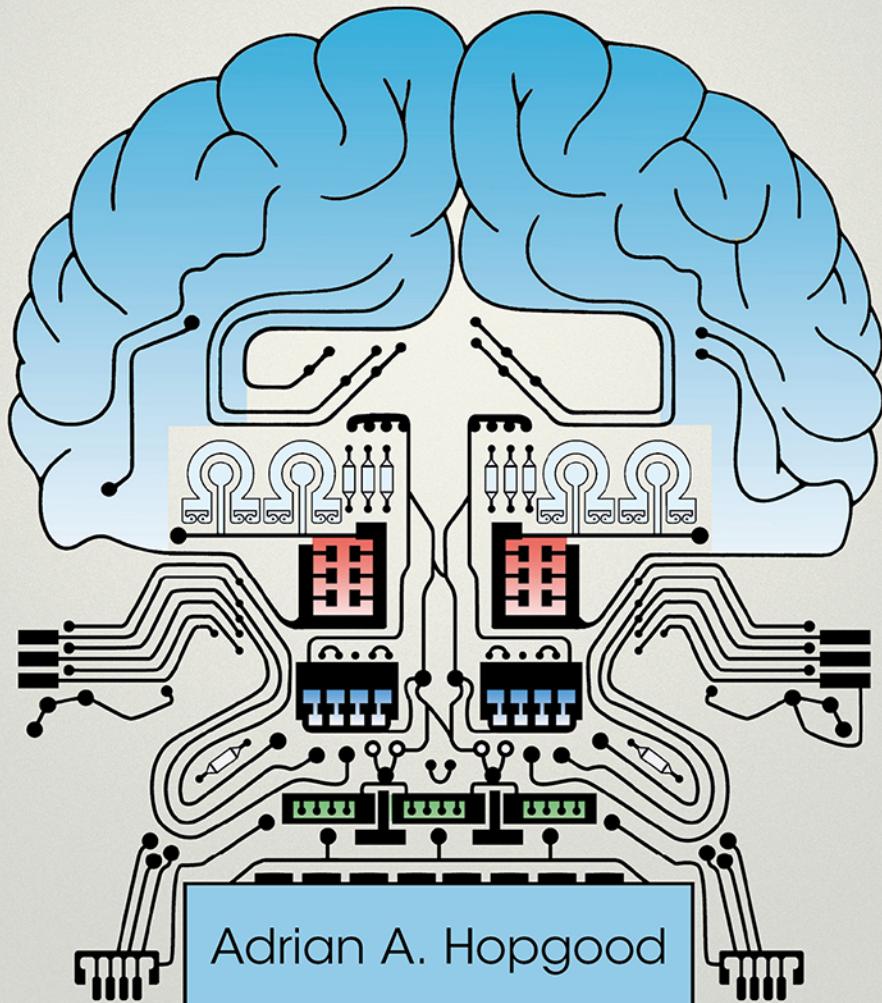


Intelligent Systems for Engineers and Scientists

A Practical Guide to Artificial Intelligence

FOURTH EDITION



 CRC Press
Taylor & Francis Group
AN AUERBACH BOOK

Intelligent Systems for Engineers and Scientists

**A Practical Guide to Artificial
Intelligence**



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Intelligent Systems for Engineers and Scientists

A Practical Guide to Artificial Intelligence

Fourth edition

Adrian A. Hopgood



CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Cover image provided by The Open University and used with its permission.
Flex, Flint, and VisiRule are trademarks of Logic programming Associates Ltd. <http://lpa.co.uk>

Fourth edition published 2022
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

© 2022 Taylor & Francis Group, LLC

First edition published by CRC Press 1993
CRC Press is an imprint of Taylor & Francis Group, LLC

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 978-0-367-33616-5 (hbk)
ISBN: 978-1-032-12676-0 (pbk)
ISBN: 978-1-003-22627-7 (ebk)

DOI: 10.1201/9781003226277

Typeset in Garamond
by SPi Technologies India Pvt Ltd (Straive)

For my family



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface	xix
Acknowledgements	xxi
Author	xxiii
1 Introduction.....	1
1.1 Artificial Intelligence and Intelligent Systems.....	1
1.2 A Spectrum of Intelligent Behavior	3
1.3 Knowledge-Based Systems (KBSS)	4
1.4 The Knowledge Base	6
1.4.1 Rules and Facts.....	6
1.4.2 Inference Networks	7
1.4.3 Semantic Networks	8
1.5 Deduction, Abduction, and Induction.....	10
1.6 The Inference Engine	11
1.7 Declarative and Procedural Programming	12
1.8 Expert Systems.....	14
1.9 Knowledge Acquisition	15
1.10 Search	15
1.11 Computational Intelligence (CI)	17
1.12 Integration with Other Software	19
Further Reading	19
2 Rule-Based Systems.....	21
2.1 Rules and Facts	21
2.2 A Rule-Based System for Boiler Control	22
2.3 Rule Examination and Rule Firing.....	24
2.4 Maintaining Consistency	25
2.5 The Closed-World Assumption	27
2.6 Use of Local Variables within Rules.....	28
2.7 Forward Chaining (a Data-Driven Strategy)	30
2.7.1 Single and Multiple Instantiation of Local Variables	31
2.7.2 Rete Algorithm.....	33

2.8	Conflict Resolution.....	35
2.8.1	First Come, First Served	35
2.8.2	Priority Values	36
2.8.3	Metarules	37
2.9	Backward Chaining (a Goal-Driven Strategy)	38
2.9.1	The Backward-Chaining Mechanism.....	38
2.9.2	Implementation of Backward Chaining.....	39
2.9.3	Variations of Backward Chaining	41
2.9.4	Format of Backward-Chaining Rules.....	43
2.10	A Hybrid Strategy	44
2.11	Explanation Facilities	48
2.12	Summary	49
	Further Reading.....	49
3	Handling Uncertainty: Probability and Fuzzy Logic.....	51
3.1	Sources of Uncertainty.....	51
3.2	Bayesian Updating	52
3.2.1	Representing Uncertainty by Probability	52
3.2.2	Direct Application of Bayes' Theorem	53
3.2.3	Likelihood Ratios	55
3.2.4	Using the Likelihood Ratios	56
3.2.5	Dealing with Uncertain Evidence	58
3.2.6	Combining Evidence.....	59
3.2.7	Combining Bayesian Rules with Production Rules	62
3.2.8	A Worked Example of Bayesian Updating	63
3.2.9	Discussion of the Worked Example	65
3.2.10	Advantages and Disadvantages of Bayesian Updating	66
3.3	Certainty Theory	68
3.3.1	Introduction.....	68
3.3.2	Making Uncertain Hypotheses	68
3.3.3	Logical Combinations of Evidence	70
3.3.3.1	Conjunction.....	71
3.3.3.2	Disjunction	71
3.3.3.3	Negation	71
3.3.4	A Worked Example of Certainty Theory.....	72
3.3.5	Discussion of the Worked Example	73
3.3.6	Relating Certainty Factors to Probabilities.....	74
3.4	Fuzzy Logic: Type-1	75
3.4.1	Crisp Sets and Fuzzy Sets	76
3.4.2	Fuzzy Rules	78
3.4.3	Defuzzification	80
3.4.3.1	Stage 1: Scaling the Membership Functions.....	81
3.4.3.2	Stage 2: Finding the Centroid.....	82
3.4.3.3	Defuzzifying at the Extremes	83

3.4.3.4	Sugeno Defuzzification.....	84
3.4.3.5	A Defuzzification Anomaly.....	85
3.5	Fuzzy Control Systems.....	86
3.5.1	Crisp and Fuzzy Control	86
3.5.2	Fuzzy Control Rules.....	87
3.5.3	Defuzzification in Control Systems	89
3.6	Fuzzy Logic: Type-2	90
3.7	Other Techniques	93
3.7.1	Dempster–Shafer Theory of Evidence.....	94
3.7.2	Inferno	95
3.8	Summary	95
	Further Reading	96
4	Agents, Objects, and Frames.....	97
4.1	Birds of a Feather: Agents, Objects, and Frames.....	97
4.2	Intelligent Agents.....	98
4.3	Agent Architectures.....	99
4.3.1	Logic-Based Architectures	100
4.3.2	Emergent Behavior Architectures	100
4.3.3	Knowledge-Level Architectures.....	100
4.3.4	Layered Architectures	101
4.4	Multiagent Systems (MASs)	102
4.4.1	Benefits of a Multiagent System.....	103
4.4.2	Building a Multiagent System.....	104
4.4.3	Contract Nets.....	105
4.4.4	Cooperative Problem-Solving (CPS)	106
4.4.5	Shifting Matrix Management (SMM).....	107
4.4.6	Comparison of Cooperative Models	108
4.4.7	Communication between Agents.....	109
4.5	Swarm Intelligence.....	111
4.6	Object-Oriented Systems.....	112
4.6.1	Introducing Object-Oriented Programming (OOP).....	112
4.6.2	An Illustrative Example	113
4.6.3	Data Abstraction	114
4.6.3.1	Classes.....	114
4.6.3.2	Instances.....	115
4.6.3.3	Attributes (or Data Members)	116
4.6.3.4	Operations (or Methods or Member Functions)	116
4.6.3.5	Creation and Deletion of Instances.....	117
4.6.4	Inheritance	119
4.6.4.1	Single Inheritance.....	119
4.6.4.2	Multiple and Repeated Inheritance.....	120
4.6.4.3	Specialization of Methods.....	124

4.6.4.4	Class Browsers	124
4.6.5	Encapsulation.....	125
4.6.6	Unified Modeling Language (UML).....	126
4.6.7	Dynamic (or Late) Binding	128
4.6.8	Message Passing and Function Calls	130
4.6.9	Metaclasses.....	131
4.6.10	Type Checking	132
4.6.11	Persistence	133
4.6.12	Concurrency	133
4.6.13	Active Values and Daemons.....	134
4.6.14	Summary of Object-Oriented Systems	135
4.7	Objects and Agents	136
4.8	Frame-Based Systems	137
4.9	Summary: Agents, Objects, and Frames	139
	Further Reading	139
5	Symbolic Learning	141
5.1	Introduction	141
5.2	Learning by Induction	143
5.2.1	Overview.....	143
5.2.2	Learning Viewed as a Search Problem.....	145
5.2.3	Techniques for Generalization and Specialization	146
5.2.3.1	Universalization.....	147
5.2.3.2	Replacing Constants with Variables	147
5.2.3.3	Using Conjunctions and Disjunctions	148
5.2.3.4	Moving Up or Down a Hierarchy.....	149
5.2.3.5	Chunking.....	149
5.3	Case-Based Reasoning (CBR)	150
5.3.1	Storing Cases.....	151
5.3.1.1	Abstraction Links and Index Links	151
5.3.1.2	Instance-Of Links.....	152
5.3.1.3	Scene Links	152
5.3.1.4	Exemplar Links	153
5.3.1.5	Failure Links.....	153
5.3.2	Retrieving Cases	153
5.3.3	Adapting Case Histories	153
5.3.3.1	Null Adaptation	154
5.3.3.2	Parameterization.....	154
5.3.3.3	Reasoning by Analogy	154
5.3.3.4	Critics	155
5.3.3.5	Reinstantiation	155
5.3.4	Dealing with Mistaken Conclusions.....	156
5.4	Summary	156
	Further Reading	157

6	Single-Candidate Optimization Algorithms.....	159
6.1	Optimization	159
6.2	The Search Space.....	160
6.3	Searching the Parameter Space	162
6.4	Hill-Climbing and Gradient-Descent Algorithms	162
6.4.1	Hill-Climbing	162
6.4.2	Steepest Gradient Descent or Ascent	163
6.4.3	Gradient-Proportional Descent or Ascent.....	163
6.4.4	Conjugate Gradient Descent or Ascent.....	163
6.4.5	Tabu Search.....	164
6.5	Simulated Annealing.....	164
6.6	Summary	168
	Further Reading.....	169
7	Genetic Algorithms for Optimization	171
7.1	Introduction: Evolutionary Algorithms	171
7.2	The Basic Genetic Algorithm	172
7.2.1	Chromosomes	172
7.2.2	Algorithm Outline	173
7.2.3	Crossover	173
7.2.4	Mutation.....	175
7.2.5	Validity Check.....	175
7.3	Selection	176
7.3.1	Selection Pitfalls	176
7.3.2	Fitness-Proportionate Selection	177
7.3.3	Fitness Scaling for Improved Selection.....	180
7.3.3.1	Linear Fitness Scaling	180
7.3.3.2	Sigma Scaling	181
7.3.3.3	Boltzmann Fitness Scaling	182
7.3.3.4	Linear Rank Scaling.....	183
7.3.3.5	Nonlinear Rank Scaling.....	184
7.3.3.6	Probabilistic Nonlinear Rank Scaling.....	184
7.3.3.7	Truncation Selection.....	185
7.3.3.8	Transform Ranking.....	185
7.3.4	Tournament Selection	186
7.3.5	Comparison of Selection Methods.....	186
7.4	Elitism	187
7.5	Multiobjective Optimization.....	187
7.6	Gray Code	188
7.7	Variable-Length Chromosomes.....	189
7.8	Building Block Hypothesis.....	191
7.8.1	Schema Theorem.....	191
7.8.2	Inversion	191

7.9	Selecting GA Parameters	192
7.10	Monitoring Evolution	192
7.11	Finding Multiple Optima	193
7.12	Genetic Programming (GP)	193
7.13	Other Forms of Population-Based Optimization	194
7.14	Summary	194
	Further Reading	195
8	Shallow Neural Networks	197
8.1	Introduction	197
8.2	Neural Network Applications	198
8.2.1	Classification	199
8.2.2	Nonlinear Estimation and Prediction	199
8.2.3	Clustering	200
8.2.4	Memory and Recall	200
8.3	Nodes and Interconnections	201
8.4	Single and Multilayer Perceptrons (SLPs and MLPs)	203
8.4.1	Network Topology	203
8.4.2	Perceptrons as Classifiers	204
8.4.3	Training a Perceptron	208
8.4.4	Hierarchical Perceptrons	211
8.4.5	Buffered Perceptrons	212
8.4.6	Some Practical Considerations	212
8.4.6.1	Overtraining	212
8.4.6.2	Leave-One-Out and K-Fold Cross-Validation	213
8.4.6.3	Data Scaling	214
8.5	Recurrent Networks	215
8.5.1	Simple Recurrent Network (SRN)	215
8.5.2	Hopfield Network	216
8.5.3	Maxnet	218
8.5.4	The Hamming Network	219
8.6	Unsupervised Networks	220
8.6.1	Adaptive Resonance Theory (ART) Networks	220
8.6.2	Kohonen Self-Organizing Networks	222
8.6.3	Radial Basis Function (RBF) Networks	223
8.7	Spiking Neural Networks (SNNs)	226
8.8	Summary	227
	Further Reading	228
9	Deep Neural Networks	229
9.1	Deep Learning	229
9.2	Convolutional Neural Networks (CNNs) for Image Recognition	230

9.2.1	Origins.....	230
9.2.2	Motivation for Convolutional Networks	230
9.2.3	CNN Structure	233
9.2.3.1	Input Layer.....	233
9.2.3.2	Feature Maps.....	235
9.2.3.3	Pooling and Classification Layers.....	237
9.2.4	Pretrained Networks and Transfer Learning.....	238
9.2.5	CNNs in Context	239
9.3	Generative Networks	239
9.3.1	Generative versus Discriminative Algorithms	239
9.3.2	Autoencoder Networks.....	240
9.3.3	Generative Adversarial Networks (GANs)	241
9.4	Long Short-Term Memory (LSTM) Networks.....	242
9.5	Summary	245
	Further Reading.....	245
10	Hybrid Systems.....	247
10.1	Convergence of Techniques.....	247
10.2	Blackboard Systems for Multifaceted Problems	248
10.3	Parameter Setting.....	250
10.3.1	Genetic–Neural Systems.....	250
10.3.2	Genetic–Fuzzy Systems	251
10.3.3	Fuzzy–Genetic Systems	252
10.4	Capability Enhancement.....	253
10.4.1	Neuro–Fuzzy Systems.....	253
10.4.2	Memetic Algorithms: Genetic Algorithms with Local Search	255
10.4.3	Learning Classifier Systems (LCSs)	257
10.5	Clarification and Verification of Neural Network Outputs.....	257
10.6	Summary	258
	Further Reading.....	258
11	AI Programming Languages and Tools	259
11.1	A Range of Intelligent Systems Tools.....	259
11.2	Features of AI Languages	261
11.2.1	Lists	261
11.2.2	Other Data Types.....	262
11.2.3	Programming Environments	264
11.3	Lisp.....	264
11.3.1	Background.....	264
11.3.2	Lisp Functions.....	265
11.3.3	A Worked Example	268
11.4	Prolog	276

11.4.1	Background.....	276
11.4.2	A Worked Example	277
11.4.3	Backtracking in Prolog	283
11.5	Python.....	287
11.5.1	Background.....	287
11.5.2	A Worked Example	288
11.6	Comparison of AI Languages.....	295
11.7	Summary.....	295
	Further Reading.....	296
12	Systems for Interpretation and Diagnosis	297
12.1	Introduction	297
12.2	Deduction and Abduction for Diagnosis.....	299
12.2.1	Exhaustive Testing.....	300
12.2.2	Explicit Modeling of Uncertainty	301
12.2.3	Hypothesize-and-Test.....	301
12.3	Depth of Knowledge.....	302
12.3.1	Shallow Knowledge.....	302
12.3.2	Deep Knowledge.....	303
12.3.3	Combining Shallow and Deep Knowledge	305
12.4	Model-Based Reasoning.....	305
12.4.1	The Limitations of Rules	305
12.4.2	Modeling Function, Structure, and State.....	306
12.4.2.1	Function.....	307
12.4.2.2	Structure	308
12.4.2.3	State.....	310
12.4.3	Using the Model	312
12.4.4	Monitoring.....	314
12.4.5	Tentative Diagnosis	315
12.4.5.1	The Shotgun Approach.....	315
12.4.5.2	Structural Isolation.....	316
12.4.5.3	The Heuristic Approach.....	316
12.4.6	Fault Simulation.....	316
12.4.7	Fault Repair.....	317
12.4.8	Using Problem Trees.....	317
12.4.9	Summary of Model-Based Reasoning	318
12.5	Case Study: A Blackboard System for Interpreting Ultrasonic Images	319
12.5.1	Ultrasonic Imaging.....	319
12.5.2	Agents in DARBS	321
12.5.3	Rules in DARBS	323
12.5.4	The Stages of Image Interpretation	325
12.5.4.1	Arc Detection Using the Hough Transform	325

12.5.4.2	Gathering the Evidence	326
12.5.4.3	Defect Classification.....	328
12.5.5	The Use of Neural Networks	328
12.5.5.1	Defect Classification Using a Neural Network.....	328
12.5.5.2	Echodynamic Classification Using a Neural Network.....	330
12.5.5.3	Combining the Two Applications of Neural Networks.....	330
12.5.6	Rules for Verifying Neural Networks	330
12.6	Summary	331
	Further Reading.....	332
13	Systems for Design and Selection	333
13.1	The Design Process	333
13.2	Design as a Search Problem.....	336
13.3	Computer-Aided Design.....	338
13.4	The Product Design Specification (PDS): A Telecommunications Case Study.....	339
13.4.1	Background.....	339
13.4.2	Alternative Views of a Network	339
13.4.3	Implementation.....	340
13.4.4	The Classes.....	340
13.4.4.1	Network	342
13.4.4.2	Link	342
13.4.4.3	Information Stream	342
13.4.4.4	Site.....	342
13.4.4.5	Equipment	343
13.4.5	Summary of PDS Case Study	343
13.5	Conceptual Design	343
13.6	Constraint Propagation and Truth Maintenance	346
13.7	Case Study: The Design of a Lightweight Beam	350
13.7.1	Conceptual Design.....	350
13.7.2	Optimization and Evaluation	352
13.7.3	Detailed Design	356
13.8	Design as a Selection Exercise	356
13.8.1	Overview.....	356
13.8.2	Merit Indices.....	357
13.8.3	The Polymer Selection Example.....	358
13.8.4	Two-Stage Selection.....	358
13.8.5	Constraint Relaxation.....	360
13.8.6	A Naive Approach to Scoring	364
13.8.7	A Better Approach to Scoring.....	366

13.8.8 Case Study: The Design of a Kettle.....	367
13.8.9 Reducing the Search Space by Classification.....	368
13.9 Failure Mode and Effects Analysis (FMEA).....	370
13.10 Summary	372
Further Reading	373
14 Systems for Planning.....	375
14.1 Introduction	375
14.2 Classical Planning Systems.....	376
14.3 Stanford Research Institute Problem Solver (STRIPS).....	378
14.3.1 General Description	378
14.3.2 An Example Problem	379
14.3.3 A Simple Planning System in Prolog	382
14.4 Considering the Side Effects of Actions.....	387
14.4.1 Maintaining a World Model.....	387
14.4.2 Deductive Rules	387
14.5 Hierarchical Planning	389
14.5.1 Description	389
14.5.2 Benefits of Hierarchical Planning	390
14.5.3 Hierarchical Planning with ABSTRIPS	392
14.6 Postponement of Commitment.....	397
14.6.1 Partial Ordering of Plans	397
14.6.2 The Use of Planning Variables	399
14.7 Job-Shop Scheduling.....	400
14.7.1 The Problem.....	400
14.7.2 Some Approaches to Scheduling.....	401
14.8 Constraint-Based Analysis (CBA)	402
14.8.1 Constraints and Preferences.....	402
14.8.2 Formalizing the Constraints	403
14.8.3 Identifying the Critical Sets of Operations.....	405
14.8.4 Sequencing in the Disjunctive Case.....	406
14.8.5 Sequencing in the Nondisjunctive Case.....	406
14.8.6 Updating Earliest Start Times and Latest Finish Times	409
14.8.7 Applying Preferences	411
14.8.8 Using Constraints and Preferences.....	413
14.9 Replanning and Reactive Planning.....	414
14.10 Summary	415
Further Reading	417
15 Systems for Control	419
15.1 Introduction	419
15.2 Low-Level Control.....	420
15.2.1 Open-Loop Control.....	420

15.2.2 Feedforward Control	421
15.2.3 Feedback Control	422
15.2.4 First- and Second-Order Models	422
15.2.5 Algorithmic Control: The PID Controller	423
15.2.6 Bang-Bang Control	425
15.3 Requirements of High-Level (Supervisory) Control	425
15.4 Blackboard Maintenance.....	427
15.5 Time-Constrained Reasoning.....	428
15.5.1 Prioritization of Processes	429
15.5.2 Approximation	429
15.5.2.1 Approximate Search.....	430
15.5.2.2 Data Approximations	431
15.5.2.3 Knowledge Approximations.....	431
15.5.3 Single and Multiple Instantiation	432
15.6 Fuzzy Control.....	433
15.7 The BOXES Controller.....	435
15.7.1 The Conventional BOXES Algorithm	435
15.7.2 Fuzzy BOXES	441
15.8 Neural Network Controllers	442
15.8.1 Direct Association of State Variables with Action Variables.....	443
15.8.2 Estimation of Critical State Variables.....	444
15.9 Statistical Process Control (SPC).....	447
15.9.1 Applications	447
15.9.2 Collecting the Data	447
15.9.3 Using the Data	449
15.10 Summary	449
Further Reading.....	451
16 The Future of Intelligent Systems	453
16.1 Benefits.....	453
16.2 Trends in Implementation.....	454
16.3 Intelligent Systems and the Internet.....	455
16.4 Computational Power	456
16.5 Ubiquitous Intelligent Systems	456
16.6 Ethics.....	458
16.7 Conclusions.....	459
References	461
Index.....	479



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Preface

About This Book

The term *intelligent systems* is broad, covering a range of artificial intelligence (AI) techniques and the ways in which they can be put into practice. It includes symbolic approaches—in which knowledge is explicitly expressed in words and symbols—and numerical approaches such as neural networks and genetic algorithms. Fuzzy logic can be considered to have a foot in both camps. In fact, many practical intelligent systems are a hybrid of different approaches. Whether any of these systems is really capable of displaying human-like intelligent behavior is a moot point. Nevertheless, they are extremely useful, and they have enabled elegant solutions to a wide variety of difficult problems.

There are plenty of other books available on artificial intelligence and related technologies, but I hope this one is substantially different. It takes a practical view, showing the issues encountered in the development of applied systems. I have deliberately described a wide range of intelligent systems techniques, with the help of realistic problems in engineering and science. The examples included here have been specifically selected for the details of the techniques that they illustrate, rather than merely to survey current practice.

The book can be roughly divided into two parts. Chapters 1 to 11 describe the techniques of artificial intelligence, while Chapters 12 to 15 look at four broad categories of applications. These latter chapters explore in depth the design and implementation issues of applied intelligent systems, together with their advantages and difficulties. The four application areas have much in common, as they all concern automated decision-making, while making the best use of the available information. Chapter 16 rounds off the book by looking at some of the trends and likely future directions for AI.

The Fourth Edition

This new edition is intended to retain the strengths of the earlier editions and to build upon them. The book has been updated throughout, with many new references, while retaining seminal sources wherever their influence remains intact. The biggest single change has been the inclusion of a new chapter on deep neural networks, reflecting the growth of machine learning as the dominant technique for AI in the 2020s. There is also a new section on the use of the Python language, which has become the de facto standard language.

As in the third edition, all the rule-based examples, including those that use uncertainty, are presented in the format of a standard artificial intelligence toolkit. The chosen toolkit comprises Flex™ and Flint™ from Logic Programming Associates Ltd., but the rules can be readily translated for other systems. All the examples have been tested and are available for download from my website.

I hope the book will appeal to a wide readership. In particular, I hope that students will be drawn toward this fascinating area from all scientific and engineering subjects, not just from the computer sciences. Beyond academia, the book will appeal to engineers and scientists who are either building intelligent systems or simply keen to know more about them.

Earlier Editions

The first edition of this book was narrower in focus and was entitled *Knowledge-Based Systems for Engineers and Scientists*. The scope of the second edition was widened to include new chapters on intelligent agents, neural networks, optimization algorithms (especially genetic algorithms), and hybrid systems. The revised title of *Intelligent Systems for Engineers and Scientists* reflected the inclusion of computational intelligence techniques alongside knowledge-based systems.

The first two editions were adopted by the Open University for its course *T396: Artificial Intelligence for Technology*, which ran successfully for 12 years. It gives me great pleasure to think of the large number of students who have been introduced to the world of intelligent systems through the book and the associated Open University course. Partly through having such a keen group of critical readers, I received a lot of useful feedback that helped remove minor errors and clarify the explanations.

The third edition was an evolution to reflect intelligent systems practice. Some of the areas that had grown in importance and were expanded included intelligent agents, fuzzy logic, genetic algorithms, and neural networks. Indeed, genetic algorithms were given their own chapter, separated from single-candidate optimization techniques. For the first time, the book was supported by a website with downloadable code.

Acknowledgements

Many people have helped me, either with this edition or the earlier ones, and I am grateful to them all. The list of people who have helped, directly or indirectly, grows with each edition and includes (in alphabetical order): Mike Brayshaw, David Carpenter, Michele Dimont, Carole Gustafson, Jon Hall, Nicholas Hallam, Jo Herbert, Tony Hirst, David Hopgood, Sue Hopgood, Adam Kowalzyk, Dawn Mesa, Priyanka Mundada, Lars Nolle, Sean Ogden, Obinwa Ozonze, Sara Parkin, Phil Picton, Chris Price, Peter Richardson, Philip Sargent, Navin Sullivan, Joy Watts, Patrick Wong, Neil Woodcock, John Wyzalek, and John Zucker.

As with the previous editions, I remain indebted to my wife for letting me get on with it.

Adrian A. Hopgood

Updates and downloads are available from adrianhopgood.com



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Author

Dr. Adrian Hopgood is Professor of Intelligent Systems and Director of Future & Emerging Technologies at the University of Portsmouth in the UK. He earned his BSc from the University of Bristol, PhD from the University of Oxford, and MBA from the Open University. After completing his PhD, he joined the AI team of Systems Designers PLC. That experience set the direction of his career toward the investigation of intelligent systems and their practical applications. After leaving Systems Designers, he spent 14 years at the Open University and remains attached as a visiting professor. During that period, he also spent two years at Telstra Research Laboratories in Australia, investigating the role of intelligent systems in telecommunications. He has subsequently worked for Nottingham Trent University, De Montfort University, Sheffield Hallam University, and the University of Liège, before joining the University of Portsmouth. Despite assuming several senior management positions during his career, he has never lost his passion for intelligent systems. His website is adrianhopgood.com.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 1

Introduction

1.1 Artificial Intelligence and Intelligent Systems

Over many centuries, tools of increasing sophistication have been developed to serve the human race. Physical tools such as chisels, hammers, spears, arrows, guns, carts, cars, and aircraft all have their place in the history of civilization. Humanity has also developed tools of communication: spoken language, written language, and the language of mathematics. These tools have not only enabled the exchange and storage of information, but they have also allowed the expression of concepts that simply could not exist outside of language.

The last century saw the arrival of a new tool—the digital computer. Computers are able to perform the same sort of numerical and symbolic manipulations that an ordinary person can, but faster and more reliably. They have therefore been able to remove the tedium from many tasks that were previously performed manually and have allowed the achievement of new feats. Such feats range from huge scientific models to the more familiar online banking facilities.

Although these developments are impressive, the computer is actually only performing quite simple operations, albeit rapidly. In such applications, the computer is still only a complex calculating machine. The intriguing idea now is whether we can build a computer (or a computer program) that can *think* (Turing 1950). As Penrose (1999) has pointed out, most of us are quite happy with machines that enable us to do physical things more easily or more quickly, such as digging a hole or traveling along a freeway. We are also happy to use machines that enable us to do physical things that would otherwise be impossible, such as flying. However, the idea of a machine that can think for us is a huge leap forward in our ambitions, and one that raises many ethical and philosophical questions.

Research in artificial intelligence (or simply AI) is directed toward building such a machine and improving our understanding of intelligence. Here is a

simple definition, adapted from Hopgood (2005) by the insertion of the phrase in parenthesis:

Artificial intelligence is the science of mimicking (or exceeding) human mental faculties in a computer.

The ultimate achievement in this field would be to construct a machine that can mimic or exceed all human mental capabilities, including reasoning, understanding, imagination, recognition, creativity, and emotions. We are a long way from achieving that ambition, but some successes have been achieved in mimicking specific areas of human mental activity. For instance, machines are now able to play chess at the highest level, to interpret spoken sentences, to translate foreign languages, and to diagnose medical complaints. An objection to these claimed successes might be that the machine does not tackle these problems in the same way that a human would. A further objection might be that the field of AI should not be limited to human intelligence, but it should include other animal species too. These objections will not be addressed in this book, which is intended as a guide to practical systems and not a philosophical thesis.

In some specialized fields, like interpreting astronomical images, AI can already exceed human capabilities (Hausen and Robertson 2020). In some other fields, the aim is not to produce the best possible form of AI, but rather to mimic human behavior or capabilities. Some examples include an AI opponent in video games, or a model of human responses in cybersecurity development.

In achieving its successes, research into AI, together with other branches of computer science, has resulted in the development of several useful computing tools that form the basis of this book. These tools have a range of potential applications, but this book emphasizes their use in engineering and science. The tools of particular interest can be roughly divided between *knowledge-based systems (KBSs)*, *computational intelligence (CI)*, and hybrid systems. KBSs include expert and rule-based systems, frame-based systems, intelligent agents, and case-based reasoning. CI includes neural networks, genetic algorithms, and other optimization algorithms. Techniques for handling uncertainty, such as fuzzy logic, fit into both categories.

Machine learning describes intelligent systems that improve their performance through experience. Typically, this phrase refers to CI and specifically to complex neural networks that learn to recognize patterns from enormous volumes of data (so-called *big data*), as described in Chapter 9: Deep Neural Networks. However, there are many other forms of data-driven machine learning, described in Chapters 6–8. Furthermore, some KBSs also have the ability to grow their knowledge, and these techniques are covered in Chapter 5: Symbolic Learning.

Intelligent systems is a broad term, covering a range of AI techniques and the ways in which they can be put into practice. They include applied KBSs, CI, machine learning, and their hybrids. Intelligent systems have not solved the problem of building an artificial mind. Indeed, some would argue that they show little, if any,

real intelligence. Nevertheless, they have enabled a range of problems to be tackled that were previously considered too difficult, and they have enabled a large number of other problems to be tackled more effectively. From a pragmatic point of view, these successes alone make intelligent systems interesting and useful.

1.2 A Spectrum of Intelligent Behavior

The definition of AI presented earlier leaves the notion of intelligence rather vague. To explore this further, a spectrum of intelligent behaviors can be drawn based on the level of understanding involved (Hopgood 2003, 2005), as shown in Figure 1.1. The lowest-level behaviors include instinctive reactions, such as withdrawing a hand from a hot object or dodging a projectile. The mid-level behaviors include vision and perception, language and interaction, and common-sense responses to unexpected events. These are capabilities that come naturally to humans, with little conscious thought. High-level behaviors demand specialist expertise such as in the legal requirements of company takeovers or the interpretation of radiological images. Such a spectrum of intelligent behaviors is useful for charting the progress of AI, although it has been criticized for oversimplifying the many dimensions of intelligence (Holmes 2003).

Conventional computing techniques have been developed to handle the low-level decision-making and control needed at the low end of the spectrum. Extremely effective computer systems have been developed for monitoring and controlling a variety of equipment. An example of the close regulation and coordination that is possible is demonstrated by various humanoid robots that show human-like mobility (Sakagami et al. 2002; Firth 2007; Gouaillier et al. 2009; Niemüller et al. 2010). As their capabilities for autonomous thought and understanding are improved through technological development, their behaviors can be expected to expand upward from the lower end of the spectrum.

Early AI research, in contrast, began with problems at the high-level end of the spectrum. Two early applications, for example, concerned the specialist areas of mass spectrometry (Buchanan et al. 1969) and bacterial blood infections (Shortliffe 1976). These early triumphs generated great optimism. If a computer could deal with difficult problems that are beyond the capabilities of most ordinary people, it was assumed that more modest human reasoning would be straightforward. Unfortunately, this assumption is false.

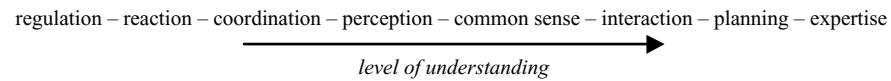


Figure 1.1 A spectrum of intelligent behavior.



Figure 1.2 The challenges of image recognition and interpretation.

The behaviors in the middle of the spectrum, that humans perform with barely a conscious thought, have proven to be the most difficult to emulate in a computer. Consider the photograph in Figure 1.2. Although most of us can recognize the three rabbits in the picture (one of which is a stone statue), the perception involved is an extremely complex behavior. First, recognizing the boundary between objects is difficult. Once an object has been delineated, recognition is far from straightforward. For instance, rabbits come in different shapes, sizes, and colors. They can assume different postures, and they may be partially occluded, like the one in the cage in Figure 1.2. Yet a fully sighted human can perform this perception in an instant, without considering it a particular mark of intelligence. The task's astonishing complexity is revealed by attempting to perform it with a computer (Hopgood 2003, 2005).

1.3 Knowledge-Based Systems (KBSs)

The principal difference between a KBS and a conventional program lies in its structure. In a conventional program, domain knowledge is intimately intertwined with software for controlling the application of that knowledge. In a KBS, the two roles are explicitly separated. In the simplest case, there are two modules: the knowledge module is called the *knowledge base*, and the control module is called the *inference engine* (Figure 1.3). In more complex systems, the inference engine itself may be

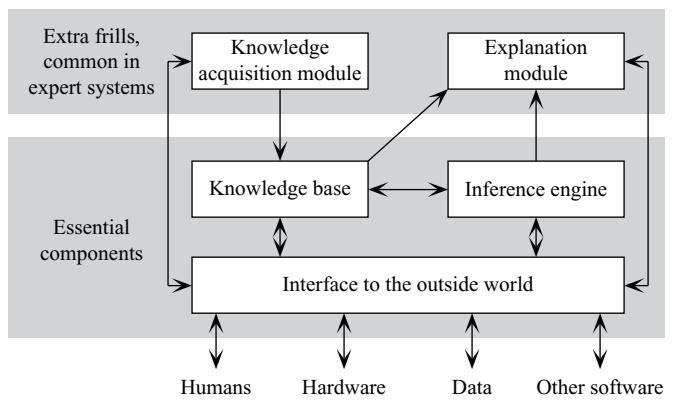


Figure 1.3 The main components of a knowledge-based system.

a KBS containing *metaknowledge*, that is, knowledge of how to apply the domain knowledge.

The explicit separation of knowledge from control of its application makes it easier to add new knowledge, either during program development or in the light of experience during the program's lifetime. There is an analogy with the brain, the control processes of which are approximately unchanging in their nature, like the inference engine. On the other hand, individual behavior is continually modified by new knowledge and experience, like updating the knowledge base.

Suppose that a professional engineer uses a conventional program to support his or her everyday work. Altering the behavior of the program would require him or her to become immersed in the details of the program's implementation. Typically, this would involve altering the control structures of the form:

`if...then...else...`

or

`for x from a to b do...`

To achieve these changes, the engineer needs to be a proficient programmer. Even if he or she does have this skill, modifications of the kind described are unwieldy and are difficult to make without unwittingly altering some other aspect of the program's behavior.

The KBS approach is more straightforward. The knowledge is represented *explicitly* in the knowledge base, not *implicitly* within the structure of a program. Thus, the knowledge can be altered with relative ease. The inference engine uses the knowledge base to tackle a particular task in a manner that is analogous to a conventional program using a data file.

1.4 The Knowledge Base

1.4.1 Rules and Facts

The knowledge base may be rich with diverse forms of knowledge. Rules and facts are among the most straightforward means of representing knowledge in a knowledge base. Nevertheless, the rules may be complex, and the facts may include sequences, structured entities, attributes of such entities, and the relationships between them. The simplest type of rule is called a production rule and takes the form:

```
if <condition> then <conclusion>
```

The details of the representation used vary from system to system. Using the syntax of the Flex™ KBS toolkit (Melioli et al. 2014; Mutawa and Alzuwawi 2019) and its VisiRule™ visual programming interface (Spenser 2007; Muraina et al. 2016), an example production rule for dealing with the payroll of ACME, Inc., might be:

```
rule r1_1
  if the employer of Person is acme
  then the salary of Person becomes large.
```

Part of the attraction of using production rules is that they can often be written in a form that closely resembles natural language, as opposed to a computer language. The use of capitalization indicates that `Person` is a local variable that can be replaced by a constant value, such as `joe_bloggs` or `mary_smith`, through the process of *pattern matching*. A fact in this KBS might be as follows, where the symbols `/*` and `*/` delineate comments that are ignored by the computer:

```
/* fact f1_1 */
  the employer of joe_bloggs is acme.
```

Facts such as this one may be made available to the KBS at the outset, in which case it is a *static fact*, or while the system is running, in which case it is a *transient fact*. In each case, these are known as *given facts*, as they are given to the KBS. One or more given facts may satisfy the condition of a rule, resulting in the generation of a new fact, known as a *derived fact*. For example, by applying rule `r1_1` to fact `f1_1`, we can derive:

```
/* derived fact f1_2 */
  the salary of joe_bloggs is large.
```

Here, the name `joe_bloggs` from fact `f1_1` has been pattern-matched to the variable `Person` from rule `r1_1` to produce the new derived fact. The derived fact may satisfy, or partially satisfy, another rule, such as:

```
rule r1_2
  if the salary of Person is large
```

or the `job_satisfaction` of Person is true
then the `professional_contentment` of Person becomes true.

This rule, in turn, may lead to the generation of a new derived fact. Rules r1_1 and r1_2 are interdependent, since the conclusion of one can satisfy the condition of the other.

1.4.2 Inference Networks

The interdependencies among rules, such as r1_1 and r1_2 given in the previous subsection, define a network. Such a network is shown in Figure 1.4 and is known as an *inference network*. The example inference network of Figure 1.4 illustrates the way in which facts can be logically combined to form new facts or conclusions. The facts can be combined logically using `and` and `or`. For example, `and` is used where *happiness* is concerned. For *happiness* to be true, both conditions, namely, *domestic bliss* and *professional contentment*, have to be true. In the inference network, this relationship is shown as a pair of arrows from the two facts to the conclusion *happiness*, with the arrows linked by `and`. *Professional contentment* looks similar, arrows from facts meeting at this conclusion, but they are linked by `or`. Thus *professional contentment* is true, if either *job satisfaction* or *large salary* is true (or both are true).

Another type of connection is `not`. According to Figure 1.4, job satisfaction is achieved through flexibility, responsibility, and the absence of stress. The negative relationship with stress is shown by a dashed line labeled `not`.

An inference network can be constructed by taking facts, and by working out what conditions have to be met for those facts to be true. After these conditions are found, they can be added to the diagram and linked by a logical expression (such

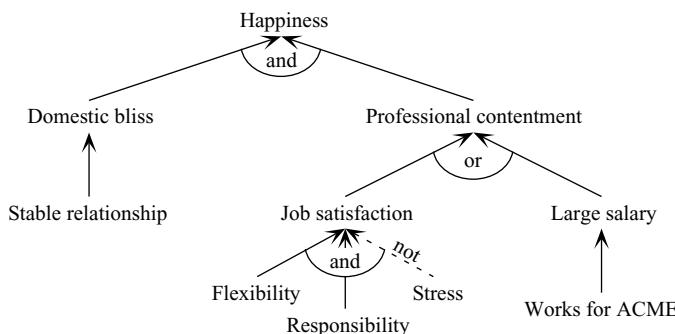


Figure 1.4 An inference network.

as and, or, not). This analysis usually involves breaking down a complex logical expression into smaller parts.

Let us now consider how we might represent these facts and the rules that link them in a conventional program. We might start by creating a data record (a data structure for grouping together different data types) for each employee. Rule r1_1 could be expressed easily enough as a conditional statement (*if...then...*), but it would need to be carefully positioned within the program so that:

- the statement is applied whenever it is needed;
- all relevant variables are in scope (the scope of a variable is that part of the program to which the declaration of the variable applies);
- any values that are assigned to variables remain active for as long as they are needed; and
- the rest of the program is not disrupted.

In effect, the facts and rules are “hard-wired,” so that they become an intrinsic part of the program. As many systems need hundreds or thousands of facts and rules, slotting them into a conventional program is a difficult task. This difficulty can be contrasted with a KBS, in which the rule and the fact are represented explicitly and can be changed at will.

Rules such as rule r1_1 are a useful way of expressing many types of knowledge and they are discussed in more detail in Chapter 2. It has been assumed so far that we are dealing with certain knowledge. This is not always the case, and Chapter 3 discusses the use of uncertainty in rules. In the case of rule r1_1, uncertainty may arise from three distinct sources:

- Uncertain evidence.
(Perhaps we are not certain that Joe Bloggs works for ACME.)
- Uncertain link between evidence and conclusion.
(We cannot be certain that a randomly selected ACME employee earns a large salary, we just know that it is likely.)
- Vague rule.
(What is a “large” salary anyway?)

The first two sources of uncertainty can be handled by Bayesian updating of probabilities or variants of this idea. The last source of uncertainty can be handled by fuzzy sets and fuzzy logic. Special types of inference networks can be constructed in which these uncertainties are attached to the inference linkages. *Bayesian inference networks* are an example that will be introduced in Chapter 3.

1.4.3 Semantic Networks

Let us now consider facts in more detail. Facts may be static, in which case they can be written into the knowledge base. Fact f1_1 falls into this category. Note

that static facts need not be permanent, but they change sufficiently infrequently that changes can be accommodated by updating the knowledge base when necessary. In contrast, some facts may be transient. Transient facts (e.g., “Oil pressure is 3000 Nm⁻²,” “the user of this program is Adrian”) apply at a specific instance only, or for a single run of the system. The knowledge base may contain defaults that can be used as facts in the absence of transient facts to the contrary. Here is a collection of English-language facts about my car:

- My car is a car (static relationship)
- A car is a vehicle (static relationship)
- A car has four wheels (static attribute)
- A car's speed is 0 mph (default attribute)
- My car is red (static attribute)
- My car is in my garage (default relationship)
- My garage is a garage (static relationship)
- A garage is a building (static relationship)
- My garage is made from brick (static attribute)
- My car is in the High Street (transient relationship)
- The High Street is a street (static relationship)
- A street is a road (static relationship)

Notice that, in this list, we have distinguished between attributes and relationships. Attributes are properties of object *instances* (such as my car) or object *classes* (such as cars and vehicles). Relationships exist among instances of objects and classes of objects. In this way, we can begin to build a model of the subject area of interest, and this reliance on a model will be a recurring topic throughout this book. Attributes and relationships can be represented as a network, known as an *associative network* or *semantic network*, as shown in Figure 1.5.

The semantic network shown in Figure 1.5 illustrates a set of facts. The ovals contain *object instances* (such as `my_car`), *classes of objects* (such as `Vehicle`), or values of *attributes* (such as `red`). Class names such as `Vehicle` are conventionally capitalized in several computer languages, a convention that has been adopted in Figure 1.5. (It is a separate convention from the capitalization of local variable names in the Flex-format rules presented earlier.) The arrows between objects represent relationships between them and are labeled according to their type, such as “is in.” Arrows are also used to link objects to their class and to the values of their attributes. So, the attribute `color` labels the link between `my_car` (an object) and `red` (an attribute value). Two ovals joined by a labeled arrow constitute a fact, such as “my car is red.”

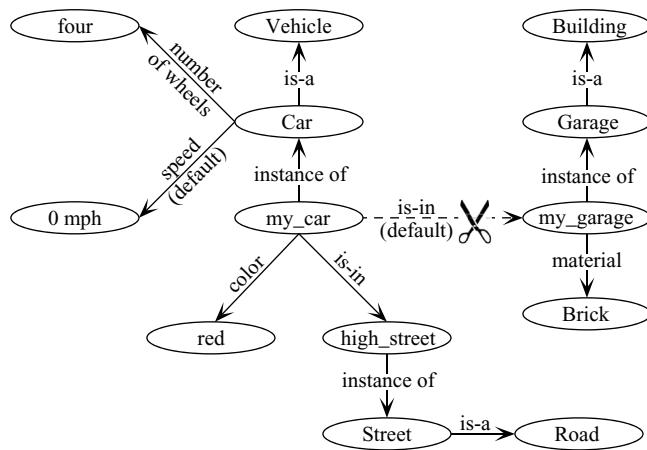


Figure 1.5 A semantic network with an overridden default. Class names are capitalized.

It is important to understand the distinction between object instances and classes of objects:

- (a) Car and Vehicle are both classes of object. They are linked in Figure 1.5 by the relation “is a,” which could be expanded to “is a sub-class of.” The direction of the arrow is important, indicating that “Car is a Vehicle” and not “Vehicle is a Car.”
- (b) On the other hand, my_car is a unique entity. There may be an identical car somewhere, but they are distinct *instances*. So, the relationship between my_car and Car takes the form “is an instance of.”

At this stage, the semantic network is an unstructured collection of object instances, classes, and attributes. In this representation, attributes are treated in the same way as relationships. In Chapter 4 we will explore agents, objects, and frames, in which relationships and attributes are represented explicitly in a formalized manner. This more structured approach turns these ideas into a powerful programming technique.

1.5 Deduction, Abduction, and Induction

The rules that make up the inference network in Figure 1.4, and the network taken as a whole, are used to link causes and effects:

```
if <cause> then <effect>
```

Using the inference network, we can infer that if Joe Bloggs works for ACME and is in a stable relationship (the causes), then he is happy (the effect). This inference is

the process of *deduction*. Many problems, such as diagnosis, involve reasoning in the reverse direction, that is, we wish to ascertain a cause, given an effect. This process is *abduction*. Given the observation that Joe Bloggs is happy, we can infer by abduction that Joe Bloggs enjoys domestic bliss and professional contentment. However, this conclusion is only valid if the inference network shows *all* of the ways in which a person can find happiness. This limitation is the *closed-world assumption*, the implications of which are discussed in Chapters 2 and 13.

The inference network therefore represents a closed world, where nothing is known beyond its boundaries. As each node represents a possible state of some aspect of the world, a model of the current overall state of the world can be maintained. Such a model is dependent on the extent of the relationships between the nodes in the inference network. In particular, if a change occurs in one aspect of the world, many other nodes could be affected. Determining what else is changed in the world model as a consequence of changing one particular thing is known as the *frame problem*. In the description of Joe Bloggs' world represented in Figure 1.4, this problem is equivalent to determining the extent of the relationships between the nodes. For example, if Joe Bloggs gets a new job, Figure 1.4 suggests that the only direct change is his salary, which could change his professional contentment and happiness. However, in a more complex model of Joe Bloggs' world, many other nodes could also be affected.

If we have many examples of causes and effects, we can infer the rule (or inference network) that links them. For instance, if every employee of ACME that we have met earns a large salary, then we might infer rule r1_1:

```
rule r1_1
  if the employer of Person is acme
  then the salary of Person becomes large.
```

Inferring a rule from a set of example cases of cause and effect is termed *induction*.

We can summarize deduction, abduction, and induction as follows:

- *deduction*: cause + rule \Rightarrow effect
- *abduction*: effect + rule \Rightarrow cause
- *induction*: cause + effect \Rightarrow rule

1.6 The Inference Engine

Inference engines vary greatly according to the type and complexity of knowledge with which they deal. Two important types of inference engines can be distinguished: *forward chaining* and *backward chaining*. These may also be known as *data driven* and *goal driven*, respectively. A KBS working in data-driven mode takes the available information (the “given” facts) and generates as many derived facts as it can. The output is therefore unpredictable. This approach may have either the advantage of

leading to novel or innovative solutions to a problem or the disadvantage of wasting time generating irrelevant information. The data-driven approach might typically be used for problems of interpretation, where we wish to know whatever the system can tell us about some data. A goal-driven strategy is appropriate when a more tightly focused solution is required. For instance, a planning system may be required to generate a plan for manufacturing a consumer product. Any other plans are irrelevant. A backward-chaining system might be presented with this proposition: “a plan exists for manufacturing a widget.” It will then attempt to ascertain the truth of this proposition by generating the plan, or it may conclude that the proposition is false and that no plan is possible. Forward and backward chaining are discussed in more detail in Chapter 2. Planning is discussed in Chapter 14.

1.7 Declarative and Procedural Programming

We have already seen that a distinctive characteristic of a KBS is that knowledge is separated from reasoning. Within the knowledge base, the programmer expresses information about the problem to be solved. Often this information is declarative, that is, the programmer states some facts, rules, or relationships without having to be concerned with the detail of *how* and *when* that information is applied. Procedural programming involves telling the computer what to *do*, but declarative programming involves telling the computer what it needs to *know*. The following are three examples of declarative programming:

```
rule r1_3
  if pressure > threshold
    then valve becomes shut.

/* declare a simple fact */
do valve_A becomes shut.

/* declare a fact in the form of a relation */
relation connected (valve_B, tank_3).
```

Each example represents a piece of knowledge that could form part of a knowledge base. The declarative programmer does not necessarily need to state explicitly how, when, and if the knowledge should be used. These details are implicit in the inference engine. An inference engine is normally programmed procedurally; a set of sequential commands is obeyed, which involves extracting and using information from the knowledge base. This task can be made explicit by using *metaknowledge* (knowledge about knowledge), for example,

```
metarule r1_4 /* not Flex format */
  examine rules about valves before rules about pipes.
```

Most conventional programming is procedural. Consider, for example, the following Python program for reading integers from a file and printing them out:

```
# open the file to read integers from a file
# and print them out
filename = input("Please enter file name: ")
file = open(filename, 'r')
try:
    # read the lines of the file
    for line in file:
        # break each line into a list of words
        mylist = line.split()
        for word in mylist:
            try:
                # convert each word to integer format and print
                myinteger = int(word)
                print(myinteger)
            except ValueError:
                # move on if a non-integer is found
                pass
finally:
    # close the file
    file.close()
```

More compact programs could be written, but this version emphasizes that there are explicit step-by-step instructions telling the computer to perform the following actions:

1. Open a data file.
2. Read each line of data from the file in turn.
3. Split each line of data into a list of words.
4. For each word in the list that is a number, convert it to integer format and store it in the variable `myinteger`.
5. Print out the value of `myinteger`.
6. If a word encountered in the list is not a number, move on.
7. Close the data file.

The data file, on the other hand, contains no instructions for the computer at all but just information in the form of a set of integers. The procedural instructions for determining what the computer should do with the integers resides in the program. The data file is therefore declarative, while the program is procedural. The data file is analogous to a trivial knowledge base, and the program is analogous to the corresponding inference engine. Of course, a proper knowledge base would be richer in content, perhaps containing a combination of rules, facts, relations, and data. The corresponding inference engine would be expected to interpret the knowledge, to combine it to form an overall view, to apply the knowledge to data, and to make decisions.

It would be an oversimplification to think that all knowledge bases are written declaratively and that all inference engines are written procedurally. In real systems, a collection of declarative information, such as a rule set, often needs to be embellished by some procedural information. Similarly, there are some inference engines that have been programmed declaratively, notably those implemented in Prolog (described in Chapter 11). Nevertheless, the declarative instructions must eventually be translated into procedural ones, as the computer can only understand procedural instructions at the machine-code level.

1.8 Expert Systems

Expert systems are a type of KBS designed to embody expertise in a particular specialized domain. Example domains might be configuring computer networks, diagnosing faults in machinery, or mineral prospecting. An expert system is intended to act as a human expert who can be consulted on a range of problems that fall within his or her domain of expertise. Typically, the user of an expert system will enter into a dialogue in which he or she describes the problem, such as the symptoms of a fault, and the expert system offers advice, suggestions, or recommendations. The dialogue may be led by the expert system, so that the user responds to a series of questions or enters information into a structured data-entry form. Alternatively, the expert system may allow the user to take the initiative in the consultation by allowing him or her to supply information without necessarily being asked for it.

Since an expert system is a KBS that acts as a specialist consultant, it is often proposed that an expert system must offer certain capabilities that mirror those of a human consultant. In particular, it is often claimed that an expert system must be capable of justifying its current line of inquiry and explaining its reasoning in arriving at a conclusion. This is the purpose of the explanation module in Figure 1.3. As the knowledge in an expert system is explicitly represented, the system can, at the very least, produce a trace of the facts and rules that have been used. The ability of a system to explain its decisions is closely linked to trust in that system. It is an advantage that KBSs have over computational intelligence, where the reasoning is more opaque.

An *expert system shell* is an expert system with an empty knowledge base. These shells are sold as software packages of varying complexity. In principle, it should be possible to buy an expert system shell, build up a knowledge base, and thereby produce an expert system. However, all domains are different, and it is difficult for a software supplier to build a shell that adequately handles them all. The best shells are flexible in their ability to represent and apply knowledge. Without this flexibility, it may be necessary to generate rules and facts in a convoluted style in order to fit the syntax or to force a certain kind of behavior from the system. This situation is not much better than building a conventional program.

1.9 Knowledge Acquisition

The representation of knowledge in a knowledge base can only be addressed once the knowledge is known. There are three distinct approaches to acquiring the relevant knowledge for a particular domain:

- The knowledge is teased out of a domain expert.
- The builder of the KBS *is* a domain expert.
- The system learns automatically from examples.

The first approach is commonly used, but it is fraught with difficulties. The person who extracts the knowledge from the expert and encodes it in the knowledge base is termed the *knowledge engineer*. Typically, the knowledge engineer interviews one or more domain experts and tries to make them articulate their in-depth knowledge in a manner that the knowledge engineer can understand. The inevitable communication difficulties can be avoided by the second approach, in which the domain expert becomes a knowledge engineer, or the knowledge engineer becomes a domain expert.

Finally, there are many circumstances in which the knowledge is either unknown or cannot be expressed explicitly. In these circumstances it may be preferable to have the system generate its own knowledge base from a set of examples. Techniques for automatic learning are discussed in Chapters 5–10.

1.10 Search

Search is the key to practically all problem-solving tasks and has been a major focus of research in intelligent systems. Problem-solving concerns the search for a solution. The detailed engineering applications discussed in this book include the search for a design, plan, control action, or diagnosis of a fault. All of these applications involve searching through the possible solutions, which comprise the *search space*, to find one or more that are optimal or satisfactory. Search is also a key issue for the internal workings of a KBS. The knowledge base may contain hundreds or thousands of rules and facts. The principal role of the inference engine is to search for the most appropriate item of knowledge to apply at any given moment (see Chapter 2).

In the case of searching the knowledge base, it is feasible (although not very efficient) to test all of the alternatives before selecting one. This approach is known as *exhaustive search*. In the search space of an application such as design or diagnosis, exhaustive search is likely to be impractical since the number of candidate solutions is so vast. A practical search therefore has to be selective. To this end, the candidate solutions that make up the search space can be organized as a *search tree*.

The expression *search tree* indicates that solutions can be categorized in some fashion, so that similar solutions are clustered together on the same branch of the tree. Figure 1.6

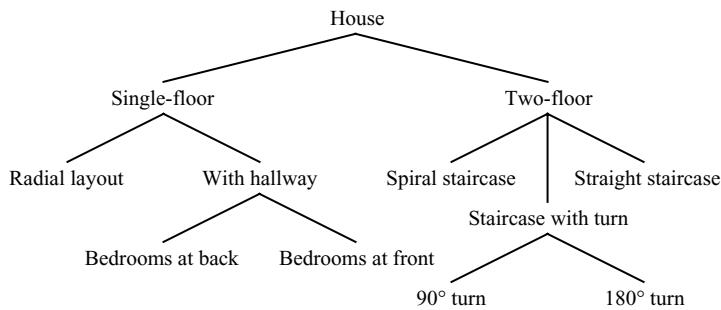


Figure 1.6 A search tree for house designs.

shows a possible search tree for designing a house. Unlike a real tree, the tree shown here has its root at the top and its leaves at the bottom. As we progress toward the leaves, the differences between the designs become less significant. Each alternative design is either generated automatically or found in a database, and then tested for suitability. This approach is described as a *generate-and-test* strategy. If a solution passes the test, the search may continue in order to find further acceptable solutions, or it may stop.

Two alternative strategies for systematically searching the tree are depth-first and breadth-first searches. An example of *depth-first* search is shown in Figure 1.7. In this example, a progressively more detailed description of a single-floor house is built up and tested before other classifications, such as a two-floor house, are considered. When a node fails the test, the search resumes at the previous node where a branch was selected. This process of *backtracking* (see Chapters 2 and 11) is indicated in Figure 1.7 by the broken arrows, which are directed toward the root rather than the leaves of the tree.

In contrast, the *breadth-first* approach (Figure 1.8) involves examining all nodes at a given level in the tree before progressing to the next level. Each node is a

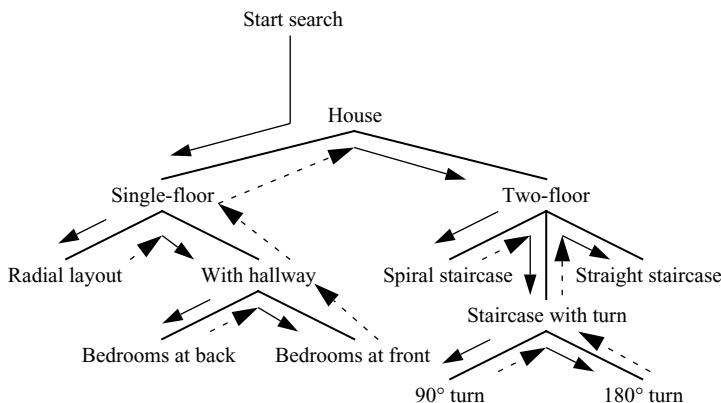


Figure 1.7 Depth-first search (the broken arrows indicate backtracking).

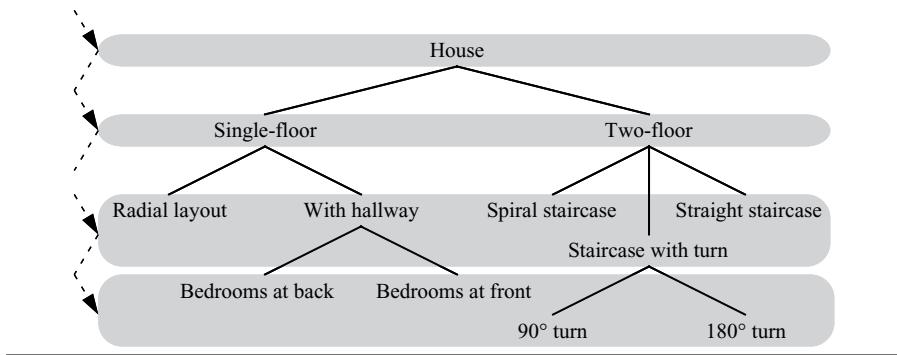


Figure 1.8 Breadth-first search.

generalization of the nodes on its subbranches. Therefore, if a node fails the test, all subcategories are assumed to fail the test and are eliminated from the search.

The systematic search strategies described in the previous paragraphs are examples of *blind search*. The search can be made more efficient either by eliminating unfeasible categories (“pruning the search tree”) or by ensuring that the most likely alternatives are tested before less likely ones. To achieve either of these aims, we need to apply heuristics to the search process. Blind search is thereby modified to *heuristic search*. Barr and Feigenbaum (1986) have surveyed the use of the word *heuristic* and produced this general description:

A heuristic is a rule of thumb, strategy, trick, simplification, or any other kind of device that drastically limits search for solutions in large search spaces. Heuristics do not guarantee optimal solutions; in fact they do not guarantee any solution at all; all that can be said for a useful heuristic is that it offers solutions that are good enough most of the time.

In a diagnostic system for plumbing, a useful heuristic may be that pipes are more likely to leak at joints than along lengths. This heuristic defines a search strategy, namely, to look for leaking joints first. In a design system for a house, we might use a heuristic to eliminate all designs that would require us to walk through a bedroom to reach the bathroom. In a short-term planning system, a heuristic might be used to ensure that no plans look ahead more than a week. In a control system for a nuclear reactor, a heuristic may ensure that the control rods are never raised while the coolant supply is turned off.

1.11 Computational Intelligence (CI)

The discussion so far has mostly concentrated on types of KBSs. They are symbolic representations, in which knowledge is explicitly represented in words and symbols that are combined to form rules, facts, relations, or other forms of knowledge

representation. As the knowledge is explicitly written, it can be read and understood by a human. As remarked in Section 1.8, this explicit representation of knowledge contributes to the system's *explainability* and the trust that can be placed in it.

These symbolic techniques (Chapters 2, 4, and 5) contrast with numerical techniques such as single-candidate optimization algorithms (Chapter 6), genetic algorithms (Chapter 7), shallow neural networks (Chapter 8), and deep neural networks (Chapter 9). There, the knowledge is not explicitly stated but is represented by numbers that are adjusted as the system improves its accuracy. Those techniques are collectively known as *CI* or *soft computing*. CI comprises data-driven forms of AI, whereas KBSs are knowledge-driven. As remarked in Section 1.1, *machine learning* is a broad term for any intelligent systems that improve their performance through experience. Nevertheless, the term is now most commonly used to describe CI techniques based around so-called *deep learning*, using deep neural networks and large data sets known as big data.

Chapter 3 describes three techniques for handling uncertainty: Bayesian updating, certainty factors, and fuzzy logic. These techniques all use a mixture of rules and associated numerical values, and they can therefore be considered as both CI tools and KBS tools. As Figure 1.9 shows, CI embraces:

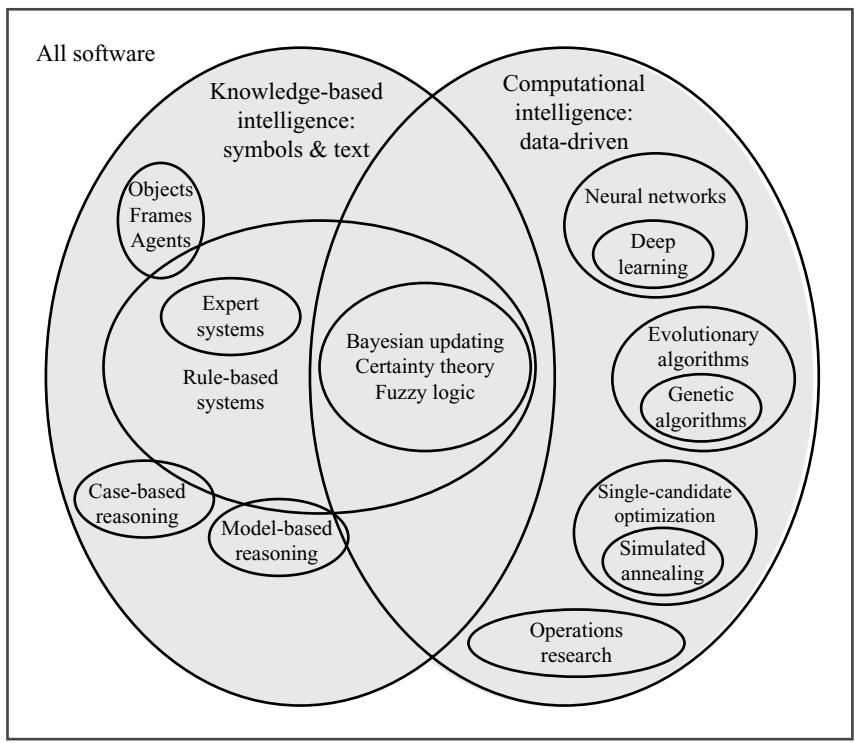


Figure 1.9 Categories of AI software.

- neural networks, including the deep neural networks most commonly associated with machine learning and big data;
- optimization algorithms including evolutionary algorithms, which in turn include genetic algorithms;
- Operations research (OR) that uses data analytics, modeling, and optimization to find solutions to problems. It predates AI as a discipline, but can be regarded as a form of CI;
- probabilistic methods such as Bayesian updating and certainty factors;
- fuzzy logic; and
- combinations of these techniques with each other and with KBSs.

1.12 Integration with Other Software

This book will take a broad view of AI and intelligent systems. It will stress the interrelationship of the various KBS and CI techniques with each other and with conventional programming (Figure 1.9). These techniques do not necessarily represent exclusive alternatives but can often be used cooperatively. For example, a designer may already have an excellent conventional program for simulating the aerodynamic properties of a car. In this case, a KBS might be used as an interface to the program, allowing the program to be used to its full potential. Similarly, an autonomous vehicle might use neural networks for image recognition, but a KBS for applying the rules of the road.

Chapter 10 describes some of the ways in which intelligent systems techniques can be used cooperatively within hybrid systems, and Chapter 11 introduces three AI programming languages. Chapters 12 to 15 describe some practical applications, most of which are hybrids of some form. If a problem can be broken down into subtasks, a blackboard system (described in Chapters 10 and 12) might provide a suitable way of tackling it. Blackboard systems allow each subtask to be handled using an appropriate technique, thereby contributing most effectively to the overall solution.

As with any other technique, KBSs and CI are not suitable for all types of problems. Each problem calls for the most appropriate tool, but KBSs, CI, and hybrids of the two can be used for many problems that would be impracticable by other means.

Further Reading

- Akerkar, R. A., and P. S. Sajja. 2009. *Knowledge Based Systems*. Jones and Bartlett, Sudbury, MA.
 Burkov, A. 2019. *The Hundred-Page Machine Learning Book*. Andriy Burkov, Quebec City, Canada.

- Carter, M. 2007. *Minds and Computers: An Introduction to the Philosophy of Artificial Intelligence*. Edinburgh University Press, Edinburgh, UK.
- Engelbrecht, A. P. 2007. *Computational Intelligence: An Introduction*. 2nd ed. John Wiley & Sons, New York, NY.
- Luger, G. F. 2008. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 6th ed. Pearson, Harlow, UK.
- Mitchell, M. 2019. *Artificial Intelligence: A Guide for Thinking Humans*. Pelican Books, London, UK.
- Negnevitsky, M. 2011. *Artificial Intelligence: A Guide to Intelligent Systems*. 3rd ed. Addison-Wesley, Reading, MA.
- Russell, S., and P. Norvig. 2016b. *Artificial Intelligence: a Modern Approach*. 3rd ed. Pearson Education, London, UK.

Chapter 2

Rule-Based Systems

2.1 Rules and Facts

A rule-based system is a knowledge-based system (KBS) where the knowledge base is represented in the form of a set, or sets, of *rules*. Rules are an elegant, expressive, straightforward, and flexible means of expressing knowledge. The simplest type of rule is called a *production rule* and takes the form:

```
if <condition> then <conclusion>.
```

An example of a production rule might be:

```
if the tap is open then water flows.
```

Part of the attraction of using production rules is that they can often be written in a form that closely resembles natural language, as opposed to a computer language. A simple rule like the foregoing one is intelligible to anyone who understands English. Although rules can be considerably more complex than this, their explicit nature still makes them more intelligible than conventional computer code.

In order for rules to be applied, and hence for a rule-based system to be of any use, the system will need to have access to *facts*. Facts are unconditional statements that are assumed to be correct at the time that they are used. For example, `the tap is open` is a fact. Facts can be:

- looked up from a database or from the Internet;
- already stored in computer memory;
- determined from sensors connected to the computer;
- obtained by prompting the user for information;
- derived by applying rules to other facts.

Given the rule if the tap is open then water flows and the fact the tap is open, the derived fact water flows can be generated. The new fact is stored in computer memory and can be used to satisfy the conditions of other rules, thereby leading to further derived facts. The collection of facts that are known to the system at any given time is called the *fact base*.

Rule-writing is a type of declarative programming (see Section 1.7), because rules represent knowledge that can be used by the computer, without specifying how and when to apply that knowledge. The ordering of rules in a program should ideally be unimportant, and it should be possible to add new rules or modify existing ones without fear of side effects. We will see by reference to some simple examples that these ideals cannot always be taken for granted.

For the declared rules and facts to be useful, an inference engine for interpreting and applying them is required (see Section 1.6). Inference engines are incorporated into a range of software tools, discussed in Chapter 11, including expert system shells, artificial intelligence (AI) toolkits, software libraries, and the Prolog language.

Rule-based systems have been used in many practical applications since the 1980s and are the dominant form of AI in computer games (Gorman and Humphrys 2007).

2.2 A Rule-Based System for Boiler Control

Whereas the foregoing discussion describes rule-based systems in an abstract fashion, a physical example is introduced in this section. We will consider a rule-based system to monitor the state of a power-station boiler and to advise appropriate actions. The boiler in our example (Figure 2.1) is used to produce steam to drive a turbine and generator. Water is heated in the boiler tubes to produce a steam and water mixture that rises to the steam drum, which is a cylindrical vessel mounted horizontally near the top of the boiler. The purpose of the drum is to separate the steam from the water. Steam is taken from the drum, passed through the superheater, and applied to the turbine that turns the generator. Sensors are fitted to the drum in order to monitor:

- the temperature of the steam in the drum;
- the voltage output from a transducer, which in turn monitors the level of water in the drum;
- the status of pressure release valve (i.e., open or closed);
- the rate of flow of water through the control valve.

The following rules have been written for controlling the boiler, using the syntax of the Flex™ KBS toolkit (Melioli et al. 2014; Mutawa and Alzuwawi 2019) and its VisiRule™ visual programming interface (Spenser 2007; Muraina et al. 2016):

```
rule r2_1
  if water_level is low
  then report('** open the control valve! **').
```

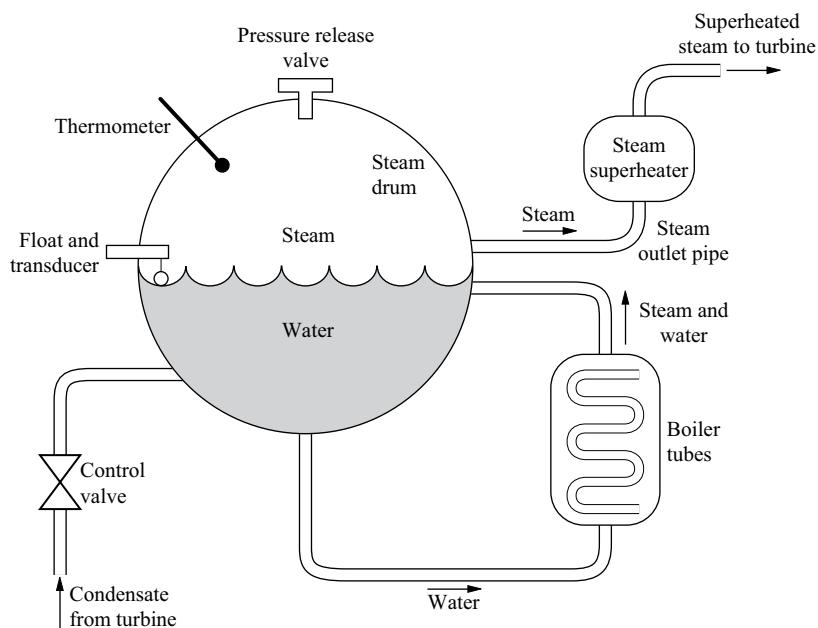


Figure 2.1 A power-station boiler.

```

rule r2_2
  if temperature is high
  and water_level is low
  then report('** open the control valve! **')
  and report('** shut down the boiler tubes! **') .

rule r2_3
  if steam_outlet is blocked
  then report('** outlet pipe needs replacing! **') .

rule r2_4
  if release_valve is stuck
  then steam_outlet becomes blocked.

rule r2_5
  if pressure is high
  and release_valve is closed
  then release_valve becomes stuck.

rule r2_6
  if steam is escaping
  then steam_outlet becomes blocked.

```

```

rule r2_7
  if temperature is high
  and water_level is not low
  then pressure becomes high.

rule r2_8
  if transducer_output is low
  then water_level becomes low.

rule r2_9
  if release_valve is open
  and flow_rate is high
  then steam becomes escaping.

rule r2_10
  if flow_rate is low
  then control_valve becomes closed.

```

The conclusions of three of these rules (r2_1, r2_2, and r2_3) consist of recommendations to the boiler operators. In a fully automated system, such rules would be able to perform their recommended actions rather than simply making a recommendation. The remaining rules all involve taking a low-level fact, such as a transducer reading, and deriving a higher-level fact, such as the quantity of water in the drum. The input data to the system (sensor readings in our example) are low-level facts; higher-level facts are facts derived from them.

Most of the rules in our rule base are specific to one particular boiler arrangement and would not apply to other situations. These rules could be described as *shallow*, because they represent shallow knowledge. On the other hand, rule r2_7 expresses a fundamental rule of physics, namely that the boiling temperature of a liquid increases with increasing applied pressure. This is valid under any circumstances and is not specific to the boiler shown in Figure 2.1. It is an example of a *deep* rule expressing deep knowledge.

The distinction between deep and shallow rules should not be confused with the distinction between *high-level* and *low-level* rules. Low-level rules are those that depend on low-level facts. Rule r2_8 is a low-level rule since it is dependent on a transducer reading. High-level rules make use of more abstract information, such as rule r2_3, which relates the occurrence of a steam outlet blockage to a recommendation to replace a pipe. Higher-level rules are those that are closest to providing a solution to a problem, while lower-level rules represent the first stages toward reaching a conclusion.

2.3 Rule Examination and Rule Firing

In Section 2.2, a rule base for boiler control was described without mention of how the rules would be applied. The task of interpreting and applying the rules belongs

to the inference engine (see Chapter 1). The application of rules can be broken down as follows:

- (i) Selecting rules to examine—these are the *available* rules.
- (ii) Determining which of these are applicable—these are the *triggered* rules; they make up the *conflict set*.
- (iii) Selecting a rule to *fire* (described in the following text).

The distinction between the examination and firing of rules is best explained by an example. Suppose the rule-based system has access to the transducer output and to the temperature readings. A sensible set of rules to *examine* would be r2_2, r2_7, and r2_8, as these rules are conditional on the boiler temperature and transducer output. If the transducer level is found to be low, then rule r2_8 is applicable. If it is selected and used to make the deduction `water_level becomes low`, then the rule is said to have *fired*. If the rule is examined but cannot fire (because the transducer reading is not low), the rule is said to *fail*.

The condition part of rule r2_2 can be satisfied only if rule r2_8 has been fired. For this reason, it makes sense to examine rule r2_8 before rule r2_2. If rule r2_8 fails, then rule r2_2 need not be examined as it too will fail. The interdependence between rules is discussed further in Sections 2.4 and 2.5.

The method for rule examination and firing described so far is a form of forward chaining. This strategy and others are discussed in more detail in Sections 2.7 through 2.10.

2.4 Maintaining Consistency

A key advantage of rule-based systems is their flexibility. New rules can be added at will, but only if each rule is written with care and without assuming the behavior of other rules. Consider rule r2_4:

```
rule r2_4
  if release_valve is stuck
  then steam_outlet becomes blocked.
```

Given the current rule base, the fact `release_valve is stuck` could only be established by first firing rule r2_5:

```
rule r2_5
  if pressure is high
  and release_valve is closed
  then release_valve becomes stuck.
```

Rule r2_5 is sensible, since the purpose of the release valve is to open itself automatically if the pressure becomes high, thereby releasing the excess pressure. Rule r2_4,

however, is less sensible. The fact `release_valve` is stuck is not in itself sufficient evidence to deduce that the steam outlet is blocked. The other necessary evidence is that the pressure in the drum must be high. The reason that the rule base works in its current form is that, in order for the system to believe the release valve to be stuck, the pressure *must* be high. Although the rule base works, it is not robust and is not tolerant of new knowledge being added. Consider, for instance, the effect of adding the following rule:

```
rule r2_11
  if pressure is low
  and release_valve is open
  then release_valve becomes stuck.
```

This rule is in itself sensible. However, the addition of the rule has an unwanted effect on rule r2_4. Because of the unintended interaction between rules r2_4 and r2_11, low pressure in the drum and the observation that the release valve is open result in the erroneous conclusion that the steam outlet is blocked. Problems of this sort can be avoided by making each rule an accurate statement in its own right. Thus, in our example, rule r2_4 should be written as:

```
rule r2_4a
  if pressure is high
  and release_valve is stuck
  then steam_outlet becomes blocked.
```

A typical rule firing order, given that the drum pressure is high and the release valve is closed, might be:

```
rule r2_5
  if pressure is high
  and release_valve is closed
  then release_valve becomes stuck.
  ↓
rule r2_4a
  if pressure is high
  and release_valve is stuck
  then steam_outlet becomes blocked.
  ↓
rule r2_3
  if steam_outlet is blocked
  then report('** outlet pipe needs replacing! **').
```

The modification that has been introduced in rule r2_4a means that the conditions of both rules r2_5 and r2_4a involve checking to see whether the drum pressure is high. This source of inefficiency can be justified through the improved robustness of the rule base. In fact, the Rete algorithm, described in Section 2.7.2, allows rule conditions to be duplicated in this way with minimal loss of efficiency.

In general, rules can be considerably more complex than the ones we have considered so far. For instance, rules can contain combinations of conditions and conclusions, exemplified by combining rules r2_3, r2_4, and r2_6 to form a new rule:

```
rule r2_12
  if [pressure is high and release_valve is stuck]
  or steam is escaping
  then steam_outlet becomes blocked
  and report('** outlet pipe needs replacing! **') .
```

Here, the first two subconditions have been bracketed to avoid any ambiguity over which combination of subconditions needs to be satisfied for the rule to fire.

2.5 The Closed-World Assumption

If we do not know that a given proposition is true, then in most rule-based systems the proposition is assumed to be false. This assumption, known as *the closed-world assumption*, simplifies the logic required as all propositions are either *true* or *false*. If the closed-world assumption is not made, then a third category, namely *unknown*, has to be introduced. The Flex rule syntax emphasizes its closed-world assumption through the use of the word “becomes” in rule conclusions, to make clear that a proposition contained in a conclusion is only made when the rule fires.

To illustrate the closed-world assumption, let us return to the boiler-control example. If `steam_outlet is blocked` is not known, then `steam_outlet is not blocked` is assumed to be true. Similarly, if `water_level is low` is not known, then `water_level is not low` is assumed to be true. The latter example of the closed-world assumption affects the interaction between rules r2_7 and r2_8:

```
rule r2_7
  if temperature is high
  and water_level is not low
  then pressure becomes high.

rule r2_8
  if transducer_output is low
  then water_level becomes low.
```

Consider the case where the temperature reading is high and the transducer output is low. Whether or not the pressure is assumed to be high will depend on the order in which the rules are selected for firing. If we fire rule r2_7 followed by r2_8, the following deductions will be made:

`temperature is high`—*true* as a given fact;
`water_level is not low`—*true* by the closed-world assumption;
`pressure is high`—*true* by the application of rule r2_7;

```
transducer_output is low—true as a given fact;
water_level is low—true by the application of rule r2_8.
```

Alternatively, we could examine rule r2_8 first:

```
transducer_output is low—true as a given fact;
water_level is low—true by the application of rule r2_8;
temperature is high—true as a given fact;
water_level is not low—false as a derived fact, so rule r2_7 fails.
```

It is most likely that the second outcome was intended by the rule-writer. There are two measures that could be taken to avoid this ambiguity, namely, to modify the rules or to modify the inference engine. The latter approach would aim to ensure that rule r2_8 is examined before r2_7, and a method for achieving this is described in Section 2.10. The former solution could be achieved by altering the rules so that they do not contain any negative conditions, as shown in the following version:

```
rule r2_7a
  if temperature is high
  and water_level is not_low
  then pressure becomes high.

rule r2_8
  if transducer_output is low
  then water_level becomes low.

rule r2_8a
  if transducer_output is not_low
  then water_level becomes not_low.
```

2.6 Use of Local Variables within Rules

The boiler shown in Figure 2.1 is a simplified view of a real system, and the accompanying ruleset is much smaller than those associated with most real-world problems. In real-world systems, local variables can be used to make rules more general, thereby reducing the number of rules needed and keeping the ruleset manageable. The sort of rule that is often required is of the form:

```
for all x, if <condition about x> then <conclusion about x>.
```

To illustrate this idea, let us imagine a more complex boiler. This boiler may, for instance, have many water-supply pipes, each with its own control valve. For each pipe, the flow rate will be related to whether or not the control valve is open. So, some possible rules might be of the form:

```
rule r2_13
  if the control_valve of tube_1 is open
  then the flow_rate of tube_1 becomes high.
```

```

rule r2_14
  if the control_valve of tube_2 is open
    then the flow_rate of tube_2 becomes high.

rule r2_15
  if the control_valve of tube_3 is open
    then the flow_rate of tube_3 becomes high.

rule r2_16
  if the control_valve of tube_4 is open
    then the flow_rate of tube_4 becomes high.

rule r2_17
  if the control_valve of tube_5 is open
    then the flow_rate of tube_5 becomes high.

```

A much more compact, elegant, and flexible representation of these rules would be:

```

rule r2_18
  if the control_valve of Tube is open
    then the flow_rate of Tube becomes high.

```

Here we have used a capitalization to denote that `Tube` is a local variable. Now if the sensors detect that a control valve in any tube is open, the identity of the control valve is matched to the value of `Tube` when the rule is fired. The local variable `Tube` is said to be *instantiated* with a value, in this case the identity of a specific tube. Thus, if the control valve of `tube_3` is open, `Tube` is instantiated with the value `tube_3` and the deduction `flow_rate of tube_3 becomes high` is made.

A local variable like `Tube` is local in the sense that its value is contained only within the rule and is not available to any other part of the program. The value may, however, be passed to a procedure or function that is directly called upon within the rule. The *scope* of a local variable is the part of the program where it has a single meaning, in this case a rule. Where the same local variable name appears more than once within a rule or a fact, it is recognized as the same local variable. However, the same local variable name can appear quite independently in another rule or fact. A value is assigned automatically to a local variable by the process of *pattern-matching*.

In this example, the possible values of `Tube` were limited, and so the use of a local variable was convenient rather than necessary. Where the possible values of a local variable cannot be anticipated in advance, the use of local variables becomes essential. This is the case when values are being looked up, perhaps from a database or from a sensor. As an example, consider the following rule:

```

rule r2_19
  if drum_pressure is P
  and P > threshold then
    tube_pressure becomes P/10.

```

Without the use of the local variable name `P`, it would not be possible to generate a derived fact that states explicitly a pressure value. Suppose that the drum pressure sensor is reading a value of 300 MNm^{-2} and `threshold` is a variable currently set to 100 MNm^{-2} . Rule `r2_19` can therefore be fired, and the derived fact is generated that `tube_pressure` is $30 (\text{MNm}^{-2} \text{ assumed})$. Here, `drum_pressure`, `threshold`, and `tube_pressure` are all examples of *global* variables. They can be accessed throughout the rule-based system and the correct value obtained.

Note that, in this example, the value of the local variable `P` has been manipulated, that is, divided by 10. More sophisticated rule-based systems allow values represented as local variables to be manipulated in this way or passed as parameters to procedures and functions.

The association of a specific value (say, 300 MNm^{-2}) with a local variable name (such as `P`) is sometimes referred to as *unification*. The term applies not only to numerical examples but to any form of data. The word arises because, from the computer's perspective, the following are contradictory pieces of information:

```
drum_pressure is P
drum_pressure is 300 (units of MNm-2 assumed).
```

This conflict can be resolved by recognizing that one of the values is a local variable name (because in this syntax it is capitalized) and by making the following assignment or unification:

```
P becomes 300 .
```

The use of local variable names within rules is integral to the Prolog language (see Chapter 11). In Prolog, just as in the Flex AI toolkit, local variables are distinguished from constants by having an underscore or uppercase letter as their first character.

2.7 Forward Chaining (a Data-Driven Strategy)

As noted in Section 2.3, the inference engine applies a strategy for deciding which rules to apply and when to apply them. Forward chaining is the name given to a data-driven strategy, that is, rules are selected and applied in response to the current fact base. The fact base comprises all facts known by the system, whether derived by rules or supplied directly (see Section 2.1).

A schematic representation of the cyclic selection, examination, and firing of rules is shown in Figure 2.2. The cycle of events shown in Figure 2.2 is just one version of forward chaining, and variations in the strategy are possible. The key points of the scheme shown in Figure 2.2 are as follows:

- Rules are examined and fired on the basis of the current fact base, independently of any predetermined goals.

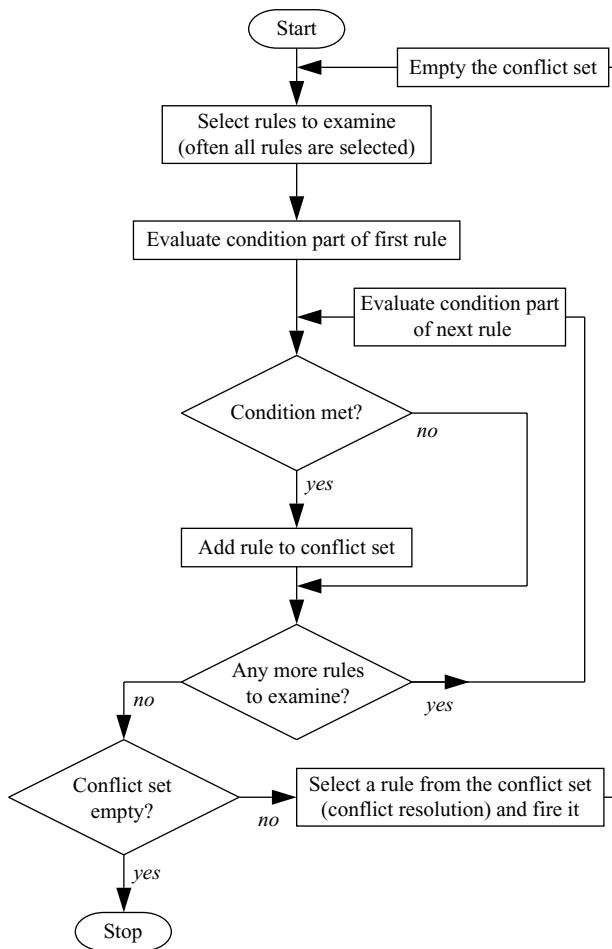


Figure 2.2 Forward chaining.

- The set of rules available for examination may comprise *all* of the rules or a subset.
- Of the available rules, those whose conditions are satisfied are said to have been *triggered*. These rules make up the *conflict set*, and the method of selecting a rule from the conflict set is *conflict resolution* (Section 2.8).
- Although the conflict set may contain many rules, only one rule is fired on a given cycle. This is because once a rule has fired, the stored deductions have potentially changed, and so it cannot be guaranteed that the other rules in the conflict set still have their condition parts satisfied.

2.7.1 Single and Multiple Instantiation of Local Variables

As noted earlier, variations on the basic scheme for forward chaining are possible. Where local variables are used in rules, the conclusions may be performed using just

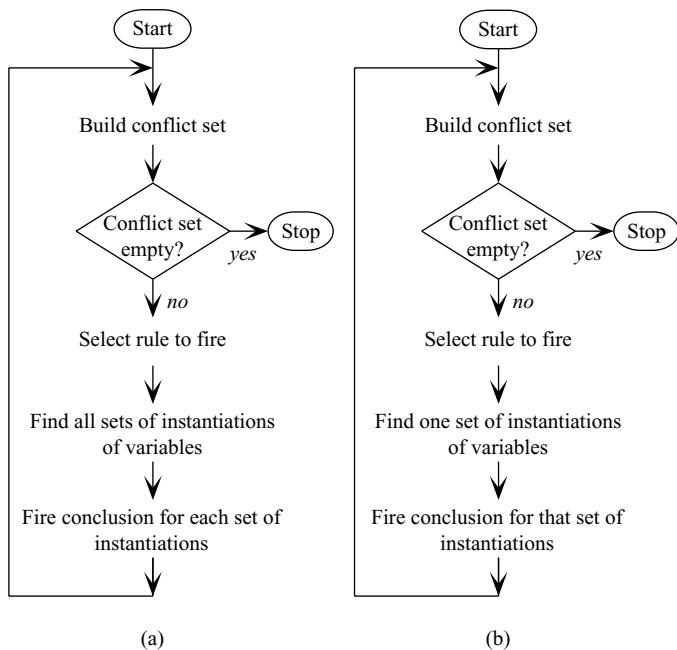


Figure 2.3 Alternative forms of forward chaining: (a) multiple instantiation of local variables; (b) single instantiation of local variables.

the first set of instantiations that are found—this is *single instantiation*. Alternatively, the conclusions may be performed repeatedly using all possible instantiations—this is *multiple instantiation*. The difference between the two approaches is shown in Figure 2.3. As an example, consider the following pair of rules:

```
rule r_20
    if control_valve of Tube is open
    then flow_rate of Tube becomes high.
```

```
rule r_21
    if flow_rate of Tube is high
    then control valve of Tube becomes needs_closing.
```

Suppose that we start with two facts:

control_valve of tube1 is open.
control valve of tube2 is open.

Under multiple instantiation, each rule would fire once, generating conclusions in the following order:

flow_rate of tube1 becomes high.
flow rate of tube2 becomes high.

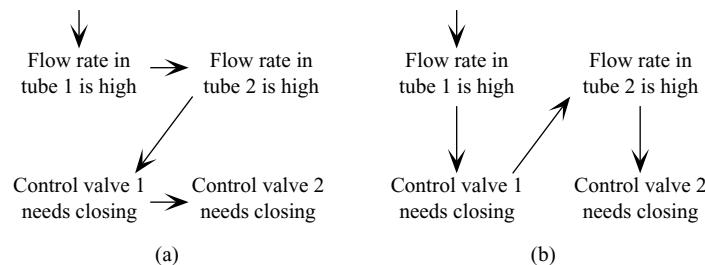


Figure 2.4 Applying rules r2_20 and r2_21: (a) multiple instantiation is a breadth-first process; (b) single instantiation is a depth-first process.

control_valve of tube1 becomes needs_closing.
control_valve of tube2 becomes needs_closing.

A different firing order would occur under single instantiation. Each cycle of the inference engine would result in a rule firing on a single instantiation of the local variable `Tube`. After four cycles, conclusions would have been generated in the following order:

```
flow_rate of tube1 becomes high.  
control_valve of tube1 becomes needs_closing.  
flow_rate of tube2 becomes high.  
control_valve of tube2 becomes needs_closing.
```

Multiple instantiation is a breadth-first approach to problem-solving and single instantiation is a depth-first approach, as illustrated in Figure 2.4.

Chapter 12 introduces an application involving the interpretation of ultrasonic images. Typically, several rules need to be fired for each occurrence of a particular feature in the image. Figure 2.5 shows an example from this system, where the task is to examine all rectangular areas larger than a specified size. This task can be achieved by either single or multiple instantiation, but the order in which the image areas are processed is different in each case. In this example, there is no strong reason to prefer one strategy over the other, although multiple instantiation is usually more efficient. Despite the reduced efficiency, it will be shown in Chapter 15 that single instantiation may be preferable in problems where a solution must be found within a limited time frame. This surprising finding is due to the fact that forward chaining under single instantiation progresses rapidly through the rule-firing sequence from the lowest-level rules to the highest-level rules, albeit only for only one instantiation at a time.

2.7.2 Rete Algorithm

The scheme for forward chaining shown in Figure 2.2 contains at least one source of inefficiency. Once a rule has been selected from the conflict set and fired, the conflict set is thrown away and the process starts all over again. This is because firing a rule

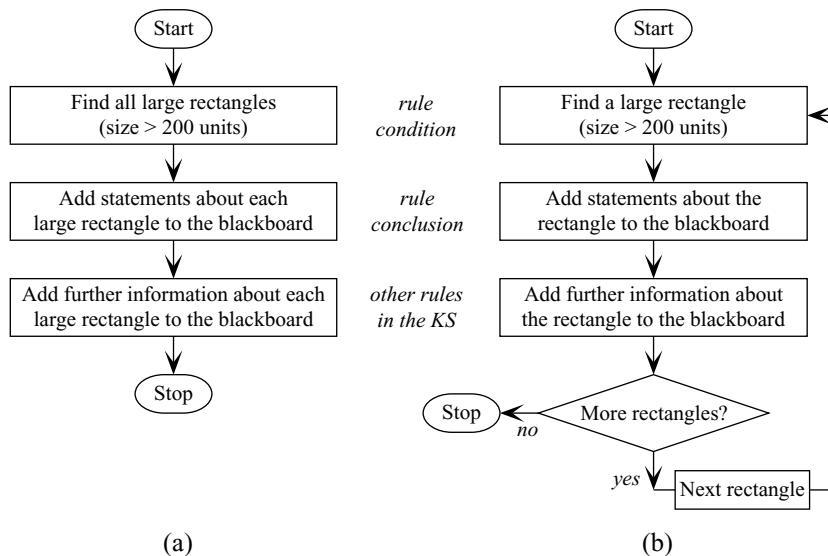


Figure 2.5 Application of a rule for examining rectangles in an image: (a) multiple instantiation of local variables; (b) single instantiation of local variables.

alters the fact base, so that a different set of rules may qualify for the conflict set. A new conflict set is therefore drawn up by reexamining the condition parts of all the available rules. In most applications, the firing of a single rule makes only slight changes to the fact base and hence to the membership of the conflict set. Therefore, a more efficient approach would be to examine only those rules whose condition is affected by changes made to the fact base on the previous cycle. The Rete (pronounced “ree-tee”) algorithm (Forgy 1982; Berstel 2002) is one way of achieving this aim.

The principle of the Rete algorithm can be shown by a simple example, using the following rule:

```
rule r2_22
  if bore of Pipe = bore of Valve
  then compatible_valve of Pipe becomes Valve
  and compatible_pipe of Valve becomes Pipe.
```

Prior to running the system, the condition parts of all the rules are assembled into a *Rete network*, where each node represents an atomic condition, that is, one that contains a simple test. There are two types of nodes: alpha nodes can be satisfied by a single fact, whereas beta nodes can only be satisfied by a pair of facts. The condition part of rule r2_22 would be broken down into two alpha nodes and one beta node:

- α1: find a pipe.
- α2: find a valve.
- β1: the bore of each must be equal.

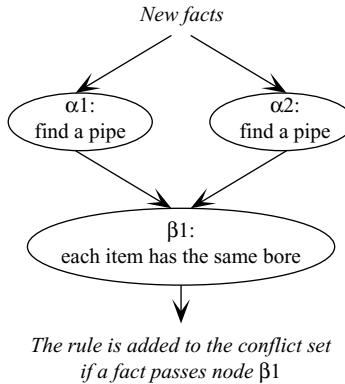


Figure 2.6 A Rete network for rule r2_22.

The Rete network for this example is shown in Figure 2.6. Suppose that, initially, the only relevant fact is:

bore of pipe1 is 100mm.

Node α_1 would be satisfied, and so the fact would be passed on to node β_1 . However, node β_1 would not be satisfied as it has received no information from node α_2 . The fact that there is a pipe of bore 100 mm would remain stored at node β_1 . Imagine now that, as a result of firing other rules, the following fact is derived:

bore of valve1 is 100mm.

This fact satisfies node α_2 and is passed on to node β_1 . Node β_1 is satisfied by the combination of the new fact and the one that was already stored there. Thus, rule r2_22 can be added to the conflict set without having to find a pipe again (the task of node α_1).

A full Rete network would contain nodes representing the subconditions of all the rules in the rule base. Every time a rule is fired, the altered facts would be fed into the network and the changes to the conflict set generated. Where rules contain identical subconditions, nodes can be shared, thereby avoiding duplicated testing of the conditions. In an evaluation of some commercially available AI toolkits that use forward chaining, those that incorporated the Rete algorithm were found to offer substantial improvements in performance (Mettrey 1991; Schmedding et al. 2007).

2.8 Conflict Resolution

2.8.1 First Come, First Served

As noted previously, conflict resolution is the method of choosing one rule to fire from those that are able to fire, that is, from the set of triggered rules, known as

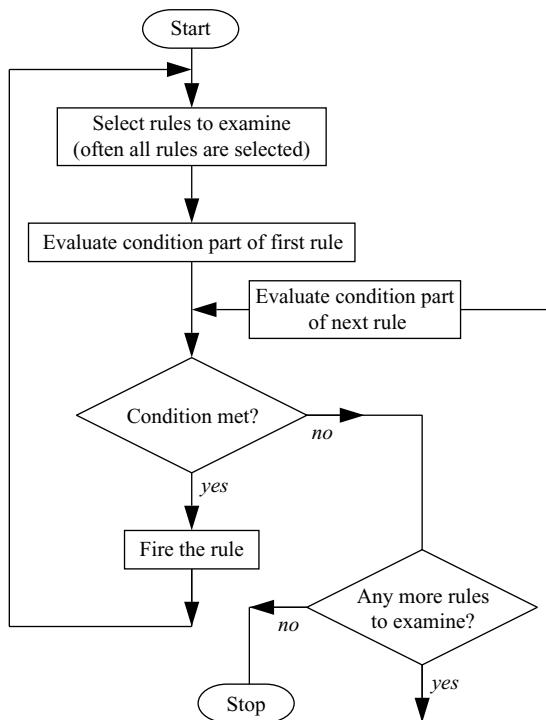


Figure 2.7 Forward chaining with “first come, first served” conflict resolution.

the conflict set. In Figure 2.2, the complete conflict set is found before choosing a rule to fire. Since only one rule from the conflict set can actually fire on a given cycle, the time spent evaluating the condition parts of the other rules is wasted unless the result is saved by using a Rete algorithm or similar technique. A strategy that overcomes this inefficiency is to fire immediately the first rule to be found that qualifies for the conflict set (Figure 2.7). In this “first come, first served” scheme, the conflict set is not assembled at all, and the order in which rules are selected for examination determines the resolution of conflict. The order of rule examination is often simply the order in which the rules appear in the rule base. If the rule-writer is aware of this, the rules can be ordered in accordance with their perceived priority.

2.8.2 Priority Values

Rather than relying on rule ordering as a means of determining rule priorities, rules can be written so that each has an explicitly stated priority value, known as a *score* in Flex. Where more than one rule is able to fire, the one chosen is the one having the highest priority. The two following rules would be available for firing if the water

level had been found to be low (i.e., rule r2_8 had fired) and if the temperature were high:

```
rule r2_1a
  if water_level is low
  then report('** open the control valve! **')
  score 4.0 .

rule r2_2a
  if temperature is high
  and water_level is low
  then report('** open the control valve! **')
  and report('** shut down the boiler tubes! **' )
  score 9.0 .
```

In the scheme shown here, rule r2_2a would be selected for firing as it has the higher priority value. The absolute values of the priorities (or scores) are not important; it is their relative values that affect the firing order of the rules.

As the examination of rules that are not fired represents wasted effort, an efficient use of priorities would be to select rules for examination in the order of their priority. Once a rule has been found that is fireable, it could be fired immediately. This scheme is identical to the “first come, first served” strategy shown in Figure 2.7, except that rules are selected for examination according to their priority value rather than their position in the rule base.

2.8.3 Metarules

Metarules are rules that are not specifically concerned with knowledge about the application at hand, but rather with knowledge about how that knowledge should be applied. Metarules are therefore “rules about rules” (or more generally, “rules about knowledge”). Some examples of metarules might be:

```
metarule r2_23 /* not Flex format */
  prefer rules about shutdown to rules about control valves.

metarule r2_24 /* not Flex format */
  prefer high-level rules to low-level rules.
```

If rules r2_1 and r2_2 are both in the conflict set, metarule r2_23 will be fired, with the result that rule r2_2 is then fired. If a conflict arises for which no metarule can be applied, then a default method such as “first come, first served” can be used.

When a conflict-resolution strategy based on priority values is used, it would be possible to write a metarule that modifies the priority attached to a rule. Techniques like this provide a way of supporting a simple form of learning within a rule-based system, as the rule base becomes altered by experience. Learning in KBSs is the subject of Chapter 5.

2.9 Backward Chaining (a Goal-Driven Strategy)

2.9.1 The Backward-Chaining Mechanism

Backward chaining is an inference strategy that assumes the existence of a goal that needs to be established or refuted. In the boiler-control example, our goal might be to establish whether it is appropriate to replace the outlet pipe, and we may not be interested in any other deductions that the system is capable of making. Backward chaining provides the means for achieving this objective. Initially, only those rules that can lead directly to the fulfillment of the goal are selected for examination. In our case, the only rule that can achieve the goal is rule r2_3, since it is the only rule whose conclusion is `report('** outlet pipe needs replacing! **')`. The condition part of rule r2_3 is examined but, since there is no information about a steam outlet blockage in the fact base, rule r2_3 cannot be fired yet. A new goal is then produced, namely, `steam_outlet is blocked`, corresponding to the condition part of rule r2_3. Two rules, r2_4 and r2_6, are capable of fulfilling this goal and are therefore *antecedents* of rule r2_3. What happens next depends on whether a depth-first or breadth-first search strategy is used. These two methods for exploring a search tree were introduced in Chapter 1, but now the nodes of the search tree are *rules*.

For the moment we will assume the use of a depth-first search strategy, as this is normally adopted. The use of a breadth-first search is discussed in Section 2.9.3. One of the two relevant rules (r2_4 or r2_6) is selected for examination. Let us suppose that rule r2_6 is chosen. Rule r2_6 can fire only if steam is escaping from the drum. This information is not in the fact base, so `steam is escaping` becomes the new goal. The system searches the rule base for a rule that can satisfy this goal. Rule r2_9 can satisfy the goal, if its condition is met. The condition part of rule r2_9 relies on the status of the release valve and on the flow rate. If the release valve is open and the flow rate is high, then r2_9 would be able to fire, the goal `steam is escaping` could be satisfied, rule r2_6 would be fireable, and the original goal thus fulfilled.

Let us suppose, on the other hand, that the release valve is found to be closed. Rule r2_9 therefore fails, with the result that rule r2_6 also fails. The system *backtracks* to the last place where a choice between possible rules was made and will try an alternative, as shown in Figure 2.8. In the example shown here, backtracking causes rule r2_4 to be examined next. This rule can fire only if the release valve is stuck but, because this information is not yet known, it becomes the new goal. This process continues until a goal is satisfied by the information in the fact base. When this happens, the original goal is fulfilled, and the chosen path through the rules is the solution. If all possible ways of achieving the overall goal have been explored and failed, then the overall goal fails.

The backward-chaining mechanism described so far has assumed a depth-first search for rules. This means that whenever a choice between rules exists, just one is selected, the others being examined only if backtracking occurs.

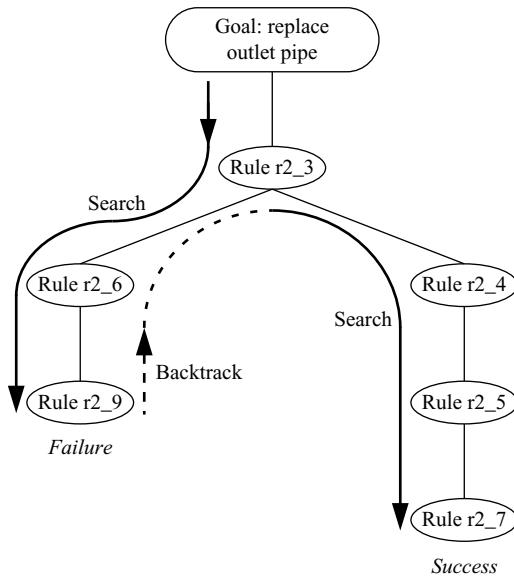


Figure 2.8 Backward chaining applied to the boiler-control rules: the search for rules proceeds in a depth-first manner.

The inference mechanism built into the Prolog language (see Chapter 11) is a depth-first backward chainer like the one described here. There are two sets of circumstances under which backtracking takes place:

1. When a goal cannot be satisfied by the set of rules currently under consideration.
2. When a goal has been satisfied and the user wants to investigate other ways of achieving the goal (i.e., to find other solutions).

2.9.2 Implementation of Backward Chaining

Figure 2.9 shows a generalized flowchart for backward chaining from a goal G_1 . In order to simplify the chart, it has been assumed that each rule has only one condition, so that the satisfaction of a condition can be represented as a single goal. In general, rules have more than one condition. The flowchart in Figure 2.9 is an attempt to represent backward chaining as an iterative process. This is difficult to achieve, as the length of the chain of rules cannot be predetermined. The flowchart has, of necessity, been left incomplete. It contains repeating sections that are identical except for the local variable names, an indication that while it is difficult to represent the process iteratively, it can be elegantly represented recursively. A recursive definition of a function is one that includes the function itself. Recursion is an important aspect of the AI languages Python, Lisp, and Prolog, discussed in

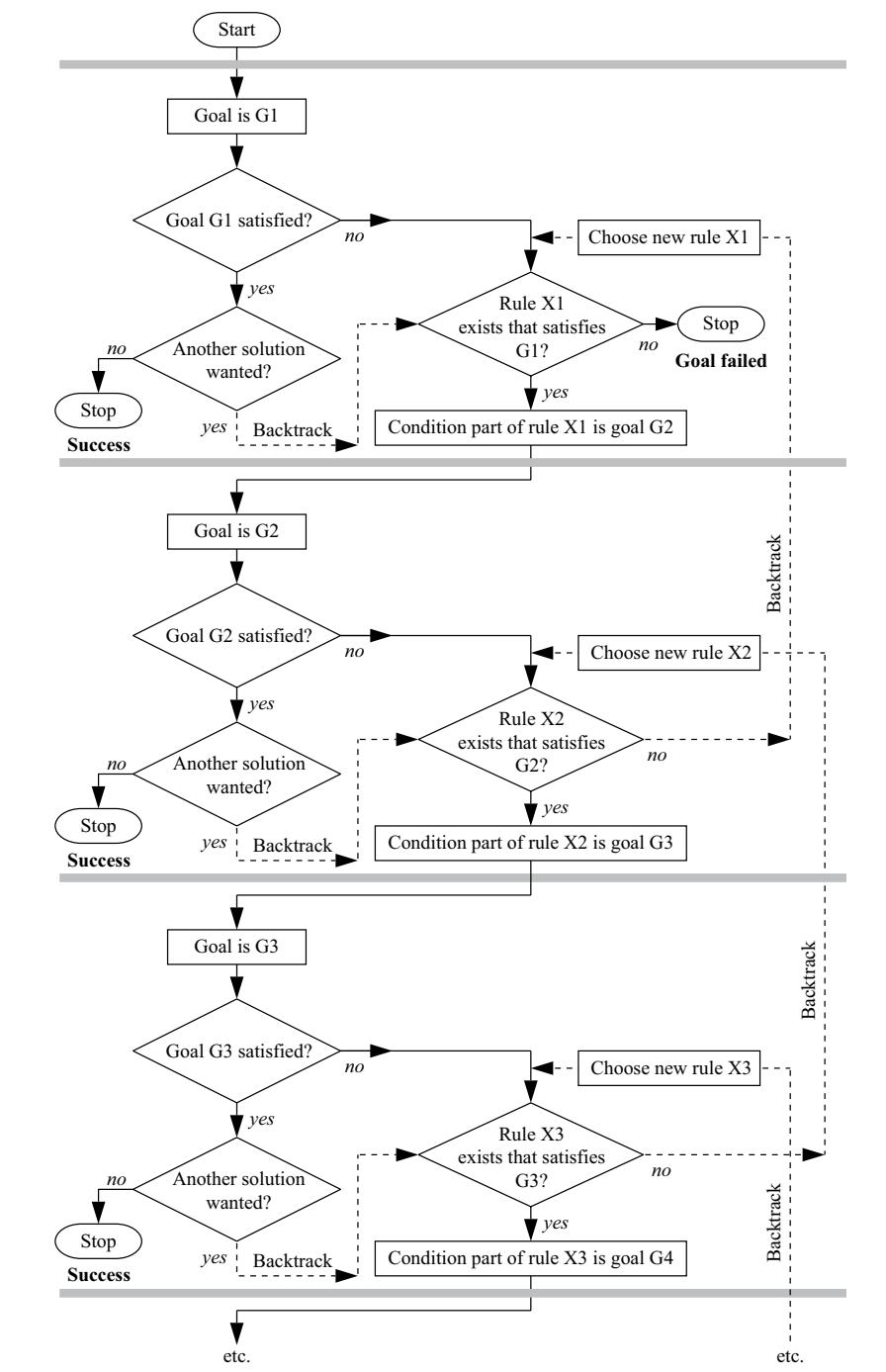


Figure 2.9 A flowchart for backward chaining.

Chapter 11, as well as many other computer languages. Box 2.1 shows a recursive definition of backward chaining, where it has again been assumed that rules have only one condition. It is not always necessary to write such a function for yourself as backward chaining forms an integral part of the Prolog language and many expert system shells and AI toolkits (see Chapter 11).

2.9.3 Variations of Backward Chaining

There are a number of possible variations to the idea of backward chaining. Some of these are as follows:

1. Depth-first or breadth-first search for rules.
2. Whether or not to pursue other solutions (i.e., other means of achieving the goal) once a solution has been found.

BOX 2.1 A RECURSIVE DEFINITION OF BACKWARD CHAINING, REPRESENTED IN PSEUDOCODE

```
define function backwardchain(G);
    /* returns a boolean (i.e., true/false) value */
    /* G is the goal being validated */
    result:= false;
    /* ':=' represents assignment of a value to a variable */
    S:= set of rules whose conclusion part matches goal G;
    if S is empty then
        result:= false;
    else
        while (result=false) and (S is not empty) do
            X:= rule selected from S;
            S:= S with X removed;
            C:= condition part of X;
            if C is true then
                result:=true
            elseif C is false then
                result:=false
            elseif (backwardchain(C)=true) then
                result:=true;
            /* note the recursive call of 'backwardchain' */
            /* C is the new goal */
            endif;
        endwhile;
    endif;
    return result;
    /* 'result' is the value returned by the
       function 'backwardchain' */
enddefine;
```

3. Different ways of choosing between branches to explore (depth-first search only).
4. Deciding upon the order in which to examine rules (breadth-first search only).
5. Having found a succession of enabling rules whose lowest-level conditions are satisfied, whether to fire the sequence of rules or simply to conclude that the goal is proven.

Breadth-first backward chaining is identical to depth-first, except for the mechanism for deciding between alternative rules to examine. In the example described in Section 2.9.1 (and in Figure 2.8), instead of choosing to explore either $r2_4$ or $r2_6$, both rules would be examined. Then the preconditions of each (i.e., rules $r2_5$ and $r2_9$) would be examined. If either branch fails, then the system will not need to backtrack, as it will simply carry on down those branches that still have the potential to fulfill the goal. The process is illustrated in Figure 2.10. The first solution to be found will be the one with the shortest (or joint-shortest) chain of rules. A disadvantage with the breadth-first approach is that large amounts of computer memory may be required, as the system needs to keep a record of progress along all branches of the search tree, rather than just one branch.

The last item (5) in the foregoing list of variations of backward chaining is important but subtle. So far, our discussion of backward chaining has made limited reference to rule firing. If a goal is capable of being satisfied by a succession of rules, the first of which is fireable, then the goal is assumed to be true and *no rules are actually fired*. If all that is required is the validation of a goal, then this approach is adequate. However,

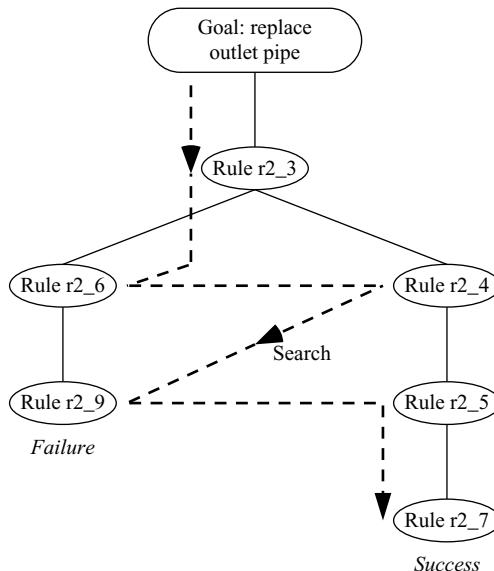


Figure 2.10 A variation of backward chaining applied to the boiler-control rules; here the search for rules proceeds in a breadth-first manner.

by actually firing the rules, all of the intermediate deductions such as `water_level becomes low` are recorded in the fact base and may be used by other rules, thereby eliminating the need to reestablish them. Therefore, in some implementations of backward chaining, once the path to a goal has been established, rules are fired until the original goal is fulfilled. In more complex rule-based systems, rules may call up and run a procedurally coded module as part of their conclusion. In such systems, the firing of the rules is essential for the role of the procedural code to be realized.

2.9.4 Format of Backward-Chaining Rules

In some backward-chaining systems, the rule syntax is reversed, compared with the examples that we have discussed so far. A possible syntax would be:

```
deduce <conclusion> if <condition>.
```

Four reasons why a separate format may be desirable for forward-chaining and backward-chaining rules are detailed below.

1. Placing the conclusion at the start of a backward-chaining rule emphasizes that the rule is goal-driven. The position of the conclusion before the condition reflects the fact that, in backward-chaining systems, it is the conclusion part of a rule that is assessed first. The condition is only examined if the conclusion is relevant. Conversely, placing the condition at the start of a forward-chaining rule emphasizes that the rule is data-driven.
2. A forward-chaining rule can have any number of conclusion statements, whereas most backward-chaining inference engines assume that a rule has only one conclusion.
3. Forward-chaining rules often contain an “action” command as part of the conclusion. As discussed in Section 2.9.2, backward chaining, on the other hand, is used to establish a logical chain of rules and facts (expressed as relations) in order to prove an assertion, that is, the goal. Establishing such a chain does not require any explicit rule firing, so any rules whose conclusions involve “actions” rather than deductions will need modification for use as backward-chaining rules.
4. KBS toolkits that are an extension of the Prolog language can make direct use of the Prolog backward-chaining mechanism by using a structure that resembles Prolog code. This is the case for the Flex KBS toolkit, where the backward-chaining rules are called *relations*.

The boiler-control ruleset considered earlier might be recast into backward-chaining relations in Flex as follows:

```
relation needs_opening(control_valve)
  if low(water_level) .
```

```

relation needs_shutting_down(boiler_tubes)
  if high(temperature)
    and low(water_level) .

relation needs_replacing(outlet_pipe)
  if blocked(steam_outlet) .

relation blocked(steam_outlet)
  if stuck(release_valve) .

relation stuck(release_valve)
  if high(pressure)
    and closed(release_valve) .

relation blocked(steam_outlet)
  if escaping(steam) .

relation high(pressure)
  if high(temperature)
    and not low(water_level) .

relation low(water_level)
  if low(transducer_output) .

relation escaping(steam)
  if open(release_valve)
    and high(flow_rate) .

relation closed(control_valve)
  if low(flow_rate) .

```

2.10 A Hybrid Strategy

A system called ARBS (Algorithmic and Rule-based Blackboard System) makes use of an inference engine that can be thought of as part forward chaining and part backward chaining (Hopgood 1994). ARBS is the forerunner of DARBS, or distributed ARBS, described in Chapters 12 and 15 (Nolle et al. 2002c; Tait et al. 2008). Conventional backward chaining involves initial examination of a rule that achieves the goal. If that rule cannot fire, its antecedent rules are recursively examined, until rules can be fired and progress made toward the goal. In all problems involving data interpretation (such as the boiler-control example), the high-level rules concerning the goal itself can never fire until lower-level rules for data manipulation have been fired. The standard mechanisms for forward or backward chaining, therefore,

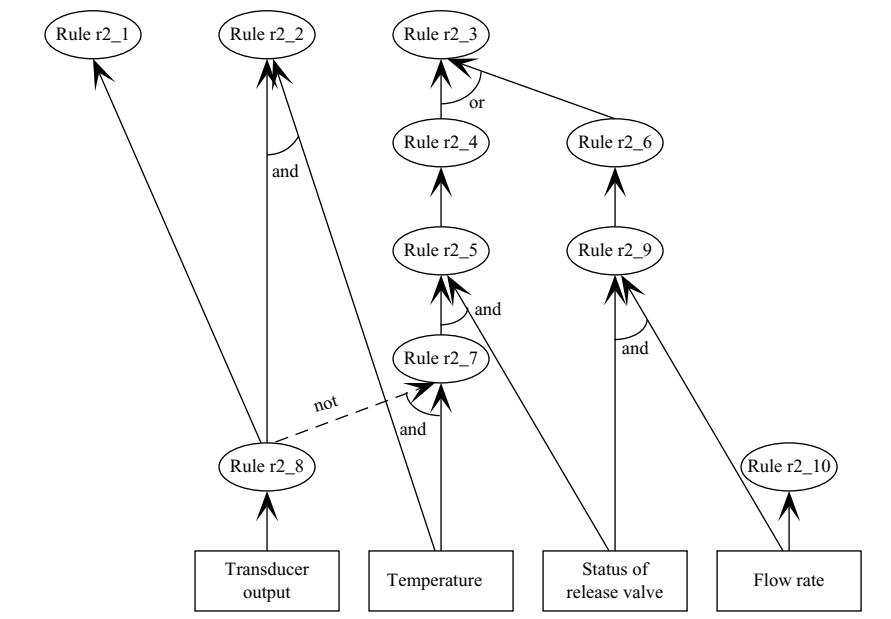


Figure 2.11 A rule dependence network.

involve a great deal of redundant rule examination. The hybrid strategy is a means of eliminating this source of inefficiency.

Under the hybrid strategy, a *rule dependence network* is built prior to running the system. For each rule, the network shows which other rules may enable it, that is, its antecedents, and which rules it may enable, that is, its dependents. The rule dependencies for the boiler-control knowledge base are shown in Figure 2.11. In its data-driven mode, known as *directed forward chaining*, the hybrid strategy achieves improved efficiency by using the dependence network to select rules for examination. Low-level rules concerning the sensor data are initially selected for examination. As shown in Figure 2.11, only rules r2_8, r2_9, and r2_10 need to be examined initially. Then higher-level rules, leading toward a solution, are selected depending on which rules have actually fired. So, if rules r2_8 and r2_9 fire successfully, the new set of rules to be examined becomes r2_1, r2_2, and r2_6. The technique is an effective way of carrying out the instruction marked “select rules to examine” in the flowchart for forward chaining (Figure 2.2).

The same mechanism can easily be adapted to provide an efficient goal-driven strategy. Given a particular goal, the control mechanism can select the branch of the dependence network leading to that goal and then backward-chain through the selected rules.

For a given rule base, the dependence network needs to be generated only once and is then available to the system at run-time. The ordering of rules in the rule base does not affect the system, because the application of rules is dictated by their position in the dependence network rather than in the ruleset.

Rule r2_7 has a “negative dependence” on rule r2_8, meaning that rule r2_7 is fireable if rule r2_8 fails to fire:

```
rule r2_7
  if temperature is high
  and water_level is not low
  then pressure becomes high.

rule r2_8
  if transducer_output is low
  then water_level becomes low.
```

As discussed in Section 2.5, the closed-world assumption will lead to `water_level is not low` being assumed true unless rule r2_8 is successfully fired. Therefore, for rule r2_7 to behave as intended, rule r2_8 must be examined (and either fire or fail) before rule r2_7 is examined. Using the dependence network to direct rule examination and firing is one way of ensuring this order of events.

The use of dependence networks is more complicated when local variables are used within rules because the dependencies between rules are less certain. Consider, for example, the following set of rules that do *not* use local variables:

```
rule r2_25
  if control_valve of pipe1 is open
  and status of pipe1 is blocked
  then release_valve becomes open.

rule r2_26
  if flow_rate of pipe1 is high
  then control_valve of pipe1 becomes open.

rule r2_27
  if pressure of pipe1 is high
  then status of pipe1 becomes blocked.
```

A dependence network would show that rule r2_25 is dependent only on rules r2_26 and r2_27. Therefore, if r2_26 and r2_27 have fired, r2_25 can definitely fire. Now consider the same rules modified to incorporate the use of the local variable `Pipe`, to which values such as `pipe1`, `pipe2`, etc., can be assigned:

```
rule r2_25a
  if control_valve of Pipe is open
  and status of Pipe is blocked
  then release_valve becomes open.
```

```

rule r2_26a
  if flow_rate of Pipe is high
    then control_valve of Pipe becomes open.

rule r2_27a
  if pressure of Pipe is high
    then status of Pipe becomes blocked.

```

Rule r2_25a is dependent on rules r2_26a and r2_27a. However, it is possible for rules r2_26a and r2_27a to have fired, but for rule r2_25a to fail. This is because the condition of rule r2_25a requires the looked-up values for control_valve and status to belong to a single value of the local variable Pipe, whereas rules r2_26a and r2_27a could each use a different value for Pipe. Thus, when rules contain local variables, a dependence network shows which rules have the *potential* to enable others to fire. Whether or not the dependent rules will actually fire cannot be determined until run-time. The dependence network shows us that rule r2_25a should be examined if rules r2_26a and r2_27a have fired, but otherwise it can be ignored.

A similar situation arises when there is a negative dependence between rules containing local variables, for example:

```

rule r2_25b
  if control_valve of Pipe is not open
  and status of Pipe is blocked
    then release_valve becomes open.

rule r2_26b
  if flow_rate of Pipe is high
    then control_valve of Pipe becomes open.

rule r2_27b
  if pressure of Pipe is high
    then status of Pipe becomes blocked.

```

Here r2_25b has a negative dependence on r2_26b and a normal (positive) dependence on r2_27b. Under these circumstances, r2_25b should be examined after both:

- (i) r2_27b has fired; *and*
- (ii) r2_26b has been examined, *whether or not it fired*.

The first subcondition of rule r2_25b, control_valve of Pipe is not open, will certainly be true if rule r2_26b fails, owing to the closed-world assumption. However, it may also be true even if rule r2_26b has fired, since Pipe could be instantiated to a different value in each rule. It is assumed that the scope of the local variable is the length of the rule, so the value of Pipe is the same throughout rule r2_25b, but may be different in rule r2_26b.

2.11 Explanation Facilities

One of the claims frequently made in support of expert systems is that they are able to explain their reasoning, and that this gives users of such systems confidence in the accuracy or wisdom of the system's decisions. As noted in Chapter 1, an expert system ought to be capable, at the very least, of producing a trace of the facts and rules that have been used. Some systems embellish the trace of rule firing with textual explanations that can serve to increase further the trust in the system.

Explanation facilities can be divided into two categories:

- *how* a conclusion has been derived;
- *why* a particular line of reasoning is being followed.

The first type of explanation would normally be applied when the system has completed its reasoning, whereas the second type is applicable while the system is carrying out its reasoning process. The latter type of explanation is particularly appropriate in an interactive expert system that involves a dialogue between a user and the computer. During such a dialogue, the user will often want to establish why particular questions are being asked. If either type of explanation is incorrect or impenetrable, the user is likely to distrust or ignore the system's findings.

Returning once more to our ruleset for boiler control (Section 2.2), the following would be a typical explanation for a recommendation to replace the outlet pipe:

```
** outlet pipe needs replacing!
because (rule r2_3) steam_outlet is blocked.

steam_outlet is blocked
because (rule r2_4) release_valve is stuck.

release_valve is stuck
because (rule r2_5) pressure is high and release_valve is
closed.

pressure is high
because (rule r2_7) temperature is high and water_level is
not low.

water_level is not low
because (rule r2_8) transducer_output is not low.

given facts:
  release_valve is closed
  and temperature is high
  and transducer_output is not low.
```

Explanation facilities are desirable for increasing user confidence in the system, as a teaching aid, and as an aid to debugging. However, a simple trace like the one shown is likely to be of little use except for debugging. The quality of explanation can be improved by placing an obligation on the rule-writer to provide an explanatory note for each rule. These notes can then be included in the rule trace or reproduced at run-time to explain the current line of reasoning. Explanation facilities can also be made more relevant by supplying the user with a level of detail tailored to his or her needs.

2.12 Summary

Rules are an effective way of representing knowledge in many application domains. They are most versatile when local variables are used within the rule. They can be particularly useful in cooperation with other processes such as procedural algorithms, or frame-based or object-oriented systems (Chapter 4). The role of interpreting, selecting, and applying rules is performed by the inference engine. Rule-writing should ideally be independent of the details of the inference engine, apart from fulfilling its syntax requirements. In practice, the rule-writer needs to be aware of the strategy for applying rules and any assumptions that are made by the inference engine. For instance, under the closed-world assumption, any facts that have not been supplied or derived are assumed to be false. Forward and backward chaining are two distinct strategies for applying rules, but many variations of these strategies are also possible.

Further Reading

- Akerkar, R. A., and P. S. Sajja. 2009. *Knowledge Based Systems*. Jones and Bartlett, Sudbury, MA.
- Brachman, R., and H. Levesque. 2011. *Knowledge Representation and Reasoning*. Elsevier, San Francisco, CA.
- Darlington, K. W. 2000. *The Essence of Expert Systems*. Prentice Hall, Upper Saddle River, NJ.
- Giarratano, J. C., and G. D. Riley. 2004. *Expert Systems: Principles and Programming*. 4th ed. Course Technology Inc., Boston, MA.
- Jannach, D., M. Zanker, A. Felfernig, and G. Friedrich. 2011. *Recommender Systems: An Introduction*. Cambridge University Press, Cambridge, UK.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 3

Handling Uncertainty: Probability and Fuzzy Logic

3.1 Sources of Uncertainty

The discussion of rule-based systems in Chapter 2 assumed that we live in a clear-cut world, where every hypothesis is either true, false, or unknown. Furthermore, it was pointed out that many systems make use of the *closed-world assumption*, whereby any hypothesis that is unknown is assumed to be false. We were then left with a binary system, where everything is either true or false. While this model of reality is useful in many applications, real reasoning processes are rarely so clear-cut. Referring to the example of the control of a power-station boiler, we made use of the following rule:

```
rule r2_8
  if transducer_output is low
  then water_level becomes low.
```

There are three distinct forms of uncertainty that might be associated with this rule:

- *Uncertainty in the rule itself*

A low level of water in the drum is not the only possible explanation for a low transducer output. Another possible cause could be that the float attached to the transducer is stuck. What we really mean by this rule is that, if the transducer output is low, then the water level is *probably* low.

■ *Uncertainty in the evidence*

The evidence upon which the rule is based may be uncertain. There are two possible reasons for this uncertainty. First, the evidence may come from a source that is not totally reliable. For instance, we may not be absolutely certain that the transducer output is low, as this information relies upon a meter to measure the voltage. Second, the evidence itself may have been derived by a rule whose conclusion was probable rather than certain.

■ *Use of vague language*

Rule r2_8 is based around the notion of a “low” transducer output. Assuming that the output is a voltage, we must consider whether “low” corresponds to 1 mV, 1 V, or 1 kV.

It is important to distinguish between these sources of uncertainty, as they need to be handled differently. There are some situations in nature that are truly random and whose outcome, while uncertain, can be anticipated on a statistical basis. For instance, we can anticipate that on average one of six throws of a die will result in a score of four. Some of the techniques that we will be discussing are based upon probability theory. These assume that a statistical approach can be adopted, although this assumption will be only an approximation to the real circumstances unless the problem is truly random.

This chapter will review some of the commonly used techniques for reasoning with uncertainty. Bayesian updating has a rigorous derivation based upon probability theory, but its underlying assumptions, for example, the statistical independence of multiple pieces of evidence, may not be true in practical situations. Certainty theory does not have a rigorous mathematical basis but has been devised as a practical way of overcoming some of the limitations of Bayesian updating. Possibility theory, or fuzzy logic, allows the third form of uncertainty, that is, vague language, to be used in a precise manner. The assumptions and arbitrariness of some of the techniques have meant that reasoning under uncertainty remains a controversial topic.

3.2 Bayesian Updating

3.2.1 Representing Uncertainty by Probability

Bayesian updating assumes that it is possible to ascribe a probability to every hypothesis or assertion, and that probabilities can be updated in the light of evidence for or against a hypothesis or assertion. This updating can either use Bayes’ theorem directly (Section 3.2.2), or it can be slightly simplified by the calculation of likelihood ratios (Section 3.2.3). One of the earliest successful applications of Bayesian updating to expert systems was PROSPECTOR, a system that assisted mineral prospecting by interpreting geological data (Duda et al. 1979; Hart et al. 1978).

Let us start our discussion by returning to our rule set for control of the power-station boiler (see Chapter 2), which included the following two rules:

```

rule r2_4
  if release_valve is stuck
  then steam_outlet becomes blocked.

rule r2_6
  if steam is escaping
  then steam_outlet becomes blocked.

```

We are going to consider the hypothesis that there is a steam outlet blockage. Previously, under the closed-world assumption, we asserted that in the absence of any evidence about a hypothesis, the hypothesis could be treated as false. The Bayesian approach is to ascribe an *a priori* probability (sometimes simply called the *prior* probability) to the hypothesis that the steam outlet is blocked. This is the probability that the steam outlet is blocked, in the absence of any evidence that it is blocked or that it is not blocked. Bayesian updating is a technique for updating this probability in the light of evidence for or against the hypothesis. So, whereas we had previously assumed that `steam is escaping` led to the derived fact `steam_outlet is blocked` with absolute certainty, now we can only say that it supports that deduction. Bayesian updating is cumulative, so that if the probability of a hypothesis has been updated in the light of one piece of evidence, the new probability can then be updated further by a second piece of evidence.

3.2.2 Direct Application of Bayes' Theorem

Suppose that the prior probability that `steam_outlet is blocked` is 0.01, which implies that blockages occur only rarely. Informally, our modified version of rule `r2_6` might look like this:

```

rule blockage1 /* not Flex format */
  if steam is escaping
  then update probability of [steam_outlet is blocked] .

```

With this new rule, the observation of steam escaping requires us to update the probability of a steam outlet blockage. As noted in the previous subsection, this conclusion contrasts with rule `r2_6`, where the conclusion that there is a steam outlet blockage would be drawn with absolute certainty. In this example, `steam_outlet is blocked` is considered to be a hypothesis (or assertion), and `steam is escaping` is its supporting evidence.

The technique of Bayesian updating provides a mechanism for updating the probability of a hypothesis $P(H)$ in the presence of evidence E . Often the evidence is a symptom, and the hypothesis is a diagnosis. The technique is based upon the application of Bayes' theorem (sometimes called Bayes' rule). Bayes' theorem provides an expression for the conditional probability $P(H|E)$ of a hypothesis H given some evidence E , in terms of $P(E|H)$, that is, the conditional probability of E given H . It is expressed as follows:

$$P(H|E) = \frac{P(H) \times P(E|H)}{P(E)} \quad (3.1)$$

The theorem is easily proved by looking at the definition of dependent probabilities. Of an expected population of events in which E is observed, $P(H|E)$ is the fraction in which H is also observed. Thus,

$$P(H|E) = \frac{P(H \& E)}{P(E)} \quad (3.2)$$

Similarly,

$$P(E|H) = \frac{P(H \& E)}{P(H)} \quad (3.3)$$

The combination of Equations 3.2 and 3.3 yields Equation 3.1. Bayes' theorem can be expanded as follows by noting that $P(E)$ is equal to $P(E \& H) + P(E \& \sim H)$:

$$P(H|E) = \frac{P(H) \times P(E|H)}{P(H) \times P(E|H) + P(\sim H) \times P(E|\sim H)} \quad (3.4)$$

where $\sim H$ means “not H.” The probability of $\sim H$ is simply given by

$$P(\sim H) = 1 - P(H) \quad (3.5)$$

Equation 3.4 provides a mechanism for updating the probability of a hypothesis H in the light of new evidence E. This is done by updating the existing value of $P(H)$ to the value for $P(H|E)$ yielded by Equation 3.4. The application of the equation requires knowledge of the following values:

- $P(H)$, the current probability of the hypothesis. If this is the first update for this hypothesis, then $P(H)$ is the prior probability.
- $P(E|H)$, the conditional probability that the evidence is present, given that the hypothesis is true.
- $P(E|\sim H)$, the conditional probability that the evidence is present, given that the hypothesis is false.

Thus, to build a system that makes direct use of Bayes' theorem in this way, values are needed in advance for $P(H)$, $P(E|H)$, and $P(E|\sim H)$ for all the different hypotheses and evidence covered by the rules. Obtaining these values might appear at first glance more formidable than the expression we are hoping to derive, namely, $P(H|E)$. However, in the case of diagnosis problems, the conditional probability of

evidence, given a hypothesis, is usually more readily available than the conditional probability of a hypothesis, given the evidence. Even if $P(E|H)$ and $P(E|\sim H)$ are not available as formal statistical observations, they may at least be available as informal estimates. So, in our example, an expert may have some idea of how often steam is observed escaping when there is an outlet blockage, but he or she is less likely to know how often a steam escape is due to an outlet blockage. Chapter 1 introduced the ideas of deduction, abduction, and induction. Bayes' theorem, in effect, performs abduction (i.e., determining causes) using deductive information (i.e., the likelihood of symptoms, effects, or evidence). The premise that deductive information is more readily available than abductive information is one of the justifications for using Bayesian updating.

3.2.3 Likelihood Ratios

Likelihood ratios, defined in the following text, provide an alternative means of representing Bayesian updating. They lead to rules of this general form:

```
rule blockage2 /* not Flex format */
  if steam is escaping
    then [steam_outlet is blocked] becomes x times more likely.
```

With a rule like this, if steam is escaping, we can update the probability of a steam outlet blockage provided we have an expression for x . A value for x can be expressed most easily if the likelihood of the hypothesis `steam_outlet is blocked` is expressed as odds rather than a probability. The odds $O(H)$ of a given hypothesis H are related to its probability $P(H)$ by the relations:

$$O(H) = \frac{P(H)}{P(\sim H)} = \frac{P(H)}{1 - P(H)} \quad (3.6)$$

and

$$P(H) = \frac{O(H)}{O(H) + 1} \quad (3.7)$$

As before, $\sim H$ means “not H .” Thus, a hypothesis with a probability of 0.2 has odds of 0.25 (or “4 to 1 against”). Similarly, a hypothesis with a probability of 0.8 has odds of 4 (or “4 to 1 on”). An assertion that is absolutely certain, that is, has a probability of 1, has infinite odds. In practice, limits are often set on odds values so that, for example, if $O(H) > 10^6$, then H is true, and if $O(H) < 10^{-6}$, then H is false. Such limits are arbitrary.

In order to derive the updating equations, start by considering the hypothesis “not H ,” or $\sim H$, in Equation 3.1:

$$P(\sim H|E) = \frac{P(\sim H) \times P(E|\sim H)}{P(E)} \quad (3.8)$$

Division of Equation 3.1 by Equation 3.8 yields

$$\frac{P(H|E)}{P(\sim H|E)} = \frac{P(H) \times P(E|H)}{P(\sim H) \times P(E|\sim H)} \quad (3.9)$$

By definition, $O(H|E)$, the conditional odds of H given E , is

$$O(H|E) = \frac{P(H|E)}{P(\sim H|E)} \quad (3.10)$$

Substituting Equations 3.6 and 3.10 into Equation 3.9 yields

$$O(H|E) = A \times O(H) \quad (3.11)$$

where

$$A = \frac{P(E|H)}{P(E|\sim H)} \quad (3.12)$$

$O(H|E)$ is the updated odds of H , given the presence of evidence E , and A is the *affirms* weight of evidence E . It is one of two likelihood ratios. The other is the *denies* weight D of evidence E . The *denies* weight can be obtained by considering the absence of evidence, that is, $\sim E$:

$$O(H|\sim E) = D \times O(H) \quad (3.13)$$

where

$$D = \frac{P(\sim E|H)}{P(\sim E|\sim H)} = \frac{1 - P(E|H)}{1 - P(E|\sim H)} \quad (3.14)$$

The function represented by Equations 3.11 and 3.13 is shown in Figure 3.1. Rather than displaying odds values, which have an infinite range, the corresponding probabilities have been shown. The weight (A or D) has been shown on a logarithmic scale over the range 0.01 to 100.

3.2.4 Using the Likelihood Ratios

Equation 3.11 provides a simple way of updating our confidence in hypothesis H in the light of new evidence E , assuming that we have a value for A and for $O(H)$, that

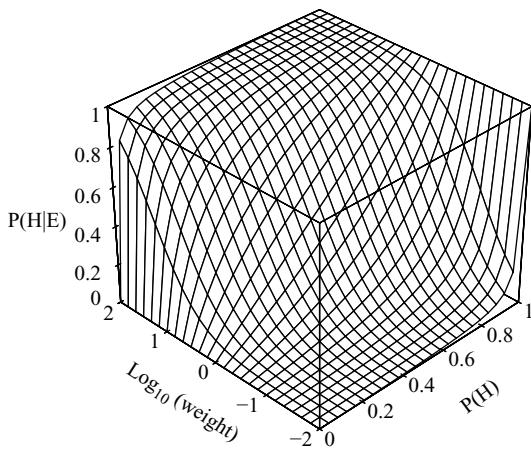


Figure 3.1 The Bayesian updating function.

is, the current odds of H. $O(H)$ will be at its *a priori* value if it has not previously been updated by other pieces of evidence. In the case of rule r2_6, H refers to the hypothesis `steam_outlet is blocked`, and E refers to the evidence `steam is escaping`.

In many cases, the absence of a piece of supporting evidence may reduce the likelihood of a certain hypothesis. In other words, the absence of supporting evidence is equivalent to the presence of opposing evidence. The known absence of evidence is distinct from not knowing whether the evidence is present, and can be used to reduce the probability (or odds) of the hypothesis by applying Equation 3.13 using the *denies* weight, D .

If a given piece of evidence E has an *affirms* weight A that is greater than 1, then its *denies* weight must be less than 1 and vice versa:

$$\begin{aligned} A > 1 &\text{ implies } D < 1, \\ A < 1 &\text{ implies } D > 1. \end{aligned}$$

If $A < 1$ and $D > 1$, then the absence of evidence is supportive of a hypothesis. Rule r2_7 provides an example of this, where `water_level is not low` supports the hypothesis `pressure is high` and, implicitly, `water_level is low` opposes the hypothesis. The rule is:

```
rule r2_7
  if temperature is high
  and water_level is not low
  then pressure becomes high.
```

The same hypothesis is also supported by the evidence `temperature is high` and, implicitly, opposed by the evidence `temperature is not high`.

Using the Flint™ toolkit extension to Flex™ (Beaudoin et al. 2005; Saini et al. 2016), a Bayesian version of this rule might be as follows, where the relative values of the *affirms* and *denies* weights determine which evidence supports the hypothesis and which evidence opposes it:

```
uncertainty_rule r2_7b
if temperature is high (affirms 18.0; denies 0.11)
and water_level is low (affirms 0.10; denies 1.90)
then pressure becomes high.
```

As with the direct application of Bayes' rule, likelihood ratios have the advantage that the definitions of A and D are couched in terms of the conditional probability of evidence, given a hypothesis, rather than the reverse. As pointed out earlier, it is usually assumed that this information is more readily available than the conditional probability of a hypothesis, given the evidence, at least in an informal way. Even if accurate conditional probabilities are unavailable, Bayesian updating using likelihood ratios is still a useful technique if heuristic values can be attached to A and D .

3.2.5 Dealing with Uncertain Evidence

So far, we have assumed that evidence is either definitely present (i.e., has a probability of 1) or definitely absent (i.e., has a probability of 0). If the probability of the evidence lies between these extremes, then the confidence in the conclusion must be scaled appropriately. There are two reasons why the evidence may be uncertain:

- The evidence could be an assertion generated by another uncertain rule, and which therefore has a probability associated with it.
- The evidence may be in the form of data that are not totally reliable, such as the output from a sensor.

In terms of probabilities, we wish to calculate $P(H|E)$, where E is uncertain. We can handle this problem by assuming that E was asserted by another rule whose evidence was B , where B is certain (has probability 1). Given the evidence B , the probability of E is $P(E|B)$. Our problem then becomes one of calculating $P(H|B)$. An expression for this has been derived by Duda et al. (1976):

$$P(H|B) = P(H|E) \times P(E|B) + P(H| \sim E) \times [1 - P(E|B)] \quad (3.15)$$

This expression can be useful if Bayes' theorem is being used directly (Section 3.2.2), but an alternative is needed when using likelihood ratios. One technique is to modify the *affirms* and *denies* weights to reflect the uncertainty in E . One means of achieving this is to interpolate the weights linearly as the probability of E varies between 1 and 0. Figure 3.2 illustrates this scaling process, where the interpolated

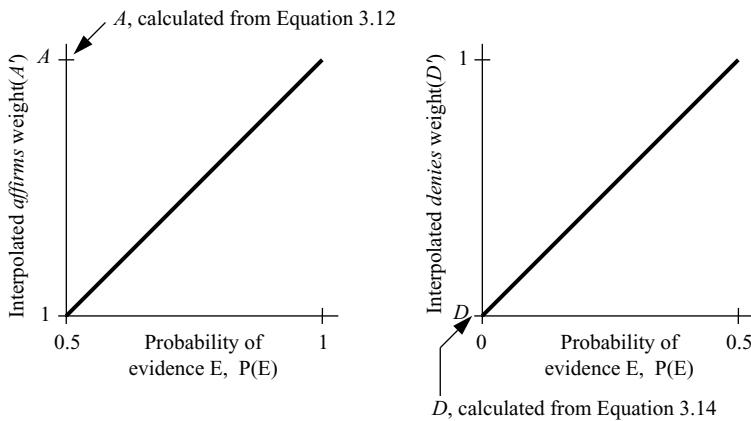


Figure 3.2 Linear interpolation of *affirms* and *denies* weights when the evidence is uncertain.

affirms and *denies* weights are given the symbols A' and D' , respectively. While $P(E)$ is greater than 0.5, the *affirms* weight is used, and when $P(E)$ is less than 0.5, the *denies* weight is used. Over the range of values for $P(E)$, A' and D' vary between 1 (neutral weighting) and A and D , respectively. The interpolation process achieves the right sort of result, but it has no rigorous basis. The expressions used to calculate the interpolated values are:

$$A' = [2(A - 1) \times P(E)] + 2 - A \quad (3.16)$$

$$D' = [2(1 - D) \times P(E)] + D. \quad (3.17)$$

3.2.6 Combining Evidence

Much of the controversy concerning the use of Bayesian updating is centered on the issue of how to combine several pieces of evidence that support the same hypothesis. If n pieces of evidence are found that support a hypothesis H , then the formal restatement of the updating equation is straightforward:

$$O(H|E_1 \& E_2 \& E_3 \dots E_n) = A \times O(H) \quad (3.18)$$

where

$$A = \frac{P(E_1 \& E_2 \& E_3 \dots E_n | H)}{P(E_1 \& E_2 \& E_3 \dots E_n | \sim H)} \quad (3.19)$$

However, the usefulness of this pair of equations is doubtful since we do not know in advance which pieces of evidence will be available to support the hypothesis H. We would have to write expressions for A covering all possible pieces of evidence E_i , as well as all combinations of the pairs $E_i \& E_j$, of the triples $E_i \& E_j \& E_k$, of quadruples $E_i \& E_j \& E_k \& E_m$, and so on. As this is clearly an unrealistic requirement, especially where the number of possible pieces of evidence (or symptoms in a diagnosis problem) is large, a simplification is normally sought. The problem becomes much more manageable if it is assumed that all pieces of evidence are *statistically independent*. It is this assumption that is one of the most controversial aspects of the use of Bayesian updating in knowledge-based systems since the assumption is rarely accurate. Statistical independence of two pieces of evidence (E_1 and E_2) means that the probability of observing E_1 given that E_2 has been observed is identical to the probability of observing E_1 given no information about E_2 . Stating this more formally, the statistical independence of E_1 and E_2 is defined as:

$$P(E_1|E_2) = P(E_1) \text{ and } P(E_2|E_1) = P(E_2) \quad (3.20)$$

If the independence assumption is made, then the rule-writer need only worry about supplying weightings of the form:

$$A_i = \frac{P(E_i|H)}{P(\sim E_i|H)} \quad (3.21)$$

and

$$D_i = \frac{P(\sim E_i|H)}{P(\sim E_i|\sim H)} \quad (3.22)$$

for each piece of evidence E_i that has the potential to update H. If, in a given run of the system, n pieces of evidence are found that support or oppose H, then the updating equations are simply:

$$O(H|E_1 \& E_2 \& E_3 \dots \& E_n) = A_1 \times A_2 \times A_3 \times \dots \times A_n \times O(H) \quad (3.23)$$

and

$$O(H|\sim E_1 \& \sim E_2 \& \sim E_3 \dots \& \sim E_n) = D_1 \times D_2 \times D_3 \times \dots \times D_n \times O(H) \quad (3.24)$$

Problems arising from the interdependence of pieces of evidence can be avoided if the rule base is properly structured. Where pieces of evidence are known to be dependent on each other, they should not be combined in a single rule. Instead, assertions—and the rules that generate them—should be arranged in a hierarchy from low-level input data to high-level conclusions, with many levels of hypotheses

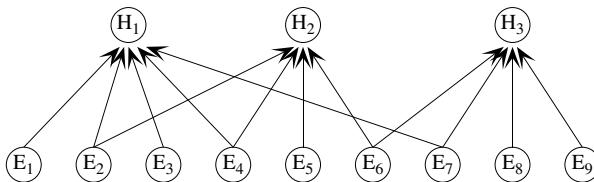


Figure 3.3 A shallow Bayesian inference network (E_i = evidence, H_i = hypothesis).

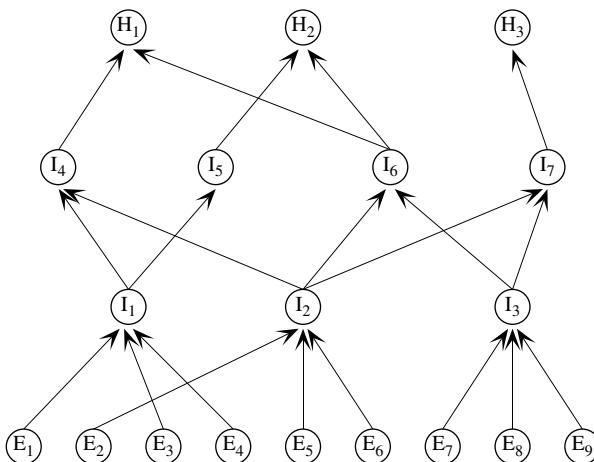


Figure 3.4 A deeper Bayesian inference network (E_i = evidence, H_i = hypothesis, I_i = intermediate hypothesis).

between. This structure does not limit the amount of evidence that is considered in reaching a conclusion, but it controls the interactions between the pieces of evidence. Inference networks are a convenient means of representing the levels of assertions from input data, through intermediate deductions, to final conclusions. Figures 3.3 and 3.4 show two possible inference networks. Each node represents either a hypothesis or a piece of evidence and has an associated probability (not shown). In Figure 3.3, the rule-writer has attempted to draw all the evidence that is relevant to particular conclusions together in a single rule for each conclusion. This produces a shallow network, with no intermediate levels between input data and conclusions. Such a system would only be reliable if there was little or no dependence between the input data.

In contrast, the inference network in Figure 3.4 includes several intermediate steps. The probabilities at each node are modified as the reasoning process proceeds, until they reach their final values. Note that the rules in the boiler-control example made use of several intermediate nodes that helped to make the rules more understandable and avoided duplication of tests for specific pieces of evidence.

In order to maintain some consistency of rule format, the condition parts of rule r3_1 given earlier have been joined by a conjunction (*and*). However, the formal treatment of Bayesian updating allows no distinction between conjoining (*and*) and disjoining (*or*) evidence. Instead, each item of evidence is assumed to contribute independently toward updating the probability of the hypothesis (i.e., the conclusion).

3.2.7 Combining Bayesian Rules with Production Rules

In a practical rule-based system, we may wish to mix uncertain rules with production rules. For instance, we may wish to make use of the following production rule, even though the assertion `release_valve is stuck` may have been established with a probability less than 1.

```
rule r3_1a
  if release_valve is stuck
    then task becomes clean_release_valve.
```

In this case, the hypothesis that the release valve needs cleaning can be asserted with the same probability as the evidence. This step avoids the issue of providing a prior probability for the hypothesis or a weighting for the evidence.

If a production rule contains multiple pieces of evidence that are independent from each other, their combined probability can be derived from standard probability theory. Consider, for example, a rule in which two pieces of independent evidence are conjoined (i.e., they are joined by *and*):

```
if <evidence E1> and <evidence E2> then <hypothesis H3>.
```

The probability of hypothesis H₃ is given by:

$$P(H_3) = P(E_1) \times P(E_2) \quad (3.25)$$

Production rules containing independent evidence that is disjoined (i.e., joined by *or*) can be treated in a similar way. So, given the rule

```
if <evidence E1> or <evidence E2> then <hypothesis H3>.
```

then the probability of hypothesis H₃ is given by:

$$P(H_3) = P(E_1) + P(E_2) - (P(E_1) \times P(E_2)) \quad (3.26)$$

The Flint™ toolkit (Beaudoin et al. 2005; Saini et al. 2016) takes a different approach in which the production rules are treated as though they were Bayesian. A default prior probability of 0.5 is assigned to the hypothesis, and the weighted *affirms* and *denies* weights, A' and D', are assumed to equal the odds of the evidence,

$O(E)$. This approach is equivalent to the method described previously if there is only one item of evidence, but it deviates from that method for multiple items of evidence.

3.2.8 A Worked Example of Bayesian Updating

We will consider the same example that was introduced in Chapter 2, namely, control of a power-station boiler. Let us start with just four rules in Flex format:

```
rule r3_1a
  if release_valve is stuck
  then task becomes clean_release_valve.

rule r3_2a
  if warning_light is on
  then release_valve becomes stuck.

rule r3_3a
  if pressure is high
  then release_valve becomes stuck.

rule r3_4a
  if temperature is high
  and water_level is not low
  then pressure becomes high.
```

The conclusion of each of these rules is expressed as an assertion. The four rules contain four assertions (or hypotheses) and three pieces of evidence that are independent of the rules, namely, the temperature, the status of the warning light (on or off), and the water level. The various probability estimates for these and their associated *affirms* and *denies* weights are shown in Table 3.1.

Table 3.1 Values Used in the Worked Example of Bayesian Updating

H	E	$P(H)$	$O(H)$	$P(E H)$	$P(E \sim H)$	A	D
release valve needs cleaning	release valve is stuck	—	—	—	—	—	—
release valve is stuck	warning light is on	0.02	0.02	0.88	0.4	2.20	0.20
release valve is stuck	pressure is high	0.02	0.02	0.85	0.01	85.0	0.15
pressure is high	temperature is high	0.1	0.11	0.90	0.05	18.0	0.11
pressure is high	water level is low	0.1	0.11	0.05	0.5	0.10	1.90

Having calculated the *affirms* and *denies* weights, we can now rewrite our production rules as probabilistic rules. We will leave the first rule as a production rule, albeit in the altered format of Flint, in order to illustrate the interaction between production rules and probabilistic rules. Our new rule set is therefore as follows:

```
uncertainty_rule r3_1b
  if release_valve is stuck
    then task becomes clean_release_valve.

uncertainty_rule r3_2b
  if warning_light is on (affirms 2.20; denies 0.20)
    then release_valve becomes stuck.

uncertainty_rule r3_3b
  if pressure is high (affirms 85.0; denies 0.15)
    then release_valve becomes stuck.

uncertainty_rule r3_4b
  if temperature is high (affirms 18.0; denies 0.11)
  and water_level is low (affirms 0.10; denies 1.90)
    then pressure becomes high.
```

Rule r3_4b makes use of two pieces of evidence. It no longer needs a negative condition, as the *affirms* and *denies* weights cover conditions that range from true to false. The requirement that *water_level* is not low be supportive evidence is expressed by the *denies* weight of *water_level* is low being greater than 1 while the *affirms* weight is less than 1.

To illustrate how the various weights are used, let us consider how a Bayesian inference engine would use the following set of input data:

```
water_level is not low.
warning_light is on.
temperature is high.
```

We will assume that the rules fire in the following order:

$$r3_4b \rightarrow r3_3b \rightarrow r3_2b \rightarrow r3_1b$$

The resultant rule trace might then appear as follows:

```
uncertainty_rule r3_4b
  H = pressure is high; O(H) = 0.11
  E1 = temperature is high; A1 = 18.0
  E2 = water_level is low; D2 = 1.90
  O(H | (E1&~E2)) = O(H) × A1 × D2 = 3.76
  /* Updated odds of "pressure is high" are 3.76 */
```

```

uncertainty_rule r3_3b
H = release_valve is stuck; O(H) = 0.02
E = pressure is high; A = 85.0
/* Because E is not certain (O(E) = 3.76, P(E) = 0.79),
the inference engine must calculate an interpolated
value A' for the affirms weight of E (see Section 3.2.5). */
A' = [2(A-1) × P(E)] + 2 - A = 49.7
O(H|(E)) = O(H) × A' = 0.99
/* Updated odds of "release_valve is stuck" are 0.99,
corresponding to a probability of approximately 0.5 */

uncertainty_rule r3_2b
H = release_valve is stuck; O(H) = 0.99
E = warning_light is on; A = 2.20
O(H|(E)) = O(H) × A = 2.18
/* Updated odds of "release_valve is stuck" are 2.18 */

uncertainty_rule r3_1b
H = task is clean_release_valve
E = release_valve is stuck;
O(E) = 2.18 implies O(H)= 2.18
/* This is a production rule, so the conclusion is
asserted with the same probability as the evidence. */
/* Updated odds of "task is clean_release_valve" are 2.18
*/

```

3.2.9 Discussion of the Worked Example

The foregoing example serves to illustrate a number of features of Bayesian updating. Our final conclusion that the release valve needs cleaning is reached with a certainty represented as:

$O(\text{task is clean_release_valve}) = 2.18$

or

$P(\text{task is clean_release_valve}) = 0.69$

Thus, there is a probability of 0.69 that the valve needs cleaning. In a real-world situation, this is a more realistic outcome than concluding that the valve definitely needs cleaning, which would have been the conclusion had we used the original set of production rules.

The initial three items of evidence were all stated with complete certainty: water level is not low; warning_light is on; and temperature is high. These are given facts and $P(E) = 1$ for each of them. Consider the evidence warning_light is on. A probability of less than 1 might be associated with this evidence if it were generated as an assertion by another probabilistic rule, or if it

were supplied as an input to the system, but the user's view of the warning light was impaired. If $P(\text{warning light is on})$ is 0.8, an interpolated value of the *affirms* weight would be used in rule r3_2b. Equation 3.16 yields an interpolated value of 1.72 for the *affirms* weight.

However, if $P(\text{warning light is on})$ were less than 0.5, then an interpolated *denies* weighting would be used. If $P(\text{warning light is on})$ were 0.3, an interpolated *denies* weighting of 0.68 is yielded by Equation 3.17.

If $P(\text{warning light is on}) = 0.5$, then the warning light is just as likely to be on as it is to be off. If we try to interpolate either the *affirms* or *denies* weight, a value of 1 will be found. Thus, if each item of evidence for a particular rule has a probability of 0.5, then the rule has no effect whatsoever.

Assuming that the prior probability of a hypothesis is less than 1 and greater than 0, the hypothesis can never be confirmed with complete certainty by the application of likelihood ratios as this would require its odds to become infinite.

While Bayesian updating is a mathematically rigorous technique for updating probabilities, it is important to remember that the results obtained can only be valid if the data supplied are valid. This is the key issue to consider when assessing the virtues of the technique. The probabilities shown in Table 3.1 have not been measured from a series of trials, but instead they are an expert's best guesses. Given that the values upon which the *affirms* and *denies* weights are based are only guesses, then a reasonable alternative to calculating them is to simply take an educated guess at the appropriate weightings. Such an approach is just as valid or invalid as calculating values from unreliable data. If a rule-writer takes such an *ad-hoc* approach, the provision of both the *affirms* and *denies* weightings becomes optional. If an *affirms* weight is provided for a piece of evidence E, but not a *denies* weight, then that rule can be ignored when $P(E) < 0.5$.

As well as relying on the rule-writer's weightings, Bayesian updating is also critically dependent on the values of the prior probabilities. Obtaining accurate estimates for these is also problematic.

Even if we assume that all of the data supplied in the foregoing worked example are accurate, the validity of the final conclusion relies upon the statistical independence from each other of the supporting pieces of evidence. In our example, as with very many real problems, this assumption is dubious. For example, pressure is high and warning_light is on were used as independent pieces of evidence, when in reality there is a cause-and-effect relationship between the two.

3.2.10 Advantages and Disadvantages of Bayesian Updating

Bayesian updating is a means of handling uncertainty by updating the probability of an assertion when evidence for or against the assertion is provided. The principal *advantages* of Bayesian updating are:

1. The technique is based upon a proven statistical theorem.

2. Likelihood is expressed as a probability (or odds), which has a clearly defined and familiar meaning.
3. The technique requires deductive probabilities, which are generally easier to estimate than abductive ones. The user supplies values for the probability of evidence (the symptoms) given a hypothesis (the cause), rather than the reverse.
4. Likelihood ratios and prior probabilities can be replaced by sensible guesses. This simplification is at the expense of advantage (1), as the probabilities subsequently calculated cannot be interpreted literally but rather as an imprecise measure of likelihood.
5. Evidence for and against a hypothesis (or the presence and absence of evidence) can be combined in a single rule by using *affirms* and *denies* weights.
6. Linear interpolation of the likelihood ratios can be used to take account of any uncertainty in the evidence (i.e., uncertainty about whether the condition part of the rule is satisfied), although this is an ad hoc solution.
7. The probability of a hypothesis can be updated in response to more than one piece of evidence.

The principal disadvantages of Bayesian updating are:

1. The prior probability of an assertion must be known or guessed at.
2. Conditional probabilities must be measured or estimated or, failing those, guesses must be taken at suitable likelihood ratios. Although the conditional probabilities are often easier to judge than the prior probability, they are nevertheless a considerable source of errors. Estimates of likelihood are often clouded by a subjective view of the importance or utility of a piece of information (Buchanan and Duda 1983).
3. The single probability value for the truth of an assertion tells us nothing about its precision.
4. Because evidence for and against an assertion are lumped together, no record is kept of how much there is of each.
5. The addition of a new rule that asserts a new hypothesis often requires alterations to the prior probabilities and weightings of several other rules. Such a requirement contravenes one of the main advantages of knowledge-based systems.
6. The assumption that pieces of evidence are independent is often unfounded. The only alternatives are to calculate *affirms* and *denies* weights for all possible combinations of dependent evidence, or to restructure the rule base so as to minimize these interactions.
7. The linear interpolation technique for dealing with uncertain evidence is not mathematically justified.

8. Representations based on odds, as required to make use of likelihood ratios, cannot handle absolute truth, that is, odds = ∞ .

3.3 Certainty Theory

3.3.1 Introduction

Certainty theory (Shortliffe and Buchanan 1975), sometimes called Stanford certainty theory, is an adaptation of Bayesian updating that is incorporated into the EMYCIN expert system shell. EMYCIN is based on MYCIN (Shortliffe 1976), an expert system that assists in the diagnosis of infectious diseases. The name EMYCIN is derived from “essential MYCIN,” reflecting the fact that it is not specific to medical diagnosis and that its handling of uncertainty is simplified. Certainty theory represents an attempt to overcome some of the shortcomings of Bayesian updating, although the mathematical rigor of Bayesian updating is lost. As this rigor is rarely justified by the quality of the data, the loss of rigor may not be a significant problem.

3.3.2 Making Uncertain Hypotheses

Instead of using probabilities, each assertion in EMYCIN has a certainty value associated with it. Certainty values can range between 1 and -1.

For a given hypothesis H, its certainty value C(H) is given by:

- C(H) = 1.0 if H is known to be true;
- C(H) = 0.0 if H is unknown;
- C(H) = -1.0 if H is known to be false.

There is a similarity between certainty values and probabilities, such that:

- C(H) = 1.0 corresponds to $P(H) = 1.0$;
- C(H) = 0.0 corresponds to $P(H)$ being at its *a priori* value;
- C(H) = -1.0 corresponds to $P(H) = 0.0$.

Each rule also has a certainty associated with it, known as its certainty factor, CF. Certainty factors serve a similar role to the *affirms* and *denies* weightings in Bayesian systems:

```
uncertainty_rule generic1
  if <evidence>
  then <hypothesis>
  with certainty factor <CF>.
```

Part of the simplicity of certainty theory stems from the fact that identical measures of certainty are attached to rules and hypotheses. The certainty factor of a rule is modified to reflect the level of certainty of the evidence, such that the modified certainty factor CF' is given by:

$$CF' = CF \times C(E) \quad (3.27)$$

If the evidence is known to be present, that is, $C(E) = 1$, then Equation 3.27 yields $CF' = CF$.

The technique for updating the certainty of hypothesis H , in the light of evidence E , involves the application of the following composite function:

$$\text{if } C(H) \geq 0 \text{ and } CF' \geq 0 \text{ then } C(H|E) = C(H) + [CF' \times (1 - C(H))] \quad (3.28)$$

$$\text{if } C(H) \leq 0 \text{ and } CF' \leq 0 \text{ then } C(H|E) = C(H) + [CF' \times (1 + C(H))] \quad (3.29)$$

$$\text{if } C(H) \text{ and } CF' \text{ have opposite signs then } C(H|E) = \frac{C(H) + CF'}{1 - \min(|C(H)|, |CF'|)} \quad (3.30)$$

where

$C(H|E)$ is the certainty of H updated in the light of evidence E ;

$C(H)$ is the initial certainty of H , that is, 0 unless it has been updated by the previous application of a rule;

$|x|$ is the magnitude of x , that is, ignoring its sign.

It can be seen from the foregoing equations that the updating procedure consists of *adding* a positive or negative value to the current certainty of a hypothesis. This contrasts with Bayesian updating, where the odds of a hypothesis are *multiplied* by the appropriate likelihood ratio. The composite function represented by Equations 3.28 to 3.30 is plotted in Figure 3.5, and it can be seen to have a broadly similar shape to the Bayesian updating equation (plotted in Figure 3.1).

In the standard version of certainty theory, a rule can only be applied if the certainty of the evidence $C(E)$ is greater than 0, that is, if the evidence is more likely to be present than not. EMYCIN restricts rule firing further by requiring that $C(E) > 0.2$ for a rule to be considered applicable. The justification for this heuristic is that it saves computational power and makes explanations clearer, as marginally effective rules are suppressed. In fact, it is possible to allow rules to fire regardless of the value of $C(E)$. The absence of supporting evidence, indicated by $C(E) < 0$, would then be taken into account since CF' would have the opposite sign from CF .

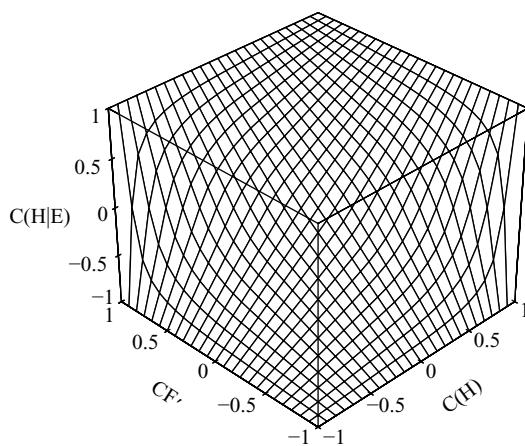


Figure 3.5 Equations 3.28–3.30 for updating certainties.

Although there is no theoretical justification for the function for updating certainty values, it does have a number of desirable properties:

1. The function is continuous and has no singularities or steps.
2. The updated certainty $C(H|E)$ always lies within the bounds -1 and $+1$.
3. If either $C(H)$ or CF' is $+1$ (i.e., definitely true), then $C(H|E)$ is also $+1$.
4. If either $C(H)$ or CF' is -1 (i.e., definitely false), then $C(H|E)$ is also -1 .
5. When contradictory conclusions are combined, they tend to cancel each other out, that is, if $C(H) = -CF'$, then $C(H|E) = 0$.
6. Several pieces of independent evidence can be combined by repeated application of the function, and the outcome is independent of the order in which the pieces of evidence are applied.
7. If $C(H) = 0$, that is, the certainty of H is at its *a priori* value, then $C(H|E) = CF'$.
8. If the evidence is certain (i.e., $C(E) = 1$), then $CF' = CF$.
9. Although not part of the standard implementation, the absence of evidence can be taken into account by allowing rules to fire when $C(E) < 0$.

3.3.3 Logical Combinations of Evidence

In Bayesian updating systems, each piece of evidence that contributes toward a hypothesis is assumed to be independent and is given its own *affirms* and *denies* weights. In systems based upon certainty theory, the certainty factor is associated with the rule as a whole rather than with individual pieces of evidence. For this reason, certainty theory provides a simple algorithm for determining the value of the certainty factor that should be applied when more than one item of evidence

is included in a single rule. The relationship between pieces of evidence is made explicit by the use of *and* and *or*. If separate pieces of evidence are intended to contribute toward a single hypothesis independently of each other, they must be placed in separate rules. The algorithm for combining items of evidence in a single rule is borrowed from Zadeh's possibility theory (Section 3.4). The algorithm covers the cases where evidence is conjoined (i.e., joined by *and*), disjoined (i.e., joined by *or*), and negated (using *not*).

3.3.3.1 Conjunction

Consider a rule of the form:

```
uncertainty_rule generic2
  if <evidence E1>
  and <evidence E2>
  then <hypothesis>
  with certainty factor <CF>.
```

The certainty of the combined evidence is given by $C(E_1 \text{ and } E_2)$, where:

$$C(E_1 \text{ and } E_2) = \min[C(E_1), C(E_2)] \quad (3.31)$$

3.3.3.2 Disjunction

Consider a rule of the form:

```
uncertainty_rule generic3
  if <evidence E1>
  or <evidence E2>
  then <hypothesis>
  with certainty factor <CF>.
```

The certainty of the combined evidence is given by $C(E_1 \text{ or } E_2)$, where:

$$C(E_1 \text{ or } E_2) = \max[C(E_1), C(E_2)] \quad (3.32)$$

3.3.3.3 Negation

Consider a rule of the form:

```
uncertainty_rule generic4
  if not <evidence E>
  then <hypothesis>
  with certainty factor <CF>.
```

The certainty of the negated evidence, $C(\neg E)$, is given by $C(\neg E) = -C(E)$, where:

$$C(\neg E) = -C(E) \quad (3.33)$$

3.3.4 A Worked Example of Certainty Theory

In order to illustrate the application of certainty theory, we can rework the example that was used to illustrate Bayesian updating. Four rules were used, which together could determine whether the release valve of a power-station boiler needs cleaning (see Section 3.2.8). Each of the four rules can be rewritten with an associated certainty factor, which is estimated by the rule-writer:

```
uncertainty_rule r3_1c
  if release_valve is stuck
  then task becomes clean_release_valve
  with certainty factor 1.0 .

uncertainty_rule r3_2c
  if warning_light is on
  then release_valve becomes stuck
  with certainty factor 0.2 .

uncertainty_rule r3_3c
  if pressure is high
  then release_valve becomes stuck
  with certainty factor 0.9 .

uncertainty_rule r3_4c
  if temperature is high
  and water_level is not low
  then pressure becomes high
  with certainty factor 0.5 .
```

Although the process of providing certainty factors might appear *ad hoc* compared with Bayesian updating, it may be no less reliable than estimating the probabilities upon which Bayesian updating relies. In the Bayesian example, the production rule r3_1b had to be treated as a special case. In a system based upon uncertainty theory, rule r3_1c can be made to behave as a production rule simply by giving it a certainty factor of 1.

As before, the following set of input data will be considered:

```
water_level is not low.
warning_light is on.
temperature is high.
```

We will assume that the rules fire in the order:

$$r3_4c \rightarrow r3_3c \rightarrow r3_2c \rightarrow r3_1c$$

The resultant rule trace might then appear as follows:

```

uncertainty_rule r3_4c; CF = 0.5
H = pressure is high; C(H) = 0
E1 = temperature is high; C(E1) = 1
E2 = water_level is low; C(E2) = -1, C(~E2) = 1
C(E1&~E2) = min[C(E1),C(~E2)] = 1
CF' = CF × C(E1&~E2) = CF
C(H|(E1&~E2)) = CF' = 0.5
/* Updated certainty of "pressure is high" is 0.5 */

uncertainty_rule r3_3c; CF = 0.9
H = release_valve is stuck; C(H) = 0
E = pressure is high; C(E) = 0.5
CF' = CF × C(E) = 0.45
C(H|(E)) = CF' = 0.45
/* Updated certainty of "release_valve is stuck" is 0.45 */

uncertainty_rule r3_2c; CF = 0.2
H = release_valve is stuck; C(H) = 0.45
E = warning_light is on; C(E) = 1
CF' = CF × C(E) = CF
C(H|(E)) = C(H) + [CF' × (1-C(H))] = 0.56
/* Updated certainty of "release_valve is stuck" is 0.56 */

uncertainty_rule r3_1c; CF = 1
H = task is clean_release_valve; C(H) = 0
E = release_valve is stuck; C(E) = 0.56
CF' = CF × C(E) = 0.56
C(H|(E)) = CF' = 0.56
/* Updated certainty of "task is clean_release_valve" is
0.56 */

```

3.3.5 Discussion of the Worked Example

Given the certainty factors shown, the example yielded the result that the release valve needs cleaning with a similar level of confidence to the Bayesian updating example.

Under Bayesian updating, rules r3_2b and r3_3b could be combined into a single rule without changing their effect:

```

uncertainty_rule r3_5b
if warning_light is on (affirms 2.20; denies 0.20)
and pressure is high (affirms 85.0; denies 0.15)
then release_valve becomes stuck.

```

With certainty theory, the weightings apply not to the individual pieces of evidence (as with Bayesian updating) but to the rule itself. If rules r3_2c and r3_3c were combined in one rule, a single certainty factor would need to be chosen to replace the two used previously. Thus, a combined rule might look like:

```
uncertainty_rule r3_5c
  if warning_light is on
  and pressure is high
  then release_valve becomes stuck
  with certainty factor 0.95 .
```

In the combined rule, the two items of evidence are no longer treated independently, and the certainty factor is the adjudged weighting if *both* items of evidence are present. If our worked example had contained this combined rule instead of rules r3_2c and r3_3c, then the rule trace would contain the following:

```
uncertainty_rule r3_5c; CF = 0.95
  H = release_valve is stuck; C(H) = 0
  E1 = warning_light is on; C(E1) = 1
  E2 = pressure is high; C(E2) = 0.5
  C(E1 & E2) = min[C(E1), C(E2)] = 0.5
  CF' = CF × C(E1 & E2) = 0.48
  C(H | (E1 & E2)) = CF' = 0.48
  /* Updated certainty of "release_valve is stuck" is 0.48 */
```

With the certainty factors used in the example, the combined rule yields a lower confidence in the hypothesis `release_valve is stuck` than rules r3_2c and r3_3c used separately. As a knock-on result, rule r3_1c would yield the conclusion `task is clean_release_valve` with a diminished certainty of 0.48.

3.3.6 Relating Certainty Factors to Probabilities

It has already been noted that there is a similarity between the certainty factors that are attached to hypotheses and the probabilities of those hypotheses, such that

- C(H) = 1.0 corresponds to P(H) = 1.0;
- C(H) = 0.0 corresponds to P(H) being at its *a priori* value;
- C(H) = -1.0 corresponds to P(H) = 0.0.

Additionally, a formal relationship exists between the certainty factor associated with a rule and the conditional probability $P(H|E)$ of a hypothesis H given some evidence E. This is only of passing interest as certainty factors are not normally

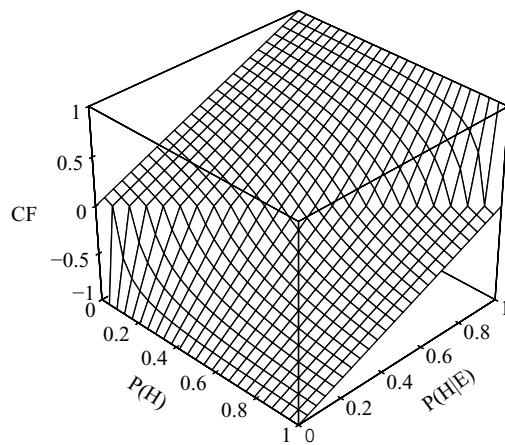


Figure 3.6 The relationship between certainty factors and probability.

calculated in this way, but instead are simply estimated or chosen to give suitable results. The formal relationships are as follows.

If evidence E supports hypothesis H, that is, $P(H|E)$ is greater than $P(H)$, then:

$$\left. \begin{array}{ll} CF = \frac{P(H|E) - P(H)}{1 - P(H)} & \text{if } P(H) \neq 1 \\ CF = 1 & \text{if } P(H) = 1 \end{array} \right\} \quad (3.34)$$

If evidence E opposes hypothesis H, that is, $P(H|E)$ is less than $P(H)$, then:

$$\left. \begin{array}{ll} CF = \frac{P(H|E) - P(H)}{P(H)} & \text{if } P(H) \neq 0 \\ CF = -1 & \text{if } P(H) = 0 \end{array} \right\} \quad (3.35)$$

The shape of Equations 3.34 and 3.35 is shown in Figure 3.6.

3.4 Fuzzy Logic: Type-1

Bayesian updating and certainty theory are techniques for handling the uncertainty that arises, or is assumed to arise, from statistical variations or randomness. Fuzzy logic, sometimes called *possibility theory*, addresses a different source of uncertainty, namely vagueness in the use of language. Fuzzy logic was developed by Zadeh (1975, 1983a, 1983b) and builds upon his theory of fuzzy sets (Zadeh 1965). Zadeh asserts that while probability theory may be appropriate for measuring the likelihood of a hypothesis, it says nothing about the *meaning* of the hypothesis. A goal of fuzzy logic is to achieve “computing with words” (Zadeh 1996). This section focuses on

the simplest and most commonly used form of fuzzy logic, known as *type-1*. Section 3.6 discusses a more sophisticated form of fuzzy logic, known as *type-2*.

3.4.1 Crisp Sets and Fuzzy Sets

The rules shown in this chapter and in Chapter 2 contain a number of examples of vague language where fuzzy sets might be applied, such as the following phrases:

- Water level is low.
- Temperature is high.
- Pressure is high.

In conventional set theory, the sets high, medium, and low—applied to a variable such as temperature—would be mutually exclusive. If a given temperature (say, 200°C) is high, then it is neither medium nor low. Such sets are said to be crisp or nonfuzzy (Figure 3.7). If the boundary between medium and high is set at 160°C, then a temperature of 161°C is considered high, while 159°C is considered medium. This distinction is rather artificial, and it means that a tiny difference in temperature can completely change the rule-firing, while a rise in temperature from 161°C to 300°C has no effect at all.

Fuzzy sets are a means of smoothing out the boundaries. The theory of fuzzy sets expresses imprecision quantitatively by introducing characteristic membership functions that can assume values between 0 and 1 corresponding to degrees of membership from “not a member” through to “a full member.” If F is a fuzzy set, then the membership function $\mu_F(x)$ measures the degree to which an absolute value x belongs to F . This degree of membership is sometimes called the *possibility* that x is described by F . The process of deriving these possibility values for a given value of x is called *fuzzification*.

Conversely, consider that we are given the imprecise statement `temperature is low`. If LT is the fuzzy set of low temperatures, then we might define the membership function μ_{LT} such that:

$$\begin{aligned}\mu_{LT}(140^\circ\text{C}) &= 0.0 \\ \mu_{LT}(120^\circ\text{C}) &= 0.0 \\ \mu_{LT}(100^\circ\text{C}) &= 0.25\end{aligned}$$

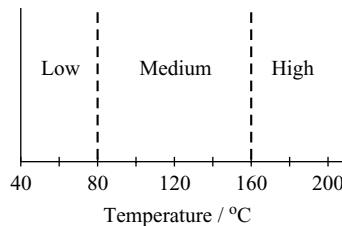


Figure 3.7 Conventional crisp sets applied to temperature.

$$\mu_{LT}(80^\circ\text{C}) = 0.5$$

$$\mu_{LT}(60^\circ\text{C}) = 0.75$$

$$\mu_{LT}(40^\circ\text{C}) = 1.0$$

$$\mu_{LT}(20^\circ\text{C}) = 1.0$$

These values correspond with the linear membership function shown in Figure 3.8a. Although linear membership functions like those in Figures 3.8a, b are convenient in many applications, the most suitable shape of the membership functions and the number of fuzzy sets depends on the particular application. Figures 3.8c, d show some nonlinear alternatives.

The key differences between fuzzy and crisp sets are that:

- an element has a degree of membership [0–1] of a fuzzy set;
- membership of one fuzzy set does not preclude membership of another.

Thus, the temperature 180°C may have some (nonzero) degree of membership to both fuzzy sets *high* and *medium*. This is represented in Figure 3.8 by the overlap between the fuzzy sets. The sum of the membership functions for a given value can be arranged to equal 1, as shown for temperature and pressure in Figure 3.8, but this is not a necessary requirement.

Some of the terminology of fuzzy sets may require clarification. The statement *temperature is low* is an example of a *fuzzy statement* involving a *fuzzy set* (low temperature) and a *fuzzy variable* (temperature). A fuzzy variable is one

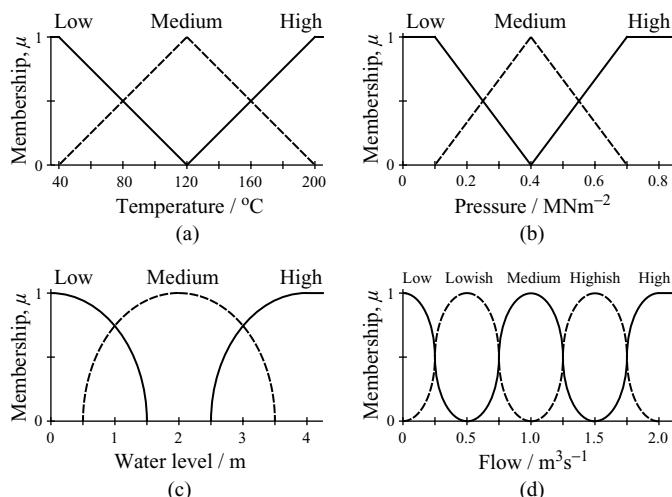


Figure 3.8 A variety of membership functions.

that can take any value from a global set (e.g., the set of all temperatures), where each value can have a degree of membership of a fuzzy set (e.g., low temperature) associated with it.

Although the discussion so far has concentrated on continuous variables such as temperature and pressure, the same ideas can also be applied to discrete variables, such as the number of signals detected in a given time span.

3.4.2 Fuzzy Rules

If a variable is set to a value by crisp rules, its value will change in steps as different rules fire. The only way to smooth those steps would be to have a large number of rules. However, only a small number of fuzzy rules are required to produce smooth changes in the outputs as the input values alter. The number of fuzzy rules required is dependent on the number of variables, the number of fuzzy sets, and the ways in which the variables are combined in the fuzzy rule conditions. Numerical information is explicit in crisp rules, for example, “if temperature > 160°C then ...” but in fuzzy rules it becomes implicit in the chosen shape of the fuzzy membership functions.

Consider a rule base that contains the following fuzzy rules:

```
fuzzy_rule r3_6f
  if temperature is high
  then pressure becomes high.
```

```
fuzzy_rule r3_7f
  if temperature is medium
  then pressure becomes medium.
```

```
fuzzy_rule r3_8f
  if temperature is low
  then pressure becomes low.
```

Suppose the measured boiler temperature is 180°C. As this is a member of both fuzzy sets *high* and *medium*, rules r3_6f and r3_7f will both fire. The pressure, we conclude, will be somewhat high and somewhat medium. Suppose that the membership functions for temperature are as shown in Figure 3.8a. The possibility that the temperature is high, μ_{HT} , is 0.75, and the possibility that the temperature is medium, μ_{MT} is 0.25. As a result of firing the rules, the possibilities that the pressure is high and medium, μ_{HP} and μ_{MP} are set as follows:

$$\mu_{HP} = \max[\mu_{HT}, \mu_{HP}]$$

$$\mu_{MP} = \max[\mu_{MT}, \mu_{MP}]$$

The initial possibility values for pressure are assumed to be zero if these are the first rules to fire, and thus μ_{HP} and μ_{MP} become 0.75 and 0.25, respectively. These values can be passed on to other rules that might have pressure is high or pressure is medium in their condition clauses.

The rules r3_6f, r3_7f and r3_8f contain only simple conditions. Possibility theory provides a recipe for computing the possibilities of compound conditions. The formulas for conjunction, disjunction, and negation are similar to those used in certainty theory (Section 3.3.3):

$$\left. \begin{array}{l} \mu_{X \text{ and } Y} = \min[\mu_X, \mu_Y] \\ \mu_{X \text{ or } Y} = \max[\mu_X, \mu_Y] \\ \mu_{\text{not } X} = 1 - \mu_X \end{array} \right\} \quad (3.36)$$

To illustrate the use of these formulas, suppose that water level has the fuzzy membership functions shown in Figure 3.8c and that rule r3_6f is redefined as follows:

```
fuzzy_rule r3_9f
  if temperature is high
  and water_level is not low
  then pressure becomes high.
```

For a water level of 1.2 m, the possibility of the water level being low, μ_{LW} , is 0.6. The possibility of the water level not being low is therefore 0.4. As this is less than 0.75, the combined possibility for the temperature being high and the water level not being low is 0.4. Thus, the possibility that the pressure is high, μ_{HP} , becomes 0.4 if it has not already been set to a higher value.

If several rules affect the same fuzzy set of the same variable, they are equivalent to a single rule whose conditions are joined by the disjunction *or*. For example, these two rules:

```
fuzzy_rule r3_6f
  if temperature is high
  then pressure becomes high.
```

```
fuzzy_rule r3_10f
  if water_level is high
  then pressure becomes high.
```

are equivalent to this single rule:

```
fuzzy_rule r3_11f
  if temperature is high
  or water_level is high
  then pressure becomes high.
```

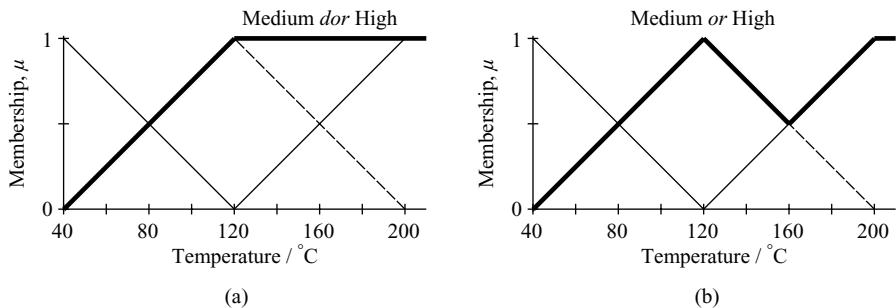


Figure 3.9 (a) Dependent or; (b) standard or.

Aoki and Sasaki (1990) have argued for treating the disjunction *or* differently when it involves two fuzzy sets of the same fuzzy variable, for example, high and medium temperature. In such cases, the memberships are clearly dependent on each other. Therefore, we can introduce a new operator *dor* for “dependent or.” For example, given the rule

```
fuzzy_rule r3_12f
  if temperature is low
  dor temperature is medium
  then pressure becomes lowish.
```

the combined possibility for the condition becomes

$$\mu_{LT \text{ dor } MT} = \min[1, \mu_{LT} + \mu_{MT}] \quad (3.37)$$

Given the fuzzy sets for temperature shown in Figure 3.8a, the combined possibility would be the same for any temperature below 120°C, as shown in Figure 3.9a. This is consistent with the intended meaning of fuzzy rule r3_12f. If the *or* operator had been used instead of *dor*, the membership would dip between 40°C and 120°C, with a minimum at 80°C, as shown in Figure 3.9b.

3.4.3 Defuzzification

In the foregoing example, at a temperature of 180°C, the possibilities for the pressure being high and medium, μ_{HP} and μ_{MP} , are set to 0.75 and 0.25, respectively, by the fuzzy rules r3_6f and r3_7f. It is assumed that the possibility for the pressure being low, μ_{LP} , remains at 0. These values can be passed on to other rules that might have pressure is high or pressure is medium in their condition clauses without any further manipulation. However, if we want to interpret these membership values in terms of a numerical value of pressure, they would need to be *defuzzified*.

Defuzzification is particularly important when the fuzzy variable is a control action such as “set current,” where a specific setting is required. The use of fuzzy logic in control systems is discussed further in Section 3.5. Defuzzification takes place in two stages, described in the following subsections.

3.4.3.1 Stage 1: Scaling the Membership Functions

The first step in defuzzification is to adjust the fuzzy sets in accordance with the calculated possibilities. A commonly used method is Larsen’s product operation rule (Lee 1990a, 1990b), in which the membership functions are multiplied by their respective possibility values. The effect is to compress the fuzzy sets so that the peaks equal the calculated possibility values, as shown in Figure 3.10. Some authors (Johnson and Picton 1995) adopt an alternative approach in which the fuzzy sets are truncated, as shown in Figure 3.11. For most shapes of fuzzy set, the difference between the two approaches is small, but Larsen’s product operation rule has the advantages of simplifying the calculations and allowing fuzzification followed by defuzzification to return the initial value, except as described in “A defuzzification anomaly” in Section 3.4.3.5.

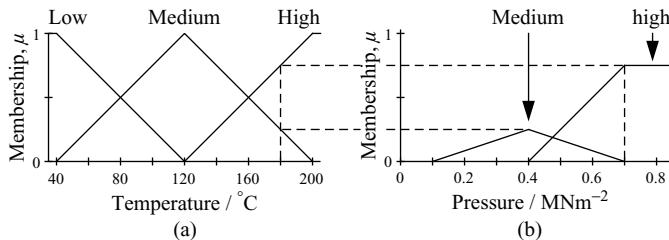


Figure 3.10 Larsen’s product operation rule for calculating membership functions from fuzzy rules. Membership functions for pressure are shown, derived from rules r3_6f and r3_7f, for a temperature of 180°C.

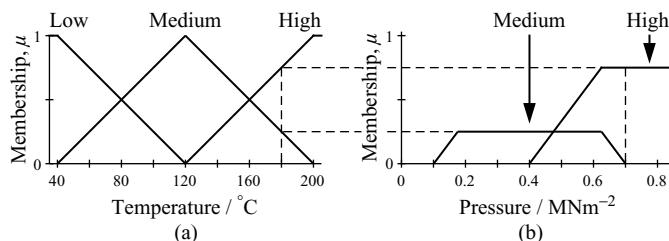


Figure 3.11 Truncation method for calculating membership functions from fuzzy rules. Membership functions for pressure are shown, derived from rules r3_6f and r3_7f, for a temperature of 180°C.

3.4.3.2 Stage 2: Finding the Centroid

The most commonly used method of defuzzification is the *centroid* method, sometimes called the center of gravity, center of mass, or center of area method. Defuzzification by determining the centroid of the fuzzy output function is part of an overall process known as Mamdani-style fuzzy inference (Mamdani 1977). The defuzzified value is taken as the point along the fuzzy variable axis that is the centroid, or balance point, of all the scaled membership functions taken together for that variable (Figure 3.12). One way to visualize this is to imagine the membership functions cut out from stiff card and pasted together where (and if) they overlap. The defuzzified value is the balance point along the fuzzy variable axis of this composite shape. When two membership functions overlap, both overlapping regions contribute to the mass of the composite shape. Figure 3.12 shows a simple case, involving only two fuzzy sets. The example that we have been following concerning boiler pressure is more complex and is described in the next subsection.

If there are N membership functions with centroids c_i and areas a_i , then the combined centroid C , that is, the defuzzified value, is:

$$C = \frac{\sum_{i=1}^N a_i c_i}{\sum_{i=1}^N a_i} \quad (3.38)$$

When the fuzzy sets are compressed using Larsen's product operation rule, the values of c_i are unchanged from the centroids of the uncompressed shapes, C_i , and a_i is simply $\mu_i A_i$, where A_i is the area of the membership function prior to compression. (This is not the case with the truncation method shown in Figure 3.11, which causes the centroid of asymmetrical membership functions to shift along the fuzzy variable axis.) The use of triangular membership functions or other simple geometries simplifies the calculations further. For triangular membership functions, A_i is one half of the base length multiplied by the height. For isosceles triangles C_i is the

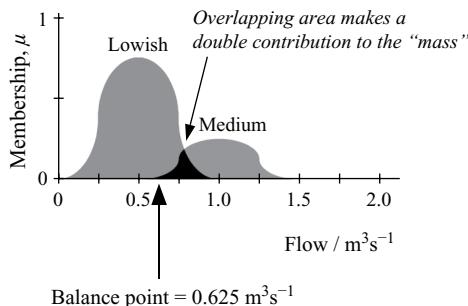


Figure 3.12 Defuzzification by the centroid method.

midpoint along the base, and for right-angle triangles, C_i is approximately 29% of the base length from the upright.

3.4.3.3 Defuzzifying at the Extremes

There is a complication in defuzzifying whenever the two extreme membership functions are involved, that is, those labeled *high* and *low* here. Given the fuzzy sets shown in Figure 3.8b, any pressure above 0.7 MNm⁻² has a membership of *high* of 1. Thus, the membership function continues indefinitely toward the right and we cannot find a balance point using the centroid method. Similarly, any pressure below 0.1 MNm⁻² has a membership of *low* of 1, although in this case the membership function is bounded because the pressure cannot go below 0.

One solution to these problems might be to specify a range for the fuzzy variable, *MIN–MAX*, or 0.1–0.7 MNm⁻² in this example. During fuzzification, a value outside this range can be accepted and given a membership of 1 for the fuzzy sets *low* or *high*. However, during defuzzification, the *low* and *high* fuzzy sets can be considered bounded at *MIN* and *MAX* and defuzzification by the centroid method can proceed. This method is shown in Figure 3.13a using the values 0.75 and 0.25 for μ_{HP} and μ_{LP} respectively, as calculated in Section 3.4.2, yielding a defuzzified pressure of 0.527 MNm⁻². A drawback of this solution is that the defuzzified value can never reach the extremes of the range. For example, if we know that a fuzzy variable has a membership of 1 for the fuzzy set *high* and 0 for the other fuzzy sets, then its actual value could be any value greater than or equal to *MAX*. However, its defuzzified value using this scheme would be the centroid of the *high* fuzzy set, in this case 0.612 MNm⁻², which is considerably below *MAX*.

An alternative solution is the *mirror rule*. During defuzzification only, the *low* and *high* membership functions are treated as symmetrical shapes centered on *MIN* and *MAX*, respectively. This is achieved by reflecting the *low* and *high* fuzzy sets in imaginary mirrors. This method has been used in Figure 3.13b, yielding a significantly different result, that is, 0.625 MNm⁻², for the same possibility values. The method uses the full range *MIN–MAX* of the fuzzy variable during defuzzification,

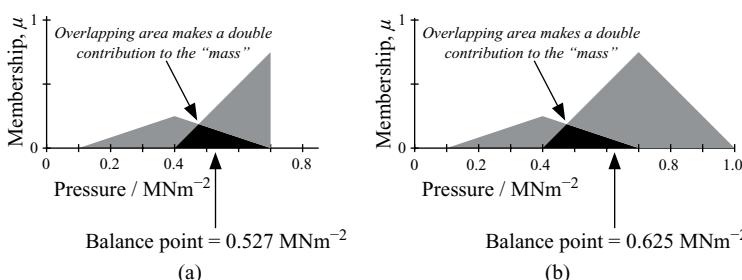


Figure 3.13 Defuzzification at the extremes: (a) bounded range and (b) mirror rule.

so that a fuzzy variable with a membership of 1 for the fuzzy set *high* and 0 for the other fuzzy sets would be defuzzified to *MAX*.

3.4.3.4 Sugeno Defuzzification

So far, we have considered defuzzification by determining the centroid of the fuzzy output function, i.e., the Mamdani method (Mamdani 1977). As the output may be a combination of several scaled membership functions, determination of the centroid can become computationally expensive. However, the combination of four assumptions can greatly simplify the defuzzification calculation:

- Larsen's product operation rule is applied.
- The mirror rule is applied.
- The membership functions for the output are symmetrical (after applying the mirror rule, if required, at the extremes).
- The membership functions for the output have the same area A_i (prior to compression through Larsen's product operation rule).

When all four assumptions are made, the calculation of the centroid of the fuzzy output function becomes equivalent to determining the weighted average of a set of singletons, that is, spike functions, as shown in Figure 3.14. With the introduction of this simplification of the defuzzification calculation, the whole process is known as *zero-order Sugeno fuzzy inference* (Takagi and Sugeno 1985). The height of each spike is the same as the height of the corresponding fuzzy output in a Mamdani-style system. The centroid, that is, the weighted sum of the singletons, is given by:

$$C = \frac{\sum_{i=1}^N \mu_i C_i}{\sum_{i=1}^N \mu_i} \quad (3.39)$$

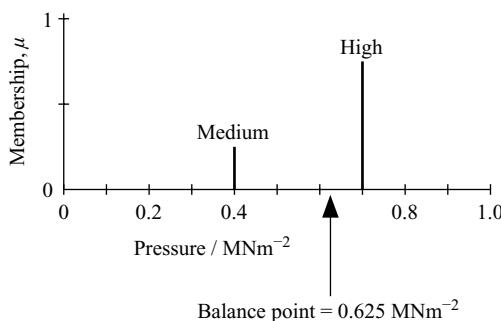


Figure 3.14 Sugeno defuzzification.

where each value of μ_i is the height of spike i and C_i is its position along the fuzzy-variable axis (Hopgood et al. 1998). Equation 3.39 represents a considerable simplification of the more general equation for defuzzification shown in Equation 3.38.

3.4.3.5 A Defuzzification Anomaly

It is interesting to investigate whether defuzzification can be regarded as the inverse of fuzzification. In the foregoing example, a pressure of 0.625 MNm^{-2} would fuzzify to a membership of 0.25 for *medium* and 0.75 for *high*. When defuzzified by the method shown in Figure 3.13b, the original value of 0.625 MNm^{-2} is returned. This observation provides strong support for defuzzification based upon Larsen's product operation rule combined with the mirror rule for dealing with the fuzzy sets at the extremes (Figure 3.13b). No such simple relationship exists if the membership functions are truncated (Figure 3.11) instead of being compressed using Larsen's product operation rule, or if the extremes are handled by imposing a range (Figure 3.13a) instead of applying the mirror rule.

However, even the full set of four assumptions listed earlier for zero-order Sugeno fuzzy inference cannot always guarantee that fuzzification and defuzzification will be straightforward inverses of each other. For example, as a result of firing a set of fuzzy rules, we might end up with the following memberships for the fuzzy variable pressure:

- Low membership = 0.25
- Medium membership = 0.0
- High membership = 0.25

Defuzzification of these membership values would yield an absolute value of 0.4 MNm^{-2} for the pressure (Figure 3.15a). If we were now to look up the fuzzy

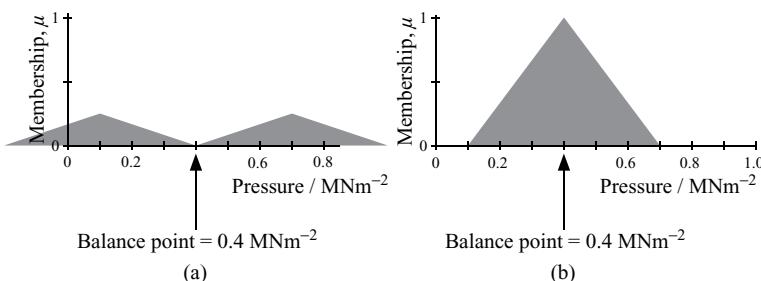


Figure 3.15 Different combinations of memberships can defuzzify to the same value.

memberships for an absolute value of 0.4 MNm^{-2} , that is, to fuzzify the value, we would obtain:

- Low membership = 0.0
- Medium membership = 1.0
- High membership = 0.0

The resulting membership values are clearly different from the ones we started with, although they still defuzzify to 0.4 MNm^{-2} , as shown in Figure 3.15b. The reason for this anomaly is that, under defuzzification, there are many different combinations of membership values that can yield an absolute value such as 0.4 MNm^{-2} . The above sets of membership values are just two examples. However, under fuzzification, there is only one absolute value, namely 0.4 MNm^{-2} , that can yield fuzzy membership values for *low*, *medium*, and *high* of 0.0, 1.0, and 0.0, respectively. Thus, defuzzification is said to be a “many-to-one” relationship, whereas fuzzification is a “one-to-one” relationship.

This observation poses a dilemma for implementers of a fuzzy system. If pressure appears in the condition part of further fuzzy rules, different membership values could be used depending on whether or not it is defuzzified and refuzzified before being passed on to those rules.

A secondary aspect of the anomaly is the observation that, in the foregoing example, we began with possibility values of 0.25 and, therefore, apparently rather weak evidence about the pressure. However, as a result of defuzzification followed by fuzzification, these values are transformed into evidence that appears much stronger. Johnson and Picton (1995) have labeled this “Hopgood’s defuzzification paradox.” The paradox arises because, unlike probabilities or certainty factors, possibility values need to be interpreted relative to each other rather than in absolute terms.

3.5 Fuzzy Control Systems

3.5.1 Crisp and Fuzzy Control

Control decisions can be thought of as a transformation from state variables to action variables (Figure 3.16). *State variables* describe the current state of the physical plant and the desired state. *Action variables* are those that can be directly altered

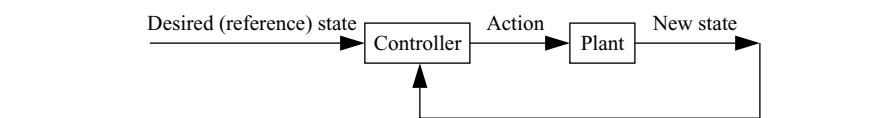


Figure 3.16 Control viewed as a transformation from state variables to action variables.

by the controller, such as the electrical current sent to a furnace, or the flow rate through a gas valve. In some circumstances, it may be possible to obtain values for the action variables by direct mathematical manipulation of the state variables. This is the case for a PID (proportional + integral + derivative) controller—see Chapter 15, Section 15.2.5. Given suitably chosen functions, this mathematical modeling approach causes values of action variables to change smoothly as values of state variables change. However, in high-level control, such as for a complex manufacturing process, analytical functions that link state variables to action variables are rarely available. This is one reason for using rules instead for this purpose.

Crisp sets are conventional Boolean sets, where an item is either a member (degree of membership = 1) or it is not (degree of membership = 0). Applying crisp sets to state and action variables corresponds to dividing up the range of allowable values into subranges, each of which forms a set. Suppose that a state variable such as temperature is divided into five crisp sets. A temperature reading can belong to only one of these sets, so only one rule will apply, resulting in a single control action. Thus, the number of different control actions is limited to the number of rules, which in turn is limited by the number of crisp sets. The action variables are changed in abrupt steps as the state variables change.

Fuzzy logic enables a small number of rules to produce smooth changes in the action variables as the state variables change. It is, therefore, particularly well-suited to control decisions where the control actions need to be scaled as input measurements change. The number of rules required is dependent on the number of state variables, the number of fuzzy sets, and the ways in which the state variables are combined in rule conditions. Numerical information is explicit in crisp rules, but in fuzzy rules it becomes implicit in the chosen shape of the fuzzy membership functions.

3.5.2 Fuzzy Control Rules

Some simple examples of fuzzy control rules for an industrial electric oven are as follows:

```
fuzzy_rule r3_13f
  if temperature is high
  or current is high
  then current_change becomes reduce.

fuzzy_rule r3_14f
  if temperature is medium
  then current_change becomes no_change.

fuzzy_rule r3_15f
  if temperature is low
  and current is high
```

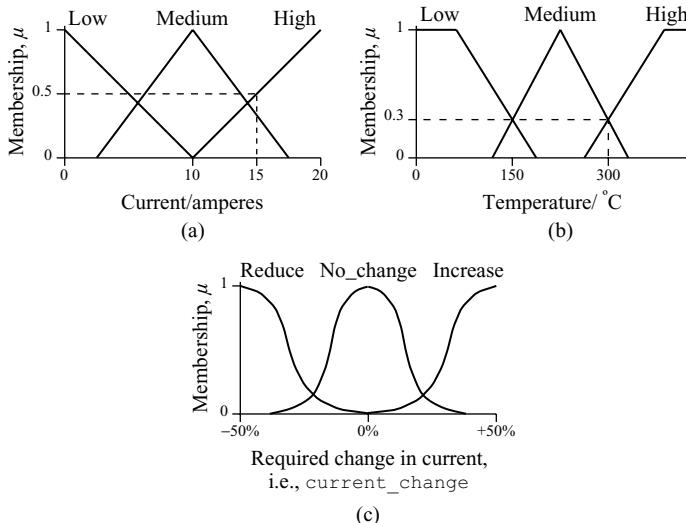


Figure 3.17 Fuzzy sets: (a) electric current (state variable), (b) temperature (state variable), and (c) change in electric current (action variable).

then `current_change` becomes `no_change`.

```
fuzzy_rule r3_16f
  if temperature is low
  and current is low
  then current_change becomes increase.
```

The rules and the fuzzy sets to which they refer are, in general, dependent on each other. Figure 3.17 shows some possible fuzzy membership functions, μ , for the state variables temperature and current, and for the required change in current, that is, for the action variable `current_change`. The state variables are the inputs to the fuzzy controller, and the action variable is the output. Since the fuzzy sets overlap, a temperature and current may have a nonzero degree of membership of more than one fuzzy set. Some variation in the shapes of the fuzzy sets has been introduced in this example, so that the sum of the memberships for a particular fuzzy variable is not necessarily equal to 1. Suppose that the recorded temperature is 300°C and the measured current is 15 A. The temperature and current are each a member of two fuzzy sets: *medium* and *high*. Rules *r3_13f* and *r3_14f* will fire, with the apparently contradictory conclusion that we should both reduce the electric current and leave it alone. Of course, what is actually required is *some* reduction in current.

Rule *r3_13f* contains a disjunction. Using Equation 3.36, the possibility value for the composite condition is as follows:

$$\max(\mu(\text{temperature is high}), \mu(\text{current is high})).$$

At 300°C and 15 A, $\mu(\text{temperature is high})$ is 0.3 and $\mu(\text{current is high})$ is 0.5. The composite possibility value is therefore 0.5, and $\mu(\text{current_change is reduce})$ becomes 0.5. Rule r3_14f is simpler, containing only a single condition. The possibility value $\mu(\text{temperature is medium})$ is 0.3 and so $\mu(\text{current_change is no_change})$ becomes 0.3.

3.5.3 Defuzzification in Control Systems

After firing rules r3_13f and r3_14f, current_change has a degree of membership, or possibility, for *reduce* and another for *no_change*. These fuzzy actions must be converted into a single precise action to be of any practical use, that is, they need to be *defuzzified*.

Assuming that we are using Larsen's Product Operation Rule (see Section 3.4.3.2), the membership functions for the control actions are compressed according to their degree of membership. Thus, the membership functions for *reduce* and for *no_change* are compressed so that their peak values become 0.5 and 0.3, respectively (Figure 3.18). Defuzzification can then take place by finding the centroid of the combined membership functions.

One of the membership functions, that is, *reduce*, covers an extremity of the fuzzy variable current_change and, therefore, continues indefinitely toward $-\infty$. As discussed in Section 3.4.3.4, a method of handling this is required in order to find the centroid. Figure 3.19 shows the effect of applying the mirror rule so that

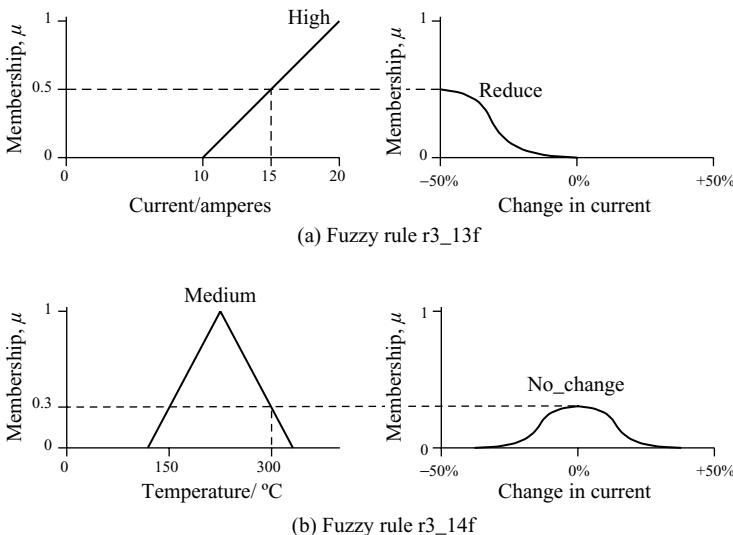


Figure 3.18 Firing fuzzy control rules using Larsen's Product Operator Rule.

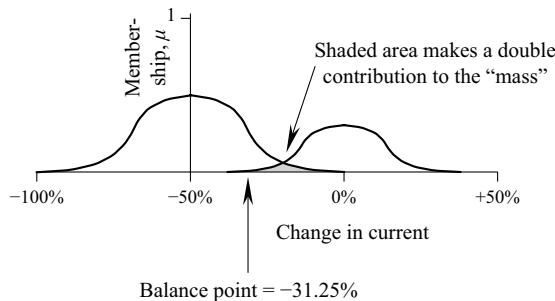


Figure 3.19 Defuzzifying a control action using the centroid method and mirror rule.

the membership function for *reduce* is treated, for defuzzification purposes only, as though it were symmetrical around -50% .

As the initial membership functions for the output, `current_change`, are symmetrical (after applying the mirror rule) and they have the same area, all of the assumptions for zero-order Sugeno fuzzy inference are now satisfied. So, the centroid C is given by:

$$C = \frac{\sum_{i=1}^N \mu_i C_i}{\sum_{i=1}^N \mu_i} \quad (3.39)$$

Inserting the values from Figure 3.18, we obtain:

$$C = \frac{(0.5 \times -50\%) + (0.3 \times 0\%)}{0.3 + 0.5} = -31.25\% \quad (3.40)$$

Thus, the defuzzified control action is a 31.25% reduction in the current, as shown in Figure 3.19.

3.6 Fuzzy Logic: Type-2

While type-1 fuzzy logic is undoubtedly useful, as demonstrated by its numerous applications in control systems and elsewhere, it is sometimes criticized for being too precise in its handling of membership functions. Since a given value of a variable, such as temperature, has a fixed membership value of a fuzzy set such as low, it might be argued that there is no uncertainty at all. This weakness was recognized by the architect of fuzzy logic himself, Zadeh, in his pioneering paper (Zadeh 1975).

To overcome this perceived weakness, he proposed that the membership of a fuzzy set could itself be treated as a fuzzy variable. The result is a “fuzzy-fuzzy” model, known as type-2 fuzzy logic. In principle, there is no need to stop there. The second-order membership could also be a fuzzy variable, resulting in type-3 fuzzy logic. The concept can be generalized to type- n fuzzy logic, although types 1 and 2 dominate current research and practice.

In the example considered in Figure 3.8a, a temperature such as 60°C has the following memberships of the type-1 fuzzy sets *low*, *medium*, and *high*:

$$\begin{aligned}\mu_{LT}(60^{\circ}\text{C}) &= 0.75 \\ \mu_{MT}(60^{\circ}\text{C}) &= 0.25 \\ \mu_{HT}(60^{\circ}\text{C}) &= 0\end{aligned}$$

This model successfully accommodates the use of vague language in the sense that words such as “low” and “medium” can represent a range of different temperatures with differing levels of membership. What it does not allow, however, are differing interpretations of the words “low” and “medium.” So, what we really need, and type-2 fuzzy logic provides, is a way to represent the idea that $\mu_{LT}(60^{\circ}\text{C})$ is “about 0.75,” and $\mu_{MT}(60^{\circ}\text{C})$ is “about 0.25.”

Mendel (2007) describes an experiment where 50 people are asked to define the endpoints of a description such as “low temperature.” (His example describes levels of eye contact, but the principle is the same.) In the context of an industrial boiler, the upper bound of “low temperature” might be set by such a sample of people to fall in the range 60°C–130°C, with a mean of 95°C. A type-1 fuzzy set could be constructed using the mean value, but this would not reflect the spread of opinion about how to interpret the phrase “low temperature” in this context. In particular, it would fail to recognize that a large proportion of the sample think that, in this context, “low temperature” can be above the boiling point of water at atmospheric pressure. (In the example used in Section 3.4, the upper bound for “low temperature” was actually set at 120°C.)

In order to model such a spread of opinion in type-2 fuzzy logic, we need to recognize the existence of a continuum of alternative fuzzy sets whose endpoints are in the specified range. The boundaries of the fuzzy set are no longer a line of infinitesimal thickness, but a band known as the *footprint of uncertainty*. Then, if we were to look up the membership of “low temperature” for a specific temperature such as 80°C, we would not find just a single membership value, as in a type-1 fuzzy set, but a range of *possibilities* from within the footprint of uncertainty. Values in this range might all be considered equally likely, in which case the fuzzy set is said to be *interval type-2* (Figure 3.20a). Alternatively, a weighting function known as the *secondary membership function*, z , can be imposed so that, for example, values close to the mean might be weighted most highly. The latter fuzzy sets are *generalized type-2* (Figure 3.20b). Interval type-2 fuzzy sets are attractive because of their

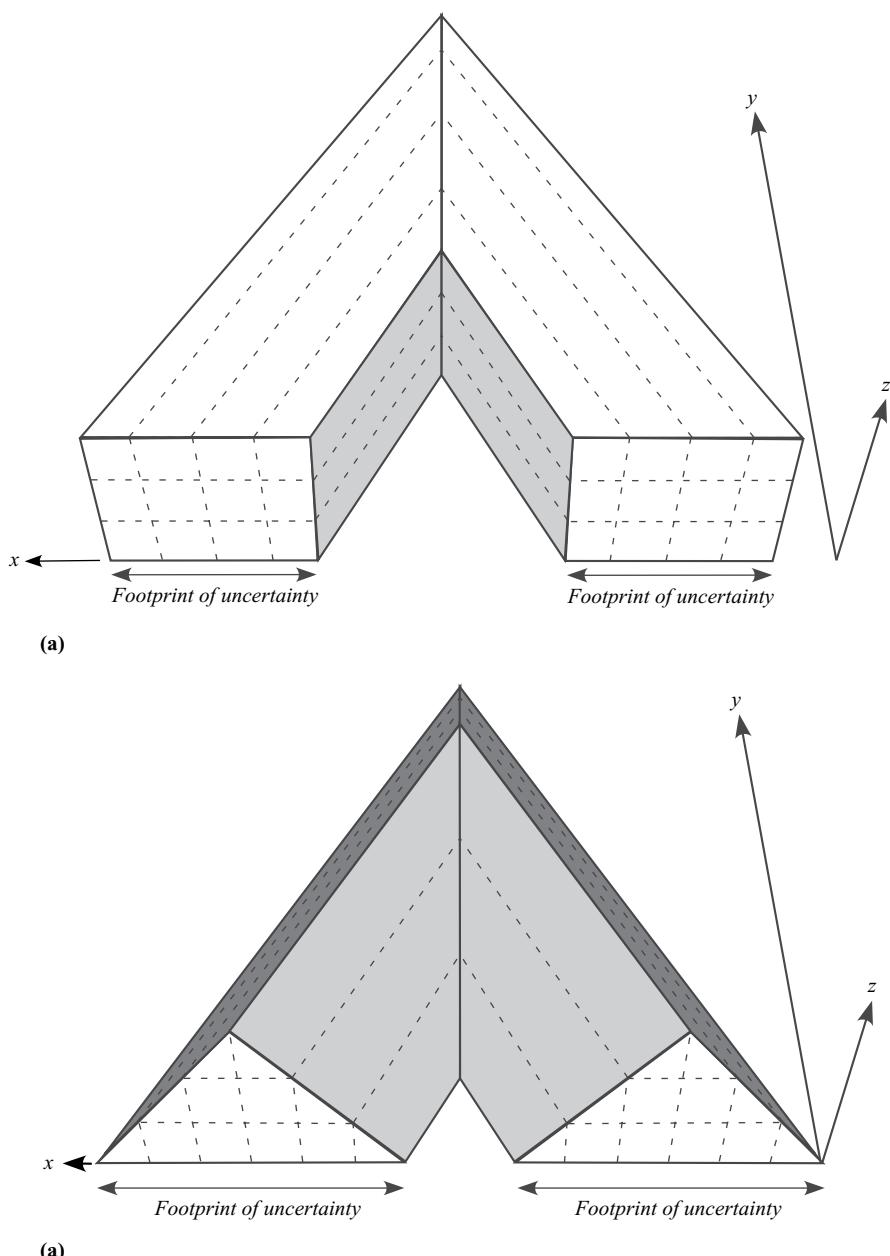


Figure 3.20 Type-2 fuzzy sets: (a) interval and (b) generalized. (Derived from Wagner, C., and H. Hagras 2010.)

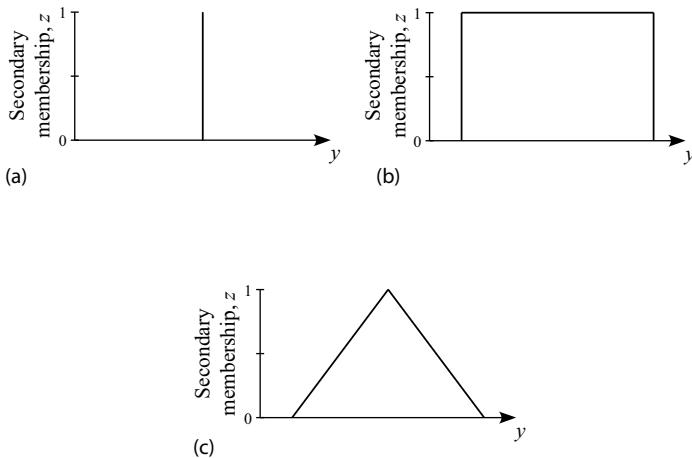


Figure 3.21 Secondary membership functions for (a) type-1 fuzzy set, (b) interval type-2 fuzzy set, and (c) generalized type-2 fuzzy set. (Derived from Wagner, C., and H. Hagras 2010.)

computational simplicity, but generalized type-2 fuzzy sets have grown in popularity owing to their greater expressiveness (John and Coupland 2007). The secondary membership functions for type-1, interval type-2, and generalized type-2 fuzzy sets are compared in Figure 3.21.

3.7 Other Techniques

Fuzzy logic, or possibility theory, occupies a distinct position among the many strategies for handling uncertainty, as it is the only established one that is concerned specifically with uncertainty arising from imprecise use of language. Techniques have been developed for dealing with other specific sources of uncertainty. For example, plausibility theory (Rescher 1976) addresses the problems arising from unreliable or contradictory sources of information. Other techniques have been developed in order to overcome some of the perceived shortcomings of Bayesian updating and certainty theory. Notable among these are the Dempster–Shafer theory of evidence and Quinlan’s Inferno, both of which are briefly reviewed here.

None of the more specialized techniques for handling uncertainty overcomes the most difficult problem, namely, obtaining accurate estimates of the likelihood

of events and combinations of events. For this reason, their use may be difficult to justify in many practical knowledge-based systems.

3.7.1 Dempster–Shafer Theory of Evidence

The theory of evidence (Barnett 1981) is a generalization of probability theory that was created by Dempster and developed by Shafer (Schafer 1976). It addresses two specific deficiencies of probability theory that have already been highlighted, namely that

- the single probability value for the truth of a hypothesis tells us nothing about its precision;
- because evidence for and against a hypothesis are lumped together, we have no record of how much there is of each.

Rather than representing the probability of a hypothesis H by a single value $P(H)$, Dempster and Shafer's technique binds the probability to a subinterval $L(H)–U(H)$ of the range 0–1. Although the exact probability $P(H)$ may not be known, $L(H)$ and $U(H)$ represent lower and upper bounds on the probability, such that:

$$L(H) \leq P(H) \leq U(H) \quad (3.41)$$

The precision of our knowledge about H is characterized by the difference $U(H) - L(H)$. If this difference is small, our knowledge about H is fairly precise, but if it is large, we know relatively little about H . A clear distinction is therefore made between uncertainty and ignorance, where uncertainty is expressed by the limits on the value of $P(H)$, and ignorance is represented by the size of the interval defined by those limits. According to Buchanan and Duda (1983), Dempster and Shafer have pointed out that the Bayesian agony of assigning prior probabilities to hypotheses is often due to ignorance of the correct values, and this ignorance can make any particular choice arbitrary and unjustifiable.

The foregoing ordering (3.41) can be interpreted as two assertions:

- The probability of H is at least $L(H)$.
- The probability of $\sim H$ is at least $1.0 - U(H)$.

Thus, a separate record is kept of degree of belief and disbelief in H . Like Bayesian updating, the theory of evidence benefits from the solid basis of probability theory for the interpretation of $L(H)$ and $U(H)$. When $L(H) = U(H)$, the theory of evidence reduces to the Bayesian updating method. It is, therefore, not surprising that the theory of evidence also suffers from many of the same difficulties.

3.7.2 Inferno

The conclusions that can be reached by the Dempster–Shafer theory of evidence are of necessity weaker than those that can be arrived at by Bayesian updating. If the available knowledge does not justify stronger solutions, then drawing weaker solutions is desirable. This theme is developed further in Inferno (Quinlan 1983), a technique that its creator, Quinlan, has subtitled “a cautious approach to uncertain inference.” Although Inferno is based upon probability theory, it avoids assumptions about the dependence or independence of pieces of evidence and hypotheses. As a result, the correctness of any inferences can be guaranteed, given the available knowledge. Thus, Inferno deliberately errs on the side of caution.

The three key motivations for the development of Inferno were as follows:

1. Other systems often make unjustified assumptions about the dependence or independence of pieces of evidence or hypotheses. Inferno allows users to state any such relationships when they are known, but it makes no assumptions.
2. Other systems take a measure of belief (e.g., probability or certainty) in a piece of evidence, and calculate from it a measure of belief in a hypothesis or conclusion. In terms of a Bayesian inference network (Figures 3.3 and 3.4), probabilities or certainty values are always propagated in one direction, namely, from the bottom (evidence) to the top (conclusions). Inferno allows users to enter values for any node on the inference network and to observe the effects on values at all other nodes.
3. Inferno informs the user of inconsistencies that might be present in the information presented to it and can make suggestions of ways to restore consistency.

Quinlan (1983) gives a detailed account of how these aims are achieved, and provides a comprehensive set of expressions for propagating probabilities throughout the nodes of an inference network.

3.8 Summary

A number of different schemes exist for assigning numerical values to assertions in order to represent levels of confidence in them and for updating the confidence levels in the light of supporting or opposing evidence. The greatest difficulty lies in obtaining accurate values of likelihood, whether measured as a probability or by some other means. The certainty factors that are associated with rules in certainty theory, and the *affirms* and *denies* weightings in Bayesian updating, can be derived from probability estimates. However, a more pragmatic approach is frequently

adopted, namely, to choose values that produce the right sort of results, even though the values cannot be theoretically justified. As the more sophisticated techniques (e.g., Dempster–Shafer theory of evidence and Inferno) also depend upon probability estimates that are often dubious, their use may be difficult to justify.

Bayesian updating is soundly based on probability theory, whereas many of the alternative techniques are ad hoc. In practice, Bayesian updating is also an ad hoc technique because:

- Linear interpolation of the *affirms* and *denies* weighting is frequently used as a convenient means of compensating for uncertainty in the evidence.
- The likelihood ratios (or the probabilities from which they are derived) and prior probabilities are often based on estimates rather than statistical analysis.
- Separate items of evidence that support a single assertion are assumed to be statistically independent, although this may not be the case in reality.

Neural networks (see Chapters 8 and 9) represent an alternative approach that avoids the difficulties in obtaining reliable probability estimates. Neural networks can be used to train a computer system using many examples, so that it can draw conclusions weighted according to the evidence supplied. Of course, given a large enough set of examples, it would also be possible to calculate accurately the prior probabilities and weightings needed in order to make Bayesian updating or one of its derivatives work effectively.

Fuzzy logic is also closely associated with neural networks, as will be discussed in Chapter 10. Fuzzy logic provides a precise way of handling vague terms such as “low” and “high” using fuzzy sets. Type-2 fuzzy logic takes this vagueness a stage further by allowing uncertainty in the membership functions that describe the fuzzy sets. Fuzzy logic enables a small set of rules to produce output values that change smoothly as the input values change. Fuzzy controllers exploit this characteristic by ensuring that deviations in the state variables of the controlled system are remedied by appropriately scaled control actions.

Further Reading

- Bouchon-Meunier, B., C. Marsala, M. Rifqi, and R. R. Yager, eds. 2008. *Uncertainty and Intelligent Information Systems*. World Scientific, Hong Kong.
- Chen, G. and T. T. Pham. 2019. *Introduction to Fuzzy Sets, Fuzzy Logic, and Fuzzy Control Systems*. CRC Press, Boca Raton, FL.
- Halpern, J. Y. 2017. *Reasoning about Uncertainty*. 2nd ed. MIT Press, Cambridge MA.
- Jantzen, J. 2013. *Foundations of Fuzzy Control*. 2nd ed. John Wiley & Sons, Chichester, UK.
- Ross, T. J. 2016. *Fuzzy Logic with Engineering Applications*. 4th ed. John Wiley & Sons, Chichester, UK.

Chapter 4

Agents, Objects, and Frames

4.1 Birds of a Feather: Agents, Objects, and Frames

System design usually involves breaking down complex problems into simpler constituents. Agents, objects, and frames are all ways of achieving this aim while maintaining the overall integrity of the system. Agents, especially *intelligent* agents, are autonomous entities that manage their own activities. Frames, on the other hand, are structures for organizing knowledge in a knowledge-based system. They were conceived as a versatile and expressive way of representing and organizing information. Object-oriented programming (OOP) is used in a wide variety of software systems including, but by no means limited to, intelligent systems. OOP is now established as a mainstay of software engineering.

All three techniques assist in the design of software and make the resultant software more maintainable, adaptable, and reusable. They provide a structure for breaking down a world representation into manageable components such as *house*, *owner*, and *dog*. In a frame-based system, each of these components could be represented by a frame, containing information about itself. Frames are passive in the sense that, like entries in a database, they don't perform any tasks themselves. Objects are similar, but as well as storing information about themselves they also have the ability to perform specified tasks. When they receive an instruction, they will perform an action as instructed, provided it is within their capability. They are therefore like obedient servants. Agents extend the idea further; they are not servants but independent entities in charge of their own actions.

This chapter will start by introducing intelligent agents, before considering the benefits and challenges of multiagent systems (MASs), in which multiple agents

interact with each other. OOP will then be described, using an example problem to illustrate the features and benefits of OOP. Finally, a review of frames focuses on their differences from objects.

4.2 Intelligent Agents

As the information world expands, people are becoming less and less able to act upon the escalating quantities of information presented to them. A way around this problem is to build *intelligent agents* that act as software assistants to take care of specific tasks for us. For instance, if you are seeking a specific piece of information, you might use an intelligent agent to apply a selection of Web search strategies and filter the recommended pages for you. In this way, you are presented with only two or three pages that precisely match your needs. An intelligent agent of this type, that personalizes itself to your individual requirements by learning your habits and preferences, is called a *user agent*.

Similarly, much of the trading on the world's stock exchanges is performed by intelligent agents. Reaping the benefits of some types of share dealing relies on reacting rapidly to minor price fluctuations. By the time a human trader has assimilated the data and made a decision, the opportunity would be lost.

Just as intelligent agents provide a way of alleviating complexity in the real world, they also fulfill a similar role within computer systems. As software systems become larger and more complex, it becomes progressively less feasible to maintain them as centralized systems that are designed and tested against every eventuality. An alternative approach is to take the idea of modular software toward its apotheosis, namely, to turn the modules into autonomous agents that can make their own intelligent decisions in a wide range of circumstances. Such agents allow the system to be largely self-managing, as they can be provided with knowledge of how to cope in particular situations, rather than being explicitly programmed to handle every foreseeable eventuality.

While noting that not all agents are intelligent, Wooldridge (2009) gives the following definition for an agent, updated from (Wooldridge 1997):

An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives.

From this definition we can see that two key characteristics of an agent are autonomy and the ability to interact with its environment. *Autonomy* refers to an agent's ability to make its own decisions based on its own experience and circumstances, and to control its own internal state and behavior. The definition implies that an agent functions continuously within its environment, i.e., it is *persistent* over time, although this is not explicitly stated. Agents are also said to be *situated*, i.e., they are

responsive to the demands of their environment and are capable of acting upon it. Interaction with a physical environment requires perception through sensors, and action through actuators or effectors. Interaction with a purely software environment is more straightforward, requiring only access to and manipulation of data and programs.

Intelligence can be added to a greater or lesser degree, but we might reasonably expect an *intelligent* agent to be all of the following:

- Reactive;
- Goal-directed;
- Adaptable; and
- Socially capable.

Social capability refers to the ability to cooperate and negotiate with other agents (or humans), which forms the basis of the section on *MASs*, below. It is quite easy to envisage an agent that is purely reactive, e.g., one whose only role is to place a warning on your computer screen when the printer has run out of paper. This behavior is sometimes described as a daemon (see Section 4.6.13). Likewise, modules of procedural computer code can be thought of as goal-directed in the limited sense that they have been programmed to perform a specific task regardless of their environment. Since it is autonomous, an intelligent agent can decide its own goals and choose its own actions in pursuit of those goals. At the same time, it must also be able to respond to unexpected changes in its environment. It, therefore, has to balance reactive and goal-directed behavior, typically through a mixture of problem-solving, planning, searching, decision-making, and learning through experience.

There is no reason why an agent should remain permanently on a single computer. If a computer is connected to a network, a *mobile agent* can travel to remote computers to carry out its designated task before returning back home with the task completed. A typical task for a mobile agent might be to determine a person's travel plan. This will require information about train and airline timetables, together with hotel availability. Instead of transferring large quantities of data across the network from the train companies, airlines, and hotels, it is more efficient for the agent to transfer itself to these remote sites, find the information it needs, and return. Clearly, there is potential for malicious use of mobile agents, so security is a prime consideration for their viability.

4.3 Agent Architectures

Any of the techniques met so far in this book could be used to provide the internal representation and reasoning capabilities of an agent. Nevertheless, several different types of approaches can be identified. There are at least four different schools of thought about how to achieve an appropriate balance between reactive and goal-directed behavior. These are reviewed in the following text.

4.3.1 Logic-Based Architectures

At one extreme, the purists favor logical deduction based on a symbolic representation of the environment (Ulrich et al. 1997; Russell and Norvig 2016). This approach is elegant and rigorous, but it relies on the environment remaining unchanged during the reasoning process. It also presents particular difficulties in symbolically representing the environment and reasoning about it.

4.3.2 Emergent Behavior Architectures

In contrast, other researchers propose that logical deduction about the environment is inappropriately detailed and time-consuming. For instance, if a heavy object is falling toward you, the priority should be to move out of the way rather than to analyze and prove the observation. These researchers suggest that agents need only a set of reactive responses to circumstances, and that intelligent behavior will emerge from the combination of such responses.

This kind of architecture is based on *reactive agents*, that is, agents that include neither a symbolic world model nor the ability to perform complex symbolic reasoning (Wooldridge and Jennings 1995). A well-known example of this approach is Brooks' *subsumption architecture* (Brooks 1991), containing behavior modules that link actions to observed situations without any reasoning at all. The behaviors are arranged into a *subsumption hierarchy*, where low-level behavior such as "avoid object" has precedence over higher-level goal-oriented behaviors such as "move across room" (Figure 4.1). This simple and practical approach can be highly effective. Its chief drawback is that the emphasis placed on the local environment can lead to a lack of awareness of the bigger picture.

4.3.3 Knowledge-Level Architectures

A third type of architecture, based on *knowledge-level* agents, treats each intelligent agent as a knowledge-based system in microcosm. Just as we saw in Chapter 1 that a knowledge-based system contains a knowledge base that is distinct and separate

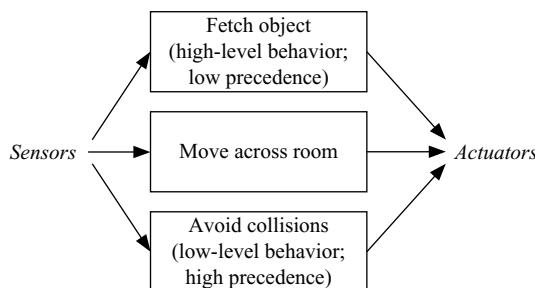


Figure 4.1 Brookes' subsumption architecture.

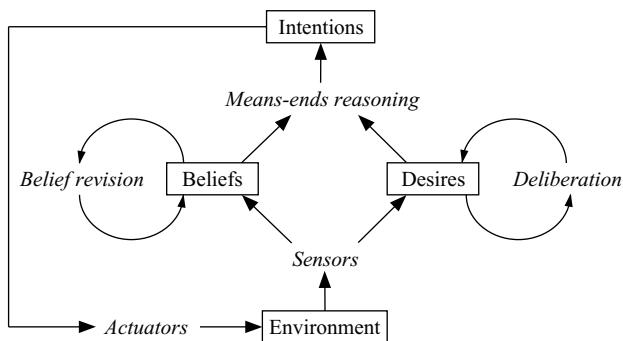


Figure 4.2 BDI architecture.

from its inference engine, so this type of agent is modeled with a *knowledge level*, distinct from its underlying mechanisms. Such agents are said to be *deliberative*. They are the antithesis of reactive agents, since they explicitly represent a symbolic model of the world and make decisions via logical reasoning based on pattern matching and symbolic manipulation (Wooldridge and Jennings 1995). A deliberative agent's knowledge determines its behavior in accordance with Newell's *Principle of Rationality* (Newell 1982), which states that:

*if an agent has knowledge that one of its actions will lead to one of its goals,
then the agent will select that action.*

One of the most important manifestations of this approach is known as the *beliefs-desires-intentions* (BDI) architecture (Bratman, Israel, and Pollack 1988). Here, knowledge of the environment is held as “beliefs” and the overall goals are “desires.” Together, they shape the “intentions,” which are the selected options that the system commits itself toward achieving (Figure 4.2). The intentions stay in place only as long as they remain both consistent with the desires and achievable according to the beliefs. The process of determining what to do, that is, the desires or goals, is *deliberation* (Wooldridge 1999). The process of determining how to do it, that is, the plan or intentions, is *means-ends analysis*. In this architecture, the balance between reactivity and goal-directedness can be restated as one between reconsidering intentions frequently (as a cautious agent might) and infrequently (as a bold or cavalier agent might). Unsurprisingly, Kinny and Georgeff (1991) found that the cautious approach works best in a rapidly changing environment, and the bold approach works best in a slowly changing environment.

4.3.4 Layered Architectures

The final approach to the balancing of reactive and goal-directed behavior is to mix modules that adopt the two different stances. This is the basis of the

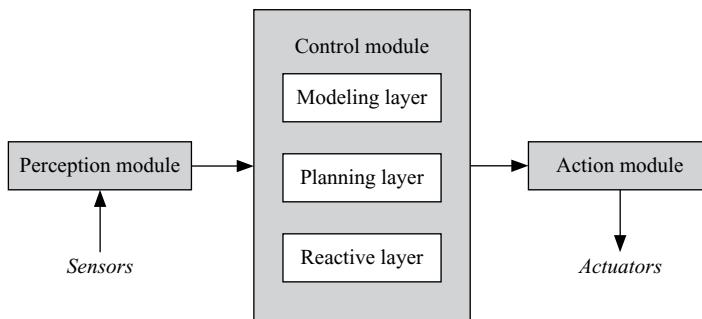


Figure 4.3 Touring Machine architecture.

layered architecture of *Touring Machines* (Ferguson 1995), so-called because their original application was as autonomous drivers for vehicles negotiating crowded streets. In fact, these agents contain three specific layers: a reactive layer, a planning layer for goal-directed behavior, and a modeling layer for modeling the environment (Figure 4.3). The problem of balancing the layers remains; in this case an intelligent control subsystem ensures that each layer has an appropriate share of power.

4.4 Multiagent Systems (MASs)

We have looked at the addition of intelligence to an individual agent, but the possibilities become especially exciting when we consider teams of intelligent agents interacting and working together. In Chapter 1, it was stated that the study of artificial intelligence has been inspired by attempts to mimic the behavior of an individual human mind. By comparison, an MAS gives us the possibility of mimicking human teams, organizations, and societies that contain a collection of individuals with their own personalities. MASs are sometimes called *agent-oriented* or *agent-based systems*. They are part of a broader field of *distributed artificial intelligence* (DAI), in which intelligent behavior is shared among parallel processes. The agents in an MAS may run concurrently on a single computer, or they may be shared among multiple machines distributed across the Internet.

An MAS can be defined as a system in which several interacting, intelligent agents pursue a set of individually held goals or perform a set of individual tasks. Though this definition is a useful starting point, it raises a number of key questions that will be addressed in the remainder of this chapter, notably:

- What are the benefits?
- How do intelligent agents interact?
- How do they pursue goals and perform tasks?

4.4.1 Benefits of a Multiagent System

There are two broad classes of problems for which MASs offer the only practicable approach:

- *Inherently complex problems*: Such problems are simply too large to be solved by a single hardware or software system. As the agents are provided with the intelligence to handle a variety of circumstances, there is some uncertainty as to exactly how an agent system will perform in a specific situation. Nevertheless, well-designed agents will ensure that every circumstance is handled in an appropriate manner, even though it may not have been explicitly anticipated.
- *Inherently distributed problems*: Here, the data and information may exist in different physical locations, or at different times, or may be clustered into groups requiring different processing methods or semantics. These types of problems require a distributed solution, which can be provided by agents that run in parallel on distributed processors. They may, alternatively, run concurrently as independent processes on a single processor.

Furthermore, an MAS offers the following general benefits:

- A more *natural view of intelligence*.
- *Speed and efficiency gains*, brought about because agents can function concurrently and communicate asynchronously.
- *Robustness and reliability*: No agent is vital, provided there are others that can take over its role in the event of its failure. Thus, the performance of an MAS will degrade gracefully if individual agents fail.
- *Scalability*: DAI systems can generally be scaled-up simply by adding components. In the case of an MAS, additional agents can be added without adversely affecting those already present.
- *Granularity*: Agents can be designed to operate at an appropriate level of detail. Many “fine-grained” agents may be required to work on the minutiae of a problem, where each agent deals with a separate detail. At the same time, a few “coarse-grained,” more sophisticated agents can concentrate on higher-level strategy.
- *Ease of development*: As with OOP, encapsulation enables individual agents to be developed separately and to be reused wherever applicable.
- *Cost*: A system comprising many small processing agents is likely to be cheaper than a large centralized system.

As a result of these benefits, Jennings has argued that MASs are not only suited to the design and construction of complex, distributed software systems, but that they are also appropriate as a mainstream software engineering paradigm (Jennings 2000).

4.4.2 Building a Multiagent System

An MAS is dependent on interactions between intelligent agents. There are, therefore, some key design decisions to be made, e.g., when, how, and with whom should agents interact? In *cooperative* models, several agents try to combine their efforts to accomplish as a group what the individuals cannot. In *competitive* models, each agent tries to get what only some of them can have. In either type of model, agents are generally assumed to be honest.

In order to achieve coherency, MASs can be designed bottom-up or top-down. In a bottom-up approach, agents are endowed with sufficient capabilities, including communication protocols, to enable them to interact effectively. The overall system performance then emerges from these interactions. In a top-down approach, conventions—sometimes called *societal norms*—are applied at the group level in order to define how agents should interact. An example might be the principle of democracy, achieved by giving agents the right to vote. If we view an agent as having a knowledge level abstracted above its inner mechanisms, then these conventions can be seen as residing at a still higher level of abstraction, namely the *social level* (Figure 4.4).

In any MAS there may be large numbers of different agents capable of providing different services, possibly located on separate computers. A *directory facilitator* (DF) agent helps to identify which services an agent can offer and where it is located. The DF agent is therefore analogous to a *Yellow Pages*© service. It does not offer services directly related to the problem in hand but facilitates distributed problem-solving in an MAS. An example of a system that uses this approach is EMADS (Extendable Multi-Agent Data-mining System), which provides an agent-enriched toolkit for data mining (Albashiri et al. 2009), implemented using the JADE (Java Agent Development) framework (Bellifemine et al. 1999).

An agent must actively register with the DF agent in order for its name, network address, and services to be listed. Agents that need to solicit other agents to perform a task can ask the facilitator for a recommendation. The DF agent may act as a broker between the soliciting agent and the recommended service-supply agent, or negotiation may take place directly between them. In some systems, a different sort of facilitator is used, which, rather than maintaining a directory of service-providing agents, broadcasts an advertisement or invitation to tender for tasks as they arise.

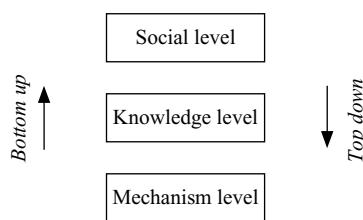


Figure 4.4 Agent levels of abstraction.

MASs are often designed as computer models of human functional roles. For example, in a hierarchical control structure, one agent is the superior of other subordinate agents. Peer group relations, such as may exist in a team-based organization, are also possible. The remainder of this section will address three models for managing agent interaction, known as *contract nets* (Smith and Davis 1981), *cooperative problem-solving* (CPS) (Wooldridge and Jennings 1994a, 1994b) and *shifting matrix management* (SMM) (Li et al. 2003). After considering each of these models in turn and comparing their performance, the semantics of communication between agents will be addressed.

4.4.3 Contract Nets

Imagine that you have decided to build your own house. You are unlikely to undertake all the work yourself. You will probably employ specialists to draw up the architectural plans, obtain statutory planning permission, lay the foundations, build the walls, install the floors, build the roof, and connect the various utilities. Each of these specialists may in turn use a subcontractor for some aspect of the work. This arrangement is akin to the contract net framework (Smith and Davis 1981) for agent cooperation (Figure 4.5). Here, a manager agent generates tasks and is responsible for monitoring their execution. The manager enters into explicit agreements with contractor agents willing to execute the tasks. Individual agents are not designated

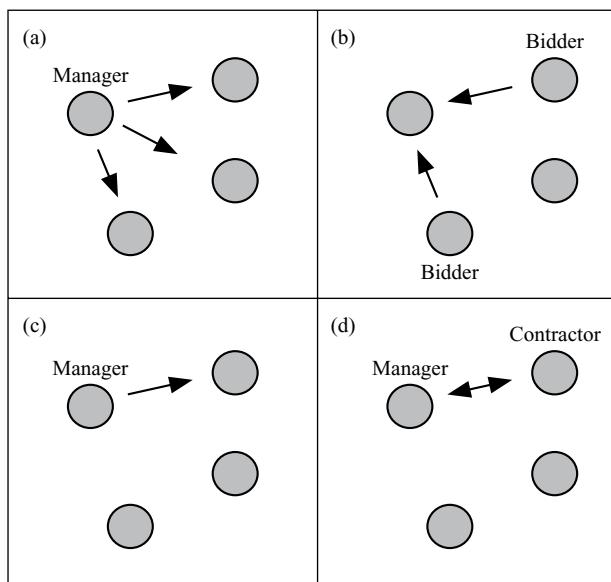


Figure 4.5 Contract nets: (a) Manager advertises a task; (b) potential contractors bid for the task; (c) manager awards the contract; (d) manager and contractor communicate privately.

a priori as manager or contractor. These are only their roles at a given time, and any agent can take on either role dynamically during problem-solving.

To establish a contract, the manager agent advertises the existence of the tasks to other agents. Agents that are potential contractors evaluate the task announcements and submit bids for those to which they are suited. The manager evaluates the bids and awards contracts for execution of the task to the agents it determines to be the most appropriate. The manager and contractor are thus linked by a contract and communicate privately while the contract is being executed. The managers supply mostly task information, and the contractor reports progress and the eventual result of the task. The negotiation process may recur if a contractor subdivides its task and awards contracts to other agents, for which it is the manager.

4.4.4 Cooperative Problem-Solving (CPS)

The CPS framework is a top-down model for agent cooperation. As in the BDI model, an agent's intentions play a key role. They determine the agent's personal behavior at any instant, while joint intentions control its social behavior (Bratman 1987). An agent's intentions are shaped by its *commitment*, and its joint intentions by its social *convention*. The framework comprises the following four stages, also shown in Figure 4.6, after which the team disbands and Stage 1 begins again:

Stage 1: Recognition. Some agents recognize the potential for cooperation with an agent that is seeking assistance, possibly because it has a goal that it cannot achieve in isolation.

Stage 2: Team formation. An agent that recognized the potential for cooperative action at Stage 1 solicits further assistance. If successful, this stage ends with a group having a joint commitment to collective action.

Stage 3: Plan formation. The agents attempt to negotiate a joint plan that they believe will achieve the desired goal.

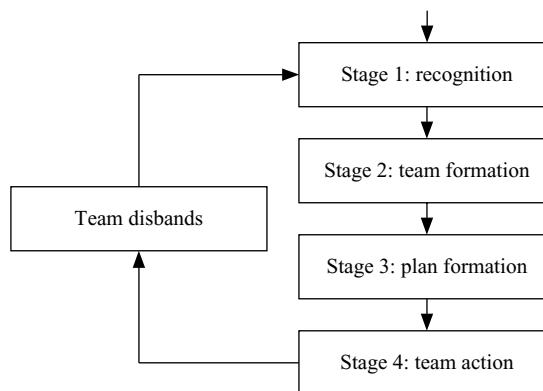


Figure 4.6 CPS framework.

Stage 4: Team action. The newly agreed plan of joint action is executed. By adhering to an agreed social convention, the agents maintain a close-knit relationship throughout.

4.4.5 Shifting Matrix Management (SMM)

SMM (Li et al. 2003) is a model of agent coordination that has been inspired by Mintzberg's SMM model of organizational structures (Mintzberg 1979), as illustrated in Figure 4.7. Unlike the traditional management hierarchy, matrix management allows multiple lines of authority, reflecting the multiple functions expected of a flexible workforce. *Shifting* matrix management takes this idea a stage further by regarding the lines of authority as temporary, typically changing as different projects start and finish. For example, in Figure 4.7, individual #1 is the coordinator for project A and the designer for project B. The agents are distinguished by their different motives, functionality, and knowledge. These differences define the agents' variety of mental states with respect to goals, beliefs, and intentions. In order to apply these ideas to agent cooperation, a cyclical six-stage framework has been devised (Figure 4.8) and outlined in the following text:

Stage 1: Goal selection. Agents select the tasks they want to perform, based on their initial mental states.

Stage 2: Action selection. Agents select a way to achieve their goals. In particular, an agent that recognizes that its intended goal is common to other agents would have to decide whether to pursue the goal in isolation or in collaboration with other agents.

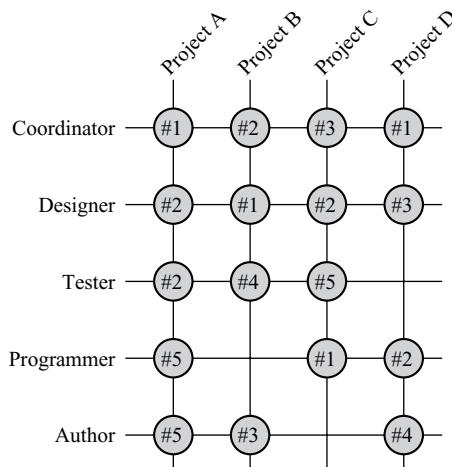


Figure 4.7 Shifting Matrix Management (SMM): the nodes represent people.

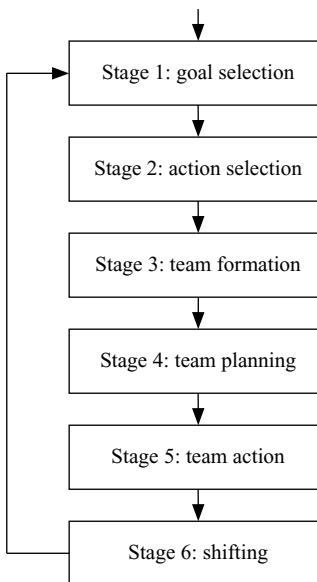


Figure 4.8 SMM multiagent framework.

Stage 3: Team formation. Agents that are seeking cooperation attempt to organize themselves into a team. The establishment of a team requires an agreed code of conduct, a basis for sharing resources, and a common measure of performance.

Stage 4: Team planning. The workload is distributed among team members.

Stage 5: Team action. The team plan is executed by the members under the team's code of conduct.

Stage 6: Shifting. The last stage of the cooperation process, which marks the disbanding of the team, involves shifting agents' goals, positions, and roles. Each agent updates its probability of team-working with other agents, depending on whether or not the completed team-working experience with that agent was successful. This updated knowledge is important, as iteration through the six stages takes place until all the tasks are accomplished.

4.4.6 Comparison of Cooperative Models

Li et al. (2003) have implemented the three cooperative models described in the preceding text using a blackboard architecture (see Chapter 10). In a series of tests, the three different models of agent cooperation were used to control two robots that were required to work together to complete a variety of tasks. A total of 50 different tasks were generated, each of which was presented twice, creating a set of 100 tasks that were presented in random order. Figure 4.9 shows the number of tasks completed for the three models as a function of time. The data are an average over

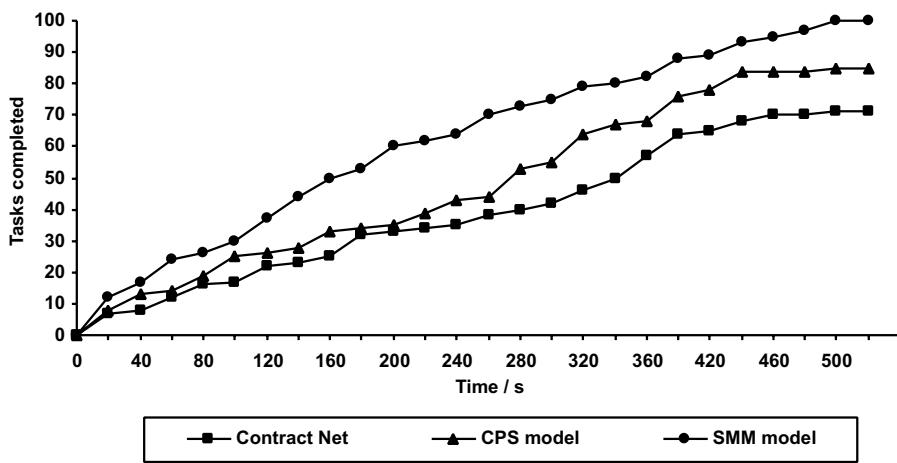


Figure 4.9 Task completion, averaged over four test runs. (From Li et al. 2003. *Engineering Applications of Artificial Intelligence* 16:191–201. Reprinted from Hopgood, A. A. 2005. The state of artificial intelligence. In *Advances in Computers* 65, edited by M. V. Zelkowitz. Elsevier, Oxford, UK, 1–75. Copyright 2005, Elsevier. With permission.)

four rounds of tests, where the tasks were presented in a different random order for each round. For these tasks and conditions, the SMM model achieves an average task completion time of 5.0 s, compared with 6.03 s for the CPS model and 7.04 s for the Contract Net model. Not only is the task completion rate greatest for SMM, so too is the total number of completed tasks. All 100 tasks were completed under the SMM model, compared with 71 for Contract Nets and 84 for the CPS model.

As the SMM model requires predominantly deliberative behaviors from the agents rather than simple reactive behaviors, it might be argued that it is inappropriate in a time-critical environment. However, in the domain of cooperation between multiple robots, the agents' computation is much faster than the robots' physical movements. Thus, Li et al. (2003) propose that the SMM model is appropriate because it emphasizes the avoidance of mistakes by reasoning before acting. If a mistake is made, actions are stopped sooner rather than later.

4.4.7 Communication Between Agents

So far, we have seen how intelligent agents can be designed and implemented, and the ways in which they can cooperate and negotiate in a society. In this section, we will examine how agents communicate with each other, both synchronously and asynchronously, in an MAS. Synchronous communication is rather like a conversation; after sending a message, the sending agent awaits a reply from the recipient. Asynchronous communication is more akin to sending an email or a letter; although

you might expect a reply at some future time, you do not expect the recipient to read or act upon the message immediately.

Agents may be implemented by different people at different times on different computers, yet still be expected to communicate with each other. Consequently, there has been a drive to standardize the structure of messages between agents, regardless of the domain in which they are operating.

A generally accepted premise is that the *form* of the message should be understandable by all agents regardless of their domain, even if they do not understand its content. Thus, the structure needs to be standardized in such a way that the domain-specific content is self-contained within it. Only specialist agents need to understand the content, but all agents need to be able to understand the form of the message. Structures for achieving this are called *agent communication languages* (ACLs), which include *Knowledge Query and Manipulation Language* (KQML) (Finin et al. 1997) and the FIPA-ACL from the *Foundation for Intelligent Physical Agents* (Poslad and Charlton 2001).

Both ACLs are designed to be as general as possible, and they are therefore defined without reference to the content of the body of the messages they transport. However, each KQML or FIPA-ACL message does say something about the meaning of the message. For instance, it may be a question, or it may be a statement of fact. In each case, the message structure contains at least the following components:

- A *performative*. This is a single word that describes the purpose of the message, e.g., achieve, advertise, ask-all, ask-one, broadcast, cancel, deny, evaluate, forward, insert, recommend, recruit, register, reply, tell.
- The identity of the *sender* agent.
- The identity of the *receiver* agent.
- The *language* used in the content of the message. Although the overall form of the message is defined, KQML allows any programming language to be used for the domain-specific content. In the case of FIPA-ACL, the use of SL (Semantic Language) is recommended.
- The *ontology*, or vocabulary, of the message. This component provides the context within which the message content is to be interpreted. For example, the problem of selecting a polymer to meet an engineering design requirement is used in Chapter 11 to demonstrate the Lisp, Prolog, and Python languages. At the programming language level, a program to tackle this problem is merely a collection of words and symbols organized as statements. It would, for instance, remain syntactically correct if each polymer name were replaced by the name of a separate type of fruit. The statements only become meaningful once they are interpreted in the vocabulary of engineering polymers.
- The message *content*.

In the polymer selection world mentioned in the preceding text, *agent1* might wish to tell *agent2* about the properties of polystyrene, encoded in Prolog. Using

KQML, it could do so with the following message. The FIPA-ACL equivalent would be similar, but SL would be recommended over Prolog.

```
(tell
  :sender    agent1
  :receiver   agent2
  :language   prolog
  :ontology   polymer-world
  :content
    "materials_database(polystyrene, thermoplastic,
      [[impact_resistance, 0.02],
       [flexural_modulus, 3.0],
       [maximum_temperature, 50]]).")
```

4.5 Swarm Intelligence

Brooks' subsumption architecture (Brooks 1991) has already been mentioned as a model that does not attempt to build intelligence into individual agents, yet intelligent behavior emerges from the combined behavior of several agents. This is the principle behind the broad area of swarm intelligence (Dorigo 2001; Dorigo and Stützle 2004). It contrasts starkly with cooperative multiagent models such as SMM, in which agents deliberate at length about their individual contributions to the overall task.

Much of the work on swarm intelligence has been inspired by the behavior of ant colonies, although other analogies exist such as worker bees, birds, spiders, and bacteria. One commonly-cited application is the traveling salesperson problem, in which the shortest route between sites is sought. The problem is identical in principle to that encountered in optimizing the layout of printed circuit boards or routing traffic on a telecommunications network. Ants tackle this problem by leaving a trail of pheromone as they travel, which encourages other ants to follow. Ants that have found the shortest route to a food source are the first to reinforce their route with pheromone as they return to the nest. The shortest route then becomes the preferred one for other members of the colony. This behavior is easily mimicked with simple agents, and the modeled pheromone can be given a degree of volatility, allowing the pheromone to evaporate. Hence, the swarm can change behavior as the environment changes (Dorigo 2001).

Other behaviors have also been successfully modeled using swarm intelligence. For instance, data-mining of bank records has been achieved by clustering customers with similar characteristics using swarm intelligence that mimics ant colonies' ability to cluster corpses and thereby clean their nest (Lumer and Faieta 1994). The ability to perceive a visual image has been modeled by simulating colonies of ants that swarm across a printed image. By reinforcing paths across the image, the photographic image can be transformed into an outline map (Ramos and Almeida 2000).

4.6 Object-Oriented Systems

4.6.1 Introducing Object-Oriented Programming (OOP)

As well as the practical advantages of maintainability, adaptability, and reusability, OOP is also a natural way of representing many real-world problems within the confines of a computer. Programs of interest to engineers and scientists normally perform calculations or make decisions about physical entities. Such programs must contain a model of those entities, and OOP is a convenient way to carry out this modeling. Every entity can be represented by a self-contained “object,” where an object contains data and the code that can be performed on those data. Thus, a system for simulating ultrasonic imaging might include an ultrasonic pulse that is reflected between features in a steel component. The pulse, the features, and the component itself can all be represented by objects. Systems theory leads directly to such an approach, given the following definition of a system (Skyttner 2005):

A system is a set of interacting units or elements that form an integrated whole intended to perform some function.

Thus, OOP can be thought of as a computer implementation of systems theory, where each interacting unit in a computer system can be represented by an object and each object is a subsystem.

Modeling with objects is not restricted to physical things but extends to include entities within the programming environment such as windows, icons, menus, clocks, and just about anything else that you can think of. In fact, the development of graphical user interfaces has been one of the main driving forces for OOP. This observation demonstrates that OOP is a powerful technique in its own right and is by no means restricted to knowledge-based systems. However, OOP does have a critical role in many knowledge-based systems, as it offers a way of representing the things that are being reasoned about, their properties and behaviors, and the relationships among them.

There are many languages and programming environments that offer object-orientation, some of which are supplied as extensions to existing languages. Four popular OOP languages are C++, Java, Smalltalk, and CLOS (Common Lisp Object System). The principles will be illustrated in this chapter using the C++ language, but they readily transfer to the other languages.

According to Pascoe (1986), an OOP language offers at least the following facilities:

- Data abstraction;
- Inheritance;
- Encapsulation (or information-hiding); and
- Dynamic (late) binding.

The following subsections describe these facilities and the advantages that they confer in the context of an ultrasonics model. As will be discussed in Section 11.5, Python is effectively an OOP language although it does not strictly meet the requirements because encapsulation is not enforced.

4.6.2 An Illustrative Example

We will illustrate the features of OOP by considering a small simulation of ultrasonic imaging. The physical arrangement to be simulated is shown in Figure 4.10. It comprises an ultrasonic probe, containing a detector and a transmitter, which sits on the surface of a component under test. The probe emits a single pulse of ultrasound, that is, sound at a higher frequency than the human audible range. The pulse is strongly reflected by the front and back walls of the component and is weakly reflected by defects within the component. The intensity and time of arrival of reflected pulses reaching the detector are plotted on a display screen. There are two possible types of ultrasonic pulses—longitudinal and shear—that travel differently through a solid. The atoms of the material travel to-and-fro, parallel to the direction of a longitudinal pulse, but they move perpendicular to the direction of a shear pulse (Figure 4.11). The two types of pulses travel at different speeds in a given material and behave differently at obstructions or edges. The emitted pulse is always longitudinal, but reflections may generate both a longitudinal and a shear pulse (Figure 4.12).

This system, like so many others, lends itself to modeling with objects. Each of the components of the system can be treated as an independent object, containing data and code that define its own particular behavior. Thus, there will be objects representing each pulse, the transmitter, the detector, the component, the front wall,

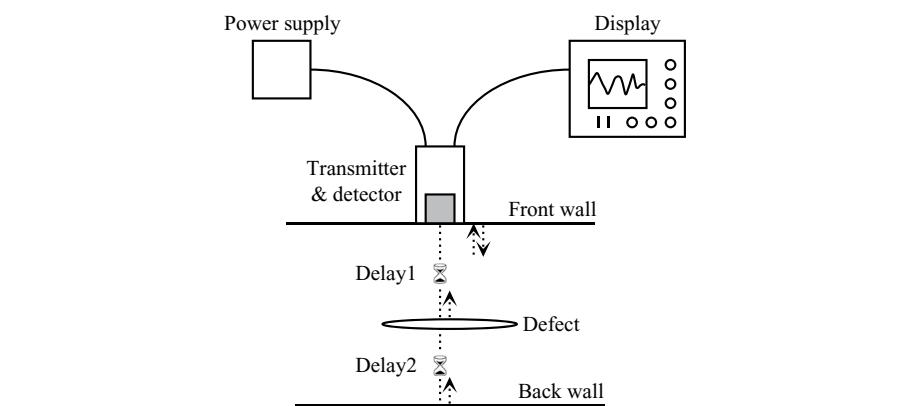


Figure 4.10 Detection of defects using ultrasonics; delays simulate the depth of the defect and the back wall.

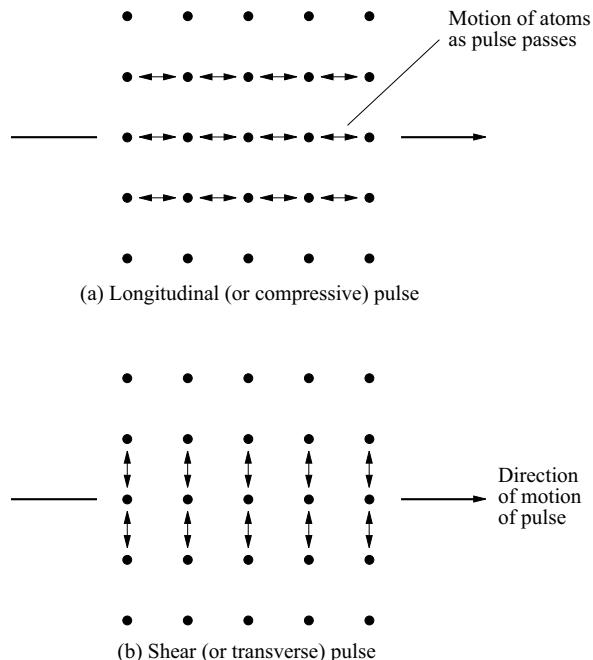


Figure 4.11 Longitudinal and shear pulses.

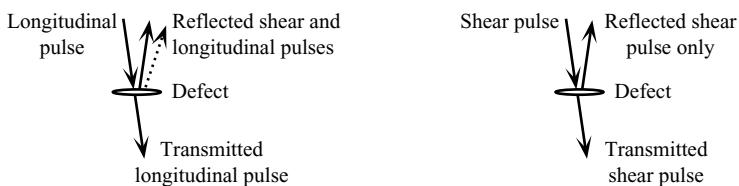


Figure 4.12 Reflection of longitudinal and shear pulses at a defect.

the back wall, the defects, and the oscilloscope. These objects stand in well-defined relationships with one another, e.g., a transmitter object can generate a pulse object.

4.6.3 Data Abstraction

4.6.3.1 Classes

The ultrasonic example involves several different types of objects. Shear pulses are objects of the same type as each other, but of a different type from the detector. Data abstraction allows us to define these types and the functions that go with them. These object types are called *classes*, and they form templates for the objects themselves. Objects that represent the same idea or concept are grouped together as

a class. The words *type* and *class* are generally equivalent and are used interchangeably here.

Most computer languages include some simple built-in data types, such as integer or real. An object-oriented language allows us to define additional data types (i.e., classes) that are treated identically, or nearly identically, to the built-in types. The user-defined classes are sometimes called *abstract data types*. These may be classes that have a specific role within a particular domain, such as `Shear_pulse`, `Circle`, or `Polymer`, or they may describe a specialized version of a standard data type, such as `Big_integer`. Here, the convention of using capitalization to denote class names has been adopted. This convention is distinct from the use of capitalization to denote local variables in the Prolog language and artificial intelligence toolkits such as Flex.

The definition of a class contains the class name, its attributes (see below), its operations (see below) and its relationships to other classes (Sections 4.6.4 and 4.6.5). C++ and some other OOP languages require that the attribute types and the visibility from other classes also be specified.

4.6.3.2 Instances

It was stated above that classes form templates for “the objects themselves,” meaning object *instances*. Once a class has been defined, instances of the class can be created that have the properties defined for the class. For example, `pulse#259` might be an ultrasonic pulse whose location at a given moment is ($x = 112$ mm, $y = 294$ mm, $z = 3.5$ mm). We would represent `pulse#259` as an instance of the class `Longitudinal_pulse`. As a more tangible example, my car is an instance of the class of car. The class specifies the characteristics and behavior of its instances. A class can, therefore, be thought of as the blueprint from which instances are built. The terms *object* and *instance* are often used interchangeably, although it is helpful to use the latter to stress the distinction from a class.

Classes and instances are just a generalization of the concepts of data types and variables that exist in other programming languages. For instance, most languages include a built-in definition of the type integer. However, we can only draw upon the properties of an integer by creating an instance, i.e., an integer variable. In C, the code would look like this:

```
int x; /* create an instance, x, of type int */
x = 3; /* manipulate x */
```

Similarly, once we have defined a class in an object-oriented language, its properties can only be used by creating one or more instances. Consider the class `Longitudinal_pulse`. One instance of this class is generated by the transmitter object (which is itself an instance of the class `Transmitter`). This instance represents one specific pulse that has position, amplitude, phase, speed, and direction. When this pulse interacts with a defect (another instance), a new instance of `Longitudinal_pulse` must be created,

since there will be both a transmitted and a reflected pulse (see Figure 4.12). The new pulse will have the same attributes as the old one because they are both derived from the same class, but some of these attributes will have different values associated with them (e.g., the amplitude and direction will be different).

4.6.3.3 Attributes (or Data Members)

A class definition includes its attributes and operations. The attributes are particular quantities that describe instances of that class. They are sometimes described as *slots* into which values are inserted. Thus, the class `Shear_pulse` might have attributes such as amplitude, position, speed, and direction. Only the names and, depending on the language, the types of the attributes need to be declared within the class, although values can also be optionally supplied. The class acts as a template, so that when a new shear pulse is created, it will contain the names of these attributes. The attribute values can be supplied or calculated for each instance, or a value provided within the class can be used as a default. The attributes can be of any type, including abstract data types, i.e., classes. Some languages, such as C++, require that the class definition defines the attribute types in advance. Amplitude and speed might be of type `float`, whereas position and direction would be of type `vector`.

If a value for an attribute is supplied in the class definition, any new instances would carry those values by default unless specifically overwritten. Most OOP languages distinguish between *class attributes* (or class variables) and *instance attributes* (or instance variables). The value of a class attribute remains the same for all instances of that class. In contrast, instances contain their own copies of each instance attribute. If a default has been specified for an instance attribute, each instance of a given class has the same initial value for that instance attribute. However, the values of the instance attribute may subsequently vary from one instance to another. Thus, in the above example, speed might be a class attribute for `Shear_pulse`, since the speed will be the same for all shear pulses in a given material. On the other hand, `amplitude` and `position` are properties of each individual pulse and are represented as instance attributes.

In C++, attributes are called *data members* and are conventionally given the prefix `m_`, so that `amplitude`, for example, would become `m_amplitude`. All data members are assumed to be instance attributes unless they are declared static. Static data members are stored at the same memory location for every instance of a class, so there is only one copy, regardless of how many instances there might be. Static data members are, therefore, equivalent to class attributes.

4.6.3.4 Operations (or Methods or Member Functions)

Each object has a set of operations or functions that it can perform. For example, a `Shear_pulse` object may contain the function `move`. This function would take as

its parameters the amount and direction of movement, and it would return a new value for its position attribute. In some OOP languages, operations belonging to objects are called *methods*. In C++, they are called *member functions*. Operations are defined for a class and can be used by all instances of that class. It may also be possible for instances of other classes to access these operations (see Sections 4.6.4 and 4.6.5).

4.6.3.5 Creation and Deletion of Instances

Creating a new instance requires the computer to set aside some of its memory for storing the instance. Given a class definition, `Myclass`, we might write the following code in C++ to create a new instance:

```
Myclass* myinstance;
    // declare a pointer to objects of type Myclass

myinstance = new Myclass;
    // pointer now points to a new instance
```

Here, the `//` symbol indicates a comment. The code can be abbreviated to:

```
Myclass* myinstance = new Myclass;
    // new pointer points to a new instance
```

Large numbers of instances may be created while running a typical object-oriented system. An important consideration, therefore, is the release of memory that is no longer required. In the example of the ultrasonic simulation, new pulse instances are generated through reflections, but they must be removed when they reach the detector. The memory that is occupied by these unwanted instances must be released again if we are to avoid building a system with an insatiable appetite for computer memory. Some OOP environments like Smalltalk reclaim unused memory automatically through a process known as *garbage collection*, which may cause the program to momentarily pause while the system carries out its memory management. In C++, the responsibility for memory management rests with the programmer. Objects that are created as described above must be explicitly destroyed when they are no longer needed:

```
delete myinstance;
```

This operation releases the corresponding memory. C++ also allows an alternative method of object creation and deletion that relies on the *scope* of objects. A new instance, `myinstance`, may be created within a block of code in the following way:

```
Myclass myinstance;
```

The instance exists only within that particular block of code, which is its scope. When the flow of execution enters the block of code, the object is automatically created. Similarly, the object is deleted as soon as the flow of execution leaves the block.

C++ allows the programmer to define special functions to be performed when a new instance is created or deleted. These are known, respectively, as the *constructor* and the *destructor*. The constructor is a member function whose name is identical to the name of the class, and it is typically used to set the initial values of some attributes. The destructor is defined in a similar way to the constructor, and its name is that of the class preceded by a tilde (~). It is mostly used to release memory when an instance is deleted. As an example, consider the definition for `Sonic_pulse` in C++:

```
// class definition:
class Sonic_pulse
{
protected:
    float amplitude;
public:
    Sonic_pulse(float initial_amplitude); // constructor
    ~Sonic_pulse(); // destructor
};
// constructor definition:
Sonic_pulse::Sonic_pulse(float initial_amplitude)
{
    amplitude=initial_amplitude; // set up initial value
}
// destructor definition:
Sonic_pulse::~Sonic_pulse()
{
    // perform any tidying up that may be necessary before
    // deleting the object
}
```

The class has a single attribute, `amplitude`, that is set to an initial value by the constructor. The constructor is automatically called immediately after a new instance of `Sonic_pulse` is created. To create a new sonic pulse whose initial amplitude is 131.4 units, we would write in C++ either:

```
Sonic_pulse* myinstance = new Sonic_pulse(131.4);
//technique 1
```

or:

```
Sonic_pulse myinstance(131.4); //technique 2
```

If a destructor has been defined for a class, it is automatically called whenever an instance is deleted. If the instance was created by technique 1, it must be explicitly deleted by the programmer. If it was created by technique 2, it is automatically deleted when it becomes out of scope from the program's thread of control.

4.6.4 Inheritance

4.6.4.1 Single Inheritance

Returning again to our example of the ultrasonic simulation, there are two classes of sonic pulses, namely `Longitudinal_pulse` and `Shear_pulse`. While there are some differences between the two classes, there are also many similarities. It would be most unwieldy if all of this common information had to be specified twice. This problem can be avoided by the use of *inheritance*, sometimes called *derivation*. A class `Sonic_pulse` is defined that encompasses both types of pulses. All of the attributes and methods that are common can be defined here. The classes `Longitudinal_pulse` and `Shear_pulse` are then designated as subclasses or specializations of `Sonic_pulse`. Conversely, `Sonic_pulse` is said to be the superclass or generalization of the other two classes. The sub/super class relationship can be thought of as *is-a-kind-of*, that is, a shear pulse is-a-kind-of sonic pulse. The following expressions, commonly used to describe the is-a-kind-of relationship, are equivalent:

- a *subclass* is-a-kind-of *superclass*;
- an *offspring* is-a-kind-of *parent*;
- a *derived* class is-a-kind-of *base* class;
- a *specialized* class is-a-kind-of *generalized* class; and
- a specialized class *inherits from* a generalized class.

In defining a derived class, it is necessary to specify only the name of the base class and the *differences* from the base class. Attributes and their initial values (where supplied) and methods are inherited by the derived class. (In C++ there are a few exceptions that are not inherited, including the constructor and destructor.) Any attributes or methods that are redefined in the derived class are said to be *specialized*, just as the derived class is said to be a specialization of the base class. Another form of specialization is the introduction of extra attributes and methods in the derived class. C++ gives the user some control over which methods and attributes of the base class are accessible from the derived class and which are not (Section 4.6.5). Many other OOP languages assume that *all* attributes and methods of the base class are accessible from the derived class, apart from those that are explicitly specialized.

Figure 4.13 shows the class definitions for `Sonic_pulse`, its superclass `Signal`, and its subclasses `Shear_pulse` and `Longitudinal_pulse`. The attributes `graphic`, `speed`, `phase`, and `amplitude` are all declared at the highest-level class and inherited by the other classes. The class variable `graphic`, which determines the screen representation of an instance, is assigned a value at the `Sonic_pulse` level. This value, as well as the declaration of the attribute, is inherited by the subclasses. Values for the instance variables `amplitude` and `phase` are inherited in the same way. Inherited values may be overridden either when an instance is created or subsequently. The class variable `speed` for `Shear_pulse` is assigned a different value from the one that would otherwise be inherited.

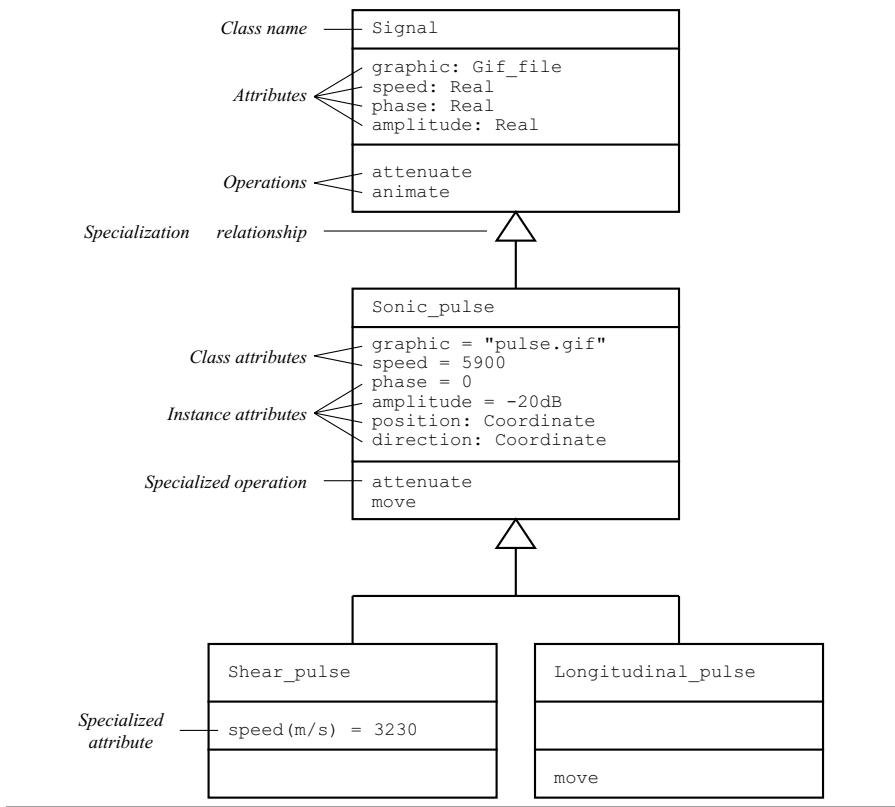


Figure 4.13 An example of inheritance.

Operations as well as attributes are inherited. Therefore, instances of **Shear_pulse** and **Longitudinal_pulse** have access to the operations `animate` and `attenuate`. The former is inherited across two generations, while the latter is specialized at the **Sonic_pulse** class level. The operation `move` is defined at the **Sonic_pulse** class level and inherited by **Shear_pulse**, but redefined for **Longitudinal_pulse**.

Figure 4.14 shows the inheritance between other objects in the ultrasonic simulation. Note that the is-a-kind-of relationship is *transitive*, i.e.:

if x is-a-kind-of y and y is-a-kind-of z , then x is-a-kind-of z .

Therefore, the class **Defect**, for example, inherits information that is defined at the **Feature** level and at the **General_object** level.

4.6.4.2 Multiple and Repeated Inheritance

In the example shown in Figure 4.14, each offspring has only one parent. The specialization relationships therefore form a hierarchy. Some OOP languages insist

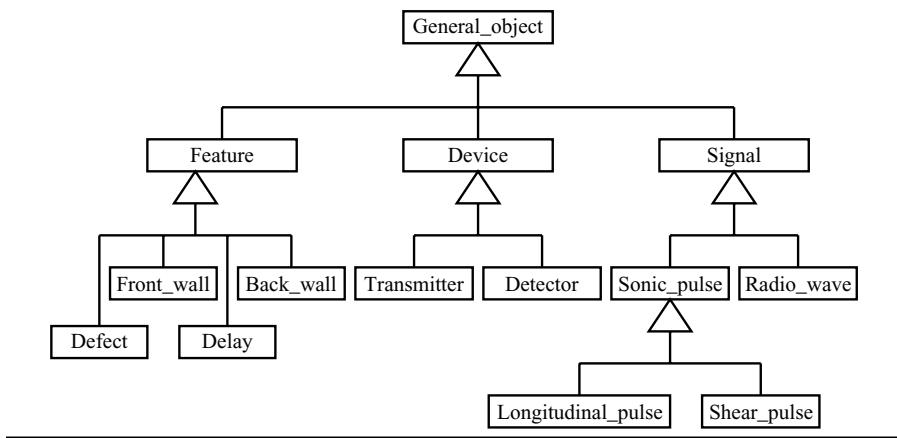


Figure 4.14 Single inheritance defines a class hierarchy.

upon hierarchical inheritance. However, others allow an offspring to inherit from more than one parent. This is known as *multiple inheritance*. Where multiple inheritance occurs, the specialization relationships between classes define a network rather than a hierarchy.

Figure 4.15 shows how multiple inheritance might be applied to the ultrasonic simulation. The parts of the component that interact with sonic pulses all inherit from the class Feature. Three of the classes derived from Feature are required to simulate partial reflection of pulses. This requirement is met by generating one or more reflected pulses, while the existing pulse is propagated in the forward direction with diminished amplitude. Code for generating pulses is contained within the class definition for Transmitter. Multiple inheritance allows those classes that need to generate pulses (Front_wall, Back_wall, and Defect) to inherit this capability from Transmitter, while inheriting other functions and attributes from Feature.

While multiple inheritance is useful, it can cause ambiguities. An example is shown in Figure 4.16, where Defect inherits the graphic feature.gif from Feature and at the same time inherits the graphic transmitter.gif from Transmitter. This situation raises two questions. Do the two attributes with the same name refer to the same attribute? If so, the class Defect can have only one value corresponding to the attribute graphic, so which value should be selected? Similarly, Defect inherits conflicting definitions for the operation send_pulse.

The most reliable way to resolve such conflicts is to have them detected by the language compiler, so that the programmer can then state explicitly the intended meaning. Some OOP environments allow the user to set a default strategy for resolving conflicts. An example might be to give preference to the parent class that is either closest to, or furthest from, the root of the inheritance tree. This approach would not help with the example in Figure 4.16 as the two parents are equidistant

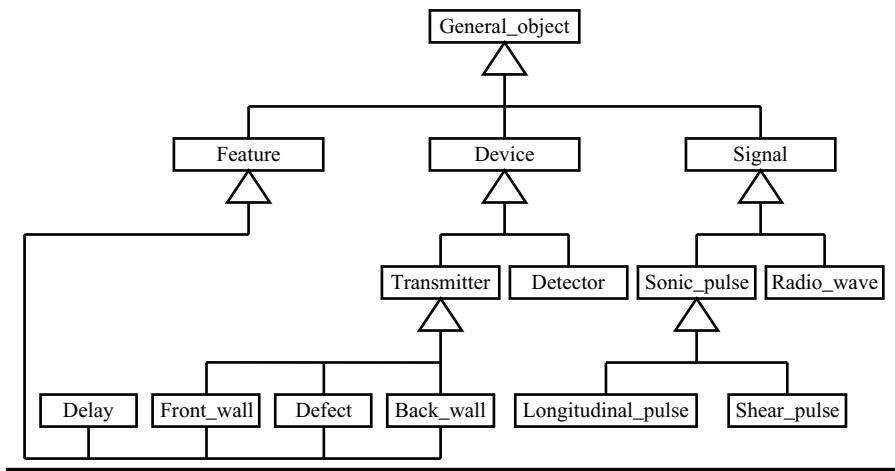


Figure 4.15 Multiple inheritance defines a network of classes.

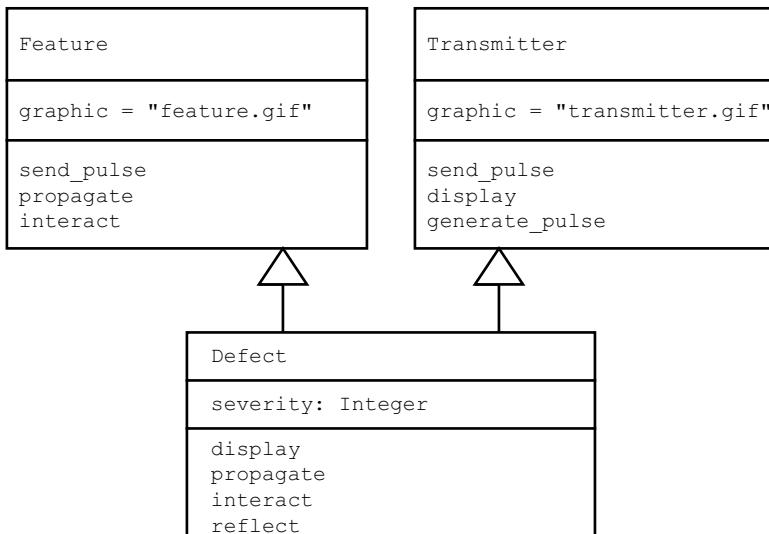


Figure 4.16 Conflicts arising from multiple inheritance: each parent has a different value for `graphic` and a different definition for `send_pulse`.

from the root. Alternatively, the class names may be deemed significant in some way, or preference may be given to the most recently defined specialization.

A further problem is that one class may find itself indirectly inheriting from another via more than one route, as shown in Figure 4.17. This is known as *repeated inheritance*. Although the meaning may be clear to the programmer, an OOP language must have some strategy for recognizing and dealing with repeated inheritance,

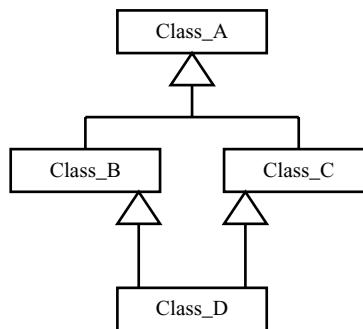


Figure 4.17 Repeated inheritance.

if it allows it at all. C++ offers the programmer a choice of two strategies. Class D can have two copies of Class A, one for each inheritance route. Alternatively, it can have just a single copy, if both Class B and Class C are declared to have Class A as a *virtual* base class.

Multiple inheritance gives rise to the idea of *mixins*, which are classes designed solely to help organize the inheritance structure. Instances of mixins cannot be created. Consider, for example, a class hierarchy for engineering materials. The materials polyethylene, Bakelite, gold, steel, and silicon nitride can be classified as polymers, metals, or ceramics. Single inheritance would allow us to construct these hierarchical relationships. Under multiple inheritance, we can categorize the materials in a variety of ways at the same time. Figure 4.18 shows the use of the mixins Cheap and Brittle for this purpose.

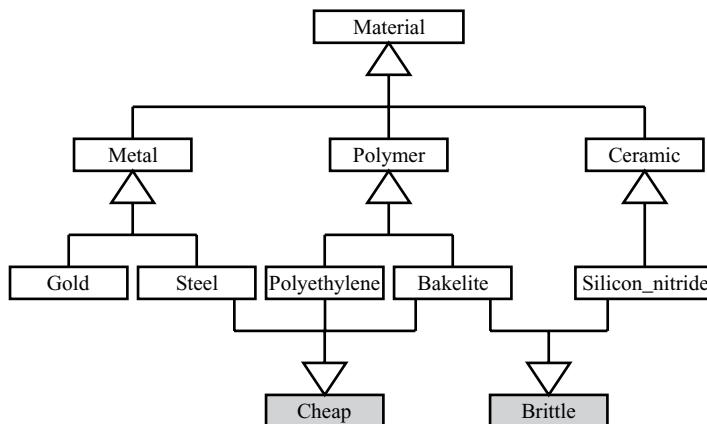


Figure 4.18 An example of the use of mixins.

4.6.4.3 Specialization of Methods

We have already seen that specialization can involve the introduction of new methods or attributes, overriding default assignments to attributes, or redefinition of inherited operations. If an operation needs to be redefined, i.e., specialized, it is not always necessary to rewrite it from scratch. In our example, a definition of the method `propagate` is inherited by the class `Defect` from the class `Feature`. The specialized version of `propagate` is the same, except that the sonic pulse must be attenuated. This can be achieved by calling the inherited definition from within the specialized one, and then adding the attenuation instruction, shown here in C++:

```
void Defect::propagate(Sonic_pulse* inst1)
{
    Feature::propagate(inst1);
    // propagate a pulse, inst1, using inherited version of
    // 'propagate'
    inst1->attenuate(0.5); // Attenuate the pulse.
    // The member function 'attenuate' must be defined for the
    // class 'Sonic_pulse'.
}
```

We might wish to call the specialized method `propagate` from within the definition of `interact`, a function that handles the overall interaction of a pulse with a defect:

```
void Defect::interact(Sonic_pulse* inst1)
{
    propagate(inst1);
    // propagate a pulse, inst1, using the locally defined
    // member function
    reflect(inst1);
    // generate a new pulse and send it in the opposite
    // direction
}
```

4.6.4.4 Class Browsers

Many OOP systems provide not only a language, but an environment in which to program. The environment may include tools that make OOP easier, and one of the most important of these tools is a class-browser. A class-browser shows inheritance relationships, often in the form of a tree such as those in Figures 4.14 and 4.15. If the programmer wants to alter a class or to specialize it to form a new one, he or she simply chooses the class from the browser and then selects the type of change from a menu. Incidentally, browsers themselves are invariably built using

OOP. Thus, the class-browser may be an instance of a class called `Class_browser`, which may be a specialization of `Browser`, itself a specialization of `Window`. The class `Class_browser` could be further specialized to provide different display or editing facilities.

4.6.5 Encapsulation

Encapsulation, or information-hiding, is a term used to express the notion that the instance attributes and the methods that define an object belong to that object and to no other. The methods and attributes are therefore private and are said to be encapsulated within the object. The interface to each object reveals as little as possible of the inner workings of the object. The object has control over its own data, and those data cannot be directly altered by other objects. Class attributes are an exception, as they are not encapsulated within any one instance but are shared between all instances of the same class.

Some languages such as Smalltalk adhere to the principle of encapsulation, whereas C++ offers flexibility in its enforcement of encapsulation through access controls. All C++ data members and member functions (collectively known as *members*) are allocated one of four access levels:

- `private`: access from member functions of this class only (the default);
- `protected`: access from member functions of this class and of derived classes;
- `public`: access from any part of the program; and
- `friend`: access from member functions of nominated classes.

Access controls are illustrated in Figure 4.19. C++ allows two types of derivation (i.e., inheritance), known as public and private. In private derivation, protected and public members in the base class become private members of the derived class. In public derivation, the access level of members in the derived class is unchanged from

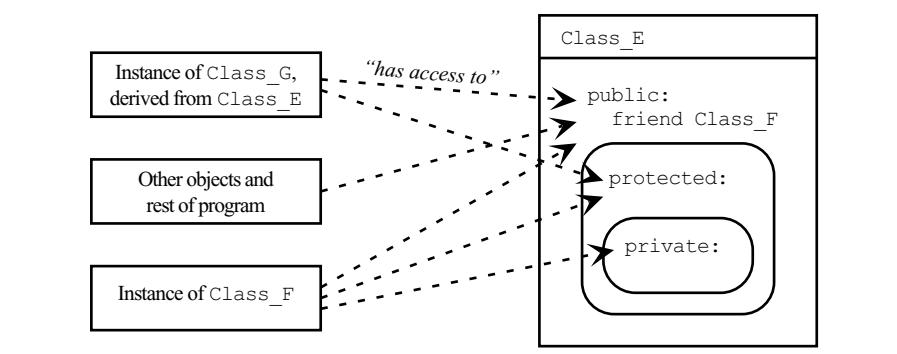


Figure 4.19 Access control in C++.

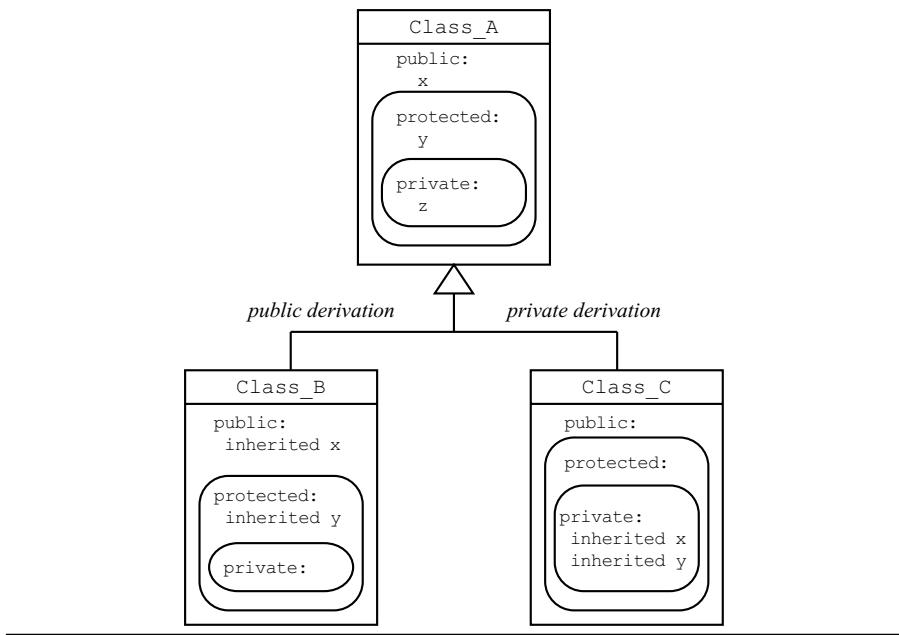


Figure 4.20 Public and private derivation in C++.

the base class. In neither case does the derived class have access to private members of the base class. The two types of derivation are shown in Figure 4.20.

4.6.6 Unified Modeling Language (UML)

The Unified Modeling Language (UML) (Booch et al. 2005; Bennett et al. 2010; Rumbaugh et al. 2010) provides a means of specifying the relationships between classes and instances, and representing them diagrammatically. We have already come across two such relationships:

- Specialization/generalization; and
- Instantiation.

Specialization describes the is-a-kind-of relationship between a subclass and superclass, and it involves the inheritance of common information. Instantiation describes the relation between a class and an instance of that class. An instance of a class and a subclass of a class are both said to be *clients* of that class, since they both derive information from it, but in different ways.

We will now consider three more types of relationships:

- Aggregation;
- Composition; and
- Association.

An *aggregation* relationship exists when an object can be viewed as comprising several subobjects. The world of software itself provides a good example, as a software library can be seen as an aggregation of many component modules. The degree of ownership of the modules by the software library is rather weak, in the sense that the same module could also belong to another software library. The *composition* relationship is a special case in which there is a strong degree of ownership of the component parts. For instance, a car comprises an engine, chassis, doors, seats, wheels, etc. This example can be recognized as a composition relation since duplication or deletion of a car would require duplication or deletion of these component objects. Returning to our ultrasonic example, we can regard the imaging equipment as a composition of a transmitter, detector, display, and image processing software. The image processing software is an aggregation of various modules. Figure 4.21 shows that assembly and composition relationships allow problems to be viewed at different levels of abstraction.

Associations are loose relationships between objects. For instance, they might be used to represent the spatial layout of objects, possibly by naming the related instance as an instance attribute. Another form of association arises when one object makes use of another by sending messages to it (or, in the case of C++, calling its member functions or accessing its data members)—see Section 4.6.8. For example, pulses may send messages to the display object so that they can be redisplayed. The senders of messages are termed *actors*, and the recipients are *servers*. Objects that both send and receive messages are sometimes termed *agents*, although this terminology is confusing, given that agents have a different meaning described in Section 4.2.

UML includes a diagrammatic representation for the relationships between classes, summarized in Figure 4.22. Wherever appropriate, this notation has been used throughout this chapter.

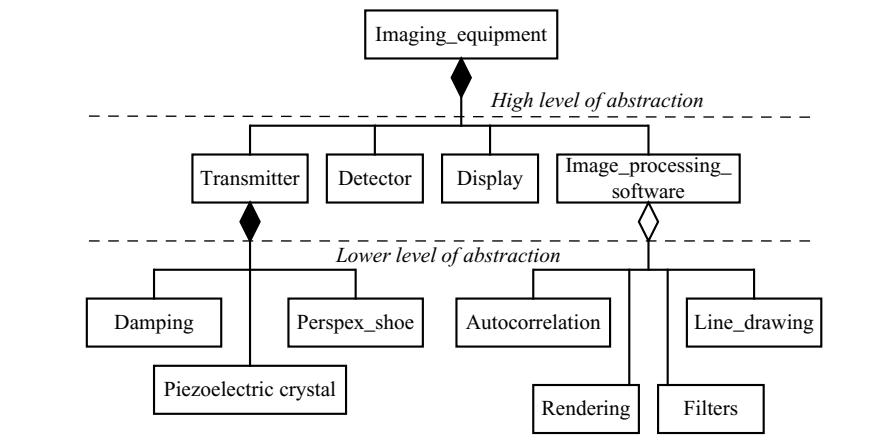


Figure 4.21 The aggregation and composition relationships allow problems to be viewed on different levels of abstraction. Filled diamonds indicate composition; unfilled diamonds indicate aggregation.

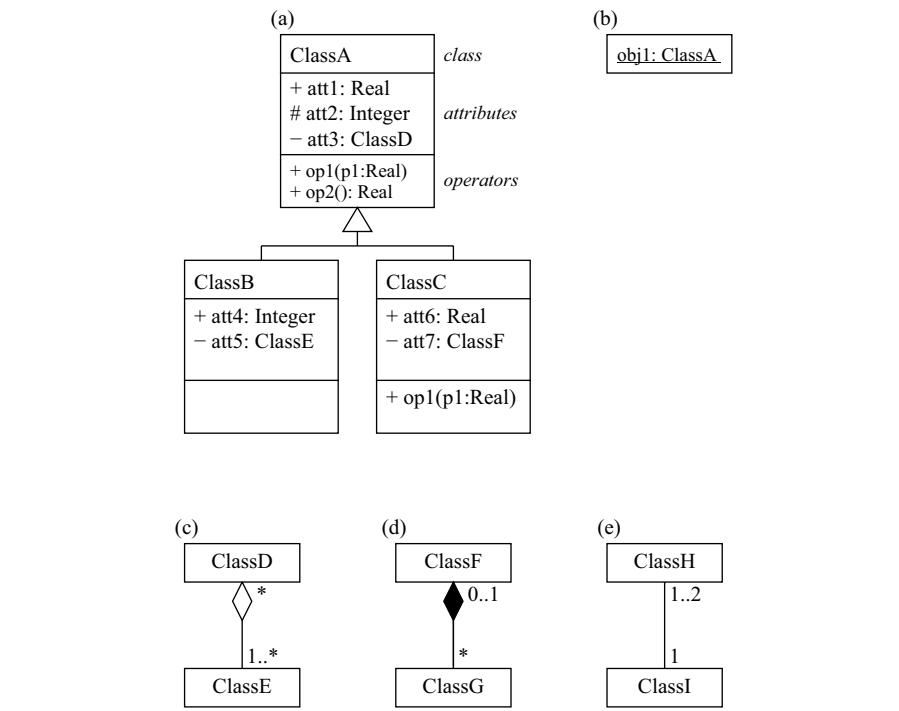


Figure 4.22 Class relationships in UML. (a) ClassA is a generalization of ClassB and ClassC; (b) obj1 is an instance of ClassA; (c) aggregation; (d) composition; (e) association. Numbers represent permissible number of objects; * indicates any number. Access controls: + indicates public, # indicates protected, and – indicates private.

4.6.7 Dynamic (or Late) Binding

Three necessary features of an OOP language, as defined in Section 4.6.1, are data abstraction, inheritance, and encapsulation. The fourth and final necessary feature is *dynamic binding* (or *late binding*). Although the parent classes of objects may be known at compilation time, the actual (derived) classes may not be. The actual class is not bound to an object name at compilation time, but instead the binding is postponed until run-time. This is known as dynamic binding and its significance is best shown by example.

Suppose that we have a method, `propagate`, defined in C++ for the class, `Feature`:

```
void Feature::propagate(Sonic_pulse* p)
{
    float x, y;
    x = getx();
```

```

y = gety(); // calculate new position for pulse, p
p->move(x,y); // move pulse p to its new position
}

```

Suppose also that `shear1` and `long1` are instances of `Shear_pulse` and `Longitudinal_pulse`, respectively. When the program is run, transmission of `shear1` and `long1` by a feature is achieved by calls to `propagate`, with pointers to each pulse passed as parameters. The parameter types are correct, since `shear1` is an instance of `Shear_pulse`, which is derived from `Sonic_pulse`. Similarly, `long1` is an instance of `Longitudinal_pulse`, which is also derived from `Sonic_pulse`. The method `propagate` calls the method `move`, but it may be specialized differently for `shear1` and for `long1`. Nevertheless, the correct definition will be chosen in each case. This is an example of late binding, since the actual method `move` that will be used is determined each time that `propagate` is called, rather than when the program is compiled.

The combined effect of inheritance and dynamic binding is that the same function call (`move` in the above example) can have more than one meaning, and the actual meaning is not interpreted until run-time. This effect is known as *polymorphism*.*

To see why polymorphism is so important, we should consider how we would tackle the above problem in a language where binding is static (or early). In such languages, which are said to use *monomorphism*, the exact meaning of each function call is determined at compilation time. In the method `propagate`, we would have to test the class of its argument and invoke a behavior accordingly. Depending on the language, we might need to include a class variable on `Sonic_pulse` to tag each instance with the name of its class (which may be `Sonic_pulse` or a class derived from it). We might then use a case statement, shown here in C++:

```

void Feature::propagate(Sonic_pulse* p)
{
    float x, y;
    char class_label;
    x = getx(); y = gety(); // calculate new position for pulse p
    class_label = p->tag; // identify the class of p from its tag
    switch (class_label) {

```

* C++ distinguishes between those member functions (i.e., methods) that can be redefined polymorphically in a derived class and those that cannot. Functions that can be redefined polymorphically are called virtual functions. In this example, `move` must be a virtual function if it is to be polymorphic. A pure virtual function is one that is declared in the base class, but no definition is supplied there. (The declaration merely states the existence of a function; the definition is the chunk of code that makes up the function.) A definition must, therefore, appear in the derived classes. Any class containing one or more pure virtual functions is termed an abstract base class. Instances of an abstract base class are not allowed since an instance, if it were allowed, would “know” that it had access to a virtual function but would not have a definition for it.

```

case 's':
    ...;      // code to move a Shear_pulse
    break;
case 'l':
    ...;      // code to move a Longitudinal_pulse
    break;
default:
    ....;    // print an error message
}
}

```

Two drawbacks are immediately apparent. First, the code that uses monomorphism is much longer and is likely to include considerable duplication, as the code for moving a `Shear_pulse` will be similar to the code for moving a `Longitudinal_pulse`. In contrast, polymorphism avoids duplication and allows the commonality between classes to be made explicit. Second, the code is more difficult to maintain than the polymorphic code, because subsequent addition of a new class of pulse would require changes to the method `propagate` in the class `feature`. This limitation runs against the philosophy of encapsulation, because the addition of a class should not require changes to existing classes.

The effect of polymorphism is that the language will always select the sensible meaning for a function call. Thus, once its importance has been understood, polymorphism should not be a source of concern for the programmer.

4.6.8 Message Passing and Function Calls

In C++, member functions (equivalent to methods) are accessed in a similar way to any other function, except that we need to distinguish the object to which the member function belongs. In the following C++ example, defects `d1` and `d2` are created using the two techniques introduced in Section 4.6.3, and the member function `propagate` is then accessed. A call to a conventional function is also included for comparison:

```

result = somefunc(parameter);
// call a conventional function, defined elsewhere.
Defect* d1 = new Defect();
// make a new instance d1 of Defect.
Sonic_pulse* p1 = new Sonic_pulse();
Shear_pulse* p2 = new Shear_pulse();
// make two new instances of pulses, p1 and p2.
d1->propagate(p1);
// call member function, 'propagate', with parameter p1.
Defect d2; // make a new instance d2 of Defect.
d2.propagate(p2);
// call member function, 'propagate', with parameter p2.

```

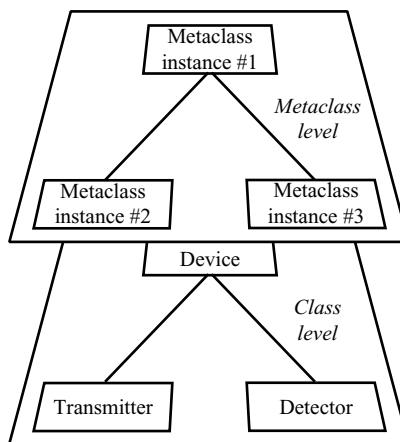


Figure 4.23 Each class can be treated as an instance of a metaclass.

In Smalltalk and other OOP languages, methods are invoked by passing *messages*. This terminology emphasizes the concept of encapsulation. Each object is independent and is in charge of its own methods. Nevertheless, objects can interact. One object can stimulate another to fire up a method by sending a message to it.

4.6.9 Metaclasses

It was emphasized in Section 4.6.3 that a class defines the characteristics (methods and attributes) that are available to instances. Methods can therefore be used only by instances and not by the class itself. This constraint is sensible, as it is clearly appropriate that methods such as `move`, `propagate`, and `reflect` should be performed by instances in the ultrasonics example. However, a problem arises with the creation of new instances, since it would not be possible to invoke the method `new` until an instance had already been created. This apparent paradox is overcome by imagining that each class is an instance of a metaclass, which is a class of classes (Figure 4.23). Thus, the method `new` is defined within one or more metaclasses and made available to each class. Likewise, class variables are simply instance variables that are declared at the metaclass level.

Although C++ does not explicitly include metaclasses, it supports class variables in the form of static data members. C++ also provides static member functions. Unlike ordinary member functions, which are encapsulated within each instance, there is only one copy of a static member function for a class. The single copy is shared by all instances of the class and is not associated with any particular instance. A static member function is, therefore, equivalent to a method (such as `new`) that is defined within a metaclass.

4.6.10 Type Checking

There are some fundamental differences between the various OOP languages, as well as their obvious differences in syntax. C++ uses static type checking whereas Smalltalk uses dynamic type checking. Thus, a C++ programmer must explicitly state the type (or class) of all variables, and the compiler checks for consistency. The Smalltalk programmer does not need to specify the classes of variables, and no checks are made during compilation. Static type checking is sometimes called *strong* type checking, and the variables are said to be *manifestly* typed. Conversely, dynamic type checking is sometimes called *weak* type checking, and the variables are said to be *latently* typed. The words *strong* and *weak* are perhaps misleading, since they might alternatively indicate the strictness or level of detail of static type checking.

Static type checking describes the ability of the compiler to check that objects of the right class are supplied to all functions and operators. In C++, the programmer must explicitly state the class of all objects (including built-in types such as `int` or `float`). Whenever assignments are made, the compiler checks that the types are compatible. Assignments can be made directly by a statement such as:

```
x = "a string";
```

or indirectly by passing an object as a parameter to a function. Consider again our definition of the method `propagate` in C++:

```
void Feature::propagate(Sonic_pulse* p)
{
    float x, y;
    x = getx();
    y = gety(); // calculate new position for Sonic_pulse p
    p->move(x, y); // move Sonic_pulse p to its new position
}
```

Let us now create some object instances:

```
Sonic_pulse* p1 = new Sonic_pulse();
Shear_pulse* p2 = new Shear_pulse();
Defect* d1 = new Defect();
Defect* d2 = new Defect();
```

If we pass as a parameter to `propagate` any object whose type is neither a pointer to `Sonic_pulse` nor a pointer to a class publicly derived from `Sonic_pulse`, we will get an error at compilation time:

```
d1->propagate(p1);
// OK because p1 is of type Sonic_pulse.

d1->propagate(p2);
```

```
// OK because p2's type is publicly derived from
// Sonic_pulse.

d1->propagate(d2);
// ERROR: d2 is not a Sonic_pulse.
```

The difference between static and dynamic typing represents a difference in programming philosophy. C++ insists that all types be stated, and it checks them all at compilation time to ensure that there are no incompatibilities. This requirement means extra work for the programmer, especially if he or she is only performing a quick experiment. On the other hand, it leads to clearly defined interactions between objects, thereby helping to document the program and making error detection easier. Static typing tends to be preferred for building large software systems that involve more than one programmer. Dynamically typed languages such as Smalltalk (and also Lisp, Prolog, and Python, introduced in Chapter 11) are ideal for trying out ideas and building prototypes, but they are more likely to contain latent bugs that may show up unpredictably at run-time.

4.6.11 Persistence

We have already discussed the creation and deletion of objects. The lifetime of an object is the time between its creation and deletion. So far, our discussion has implicitly assumed that instances are created and destroyed in a timeframe corresponding to one of the following:

- The evaluation of a single expression;
- Running a method or other block of code;
- Specific creation and deletion events during the running of a program; and
- The duration of the program.

However, object instances can outlive the run of the program in which they were created. A program can create an object and then store it. That object is said to be persistent across time (because it is stored) and space (because it can be moved elsewhere). Persistent objects are particularly important in database applications. For example, a payroll system might store instances of the class `Employee`. Such applications require careful design, since the stored instances are required not only between different executions of the program, but also between different versions of the program.

4.6.12 Concurrency

Each object acts independently, on the basis of messages that it receives. Objects can, therefore, perform their own individual tasks concurrently. This independence makes them strong candidates for implementation on parallel processing computers.

Returning once more to our object-oriented ultrasonic simulation, a pulse could arrive at one feature (say the front wall) at the same time as another arrives at a different feature (say a defect), and both would require processing by the respective features. Although these processing tasks could be achieved on a parallel machine, the actual implementation was on a serial computer, where concurrency was simulated. A clock (implemented as an object) marked simulated time. Thus, if two pulses arrived at different features at the same time t , it would not matter which was processed first, as according to the simulation clock, they would both be processed at time t .

4.6.13 Active Values and Daemons

So far, we have considered programs in which methods are explicitly called, and where these methods may involve reading or altering data attached to object attributes. Control is achieved through function (method) calls, and data are accessed as a side effect. Active values and daemons allow us to achieve the opposite effect, namely, function calls are made as a side effect of accessing data. Attributes that can trigger function calls in this way are said to be *active*, and their data are *active values*. The functions that are attached to the data are called *daemons*. A daemon (sometimes spelled “demon”) can be thought of as a piece of code that lies dormant, watching an active value, and which springs to life as soon as the active value is accessed.

Daemons can pose some problems for the flow of control in a program. Suppose that a method `method1` accesses an active value that is monitored by daemon `daemon1`. If `daemon1` were to fire immediately, before `method1` has finished, then `daemon1` might disrupt some control variables upon which `method1` relies. The safer solution is to treat the methods and daemons as indivisible “atomic” actions, so that `method1` has to run to completion before `daemon1` fires.

Active values need not be confined to attributes (data members) but may also include methods (member functions). Thus, a daemon may be set to fire when one or more of the following occur:

- An attribute is read;
- An attribute is set; or
- A method is called.

An example of the use of daemons is in the construction of gauges in graphical user interfaces (Figure 4.24). A gauge object may be created to monitor some attribute of an object in a physical model, such as the voltage across a transducer. As soon as the voltage value is altered, the gauge should update its display, regardless of what caused the change in voltage. The `voltage` attribute on the object `transducer` is active, and the `display` method on the object `gauge` is a daemon.



Figure 4.24 Using a daemon to monitor an active value.

4.6.14 Summary of Object-Oriented Systems

We began with the premise that OOP requires the following language capabilities:

- **Data Abstraction:** New types (classes) can be defined and a full set of operations provided for each, so that the new classes behave like built-in ones such as int and float.
- **Inheritance:** Class definitions can be treated as specializations of other (parent) classes. This ability maximizes code re-use and minimizes duplication. OOP therefore helps us to build a world model that maps directly onto the real world. This model is expressed in terms of the classes of objects that the world manipulates, and OOP helps us to refine successively our understanding of these classes.
- **Encapsulation:** Data and code are bundled together into objects and are hidden from other objects. This encapsulation simplifies program maintenance, helps to ensure that interactions between objects are clearly defined and predictable, and provides a realistic world model.
- **Dynamic (late) binding:** The object (and, hence, its class) that is associated with a given variable is not determined until run-time. Although the parent class may be specified and checked at compile-time (depending on the language), the instance used at run-time could legitimately belong to a subclass of that parent. Therefore, it cannot be known at compile-time whether the object will use methods defined within the parent class or specialized versions of them. Different objects can, therefore, respond to a single command in a way that is appropriate for those objects, and which may be different from the response of other objects. This property is known as *polymorphism*.

Stroustrup (1988) has pointed out that, although it is *possible* to build the above capabilities in many computer languages, a language can only be described as object-oriented if it *supports* these features, that is, it makes these features convenient for the programmer.

Class definitions act as templates from which multiple instances (i.e., objects) can be created. Classes and instances are easily inspected, modified, and reused. All of these capabilities assist in the construction of large, complex software systems by breaking the system down into smaller, independent, manageable chunks. The problem representation is also more natural, allowing more time to be spent designing a system and less on coding, testing, and integration.

4.7 Objects and Agents

Although the history of agent-based programming can be traced back to the 1970s, it is now seen by many as the next logical development from OOP, which came to prominence during the 1980s. OOP has become the preferred style of programming for most practical applications as it allows complex problems to be broken down into simpler constituents while maintaining the integrity of the overall system. If objects are viewed as obedient servants, as suggested in Section 4.1, then intelligent agents can be seen as independent beings. When an autonomous agent receives a request to perform an action, it will make its own decision, based on its beliefs and in pursuit of its goals. Thus, it behaves more like an individual with his or her own personality. In consequence, agent-based systems are analogous to human teams, organizations, or societies.

As an encapsulated software entity, an intelligent agent bears some resemblance to an object. However, it is different in three ways:

- *Autonomy*: Once an object has declared a method as public, as it must for the method to be useful, it loses its autonomy. Other objects can then invoke that method by sending a message. In contrast, agents cannot invoke the actions of another agent, they can only make *requests*. The decision over what action to take rests with the receiver of the message, not the sender. Autonomy is not required in an object-oriented system because each object is designed to perform a task in pursuit of the developer's overall goal. However, it cannot necessarily be assumed that agents will share a common goal.
- *Intelligence*: Although intelligent behavior can be built into an object, it is not a requirement of the OOP model.
- *Persistence*: It was stated in Section 4.6.11 that objects could be made to persist from one run of a program to another by storing them. In contrast, agents persist in the sense that they are constantly “switched on” and thus they operate concurrently. They are said to have their own thread of control, which can be thought of as another facet of autonomy since an agent decides for itself when it will do something. In contrast, a standard object-oriented system has a single thread of control, with objects performing actions sequentially. One exception to this is the idea of an active object (Booch et al. 2007) that has its own thread of control and is, therefore, more akin to an agent in this respect.

While OOP languages such as C++, Smalltalk, and Java are quite closely defined, there are no standard approaches to the implementation of agent-based systems. Indeed, an object-oriented approach might typically be used in the implementation of an agent-based system, while the reverse is unlikely.

4.8 Frame-Based Systems

Frame-based systems are closely allied to object-oriented systems. Frame-based systems evolved from artificial intelligence research, whereas OOP has its origins in more general software engineering. Frames provide a means of representing and organizing knowledge. They use some of the key features of object-orientation for organizing information, but they do not have the behaviors or methods of objects. As relations can be created between frames, they can be used to model physical or abstract connections. Just as object instances are derived from object classes, so frame instances can be derived from frame classes, taking advantage of inheritance.

Section 4.6.1 quoted a definition of OOP from Pascoe (1986). Frame-based systems are less rigidly defined, so a corresponding definition might be:

Frame-based systems offer data abstraction and inheritance. In general, they do not offer either encapsulation or dynamic (late) binding.

The absence of encapsulation is the key feature that distinguishes frames from objects. Since frames do not have ownership of their own functions, they are passive structures. Without encapsulation, the concept of dynamic binding becomes meaningless. This does not, however, detract from their usefulness. In fact, they provide an extremely useful way of organizing data and knowledge in a knowledge-based system.

The frame-based view is that an item such as my truck can be represented by a data structure, i.e., a frame instance, that we can call `my_truck`. This instance can inherit from a class of frame such as `Truck`, which can itself inherit from a parent class such as `Vehicle`. Thus, frame-based systems support data abstraction by allowing the definition of new classes. They also support inheritance between classes and from a class to its instances.

We can hang extra information onto a frame, such as the number of wheels on my truck. Thus `number_of_wheels` could be a slot (see Section 4.6.3) associated with the frame instance `my_truck`. This slot could use the default value of 4 inherited from `Vehicle` or it may be a locally defined value that overrides the default. The value associated with a slot can be a number, a description, a number range, a procedure, another frame, or anything allowed by the particular implementation. Some frame-based systems allow us to place multiple values in a slot. In such systems, the different pieces of information that we might want to associate with a slot are known as its *facets*. Each facet can have a value associated with it, as shown in Figure 4.25. For instance, we may wish to specify limits on the number of wheels, provide a default, or calculate a value using a function known as *an access function*. In this example, an access

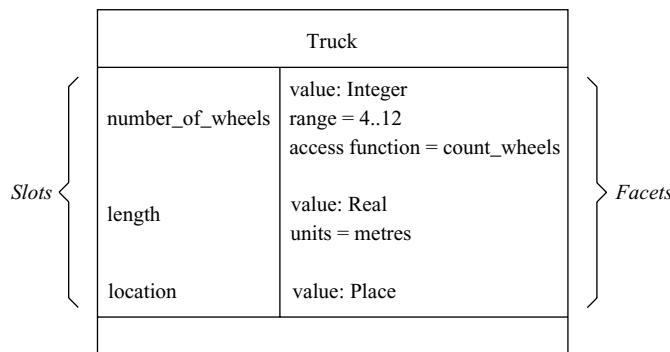


Figure 4.25 An example of a frame-based representation.

function `count_wheels` could calculate the number of wheels when a value is not previously known.

It was pointed out above that frame-based systems do not, in general, provide encapsulation. Consider the following example, written using the syntax of the Flex™ artificial intelligence toolkit (Melioli et al. 2014; Mutawa and Alzuwawi 2019), which includes frame-based capabilities:

```

/* define a frame */
frame vehicle;
    default location is garage and
    default number_of_wheels is 4
    and default mass_in_tonnes is 1 .

/* define another frame */
frame truck is a kind of vehicle;
    default mass_in_tonnes is 10 .

/* create an instance of frame truck */
instance my_truck is a truck .

/* create another instance of frame truck */
instance your_truck is a truck;
    number_of_wheels is my_truck`'s number_of_wheels + 2 .

```

In this example, the default number of wheels is inherited by `my_truck`. The number of wheels on `your_truck` is derived by an access function defined within the frame and which does not need to be explicitly named. The access function in `your_truck` makes use of a value in `my_truck`, although this would be forbidden under the principle of encapsulation. As the access function is not calculated until run-time, the example also demonstrates that Flex offers dynamic binding.

Frames can be applied to physical things such as trucks or abstract things such as plans or designs. A special kind of frame, known as a *script*, can be used to describe actions or sequences of actions. For instance, in a system for fixing plumbing faults we might construct scripts called `changing_washer` or `unblocking_pipe`.

The term *frame* has been used so far here to imply a framework onto which information can be hung. However, frames are also analogous to the frames of a movie or video. In associating values with the slots of a frame we are taking a snapshot of the world at a given instant. At one moment the slot `location` on `my_truck` might contain the value `smallville`, while sometime later it might contain the value `targettown`.

4.9 Summary: Agents, Objects, and Frames

In this chapter, we have seen that OOP is built around the concepts of data abstraction, inheritance, encapsulation, and dynamic (or late) binding. Intelligent agents extend the ideas of objects by giving them autonomy, intelligence, and persistence. Intelligent agents act in pursuit of their personal goals, which may or may not be the same as those of other agents. Intelligent agents can be made mobile, so they can travel across a network—possibly the Internet—to perform a set of tasks. This mobility can reduce the amount of data transfer required compared with the whole task being performed on the originating computer.

MASs contain several interacting intelligent agents, often designed as computer models of human functional roles. The overall effect is to mimic human teams or societies. There are a variety of ways for organizing an MAS, and the agents within it may work cooperatively or competitively. There are also a variety of ways of achieving inter-agent communication, including the use of KQML.

We saw that frame-based systems support data abstraction and inheritance, but that they do not necessarily offer encapsulation or dynamic binding. As a result, frames are passive in the sense that, like entries in a database, they do not perform any tasks themselves. Their ability to calculate values for slots through access functions might be regarded as an exception to this generality. Although frames are more limited than a full-fledged object-oriented system, they nonetheless provide a powerful way of organizing and managing knowledge in a knowledge-based system.

Further Reading

- Bennett, S., S. McRobb, and R. Farmer. 2010. *Object-Oriented Systems Analysis and Design using UML*. 4th ed. McGraw-Hill, London, UK.
- Booch, G., R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. 2007. *Object-oriented Analysis and Design with Applications*. 3rd ed. Addison-Wesley, Reading, MA.

- Shoham, Y., and K. Leyton-Brown. 2009. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, Cambridge, UK.
- Stroustrup, B. 2014. *Programming: Principles and Practice using C++*. 2nd ed. Addison-Wesley, Reading, MA.
- Weiss, G., ed. 2017. *Multiagent Systems*. 2nd ed. MIT Press, Cambridge, MA.
- Wooldridge, M. J. 2009. *An Introduction to Multiagent Systems*. 2nd ed. Wiley, Chichester, UK.

Chapter 5

Symbolic Learning

5.1 Introduction

The preceding chapters have discussed ways of representing knowledge and drawing inferences. It was assumed that the knowledge itself was readily available and could be expressed explicitly. However, there are many circumstances where this is not the case, such as those listed here, adapted from (Alpaydin 2010).

- The software engineer may not possess the domain expertise. The knowledge would need to be obtained from a domain expert through a process of *knowledge acquisition* (O’Leary 1998) or, conversely, the domain expert would need to acquire software engineering skills.
- The rules that describe a particular domain may not be completely understood, such as listening to and interpreting a spoken voice.
- Even though a domain may be well understood, it may not be expressible explicitly in terms of rules, facts or relationships. This category includes *skills*, such as welding or painting.
- The problem may change over time. For example, the optimal routing of packets on a computer network varies according to network loading patterns.

One way around these difficulties is to have the system learn for itself from a set of example solutions. Two approaches can be broadly recognized: *symbolic learning* and *numerical learning*. Symbolic learning describes systems that formulate and modify rules, facts, and relationships, explicitly expressed in words and symbols. In other words, they create and modify their own knowledge base. Numerical learning refers to systems that use numerical models; learning in this context refers to techniques for optimizing the numerical parameters. Numerical

learning includes artificial neural networks (Chapters 8 and 9) and a variety of optimization algorithms such as genetic algorithms (Chapter 7) and simulated annealing (Chapter 6).

A learning system is normally given some feedback on its performance. The source of this feedback is called the *teacher* or the *oracle*. Often the teacher role is fulfilled by the environment within which the knowledge-based system is working. In other words, the reaction of the environment to a decision is sufficient to indicate whether the decision was right or wrong. Learning with a teacher is sometimes called *supervised* learning. Learning can be classified as follows, where each category involves a different level of supervision:

1. Rote learning.

The system receives confirmation of correct decisions. When it produces an incorrect decision, it is shown the correct rule or relationship that it should have used.

2. Learning from advice.

Rather than being given a specific rule that should apply in a given circumstance, the system is given a piece of general advice, such as “gas is more likely to escape from a valve than from a pipe.” The system must sort out for itself how to move from this high-level abstract advice to an immediately usable rule.

3. Learning by induction.

The system is presented with sets of example data and is told the correct conclusions that it should draw from each. The system continually refines its rules and relations so as to correctly handle each new example.

4. Learning by analogy.

The system is told the correct response to a similar, but not identical, task. The system must adapt the previous response to generate a new rule applicable to the new circumstances.

5. Explanation-based learning (EBL).

The system analyzes a set of example solutions and their outcomes to determine *why* each one was successful or otherwise. Explanations are generated, which are used to guide future problem-solving. EBL is incorporated into PRODIGY, a general-purpose problem-solver (Minton et al. 1989).

6. Case-based reasoning (CBR).

Any case about which the system has reasoned is filed away, together with the outcome, regardless of the degree of success of the reasoning process. Whenever a new case is encountered, the system adapts its stored behavior to fit the new circumstances. CBR is discussed in further detail in Section 5.3.

7. Explorative or unsupervised learning.

Rather than having an explicit goal, an explorative system continuously searches for patterns and relationships in the input data, perhaps marking

some patterns as interesting and warranting further investigation. Examples of the use of unsupervised learning include:

- *Data mining*, where patterns are sought among large or complex data sets.
- Identifying *clusters*, possibly for compressing the data.
- Learning to *recognize* fundamental features, such as edges, from pixel images.
- *Designing* products, where innovation is a desirable characteristic.

In rote learning and learning from advice, the sophistication lies in the ability of the teacher rather than the learning system. If the teacher is a human expert, these two techniques can provide an interactive means of eliciting the expert's knowledge in a suitable form for addition to the knowledge base. However, most of the interest in symbolic learning has focused on learning by induction and CBR, discussed in the following two sections. Reasoning by analogy is similar to CBR, while many of the problems and solutions associated with learning by induction also apply to the other categories of symbolic learning.

5.2 Learning by Induction

5.2.1 Overview

Rule induction involves generating from specific examples a general rule of the type:

```
if <general circumstance> then <general conclusion>
```

Since it is based on trial-and-error, induction can be said to be an empirical approach. We can never be certain of the accuracy of an induced rule, since it may be shown to be invalid by an example that we have not yet encountered. The aim of induction is to build rules that are successful as often as possible, and to modify them quickly when they are found to be wrong. Whatever is being learned—typically, rules and relationships—should match the positive examples but not the negative ones.

The first step is to generate an initial prototype rule that can subsequently be refined. The initial prototype may be a copy of a general-purpose template, or it can be generated by hypothesizing a causal link between a pair of observations. For instance, if a specific valve, *valve_1*, in a particular plant is open and the flow rate through it is $0.5\text{m}^3\text{s}^{-1}$, we can propose two initial prototype rules in Flex format:

```
rule r5_1
  if status of valve_1 is open
  then flow_rate of valve_1 becomes 0.5 .

rule r5_2
  if flow_rate of valve_1 is 0.5
  then status of valve_1 becomes open.
```

The prototype rules can then be modified in the light of additional example data or rejected. Rule modifications can be classified as either strengthening or weakening. The condition is made stronger (or *specialized*) by restricting the circumstances to which it applies, and conversely it is made weaker (or *generalized*) by increasing its applicability. The pair of preceding examples could be made more general by considering other valves or a less precise measure of flow. On the other hand, they could be made more specific by specifying the necessary state of other parts of the boiler. A rule needs to be generalized if it fails to fire for a given set of data, where we are told by the teacher that it should have fired. Conversely, a rule needs to be specialized if it fires when it should not.

Assume for the moment that we are dealing with a rule that needs to be generalized. The first task is to spot where the condition is deficient. This is easy when using pattern matching, provided that we have a suitable representation. Consider the following prototype rule:

```
rule r5_3
  if status of X is open
  and X is a gas_valve
  then flow_rate of X becomes high.
```

In Chapter 4, names beginning with an uppercase letter were used for classes, consistent with conventions in C++ and UML. However, in Flex, class names are in lowercase, while names beginning with an uppercase letter are used to indicate a local variable that can be matched to something. Rule r5_3 would fire, given the scenario:

```
status of valve_1 is open.
valve_1 is a gas_valve.
```

The rule is able to fire because `status of valve_1 is open` matches `status of X is open` and `valve_1 is a gas_valve` matches `X is a gas_valve`. The conclusion `flow_rate of valve_1 is high` would be drawn. Now consider the following scenario:

```
status of valve_2 is open.
valve_2 is a water_valve.
```

The rule would not fire. However, the teacher may tell us that the conclusion `flow_rate of valve_2 is high` should have been drawn. We now look for matches as before and find that `status of valve_1 is open` matches `status of X is open`. However, there is no match to `X is a gas_valve`, so this part of the condition needs to be generalized to embrace the circumstance `X is a water_valve`.

We can therefore recognize where a rule is deficient by pattern-matching between the condition part of the rule and the scenario description. This approach is analogous to *means–ends analysis*, which is used to determine a plan for changing

the world from its current state to a goal state (see Section 4.3.3 and Section 14.3). Means–ends analysis typically uses pattern-matching to determine which rules can lead to such a change.

5.2.2 Learning Viewed as a Search Problem

The task of generalizing or specializing a rule is not straightforward, as there are many alternative ways in which a rule can be changed. Finding the correctly modified rule is a search problem, where the search field can be enormous. Figure 5.1 shows a possible form of the search tree, where each branch represents a generalization or specialization that would correctly handle the most recently encountered input. Subsequent inputs may reveal an incorrect choice (indicated by a dot at the end of a branch). The system must keep track of its current position in the search tree, as it must backtrack whenever an unsuitable choice is found to have been made.

Recording all of the valid options (or branches) that make up a search tree is likely to be unwieldy. The Lex system (Mitchell et al. 1983) offers a neat solution to this problem. Rather than keeping track of all the possible solutions, it simply records the most general and the most specific representations that fit the examples. These boundaries define the range of acceptable solutions. The boundaries move as further examples are encountered, converging on a smaller and smaller choice of rules, and ideally settling on a single rule.

There are other difficulties associated with the search problem. Suppose that a rule-based system is controlling a boiler. It makes a series of adjustments, but ultimately the boiler overheats, that is, the control objective has not been achieved. In this case, the feedback of the temperature reading to the learning system serves as

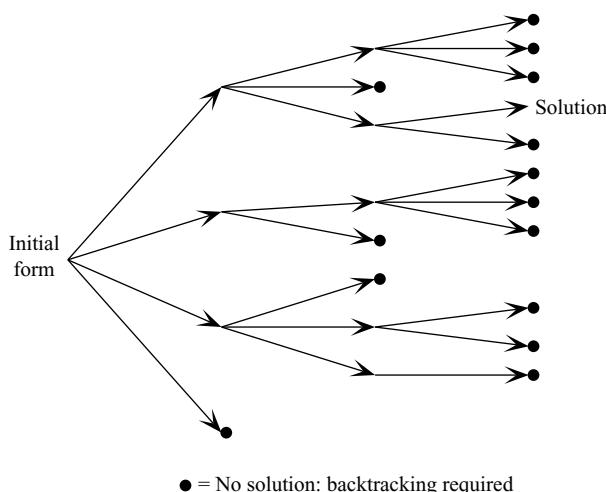


Figure 5.1 A search tree for rules.

the teacher. The system is faced with the difficulty of knowing where it went wrong, that is, which of its decisions were good and which ones were at fault. This is the *credit-assignment problem*. Credit assignment applies not only to negative examples such as this (which could be termed “blame assignment”), but also to cases where the overall series of decisions was successful. For example, a boiler controller that succeeds in keeping the temperature within the specified range might do so because several good decisions compensate for poorer ones.

The *frame problem* (or *situation-identification problem*), introduced in Chapter 1, affects many areas of artificial intelligence, particularly planning (Chapter 14). It is also pertinent here, where the problem is to determine which aspects of a given example situation are relevant to the new rule. A system for control of a boiler will have access to a wide range of information. Suppose that it comes across a set of circumstances where it is told by the teacher that it should shut off `valve_2`. The current world state perceived by the system is determined by stored data and sensor values, for example:

```
valve_1 is shut.
valve_2 is open.
gas_flow_rate is high.
gas_temperature is 300 . /* Celsius assumed */
steam_temperature: 150 . /* Celsius assumed */
fuel_oil_stored: 100 . /* gallons assumed */
fuel_oil_supplier: acme.
```

In order to derive a rule condition such as that in rule `r5_4` that follows, the system must be capable of deducing which parts of its world model to ignore—such as the information about fuel oil—and which to include:

```
rule r5_4
  if gas_flow_rate is high
  and status of valve_2 is open
  then report('** shut valve_2 **').
```

The ability to distinguish the relevant information from the rest places a requirement that the system must already have some knowledge about the domain.

5.2.3 Techniques for Generalization and Specialization

We can identify at least five methods of generalizing (or specializing through the inverse operation):

1. Universalization;
2. Replacing constants with variables;
3. Using disjunctions (generalization) and conjunctions (specialization);
4. Moving up a hierarchy (generalization) or down it (specialization); and
5. Chunking.

5.2.3.1 Universalization

Universalization involves inferring a new general rule from a set of specific cases. Consider the following series of separate scenarios:

```
status of valve_1 is open and flow_rate of valve_1 is high.
status of valve_2 is open and flow_rate of valve_2 is high.
status of valve_3 is open and flow_rate of valve_3 is high.
```

From these we might induce the following general rule:

```
rule r5_5
  if status of X is open
    then flow_rate of X becomes high.
```

Thus, we have generalized from a few specific cases to a general rule. This rule turns out to be *too* general as it does not specify that *X* must be a valve. However, this could be fixed through subsequent specialization based on some negative examples.

5.2.3.2 Replacing Constants with Variables

Universalization shows how we might induce a rule from a set of example scenarios. Similarly, general rules can be generated from more specific ones by replacing constants with local variables (see Section 2.6). Consider, for example, the following specific rules:

```
rule specific1
  if status of gas_valve_1 is open
    then flow_rate of gas_valve_1 becomes high.

rule specific2
  if status of gas_valve_2 is open
    then flow_rate of gas_valve_2 becomes high.

rule specific3
  if status of gas_valve_3 is open
    then flow_rate of gas_valve_3 becomes high.

rule specific4
  if status of gas_valve_4 is open
    then flow_rate of gas_valve_4 becomes high.

rule specific5
  if status of gas_valve_5 is open
    then flow_rate of gas_valve_5 becomes high.
```

From these specific rules we might induce a more general rule:

```
rule r5_5
  if status of X is open
  then flow_rate of X becomes high.
```

However, even this apparently simple change requires a certain amount of meta-knowledge (knowledge about knowledge). The system must “know” to favor the creation of rule r5_5 rather than, for example:

```
rule r5_6
  if status of X is open
  then flow_rate of Y becomes high.
```

or:

```
rule r5_7
  if status of X is open
  then Y of X becomes high.
```

Rule r5_6 implies that if any valve is open (we'll assume for now that we are only dealing with valves) then the flow rate through all valves is deduced to be high. Rule r5_7 implies that if a valve is open then everything associated with that valve (e.g., cost and temperature) becomes high.

5.2.3.3 Using Conjunctions and Disjunctions

Rules can be made more specific by adding conjunctions to the condition and more general by adding disjunctions. Suppose that rule r5_5 is applied when the world state includes the information `status of office_door is open`. We will draw the nonsensical conclusion that `the flow_rate of office_door is high`. The rule clearly needs to be modified by strengthening the condition. One way to achieve this is by use of a conjunction (*and*):

```
rule r5_8
  if status of X is open
  and X is a gas_valve
  then flow_rate of X becomes high.
```

We are relying on the teacher to tell us that `flow_rate through office_door is high` is not an accurate conclusion. We have already noted that there may be several alternative ways in which local variables might be introduced. The number of alternatives greatly increases when compound conditions are used. Here are some examples:

```
if valve_1 is open and valve_1 is a gas_valve then...
if valve_1 is open and X is a gas_valve then...
if X is open and valve_1 is a gas_valve then...
if X is open and X is a gas_valve then...
if X is open and Y is a gas_valve then...
```

The existence of these alternatives is another illustration that learning by rule induction is a search process in which the system searches for the correct rule.

Suppose now that we wish to extend rule r5_8 so that it includes water valves as well as gas valves. One way of doing this is to add a disjunction (*or*) to the condition part of the rule:

```
rule r5_9
  if status of X is open
  and [X is a gas_valve or X is a water_valve]
  then flow_rate of X becomes high.
```

This is an example of generalization by use of disjunctions. The use of disjunctions in this way is a “cautious generalization,” as it caters for the latest example, but does not embrace any novel situations. The other techniques risk overgeneralization, but the risky approach is necessary if we want the system to learn to handle data that it may not have seen previously. Overgeneralization can be fixed by specialization at a later stage when negative examples have come to light. However, the use of disjunctions should not be avoided altogether, as there are cases where a disjunctive condition is correct. This dilemma between the cautious and risky approaches to generalization is called the *disjunctive-concept* problem. A reasonable approach is to look for an alternative form of generalization, only resorting to the use of disjunctions if no other form can be found that fits the examples.

5.2.3.4 Moving Up or Down a Hierarchy

Rule r5_9 showed a cautious way of adapting rule r5_8 to include both water valves and gas valves. Another approach would be to modify rule r5_8 so that it can deal with valves of any description:

```
rule r5_10
  if status of X is open
  and X is a valve
  then flow_rate of X becomes high.
```

Here, we have made use of an is-a-kind-of relationship in order to generalize (see Section 4.6.4). In the class hierarchy for valves, `water_valve` and `gas_valve` are both specializations of the class `valve`, as shown in Figure 5.2, which uses UML-style capitalization for class names (see Section 4.6.6).

5.2.3.5 Chunking

Chunking is a mechanism for automated learning that is used in SOAR (Laird et al. 1986; Laird et al. 1987). SOAR is a production system (i.e., one that uses production rules—see Chapter 2) that has been modeled on a theory of human cognition. It works on the premise that, given an overall goal, every problem encountered along the way can be regarded as a subgoal. Problems are, therefore, tackled hierarchically.

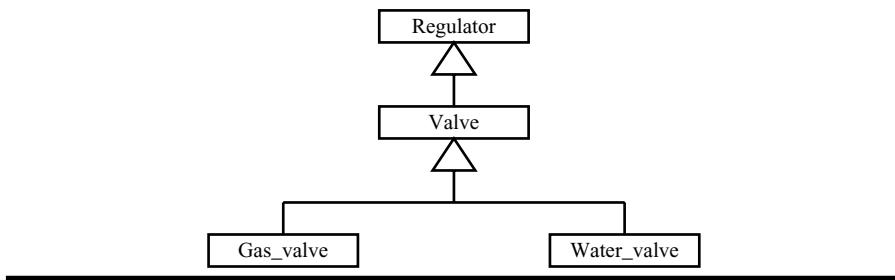


Figure 5.2 Class hierarchy for valves.

So, if a goal cannot be met at one level, it is broken down into subgoals. These automatically generated subgoals may be satisfied through the application of production rules stored in a long-term memory from previous runs of the system. However, the application of such rules may be slow, as several of them may need to be fired successively to satisfy the subgoal. Once SOAR has recognized the series of rules required to satisfy a subgoal, it can collapse them down into a single production rule. This is the process of *chunking*. The new rule, or *chunk*, is then stored so that it can rapidly solve the same subgoal if it should arise again in the future.

SOAR also offers a novel method of conflict resolution (see Section 2.8). The usual role of a production rule is to change some aspect of the system's state, thereby updating the model of the problem being tackled. In SOAR, any rule that can fire does so, but the changes to the current state proposed by the rule are not applied immediately. Instead, they are added to a list of suggested changes. A separate set of production rules is then applied to arrange conflicting suggestions in order of preference—a process known as *search control*. Once the conflicts have been resolved through search control, changes are made to the actual state elements.

5.3 Case-Based Reasoning (CBR)

The design of intelligent systems is often inspired by attempts to emulate the characteristics of human intelligence. One such characteristic is the ability to recall previous experience whenever a similar problem arises. This is the essence of CBR. As Riesbeck and Schank (1989) put it,

A case-based reasoner solves new problems by adapting solutions that were used to solve old problems.

Consider the example of diagnosing a fault in a refrigerator, an example that will be revisited in Chapter 12. If an expert system has made a successful diagnosis of the fault, given a set of symptoms, it can file away this information for future use. If the expert system is subsequently presented with details of another faulty refrigerator

of exactly the same type, displaying exactly the same symptoms in exactly the same circumstances, then the diagnosis can be completed simply by recalling the previous solution. However, a full description of the symptoms and the environment would need to be very detailed, and the chances of it ever being exactly reproduced are remote. What we need is the ability to identify a previous case, the solution of which can be modified to reflect the slightly altered circumstances, and then save it for future use. Aamodt and Plaza (1994) originally proposed that CBR can be described by a four-stage cycle, reviewed by de Mantaras et al. (2006):

- Retrieve the most similar case(s).
- Reuse the case(s) to attempt to solve the problem.
- Revise the proposed solution if necessary.
- Retain the new solution as a part of a new case.

Such an approach is arguably a good model of human reasoning. Indeed, CBR is often used in a semiautomated manner, where a human can intervene at any stage in the cycle.

5.3.1 Storing Cases

The stored cases form a *case base*. An effective way of representing the relevance of cases in the case base is by storing them as objects. Riesbeck and Schank (1989) have defined a number of types of link between classes and instances in order to assist in locating relevant cases. These links are described in the following subsections and examples of their use are shown in Figure 5.3.

5.3.1.1 Abstraction Links and Index Links

The classes may form a structured hierarchy, in which the different levels correspond to the level of detail of the case descriptions. Riesbeck and Schank (1989) distinguish two types of link between classes and their specializations: abstraction links and index links. A subclass connected by an abstraction link provides additional detail to its superclass, without overriding any information. An index link is a specialization in which the subclass has an attribute value that is different from the default defined in its superclass. For example, the class `Refrigerator_fault` might have an index link to the class `Electrical_fault`, whose defaults relate to faults in vacuum cleaners (Figure 5.3a).

Suppose that we wish to save an instance of fixing a refrigerator, where the refrigerator exhibited the following symptoms:

- Failed to chill food.
- Made no noise.
- The light would not come on.

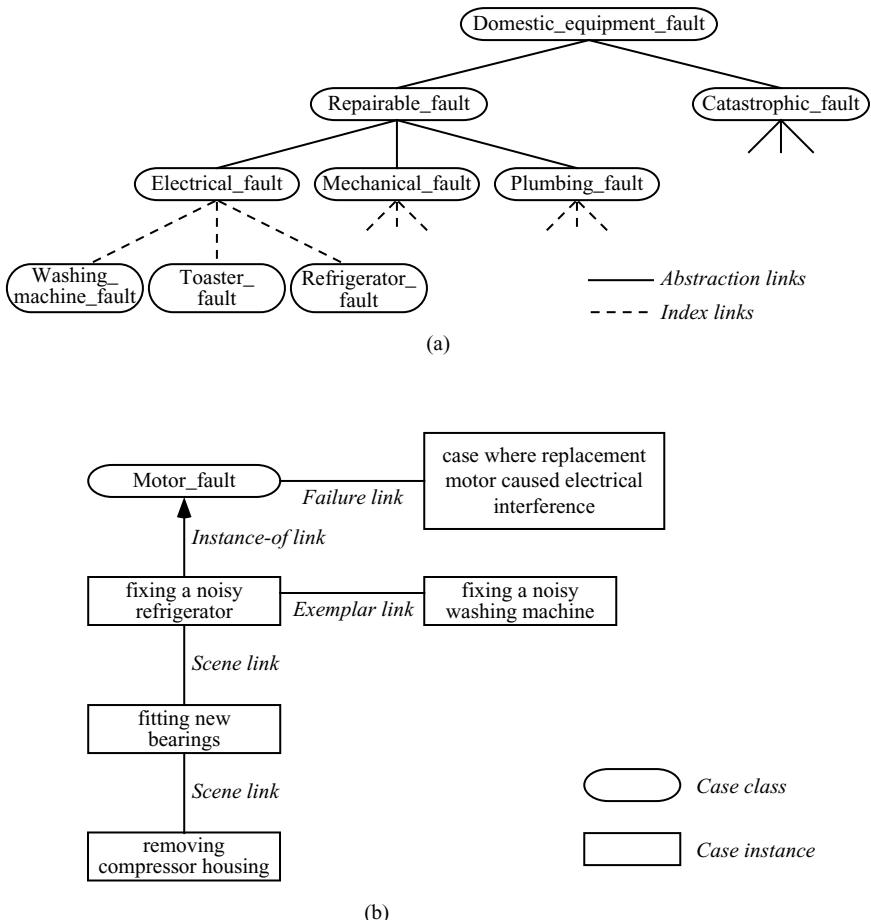


Figure 5.3 Classifying case histories: (a) abstraction links and index links between classes; (b) links between instances.

This case might be stored as an instance of the class `Refrigerator_fault`, which, as already noted, has an index link to the class `Electrical_fault`.

5.3.1.2 Instance-Of Links

Each case is an instance of a specific class of cases (see Chapter 4 for a discussion of object classes and instances). Thus, each case has an *instance-of* relation with its parent class, as shown in Figure 5.3b.

5.3.1.3 Scene Links

Scene links are used to link a historical event to its subevents. For example, removing the outer casing of a refrigerator's compressor is a subevent of the “changing

compressor bearings” event. Both the event and subevent are stored as case instances, with a scene link between them (Figure 5.3b).

5.3.1.4 Exemplar Links

Exemplar links are used to link instances to other similar instances. Suppose that a refrigerator motor fault was diagnosed by referring to a previous case involving a failed washing machine motor. This new case could have an exemplar link to the case from which it was adapted (Figure 5.3b).

5.3.1.5 Failure Links

A failure link is a special type of class–instance relation, where the instance represents a specific case where things did not turn out as expected. It is a convenient way of storing cases that form exceptions to their general category. For example, the class `Motor_fault` might have a failure link to a case in which a replacement motor created a new problem, such as radio interference (Figure 5.3b).

5.3.2 Retrieving Cases

The problem of retrieving cases that match a new case, termed the *probe* case, is eased if the case base is carefully indexed using links, as described in the preceding section. Such links are often used as a basis for the first stage of a two-stage interrogation of the case base. The second stage involves ranking the matched cases according to a measure of similarity to the probe case (Ferguso and Bridge 2000).

For a given probe case, a suitable similarity measure S_i for case i in the case base could be:

$$S_i = \sum_{j=1}^P w_j m_{ij} \quad (5.1)$$

where P is the number of parameters considered, w_j is an importance weighting applied to parameter j , and m_{ij} is a degree of match between case i and the probe case for the parameter j . For some parameters, such as the presence or absence of a given observation, m_{ij} is binary, that is, it is either 0 or 1. Other parameters may concern continuous variables such as temperature, or nearly continuous variables such as cost. These can still yield binary values for m_{ij} if the measure is whether the parameter is within an acceptable range or tolerance. Alternatively, a sliding scale between 0 and 1 can be applied to m_{ij} , which is akin to using a fuzzy membership value rather than a crisp one (see Chapter 3).

5.3.3 Adapting Case Histories

There are two distinct categories of techniques for adapting a case history to suit a new situation, namely, structural and derivational adaptation. *Structural* adaptation

describes techniques that use a previous solution as a guide and adapt it to the new circumstances. *Derivational* adaptation involves looking at the *way* in which the previous solution was derived, rather than the solution itself. The same reasoning processes are then applied to the set of data describing the new circumstances, that is, the probe case. Four structural techniques and one derivational technique are outlined below.

5.3.3.1 Null Adaptation

Null adaptation is the simplest approach and, as the name implies, involves no adaptation at all of the previous solution. Instead, the previous solution is given as the solution to the new case. Suppose that the case history selector decides that a failed refrigerator is similar to the case of a failed washing machine. If the washing machine failure was found to be due to a severed power lead, then this same solution is offered for the refrigerator problem.

5.3.3.2 Parameterization

Parameterization is a structural adaptation technique that is applicable when both the symptoms and the solution have an associated magnitude or extent. The previous solution can then be scaled up or down in accordance with the severity of the symptoms. Suppose, for example, that a case history was as follows:

- Symptom: Fridge cabinet temperature is 15°C, which is too warm.
- Solution: Reduce thermostat setting by 11°C.

If our new scenario involves a fridge whose cabinet temperature is 10°C, which is still warmer than it should be, the new solution would be to turn down the thermostat, but by a modified amount (say, 6°C).

5.3.3.3 Reasoning by Analogy

Reasoning by analogy is another structural adaptation technique. If a case history cannot be found in the most appropriate class, then analogous case histories are considered. Given a hierarchically organized database of case histories, the search for analogous cases is relatively straightforward. The search begins by looking at siblings, and then cousins, in a class hierarchy like the one shown in Figure 5.3a. Some parts of the historical solution may not be applicable, as the solution belongs to a different class. Under such circumstances, the inapplicable parts are replaced by referring back to the class of the current problem.

As an example, consider a refrigerator that is found to be excessively noisy. There may not be a case in the database that refers to noisy refrigerators, but there may be a case that describes a noisy washing machine. The solution in that case may

have been that the bearings on the washer's drum were worn and needed replacing. This solution is not directly applicable to the refrigerator, as a refrigerator does not have a drum. However, the refrigerator has bearings on the compressor, and so it is concluded that the compressor bearings are worn and are in need of replacement.

5.3.3.4 Critics

The use of critics has stemmed from work on a planning system called HACKER (Sussman 1975) that could rearrange its planned actions if they were found to be incorrectly ordered (see Chapter 14). The ideas are also applicable to other problems, such as diagnosis (see Chapter 12). Critics are modules that can look at a nearly correct solution and determine what flaws it has, if any, and suggest modifications. In the planning domain, critics would look for unnecessary actions, or actions that make subsequent actions more difficult. Adapting these ideas to diagnosis, critics can be used to "fine-tune" a previous solution so that it fits the current circumstances. For instance, a case-based reasoner might diagnose that the compressor bearings in a refrigerator need replacing. Critics might notice that most compressors have two sets of bearings and that, in this particular refrigerator, one set is fairly new. The modified solution would then be to replace only the older set of bearings.

5.3.3.5 Reinstantiation

The preceding adaptation techniques are all structural, that is, they modify a previous solution. Reinstantiation is a derivational technique, because it involves replaying the *derivation* of the previous solution using the new data. Previously used names, numbers, structures, and components are reinstated to the corresponding new ones. Suppose that a case history concerning a central heating system contained the following abduction to explain from a set of informally stated rules why a room felt cold:

```
if thermostat is set too low
then boiler will not switch on.
```

```
if boiler is not switched on
then radiators stay at ambient temperature.
```

```
if radiators are at ambient temperature
then room will not warm up.
```

Abductive conclusion: thermostat is set too low.

By suitable reinstatement, this case history can be adapted to diagnose why food in a refrigerator is not chilled:

```
if thermostat is set too high
then compressor will not switch on.
```

if compressor is not switched on
 then cabinet stays at ambient temperature.

if cabinet is at ambient temperature
 then food will not be chilled.

Abductive conclusion: thermostat is set too high.

5.3.4 Dealing with Mistaken Conclusions

Suppose that a system has diagnosed that a particular component is faulty, and that this has caused the failure of an electronic circuit. If it is then discovered that there was in fact nothing wrong with the component, or that replacing it made no difference, then the conclusion needs *repair*. Repair is conceptually similar to adaptation, and similar techniques can be applied to modify the incorrect conclusion in order to reach a correct one. If the modification fails, then a completely new solution must be sought. In either case, it is important that the failed conclusion be recorded in the database of case histories with a link to the correct conclusion. If the case is subsequently retrieved in a new scenario, the system will be aware of a possible failure and of a possible way around that failure.

5.4 Summary

Systems that can learn offer a way around the so-called “knowledge acquisition bottleneck.” In cases where it is difficult or impossible to extract accurate knowledge about a specific domain, it is clearly an attractive proposition for a computer system to derive its own representation of the knowledge from examples. We have distinguished between two categories of learning systems—symbolic and numerical—with this chapter focusing on the former. Inductive and cased-based methods are two particularly important types of symbolic learning.

Rule induction involves the generation and refinement of prototype rules from a set of examples. An initial prototype rule can be generated from a template or by hypothesizing a causal link between a pair of observations. The prototype rule can then be refined in the light of new evidence by generalizing, specializing, or rejecting it. Rule induction from numerical systems such as neural networks is also possible, but discussion of this approach is deferred until Chapter 10 on hybrid systems.

CBR involves storing the details of every case encountered, successful or not. A stored case can subsequently be retrieved and adapted for new, but related, sets of circumstances. This model is arguably a good representation of human reasoning. The principal difficulties are in recognizing relevant cases, which necessitates a suitable storage and retrieval system, and in adapting stored cases to the new circumstances. The concepts of CBR have been illustrated here with reference to fault diagnosis,

but CBR has also found a wide range of other applications including engineering sales (Watson and Gardingen 1999), call-center customer support (Cheetham and Goebel 2007), decision support for color matching (Cheetham 2005), planning (Marefat and Britanik 1997), and health sciences (Bichindaritz and Marling 2006).

Further Reading

- Alpaydin, E. 2010. *Introduction to Machine Learning*. 2nd ed. MIT Press, Cambridge, MA.
- Kolodner, J. 1993. *Case-Based Reasoning*. Morgan Kaufmann, San Francisco, CA.
- Langley, P. 1995. *Elements of Machine Learning*. Morgan Kaufmann, San Francisco, CA.
- Leeland, A. M. 2010. *Case-Based Reasoning: Processes, Suitability & Applications*. Nova Science, Hauppauge, NY.
- Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill, New York.
- Richter, M. M. and R. O. Weber. 2013. *Case-Based Reasoning: A Textbook*. Springer-Verlag, Berlin, Germany.
- Watson, I. D. 1997. *Applying Case-based Reasoning: Techniques for Enterprise Systems*. Morgan Kaufmann, San Francisco, CA.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 6

Single-Candidate Optimization Algorithms

6.1 Optimization

We have already seen that symbolic learning by induction is a search process, where the search for the correct rule, relationship, or statement is steered by the examples that are encountered. Numerical learning systems can be viewed in the same light. An initial model is set up, and its parameters are progressively refined in the light of experience. The goal is invariably to determine the maximum or minimum value of some function of one or more variables. This is the process of *optimization*.

Often the optimization problem is considered to be one of determining a minimum, and the function that is being minimized is referred to as a *cost function*. The cost function might typically be the difference, or *error*, between a desired output and the actual output. Alternatively, optimization is sometimes viewed as maximizing the value of a function, known then as a *fitness function*. In fact, the two approaches are equivalent, because the fitness f_i of solution i can simply be taken to be the negation of its cost c_i and vice versa, with the optional addition of a positive constant k that can be chosen to keep both cost and fitness positive:

$$f_i = k - c_i \quad (6.1)$$

An alternative approach is to take the cost and fitness as the reciprocals of each other:

$$f_i = 1/c_i \quad (6.2)$$

Whether the selected relationship between fitness and cost is as shown in Equation 6.1 or 6.2, or some other function, the fitness always increases as the cost

decreases. The term *objective function* embraces both fitness and cost. Optimization of the objective function might mean either minimizing the cost or maximizing the fitness.

Deterministic methods use predictable and repeatable steps, so that the same solution is always found, provided the starting conditions are the same. For example, linear programming provides an algebraic solution to find the optimum combination of linearly scaled variables (Dantzig 1998). For more challenging searches, a computational intelligence approach is required, incorporating stochastic methods. *Stochastic* methods include an element of randomness that prevents guaranteed repeatability. Their benefit is that they can be designed to find a solution that is near optimal within an acceptable timeframe, even when the number of potential solutions is vast.

This chapter addresses single-candidate optimization algorithms. At all stages, there is a single trial solution that the algorithm continuously aims to improve until a stopping criterion is reached. In contrast, genetic algorithms, considered in the next chapter, maintain a *population* of many trial solutions.

6.2 The Search Space

The potential solutions to a search problem constitute the *search space* or *parameter space*. If a value is sought for a single variable, or parameter, the search space is one-dimensional. If simultaneous values of n variables are sought, the search space is n -dimensional. Invalid combinations of parameter values can be either explicitly excluded from the search space or included on the assumption that they will be rejected by the optimization algorithm.

In combinatorial problems, the search space comprises combinations of values, the order of which has no particular significance, provided that the meaning of each value is known. For example, in a steel rolling mill the combination of parameters that describe the profiles of the rolls can be optimized to maximize the flatness of the manufactured steel (Nolle et al. 2002a). Here, each possible combination of parameter values represents a point in the search space. The extent of the search space is constrained by any limits that apply to the variables.

In contrast, permutation problems involve the ordering of certain attributes. One of the best-known examples is the traveling salesperson problem, where the salesperson must find the shortest route between cities of known location, visiting each city only once. This sort of problem has many real applications, such as in the routing of electrical connections on a semiconductor chip. For each permutation of cities, known as a tour, we can evaluate the cost function as the sum of distances traveled. Each possible tour represents a point in the search space. Permutation problems are often cyclic, so the tour ABCDE is considered the same as BCDEA, where the letters represent a city visited.

The metaphor of space relies on the notion that certain points in the search space can be considered closer together than others. In the traveling salesperson example, the tour ABCDE is close to ABDCE, but DACEB is distant from both of them. This separation of patterns can be measured intuitively in terms of the number of pairwise swaps required to turn one tour into another. In the case of binary patterns, the separation of the patterns is usually measured as the *Hamming distance* between them, that is, the number of bit positions that contain different values. For instance, the binary patterns 01101 and 11110 have a Hamming separation of 3.

We can associate a fitness value with each point in the search space. By plotting the fitness for a two-dimensional search space, we obtain a *fitness landscape* (Figure 6.1). Here, the two search parameters are x and y , constrained within a range of allowed values. For higher dimensions of search space, a fitness landscape still exists but is difficult to visualize. A suitable optimization algorithm would involve finding peaks in the fitness landscape or valleys in the cost landscape. Regardless of the number of dimensions, there is a risk of finding a local optimum rather than the global optimum for the function. A global optimum is the point in the search space with the highest fitness. A local optimum is a point whose fitness is higher than all its near neighbors but lower than that of the global optimum.

If neighboring points in the search space have a similar fitness, the landscape is said to be *smooth* or *correlated*. The fitness of any individual point in the search space is, therefore, representative of the quality of the surrounding region. Where neighboring points have very different fitnesses, the landscape is said to be *rugged*. Rugged landscapes typically have large numbers of local optima, and the fitness of an individual point in the search space will not necessarily reflect that of its neighbors.

The idea of a fitness landscape assumes that the function to be optimized remains constant during the optimization process. If this assumption cannot be made, as might be the case in a real-time system, we can think of the problem as finding an optimum in a *fitness seascape* (Mustonen and Lässig 2009, 2010).

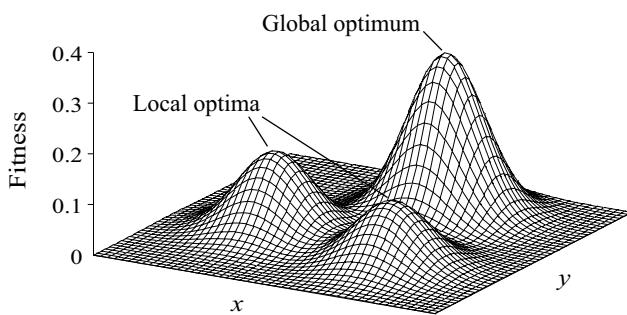


Figure 6.1 A fitness landscape.

In many problems, the search variables are not totally independent from each other. This phenomenon, in which changes in one variable lead to changes in another, is known as *epistasis*, and the search space is said to be *epistatic*.

6.3 Searching the Parameter Space

Determining the optimum for an objective function of multiple variables is not straightforward, even when the landscape is static. Although exhaustively evaluating the fitness of each point in the search space will always reveal the optimum, this is usually impracticable because of the enormity of the search space. Thus, the essence of all numerical optimization techniques is to determine the optimum point in the search space by examining only a fraction of all possible candidates.

The techniques described here are all based upon the idea of choosing a starting point and then altering one or more variables in an attempt to increase the fitness or reduce the cost. All of the methods described in this chapter maintain a single “best solution so far” that is refined until no further increase in fitness can be achieved. Genetic algorithms, which are the subject of Chapter 7, maintain a population of candidate solutions. The overall fitness of the population generally improves with each generation, although some decidedly unfit individual candidates may be added along the way.

The methods described in this chapter proceed in small steps from the start point, so each new candidate solution can never be far from the current one in the search space. The iterative improvement continues until the search reaches either the global optimum or a local optimum. To guard against missing the global optimum, it is advisable to repeat the process several times, starting from different points in the search space.

6.4 Hill-Climbing and Gradient-Descent Algorithms

6.4.1 Hill-Climbing

The name *hill-climbing* implies that optimization is viewed as the search for a maximum in a fitness landscape. However, the method can equally be applied to a cost landscape, in which case a better name might be *valley descent*. It is the simplest of the optimization procedures described here. The algorithm is easy to implement but is inefficient and offers no protection against finding a local minimum rather than the global one. From a randomly selected start point in the search space, which is the initial trial solution, a step is taken in a random direction. If the fitness of the new point is greater than the previous position, it is accepted as the new trial solution. Otherwise the trial solution is unchanged. The process is repeated until the algorithm no longer accepts any steps from the trial solution. At this point, the trial

solution is assumed to be the optimum. As noted in the preceding section, one way of guarding against the trap of detecting a local optimum is to repeat the process many times with different starting points.

6.4.2 Steepest Gradient Descent or Ascent

Steepest gradient descent (or ascent) is a refinement of hill-climbing that can speed the convergence toward a minimum cost (or maximum fitness). It is only slightly more sophisticated than hill-climbing, and it offers no protection against finding a local minimum rather than the global one. From a given starting point, that is, a trial solution, the direction of steepest descent is determined. A point lying a small distance along this direction is then taken as the new trial solution. The process is repeated until it is no longer possible to descend, at which point it is assumed that the optimum has been reached.

If the search space is not continuous but discrete—that is, it is made up of separate individual points—at each step the new trial solution is the neighbor with the highest fitness or lowest cost. The most extreme form of discrete data is where the search parameters are binary, i.e., they have only two possible values. The parameters can then be placed together so that any point in the search space is represented as a binary string and neighboring points are those at a Hamming distance (see Section 6.2) of 1 from the current trial solution.

6.4.3 Gradient-Proportional Descent or Ascent

Gradient-proportional descent, often simply called *gradient descent*, is a variant of steepest gradient descent. Rather than choosing a fixed step size, the size of the steps is allowed to vary in proportion to the local gradient of descent of the cost function. This technique can be applied in a cost landscape that is continuous and differentiable, that is, where the variables can take any value within the allowed range and the cost varies smoothly. Gradient-proportional ascent is equivalent, with the step size proportional to the gradient of ascent of the fitness function.

6.4.4 Conjugate Gradient Descent or Ascent

Conjugate gradient descent (or ascent) is a simple attempt at avoiding the problem of finding a local, rather than global, optimum in the cost (or fitness) landscape. We will consider conjugate gradient descent for cost minimization, but conjugate gradient ascent for fitness maximization is equivalent except that the gradient is climbed instead of descended. From a given starting point in the cost landscape, the direction of steepest descent is initially chosen. New trial solutions are then taken by stepping along this direction, with the same direction being retained until the slope begins to curve uphill. When this happens, an alternative

direction having a downhill gradient is chosen. When the direction that has been followed curves uphill, and all of the alternative directions are also uphill, it is assumed that the optimum has been reached. As the method does not continually hunt for the sharpest descent in the cost landscape, it may be more successful than the steepest gradient descent method in finding the global minimum. However, the technique will never cause a cost gradient to be climbed, even though this would be necessary in order to escape a local minimum and thereby reach the global minimum.

6.4.5 Tabu Search

Tabu search is a refinement of hill-climbing methods, designed to provide some protection against becoming trapped in a local optimum (Cvijovic and Klinowski 1995; Glover and Laguna 1997). It is described by its originators as a *metaheuristic algorithm*, that is, it comprises some guiding rules that sit astride an existing local search method. Its key feature is a short-term memory of recent trial solutions, which are marked as “taboo” or “tabu.” These trial solutions are forbidden, which has the dual benefits, in certain circumstances, of forcing the search algorithm out of a local optimum and avoiding oscillation between alternative trial solutions. Tabu search is applicable to discrete search spaces, that is, where the search variables have discreet values, so that each candidate solution is distinct from its neighbors.

When tabu search is used alongside a standard hill-climbing algorithm, the requirement to accept new solutions only if they represent an improvement may be relaxed at heuristically determined junctures, typically when progress has stalled. Indeed, without this relaxation, previous trial solutions would be effectively taboo in any case, as they would have a lower fitness than the current trial solution. A long-term memory of the best trial solution so far would typically be used to guard against an overall deterioration of fitness.

The set of taboo solutions may be expressed in terms of complete solutions or, alternatively, as partial solutions. Partial solutions are specific attributes within a solution (e.g., a certain value for one attribute, or a particular arc in candidate circuit designs). A drawback of marking partial solutions as taboo is the risk that good complete solutions may become forbidden. This problem can be avoided by *aspiration criteria*, which are extra heuristics that override a trial solution’s taboo status. A typical example would be to allow a solution if it is better than the best trial solution so far, as recorded in the long-term memory.

6.5 Simulated Annealing

Simulated annealing (Kirkpatrick et al. 1983) owes its name to its similarity to the problem of atoms rearranging themselves in a cooling metal. In the cooling metal, atoms move to form a near-perfect crystal lattice, even though they may have to

overcome a localized energy barrier called the activation energy, E_a , in order to do so. The atomic rearrangements within the crystal are probabilistic. The probability P of an atom jumping into a neighboring site is given by:

$$P = \exp(-E_a/kT) \quad (6.3)$$

where k is Boltzmann's constant and T is temperature. At high temperatures, the probability approaches 1, while at $T = 0$ the probability is 0.

In simulated annealing, a trial solution is chosen, and the effects of taking a small random step from this position are tested. If the step results in a reduction in the cost function, it replaces the previous solution as the current trial solution. If it does not result in a cost saving, the solution still has a probability P of being accepted as the new trial solution given by:

$$P = \exp(-\Delta E/T) \quad (6.4)$$

This function is shown in Figure 6.2a. Here, ΔE is the increase in the cost function that would result from the step and is, therefore, analogous to the activation energy in the atomic system. There is no need to include Boltzmann's constant, as ΔE and T no longer represent real energies or temperatures.

The temperature T is simply a numerical value that determines the stability of a trial solution. If T is high, new trial solutions will be generated continually. If T is low, the trial solution will move to a local or global cost minimum, if it is not there already, and will remain there. The value of T is initially set high and is periodically reduced according to a cooling schedule. A commonly used simple cooling schedule is as follows:

$$T_{t+1} = \alpha T_t \quad (6.5)$$

where T_t is the temperature at step number t , and α is a constant close to, but below, 1.

While T is high, the optimization routine is free to accept many varied solutions, but as it drops, this freedom diminishes. At $T = 0$, the method is equivalent to the hill-climbing algorithm, as shown in Figure 6.2b. So, at the start of the process, while T is high, the trial solution can roam around the whole of the search space in order to find the regions of highest fitness. This initial *exploration* phase is followed by *exploitation* as T cools, that is, a detailed search of the best region of the search space identified during exploration.

If the optimization is successful, the final solution will be the global minimum. The success of the technique is dependent upon values chosen for starting temperature, the rapidity of the cooling schedule, and the size of perturbations applied to the trial solutions. A flowchart for the simulated annealing algorithm is given in Figure 6.3.

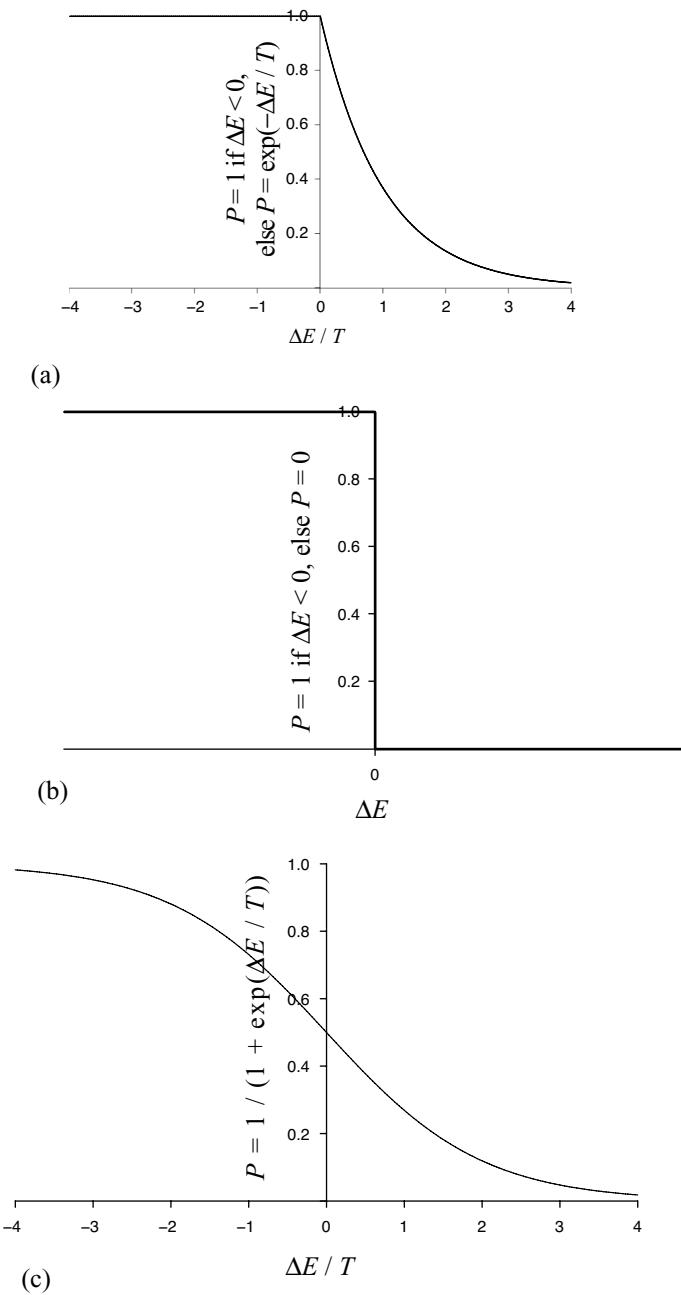


Figure 6.2 Three probability functions.

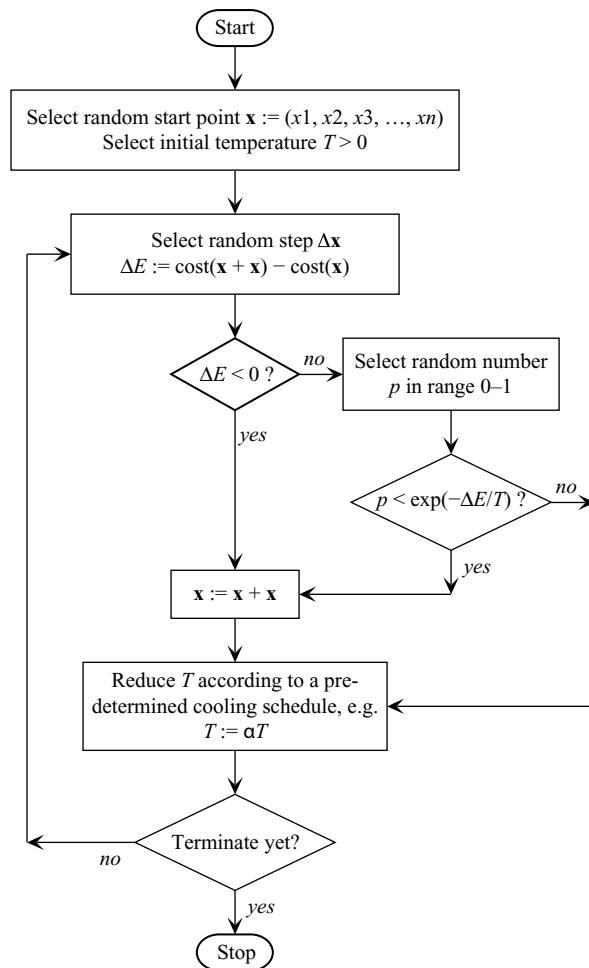


Figure 6.3 Simulated annealing.

Johnson and Picton (1995) have described a variant of simulated annealing in which the probability of accepting a trial solution is always probabilistic, even if it results in a decrease in the cost function (Figure 6.2c). Under their scheme, the probability of accepting a trial solution is as follows:

$$P = \frac{1}{1 + \exp(\Delta E/T)} \quad (6.6)$$

where ΔE is positive if the new trial solution increases the cost function, or negative if it decreases it. In the former case P is in the range 0–0.5, and in the latter case it is in the range 0.5–1. At high temperatures, P is close to 0.5 regardless of the fitness of the new trial solution. As with standard simulated annealing, at $T = 0$ the method becomes equivalent to the hill-climbing algorithm, as shown in Figure 6.2b.

Problems that require the optimum combination of many parameters are said to be *multiobjective* optimization problems. In such problems, there is a choice in the direction of the trial step in the multidimensional search space. Improved performance has been reported through the use of algorithms for selecting the step direction (Suman et al. 2010). An additional refinement is the inclusion of a variable step size. A gradual reduction in the maximum allowable step size as the iteration count rises has been shown to lead to fitter solutions in specific applications (Nolle et al. 2005).

Other modifications to simulated annealing include the use of an archive of past trial solutions, which is included alongside the current trial solution in assessing whether to jump to a new trial solution (Bandyopadhyay et al. 2008). Suman and Kumar have provided a review of some of the refinements to the basic simulated annealing algorithm (Suman and Kumar 2006).

Simulated annealing has been successfully applied to a wide variety of problems including electronic circuit design (Kirkpatrick et al. 1983), tuning specialist equipment for plasma diagnostics (Nolle et al. 2002b), cluster analysis of weather patterns (Philipp et al. 2007), allocation of machines to manufacturing cells (Wu et al. 2008), and parameter-setting for a finisher mill in the rolling of sheet steel (Nolle et al. 2002a).

6.6 Summary

This chapter has reviewed some numerical optimization techniques, all of which are based on minimizing a cost or maximizing a fitness. Often the cost is taken as the error between the output and the desired output. These numerical search techniques can be contrasted with a knowledge-based search. A knowledge-based system is typically used to find a path from a known, current state, such as a set of observations, to a desired state, such as an interpretation or action. In numerical optimization, we know the properties we require of a solution in terms of some fitness measure but have no knowledge of where it lies in the search space. The problem is one of searching for locations within the search space that satisfy a fitness requirement.

All the numerical optimization techniques carry some risk of finding a local optimum rather than the global one. An attempt to overcome this is made in simulated annealing by the inclusion of an *exploration* phase during the early stages of the search. In these early stages, while the temperature is high, the algorithm is free to roam the search space seeking good quality regions. As the temperature cools, the

algorithm migrates from exploration to *exploitation*, meaning that it seeks the peak fitness within the region of the current trial solution. This region is assumed, by the exploitation stage, to contain the global optimum.

The next chapter is dedicated to a particular family of optimization algorithms that further develop the concepts of exploration and exploitation, namely genetic algorithms. Whereas all the techniques in this chapter have been based on improving a single trial solution, genetic algorithms maintain a *population* of trial solutions.

Further Reading

- Antoniou, A., and W.-S. Lu. 2021. *Practical Optimization: Algorithms and Engineering Applications*. 2nd ed. Springer, Berlin, Germany.
- Kochenderfer, M. J. and Wheeler, T. A. 2019. *Algorithms for Optimization*. MIT Press, Cambridge, MA.
- van Laarhoven, P. J. M. 2010. *Simulated Annealing: Theory and Applications*. Kluwer, Dordrecht, Netherlands.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 7

Genetic Algorithms for Optimization

7.1 Introduction: Evolutionary Algorithms

Evolutionary algorithms are a special family of optimization algorithms. Instead of striving to improve a single trial solution, like the techniques presented in Chapter 6, they maintain a *population* of candidate solutions. The population as a whole evolves toward the optimum, although the population may contain many poor solutions, especially during the early stages of evolution. In common with simulated annealing, evolutionary algorithms are generally designed to start with an *exploration* phase. During this phase, the population roams the search space seeking good quality regions. As there is a population of candidate solutions, several regions can be explored at the same time. As evolution progresses, the algorithm migrates from exploration to *exploitation*, that is, seeking the peak fitness of the region that is assumed, by this stage, to contain the global optimum.

Genetic algorithms (GAs) are the most popular type of evolutionary algorithm and will be described in detail in this chapter. GAs use only binary representations of candidate solutions, whereas no such restriction applies to the broader category of evolutionary algorithms.

Evolutionary algorithms have been inspired by natural evolution, the process by which successive generations of animals and plants are modified so as to approach an optimum form. Each offspring has different features from its parents; it is not a perfect copy. If the new characteristics are favorable, the offspring is more likely to flourish and pass its characteristics to the next generation, via natural selection. However, an offspring with unfavorable characteristics is likely to die without reproducing. These ideas have been applied to mathematical optimization, where a population of candidate solutions “evolves” toward an optimum (Holland 1975).

The type of reproduction modeled in GAs is sexual, that is, involving two parents. Such reproduction is found in eukaryotes, which include plants, animals, fungi, and protists. In principle, it would be possible to build a GA modeled on prokaryotes, which comprise archaea and bacteria. However, as they reproduce by copying their own cells to produce clones, this asexual reproduction provides reduced scope for introducing genetic diversity.

Each cell of a living organism contains a set of chromosomes that define the organism's characteristics. The chromosomes are made up of genes, where each gene determines a particular trait such as eye color. The complete set of genetic material is referred to as the *genome*, and a particular set of gene values constitutes a *genotype*. The resulting set of traits is described as the *phenotype*.

Each individual in the population of candidate solutions is graded according to its fitness. The higher the fitness of a candidate solution, the greater are its chances of reproducing and passing its characteristics to the next generation. In order to implement an evolutionary algorithm or a GA specifically, the following design decisions need to be made:

- How to use sequences of numbers, known as *chromosomes*, to represent the candidate solutions;
- The size of the population;
- How to evaluate the fitness of each member of the population;
- How to select individuals for reproduction using fitness information (conversely, how to determine which less-fit individuals will not reproduce);
- How to reproduce candidates, i.e., how to create a new generation of candidate solutions from the existing population; and
- When to stop the evolutionary process.

These decisions will be addressed in detail in subsequent subsections but, for now, let us look at the most basic form of GA.

7.2 The Basic Genetic Algorithm

All of the other numerical optimization techniques described in the previous chapter involved storing just one “best so far” candidate solution. In each case, a new trial solution was generated by taking a small step in a chosen direction. GAs are different in both respects. First, a population of several candidate solutions, in the form of chromosomes, is maintained. Second, the members of one generation can be a considerable distance in the search space from the previous generation.

7.2.1 Chromosomes

Each point in the search space can be represented as a unique chromosome, made up of *genes*. Suppose, for example, we are trying to find the maximum value of a fitness function, $f(x, y)$. In this example, the search space variables, x and y , are constrained

to the 16 integer values in the range 0–15. A chromosome corresponding to any point in the search space can be represented by two genes:

x	y
-----	-----

Thus, the point (2, 6) in search space would be represented by the following chromosome:

2	6
---	---

The possible values for the genes are called *alleles*, so there are 16 alleles for each gene in this example. Each position along the chromosome is known as a *locus*; there are two loci in the preceding example. In a GA, the loci are constrained to hold only binary values. (As noted in the previous section, the term *evolutionary algorithm* describes the more general case where this constraint is relaxed.) The chromosome could therefore be represented by eight loci comprising the binary numbers 0010 and 0110, which represent the two genes:

0	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---

Although there are still 16 alleles for the genes, there are now only two possible values (0 and 1) for the loci. The chromosome can be made as long as necessary for problems involving many variables, or where many loci are required for a single gene. In general, there are 2^N alleles for a binary-encoded gene that is N bits wide.

7.2.2 Algorithm Outline

A flow chart for the basic GA is shown in Figure 7.1. In the basic algorithm, the following assumptions have been made:

- The initial population is randomly generated.
- Individuals are evaluated according to the fitness function.
- Individuals are selected for reproduction on the basis of fitness. The fitter an individual, the more likely that individual is to be selected. Further details are given in Section 7.3.
- Reproduction of chromosomes to produce the next generation is achieved by “breeding” between pairs of chromosomes using the crossover operator and then applying a mutation operator to each of the offspring. The crossover and mutation operators are described in the next two sections; the balance between them is another decision facing the GA designer.

7.2.3 Crossover

In crossover, child chromosomes are produced by aligning two parents, picking a random position along their length, and swapping the tails with a probability P_c , known as the crossover probability. An example for an eight-loci

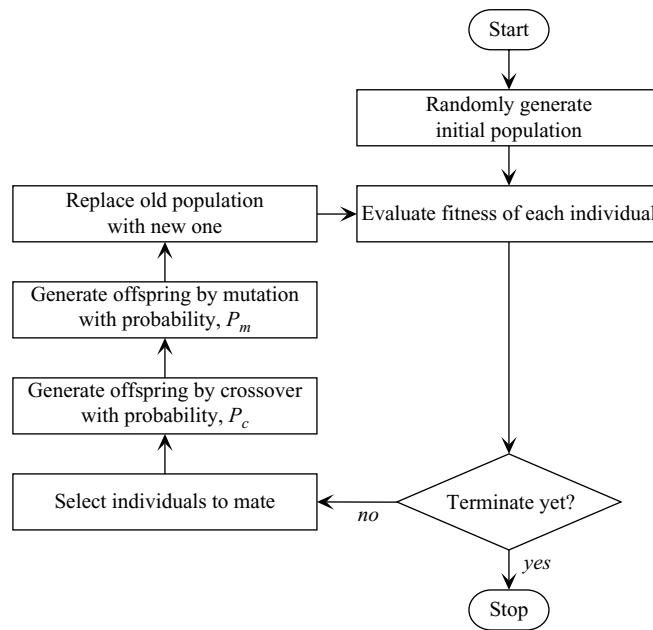
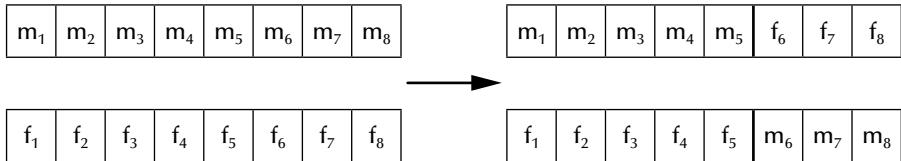
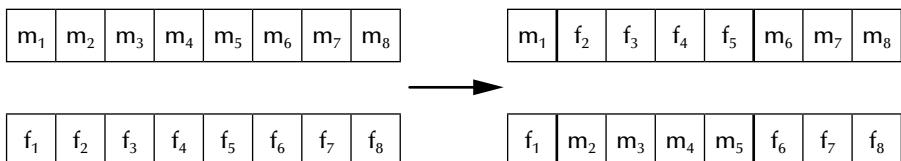


Figure 7.1 The basic GA.

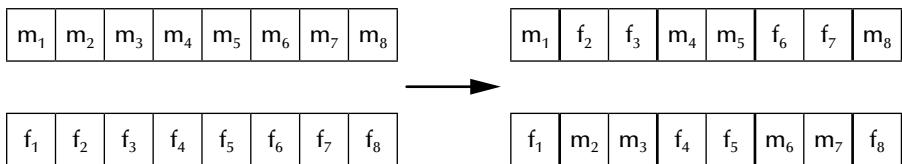
chromosome, where the mother and father genes are represented by m_i and f_p , respectively, would be:



This arrangement is known as single-point crossover, as only one position is specified for separating the swapped and unswapped loci. In fact, this terminology is a misnomer, as a second cross-over position is always required. In single-point crossover, the second crossover position is assumed to be the end of the chromosome. This assumption can be made clearer by considering two-point crossover, where the chromosomes are treated as though they were circular, so that m_1 and m_8 are neighboring loci:



In general, multipoint crossover is also possible, provided there is an even number of crossover points:



In the extreme case, each locus is considered for crossover, independently of the rest, with crossover probability P_c . This arrangement is known as *uniform crossover* (Syswerda 1989).

7.2.4 Mutation

Unlike crossover, mutation involves altering the values of one or more loci. This technique creates new possibilities for the gene combinations that can be generated by crossover. Mutation can be carried out at either the gene level or the locus level:

- At the gene level, a randomly selected gene can be replaced by a randomly generated allele.
- At the locus level, where values are binary, randomly selected loci can be toggled so that 1 becomes 0 and 0 becomes 1.

Genes or loci are selected randomly for mutation with a probability P_m , known as the mutation probability. The main advantage of mutation is that it puts variety into the gene pool, enabling the GA to explore potentially beneficial regions of the search space that might otherwise be missed. This capability helps to counter premature convergence, described in Section 7.3.

7.2.5 Validity Check

Depending on the optimization problem, an additional check may be required to ensure that the chromosomes in the new generation represent valid points in the search space. Consider, for example, a chromosome comprising four genes, each of which can take three possible values: A , B , or C . The binary representation for each gene would require two bits, where each gene has redundant capacity of one extra value. In general, binary encoding of a gene with n alleles requires X bits, where X is $\log_2 n$ rounded up to the nearest integer. Thus, there is redundant capacity of $2^X - n$ values per gene. Using the binary coding $A = 01$, $B = 10$, $C = 11$, a binary chromosome to represent the gene combination BACA would look like this:

1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

A mutation that toggled the last locus would generate an invalid chromosome, since a gene value of 00 is undefined:

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

Similarly, defective chromosomes can also be generated by crossover. In each case, the problem can be avoided by using *structured* operators, whereby crossover and mutation are required to operate at the level of genes rather than loci. Thus, crossover points could be forced to coincide with gene boundaries and mutation could randomly select new values for whole genes. These restrictions would ensure the generation of valid chromosomes, but they also risk producing insufficient variety in the chromosome population.

An alternative approach is to detect and *repair* invalid chromosomes. Once a defective chromosome has been detected, a variety of ways exist to repair it. One approach is to generate “spare” chromosomes in each generation, which can then be randomly selected as replacements for any defective ones.

7.3 Selection

7.3.1 Selection Pitfalls

It has already been stated that individuals are selected for reproduction on the basis of their fitness, that is, the fittest chromosomes have the highest likelihood of reproducing. Selection determines not only which individuals will reproduce, but how many offspring they will have. The selection method can have an important impact on the effectiveness of a GA.

Selection is said to be *strong* if the fittest individuals have a much greater probability of reproducing than less fit ones. Selection is said to be *weak* if the fittest individuals have only a slightly greater probability of reproducing than the less fit ones. If the selection method is too strong, the genes of the fittest individuals may dominate the next generation population even though they may be suboptimal. This outcome is known as *premature convergence*, resulting in the exploitation of a small region of the search space before a thorough exploration of the whole space has been achieved.

On the other hand, if the selection method is too weak, less-fit individuals are given too much opportunity to reproduce and evolution may become too slow. This problem is particularly common during the latter stages of evolution, when the whole population may have congregated within a smooth and fairly flat region of the search space. All individuals in such a region would have similar, relatively high, fitnesses and, thus, it may be difficult to select among them. This outcome is *stalled evolution*, where there is insufficient variance in fitness across the population to drive further evolution.

Some alternative methods of selection, based on a fitness for each member of the population, are now reviewed. The first type of approach, fitness-proportionate selection, is prone to both premature convergence and stalled evolution. The other methods are designed to counter these effects.

7.3.2 Fitness-Proportionate Selection

In this method of selection, an individual's expected number of offspring is proportional to its fitness. The number of times an individual would expect to reproduce is, therefore, equal to its fitness divided by the mean fitness of the population. A method of achieving this, as originally proposed by Holland (1975), is *roulette wheel selection with replacement*. The fitness of each individual is first normalized by dividing it by the sum of fitness values for all individuals in the population to yield its selection probability, P_r . Individuals are then imagined on a roulette wheel, with each one allocated a proportion of the circumference equal to their selection probability. Figure 7.2 illustrates the allocations for a population of eight individuals labeled A–H. In this example, the normalized fitnesses (i.e., selection probabilities) are 0.0625 for A, D, G, and H; 0.125 for B and F; and 0.25 for C and E.

Individuals are selected for reproduction by spinning the notional roulette wheel. This concept is achieved in software by generating a random number in the range 0–1. From a fixed starting point, a notional pointer moves around the wheel by the fraction of a revolution determined by the random number. In the example in Figure 7.2, the random number 0.29 was spun and individual C selected. The pointer is reset at the origin and spun again to select the next individual. To say that selection proceeds “with replacement” means that previously selected individuals remain available for selection with each spin of the wheel. In all, the wheel is spun N times per generation, where N is the number of individuals in the population. The pseudocode for implementing roulette wheel selection with replacement is shown in Box 7.1.

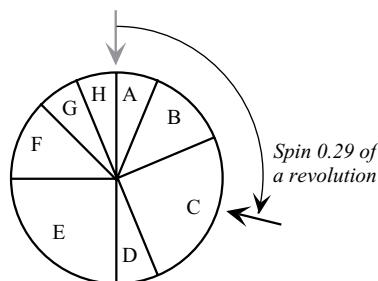


Figure 7.2 Roulette wheel selection with replacement, shown here for a population size of eight and a randomly generated spin of 0.29.

BOX 7.1 PSEUDOCODE IMPLEMENTATION OF ROULETTE WHEEL SELECTION WITH REPLACEMENT (RWSR)

```

***** Part 1: allocate segments of the wheel *****
***** This part is the same for RWSR and SUS *****
/* calculate total fitness */
totalfitness = 0;
for i from 1 to N do /* individual I in population N */
    totalfitness = totalfitness + fitness[i];

/* normalise, then derive cumulative values */
cumulativefitness[0] = 0;
for i from 1 to N do
{
    normalisedfitness[i] = fitness[i] / totalfitness;
    cumulativefitness[i] = cumulativefitness[i-1] +
        normalisedfitness[i]
}
***** Part 2: spinning the wheel *****
/* spin the wheel N times to select parents */
for i from 1 to N do /* for N spins of the wheel */
{
    selector = rand(1); /* random value in range 0-1 */
    for j from 1 to N do /* look around the wheel */
    {
        if cumulativefitness[j] > selector
            and cumulativefitness[j-1] <= selector
            then parent[i] = population[j]
    }
}

```

A drawback of roulette wheel selection is its high degree of variance. An unfit individual could, by chance, reproduce more times than a fitter one. *Stochastic universal selection* (SUS) is a refinement of fitness-proportional selection that overcomes this problem. As with RWSR, individuals are allocated a proportion of the circumference of a wheel according to their fitness value. Rather than a single pointer, there are N equally spaced pointers for a population size N (Figure 7.3). All pointers are moved together around the wheel by the fraction of a revolution determined by the random number. Thus, only one spin of the wheel is required in order to select all reproducing individuals and so the method is less computationally demanding than roulette wheel selection. The pseudocode for implementing SUS is shown in Box 7.2.

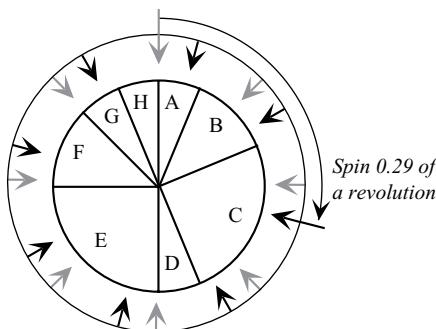


Figure 7.3 Stochastic universal selection, shown here for a population size of eight and a randomly generated spin of 0.29. Gray arrows represent the original pointer positions; black arrows represent the pointer positions after spinning.

BOX 7.2 PSEUDOCODE IMPLEMENTATION OF STOCHASTIC UNIVERSAL SELECTION (SUS).

```

***** Part 1: allocate segments of the wheel *****/
***** This part is the same for RWSR and SUS *****/
/* calculate total fitness */
totalfitness = 0;
for i from 1 to N do /* individual I in population N */
    totalfitness = totalfitness + fitness[i];

/* normalise, then derive cumulative values */
cumulativefitness[0] = 0;
for i from 1 to N do
{
    normalisedfitness[i] = fitness[i] / totalfitness;
    cumulativefitness[i] = cumulativefitness[i-1] +
        normalisedfitness[i]
}
***** Part 2: spinning the wheel *****/
/* spin the wheel only once to select parents */
selector = rand(1); /* random value in range 0-1 */

for i from 1 to N do /* for the N pointers */
{
    pointer = selector + (i-1)/N;
    for j from 1 to N do /* look around the wheel */
    {
        if cumulativefitness[j] > pointer
            and cumulativefitness[j-1] <= pointer
        then parent[i] = population[j]
    }
}

```

7.3.3 Fitness Scaling for Improved Selection

Both variants of fitness-proportionate selection suffer from the risk of premature convergence and, later on, stalled evolution. Fitness-scaling techniques are intended to counter both problems. At the start of evolution, the variations in fitness are typically large, as the initial population is chosen randomly. If selection is too strong, the fittest individuals dominate reproduction even though they may not be near the global optimum, leading to premature convergence. Fitness scaling can be applied at these early stages of evolution to weaken selection and thereby encourage exploration of the whole search space. Fitness scaling counters premature convergence by slowing evolution and maintaining diversity in the early generations.

During the later stages of evolution, a successful algorithm would have finished exploring the search space and would be exploiting the region of the global optimum. The differences between the population fitnesses can be relatively small at this stage, leading to weak selection and stalled evolution. The application of fitness scaling at these late stages of evolution is intended to strengthen the selection pressure in order to converge near the exact optimum. Fitness scaling counters stalled evolution by spreading out the selection rates for the population in the later stages.

Eight approaches to fitness scaling in fitness-proportional selection are described in the following subsections. The subsequent section (Section 7.3.4) presents tournament selection, which has a different basis from fitness-proportional selection. All nine methods are designed to reduce the selection pressure at the start of evolution and to strengthen it in the latter stages.

7.3.3.1 Linear Fitness Scaling

The simplest form of scaling is linear scaling, where the scaled fitness s_i of individual i is scaled linearly with its fitness f_i :

$$s_i = af_i + b \quad (7.1)$$

where a and b are chosen for each generation to either stretch or compress the range of fitnesses. The raw fitness is sometimes referred to as the *evaluation* or *objective* value. The scaled value, which forms the basis of the selection, is referred to as the *selective* value. Selection can proceed by either roulette wheel selection or SUS, where the individuals are now allocated a proportion of the circumference of the wheel according to their selective values instead of their objective values.

The selection probability P_i and the expected number of offspring E_i can be calculated by normalizing the scaled fitness:

$$P_i = \frac{s_i}{\sum_i s_i} \quad (7.2)$$

$$E_i = P_i N \quad (7.3)$$

As defined here, E_i is measured in terms of the amount of genetic material that is passed to the next generation. For instance, an individual of average fitness would have $E_i = 1$, even though this genetic material might be shared among two offspring as a result of crossover.

The key to linear scaling is the selection of suitable values for a and b for each generation. Kreinovich et al. (1993) have demonstrated mathematically that linear scaling is the optimal form of scaling, but this is only a theoretical benefit since it relies on optimal values for a and b being known. In general, these parameters will be altered with successive generations in order to compress the selective fitnesses, s_p , during the early explorative phase and to expand them during the later exploitative phase. Goldberg (1989) has proposed the following formulation for finding suitable values for a and b :

$$a = \frac{(c-1)\bar{f}}{f_{\max} - \bar{f}} \quad (7.4)$$

$$b = \frac{f_{\max} - c\bar{f}}{f_{\max} - \bar{f}} \quad (7.5)$$

where f_{\max} and \bar{f} are the maximum and mean fitness, respectively, of the population and c is an adjustable coefficient.

7.3.3.2 Sigma Scaling

Sigma scaling, also known as *sigma truncation*, is a variant of linear scaling, introduced by Forrest and reported by Mitchell (1996). Here an individual's fitness is scaled according to its deviation from the mean fitness \bar{f} of the population, measured in standard deviations (i.e., "sigma", σ). The scaled fitness s_i for an individual i is given by:

$$s_i = E_i = \begin{cases} 1 + \frac{f_i - \bar{f}}{2\sigma} & \text{if } \sigma \neq 0 \text{ and } f_i \geq \bar{f} - 2\sigma \\ 1 & \text{if } \sigma = 0 \\ 0 & \text{if } \sigma \neq 0 \text{ and } f_i < \bar{f} - 2\sigma \end{cases} \quad (7.6)$$

The first part of this equation is linear and equivalent to Equation 7.1 where:

$$a = \frac{1}{2\sigma} \quad (7.7)$$

and

$$b = \frac{2\sigma - \bar{f}}{2\sigma} \quad (7.8)$$

That is:

$$s_i = \frac{1}{2\sigma} f_i + \frac{2\sigma - \bar{f}}{2\sigma} \quad (7.9)$$

The second and third parts of Equation 7.6 are simply safeguards against anomalies. The second part deals with the case of zero variance, and the third part prevents the scaled fitness from becoming negative. In Tanese's implementation as reported by Mitchell (1996), the equation was modified to ensure that even the least fit individuals had a small chance of reproducing:

$$s_i = E_i = \begin{cases} 1 + \frac{f_i - \bar{f}}{2\sigma} & \text{if } \sigma \neq 0 \text{ and } f_i \geq \bar{f} - 1.8\sigma \\ 1 & \text{if } \sigma = 0 \\ 0.1 & \text{if } \sigma \neq 0 \text{ and } f_i < \bar{f} - 1.8\sigma \end{cases} \quad (7.10)$$

Sigma scaling tends to maintain a fairly consistent contribution to the gene pool from highly fit individuals. During the early stages of evolution, premature convergence is avoided because even the fittest individuals are only two or three standard deviations above the mean fitness. Later on, when the population has converged, stalled evolution is avoided because the diminished value of σ has the effect of spreading out the scaled fitnesses.

7.3.3.3 Boltzmann Fitness Scaling

Boltzmann fitness scaling is a nonlinear method. This technique borrows from simulated annealing the idea of a “temperature” T that drops slowly from generation to generation. The simple Boltzmann scaling function is:

$$s_i = \exp(f_i / T) \quad (7.11)$$

A cooling schedule is used to lower the temperature, typically by multiplying it by a cooling parameter α at each generation, where α is slightly less than 1.

When the temperature is high, this expression serves to compress the scaled fitnesses and, conversely, it spreads them out as the temperature falls as shown in Figure 7.4. When the temperature is high, all chromosomes have a good chance of reproducing as the fittest are only slightly favored over less fit individuals. This compression of fitnesses avoids premature convergence and allows extensive exploration of the search space. As the temperature cools according to the preset schedule, less fit individuals are progressively suppressed, allowing exploitation of what should then be the right part of the search space. Stalled evolution is therefore avoided as well.

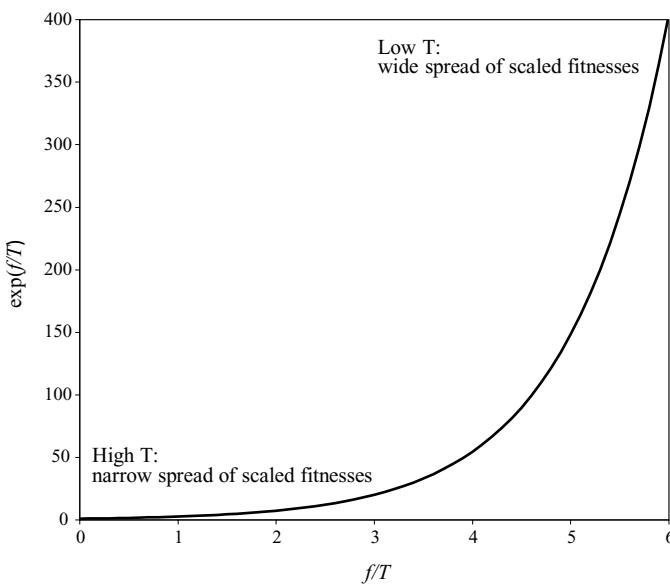


Figure 7.4 Bolzmann scaling of fitness.

7.3.3.4 Linear Rank Scaling

Rank scaling, or rank selection (Baker 1985), offers a fundamentally different approach. It can, nevertheless, be thought of as an alternative way of scaling the fitness of the chromosomes in the population. Instead of being derived from the raw fitness values, the scaled fitness is derived only from the rank ordering of the chromosomes from the fittest to the least fit. In the original form of rank scaling proposed by (Baker 1985), shown in Figure 7.5, the scaled fitness maps linearly to the rank ordering:

$$s_i = \text{Min} + (\text{Max} - \text{Min}) \frac{\text{rank}_i - 1}{N - 1} \quad (7.12)$$

where N is the population size, Min is the fitness for the lowest ranking individual (rank_1) and Max is the fitness for the highest-ranking individual (rank_N). If this expression is normalized so that $s_i = E_p$, and we require that $\text{Max} \geq 0$, then the values of Max and Min are bounded such that $1 \leq \text{Max} \leq 2$ and $\text{Min} = 2 - \text{Max}$.

The likelihood of being selected for reproduction is dependent only on rank within the current population and not directly on the fitness value, except insofar as this value determines rank. This approach helps to avoid both premature convergence and stalled evolution because the spread of scaled fitnesses (i.e., selective values) is maintained at a consistent level, regardless of the distribution of the underlying raw fitnesses (i.e., objective values).

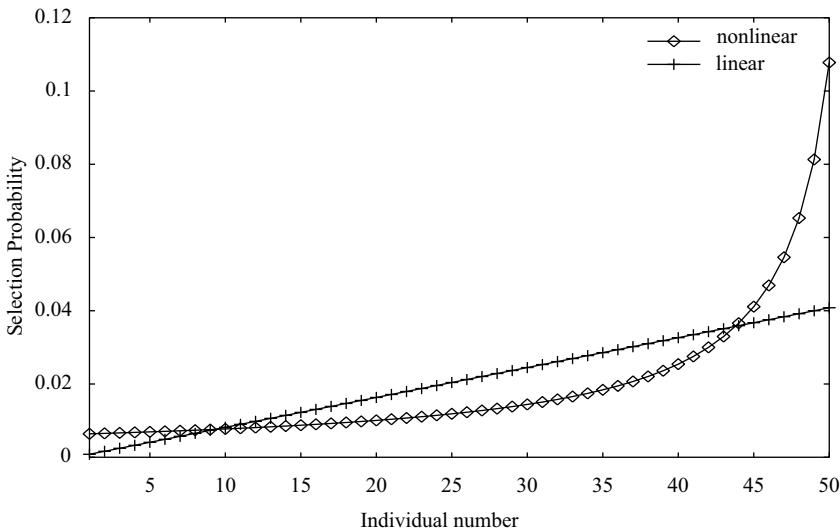


Figure 7.5 Comparison of linear ($\text{Max} = 2.0$, $\text{Min} = 0$) and nonlinear ($c = 3.0$) rank-based selection.

7.3.3.5 Nonlinear Rank Scaling

Baker (1985) also proposed a nonlinear form of rank scaling that increases the selection pressure:

$$s_i = \frac{1}{c} \ln \left(\frac{N - (\text{rank}_i - 1)(1 - e^{-c})}{N - \text{rank}_i (1 - e^{-c})} \right) \quad (7.13)$$

where c is a constant chosen to control the nonlinearity of the function. In Figure 7.5, this nonlinear algorithm is compared with the standard linear rank-based approach. The nonlinear algorithm stretches the distribution of selection probabilities for the fittest individuals, thereby countering the effects of stalled evolution.

7.3.3.6 Probabilistic Nonlinear Rank Scaling

Nolle et al. (1999) have integrated nonlinear rank scaling into roulette wheel selection and SUS. This implementation has been labeled probabilistic nonlinear rank scaling (Hopgood and Mierzejewska 2008) as it integrates nonlinear rank scaling with the probabilistic nature of the roulette wheel:

$$n_i = \text{roundup} \left(\frac{N - Ne^{-cx_i}}{1 - e^{-c}} \right) \quad (7.14)$$

where n_i is the rank of individual i selected for mating, x_i is a random number in the range 0–1 for spinning the roulette wheel (once in the case of SUS, N times in the case of roulette wheel selection), c is a constant that controls the degree of nonlinearity, and roundup is a function that rounds upward to the nearest integer.

7.3.3.7 Truncation Selection

Truncation selection is a variant of rank selection. Here a cutoff rank position $rank_0$ is chosen in advance and the scaled fitness becomes:

$$s_i = \begin{cases} \frac{1}{N - rank_0 + 1} & \text{if } rank_i \geq rank_0 \\ 0 & \text{if } rank_i < rank_0 \end{cases} \quad (7.15)$$

The individuals at or above the cutoff have an equal number of expected offspring, while those below the cutoff do not reproduce at all. Premature convergence is avoided as the fittest individuals have the same reproductive chances as any others that meet the threshold requirement, and stalled evolution is avoided as the method always dispenses with the least fit individuals.

7.3.3.8 Transform Ranking

As shown in Figure 7.5, linear rank scaling ensures an even spread of scaled fitnesses and hence a lower selection pressure than the nonlinear form. Hopgood and Mierzejewska (2008) have, therefore, suggested that linear rank scaling is well-suited to the early stages of evolution, when exploration of the search space is to be encouraged. They have further suggested that nonlinear rank selection is better suited to the later stages of evolution, when exploitation of the optimum is to be encouraged.

Transform ranking is a scaling mechanism that progresses from almost linear to increasingly nonlinear (Hopgood and Mierzejewska 2008). Nolle et al. (1999) have shown that Equation 7.14 is close to linear rank scaling at $c = 0.2$, but becomes highly nonlinear at $c = 3.0$. So, the transition between the two modes can be achieved by a progressive increase in c , analogous to the cooling schedule in Boltzmann scaling. The transition schedule can be either linear:

$$c_{t+1} = c_t + \Delta \quad (7.16)$$

or geometric:

$$c_{t+1} = \frac{(100 + k)c_t}{100} \quad (7.17)$$

where c_t and c_{t+1} are the values of c at successive generations, Δ is the increment added at each generation, and k is a percentage increase at each generation. The method

was designed to benefit from the known performance of both linear and nonlinear rank scaling, with the only foreseen drawbacks being computational cost and the need for careful selection of the transition schedule parameters.

7.3.4 Tournament Selection

In each of the preceding techniques for selection, an individual's expected number of offspring is proportional to either its fitness or a scaled version of its fitness. Tournament selection is a different approach in which individuals are selected on the basis of direct competition between them. The most commonly used form is a binary tournament, in which two individuals are selected at random and their fitnesses evaluated. The fitter of the two is then selected for breeding. Further pairs of individuals are then selected at random from the population of size N , which includes previously selected individuals, until N individuals have been selected for reproduction. The fittest individuals would be expected to be selected several times, thereby increasing their probability of reproduction. The least fit individual in the population can never be selected, as it will always lose any tournament in which it participates.

The victor in a tournament is determined by the rank ordering of the fitnesses of the contestants rather than their absolute values. Fitness scaling would have no effect on tournament selection, as it does not alter the rank ordering of fitnesses. Tournament selection is computationally cheap, as it requires neither the calculation of scaling parameters nor the application of a roulette wheel. Although binary tournaments are the norm, tournaments with more than two competitors are possible.

7.3.5 Comparison of Selection Methods

Some early attempts to assess the effects of fitness scaling are reported in (Goldberg 1989), but these were largely inconclusive beyond demonstrating the benefits of SUS over the original roulette wheel method. Pedersen and Goldberg (2004) have considered badly scaled optimization problems, typically those involving very large and very small coefficients. They recommended that dynamic scaling, using a locally based normalization scheme, be employed in these cases.

Hopgood and Mierzejewska (2008) have presented a systematic comparison of selection methods against two challenging benchmark optimization problems, namely the Schwefel function (Schwefel 1981) and the Griewank function (Griewank 1981). The highest fitness solution was found to be improved through each of the following scaling methods: sigma scaling, linear rank scaling, nonlinear rank scaling, probabilistic nonlinear rank scaling, and transform ranking. Generic linear scaling and Boltzmann scaling were each of benefit for one fitness landscape but not the other. The best improvement of all was achieved by probabilistic nonlinear rank scaling for the Schwefel function and by transform ranking with a linear transform schedule ($\Delta = 0.1$) for the Griewank function. The poor performance of

Boltzmann scaling contrasts with the findings of Sadjadi (2004), who found that Boltzmann scaling led to the most rapid convergence in their tests. Nevertheless, Sadjadi had expressed concern over the method's susceptibility to premature convergence if faced with more complex fitness landscapes, such as those in Hopgood and Mierzejewska's experiments.

All of the fitness scaling methods come at a computational cost that has been evaluated by (Hopgood and Mierzejewska 2008). Tournament selection, on the other hand, was shown to be the computationally cheapest method of all. As noted in the previous subsection, it requires neither the calculation of scaling parameters nor the application of a roulette wheel.

7.4 Elitism

The selection methods described in the previous section have all assumed that the whole of the population is replaced with every generation. Such GAs are said to be *generational*. Even in a generational GA, some individuals may, by chance, be identical to members of the previous generation.

Some researchers have found it useful to modify the basic GA so that it is only partially generational, through the use of *elitism*. Elitism refers to passing one or more of the fittest individuals unchanged through to the next generation. The fittest solutions found so far are, therefore, preserved within the population. Elitism can be thought of as the allowance of cloning alongside reproduction.

The proportion of new individuals in successive generations is termed the *generation gap*. *Steady-state selection* refers to the extension of elitism so that the generation gap is rather small, with only a few of the least fit individuals being replaced on successive generations. They would typically be replaced by crossover and mutation of the fittest individuals.

7.5 Multiobjective Optimization

The treatment so far has assumed that there is a single fitness function to optimize, whereas in fact there may be more than one. For instance, Chapter 13 considers the design of a lightweight beam, where both stiffness and strength criteria are considered. A simple approach to the application of a GA to this sort of problem is to combine the fitnesses into a single function. A multiobjective function could be defined as a weighted sum of the individual fitness functions. So, if the strength of a composite beam were considered twice as important as its stiffness, a suitable function might be:

$$\text{combined fitness} = \frac{(2 \times \text{strength}) + \text{stiffness}}{3} \quad (7.18)$$

The difficulty with this approach is that, although the relative weightings are vital, they are likely to be arbitrary. A more sophisticated view is to recognize that there is often no single best solution. Typically, there is a trade-off between one fitness measure and the others. Rather than a single optimal solution, a set of solutions exists, from which any improvement in one fitness measure would cause a deterioration in the others. This set is said to be *Pareto optimal*. Ideally, an optimization process will identify all members of the Pareto optimal set, leaving selection of the ultimate solution to a further decision process, possibly knowledge-based.

7.6 Gray Code

The structure of the chromosome needs to be designed according to the sort of candidate solution we wish to discover. Choosing a good representation is essential if a GA is to work effectively. Section 7.2 showed a simple case where the objective was to maximize the value of $f(x, y)$ within a specified range of x and y . The chromosome was defined as two genes, one for each of x and y . Each gene was represented by an integer and was encoded in a specified number of bits using standard binary encoding. Some researchers have pointed out that this may not be the most suitable encoding, as consecutive values, such as 7 (binary 0111) and 8 (binary 1000), often have large Hamming separations (Goldberg 1989; Davis 1991). Conversely, values that have small Hamming separations may be far apart in value, for example, 8 (binary 1000) and zero (binary 0000).

An alternative form of encoding is known as Gray code (Gray 1953), shown in Table 7.1. Its key characteristic is that consecutive Gray-coded representations always have a Hamming separation of 1. Therefore, the alteration of a single bit through mutation or crossover would only lead to a small movement through the

Table 7.1 Binary and Gray Codes for Integers 0–15

Denary	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101

(Continued)

Table 7.1 (Continued)

<i>Denary</i>	<i>Binary Code</i>	<i>Gray Code</i>
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

search space. The Gray code, g_i , for integer i is formed from its binary form, b_i , as follows:

$$g_i = b_i \text{ XOR } \text{trunc}\left(\frac{b_i}{2}\right) \quad (7.19)$$

where XOR is the logical exclusive-OR operator, and *trunc* is a function that rounds down its argument to the nearest integer. For example, the Gray code for 7 is (0111 XOR 0011) = 0100 and the Gray code for 8 is (1000 XOR 0100) = 1100.

7.7 Variable-Length Chromosomes

Some researchers have experimented with variable-length chromosomes in the hope of achieving more open-ended solutions. The so-called *messy* GAs are one such approach, using messy chromosomes containing messy genes (Goldberg et al. 1989). In a conventional fixed-length chromosome, the position of a gene on a chromosome determines how it is decoded. This is the *fixed-locus assumption*. In a simple GA with three binary genes, the string 110 would be decoded as:

gene #1 = 1
gene #2 = 1
gene #3 = 0

In a messy chromosome, the position of the genes does not matter as each gene carries an identifier to indicate what it represents. Instead of containing just a value,

each gene comprises two components, shown here bracketed together: (identifier, value). The fixed-length chromosome 110 could be equally represented by any of the following messy chromosomes, all of which are equivalent:

$[(1, 1) (2, 1) (3, 0)]$
 $[(1, 1) (3, 0) (2, 1)]$
 $[(2, 1) (1, 1) (3, 0)]$
 $[(2, 1) (3, 0) (1, 1)]$
 $[(3, 0) (1, 1) (2, 1)]$
 $[(3, 0) (2, 1) (1, 1)]$

This structure means that messy chromosomes may be *over-specified* or *under-specified*. An over-specified chromosome has some duplication of messy genes. For example:

$[(1, 1) (3, 0) (1, 1) (2, 1)]$

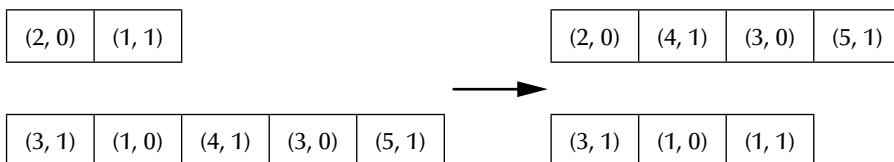
defines gene #1 twice. Over-specification can result in contradictions, shown here for gene #1:

$[(1, 1) (3, 0) (1, 0) (2, 1)]$

Such contradictions are normally handled by accepting the first, or leftmost, instance of the gene. So, in this specific case, the position of the gene *does* matter.

In an under-specified chromosome, such as $[(3, 1) (1, 0)]$, no value is provided for at least one of the genes, namely gene #2. Chromosome $[(1, 1) (3, 0) (1, 0)]$ is both under-specified and over-specified since it lacks gene #2 but contains multiple occurrences of gene #1. Whether under-specification is problematic depends on the nature of the optimization task. In cases where values must be found for each parameter, under-specified chromosomes must be repaired in some way. The unspecified genes may be set to random values, or to the values used in previously discovered fit chromosomes.

The messy GA itself is broadly similar to the standard GA. Tournament selection (see Section 7.3) is usually used to select messy chromosomes for reproduction. Competing individuals are chosen on the basis of their similarity, measured as the number of genes for which both have a value. If the similarity of two individuals is above a specified threshold, a tournament between them is allowed. Mutation is rarely used, and the *cut-and-splice* operator replaces crossover. Each chromosome is cut at a randomly selected point and then chromosome segments from different individuals are spliced together, for example:



7.8 Building Block Hypothesis

7.8.1 Schema Theorem

Holland (1975) has proposed the *building block hypothesis* that successful chromosomes are made up of good quality blocks of genes. He formalized this idea as *schemata* (plural of *schema*), which are sets of gene values that can be represented by a template. For example, the template 1**0 defines a four-bit gene sequence whose first and fourth positions have defined values, and whose other two positions can take any value.

Holland developed the following equation as the basis of his *schema theorem*:

$$n(H,t+1) \geq n(H,t) \frac{\bar{f}(H,t)}{\bar{f}(t)} \left(1 - \frac{P_c d(H)}{l-1}\right) (1 - P_m)^{o(H)} \quad (7.20)$$

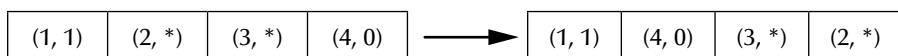
where H is a schema with at least one instance in the last generation; $d(H)$ is the defining length of H , i.e., the distance between crossover points; l is the chromosome length, $n(H, t)$ is the number of instances of H at time t , $\bar{f}(H,t)$ is the mean fitness of H at time t , $\bar{f}(t)$ is the mean fitness of the population at time t ; P_c is the crossover probability; P_m is the mutation probability; and $o(H)$ is the number of defined bits in H , known as the *order* of H . The schema theorem provides a theoretical foundation for GAs, since it can be used to demonstrate that the number of schemata that are fitter than average increases with each generation, while the number of schemata that are less fit than average diminishes (Goldberg 1989).

7.8.2 Inversion

The crossover operator risks breaking up good quality schemata. Holland, therefore, devised a further operator—*inversion*—that may be able to protect good combinations of genes. In practice, it is rarely used. The process of inversion consists of reversing the order of a short section of loci within a chromosome. It is intended to operate on chromosomes whose interpretation is independent of the ordering of the genes. Although not envisaged by Holland, the messy chromosomes described in the previous section provide a good example. The template 1**0 could be represented as the following messy chromosome:

$[(1, 1) (2, *) (3, *) (4, 0)]$

Rearranging the ordering of the messy genes has no effect on their fitness. The inversion operator could be applied to this messy gene to reverse the order of the last three loci:



The effect has been to bring together the two genes with good quality values, namely the genes with identifier 1 and 4. Although the fitness of the chromosome

has been unchanged, these two good quality values are now more likely to be retained as a good quality schema when crossover is applied.

For chromosomes where the order of the genes does affect their interpretation, inversion can be used as a nonsexual means of generating new individuals, additional to mutation. Used in this way, it might be applicable to permutation problems such as the traveling salesperson.

7.9 Selecting GA Parameters

One of the main difficulties in building a practical GA is in choosing suitable values for parameters such as population size, mutation rate, and crossover rate. De Jong's guidelines, as cited in (Mitchell 1996), are still widely followed, namely, to start with:

- a relatively high crossover probability (0.6–0.7);
- a relatively low mutation probability (typically set to $1/l$ for chromosomes of length l); and
- a moderately sized (50–500) population.

Some of the parameters can be allowed to vary. For example, the crossover rate may be started at an initially high level and then progressively reduced with each generation or in response to particular performance measures. Such a variation would promote exploration at the start of evolution and exploitation toward the end.

Given the difficulties in setting the GA parameters, it is unsurprising that many researchers have tried encoding them so that they too might evolve toward optimum values. These *self-adaptive* parameters can be encoded in individual chromosomes, providing values that adapt specifically to the characteristics of the chromosome. Typically, a minimal background mutation rate applies to the population as a whole, and each chromosome includes a gene that encodes a mutation rate to apply to the remaining genes on that chromosome. Self-adaptive parameters do not completely remove the difficulties in choosing parameters, but by deferring the choice to the level of *metaparameters*, that is, the parameters' parameters, the choice may become less critical.

7.10 Monitoring Evolution

There are a variety of measures that can be made at run-time in order to monitor the evolutionary progress of a GA, including:

- Highest fitness in the current population
- Lowest fitness in the current population

- Mean fitness in the current population
- Standard deviation of fitness in the current population
- Mean Hamming separation between randomly selected sample pairs in the current population
- Bitwise convergence, in which the proportion of the population that has the most popular value is calculated for each locus, and then averaged over the whole chromosome

The first four all relate to fitness, while the last two provide measures of similarity between chromosomes.

7.11 Finding Multiple Optima

In certain classes of problems, we may want a population to identify several different optima, referred to as *niches*. In such cases, we must provide a mechanism for decomposing the population into several distinct sub-populations, or *species*. These mechanisms fall into one of four broad categories:

- *Incest prevention*, in which similar individuals are prevented from mating
- *Speciation*, in which only similar individuals may breed together
- *Crowding*, in which each offspring replaces the member in the current population most similar to it
- *Fitness sharing*, in which an individual's selective value is scaled as a decreasing function of its similarity to other members of the population

7.12 Genetic Programming (GP)

Since GAs can evolve numerical populations, Koza et al. (1999) reasoned that it ought to be possible to evolve computer programs in a similar way. This concept has led to the research field of genetic programming (GP). The idea is that hierarchically arranged computer programs can be automatically generated to solve a specified problem.

The building blocks of GP are *functions* and *terminals*. Functions are simple pieces of program code that take one or more arguments, perform an operation on them, and return a value. Terminals are constants or variables that can be used as arguments to the functions. An initial population of hierarchically arranged programs is generated by randomly combining functions and terminals. The fitness of these primitive programs for the specified task can be evaluated, and the fittest individuals selected for reproduction. Crossover is achieved by swapping branches of the hierarchical programs between parents. Mutation may occur by changing the terminals, or by replacing a function with another that takes the same number of

arguments. If a particularly useful combination of functions and terminals is discovered, it can be nominated as a function in its own right, thereby preserving it within the population.

7.13 Other Forms of Population-Based Optimization

Swarm intelligence was described in Section 4.5 as a form of multiagent system. This field of work has been inspired by the observation that large numbers of simple agents in nature, such as ants, can achieve an apparently intelligent overall behavior. Swarm optimization applies these principles to optimization problems. Ant colony optimization is a specific example, based on the idea of reinforcement of good solutions through the addition of a pheromone, as described in Section 4.5. The volatility of the pheromone ensures that solutions lose their attractiveness if they become suboptimal as a result of external changes.

Artificial immune systems (AISs) are another category of biologically inspired algorithms. They are not as explicitly defined as GAs, as is clear from the following definition (de Castro and Timmis 2003):

Artificial immune systems are defined as computational systems inspired by theoretical immunology and observed immune functions, principles and models, applied to solve problems.

Within this broad definition, many forms of AIS exist, which de Castro and Timmis have divided into population-based and network-based algorithms. The applications of AISs are also left loosely specified in this definition. As well as optimization, AISs have been used to tackle a range of problems including control, diagnosis, and anomaly detection (Klarreich 2002). A key strength of AIS is their adaptability to new or unforeseen circumstances, such as partial loss of input data.

7.14 Summary

This chapter has mostly focused on one specific numerical optimization technique, namely GAs. Like the other optimization techniques considered in Chapter 6, GAs are designed to minimize a cost or maximize a fitness.

All optimization techniques, apart from exhaustive search, carry some risk of finding a local optimum rather than the global one. GAs reduce this risk through crossover and mutation, which can allow the algorithm to explore the search space thoroughly during the early stages of evolution. In this exploration phase, the

algorithm is free to roam the search space seeking good quality regions. Nevertheless, the problem may still remain.

There are some occasions where the precise global optimum may not be the best solution, specifically if it is on a knife-edge or narrow peak in the fitness landscape. Design engineers, for example, know that their products can only be manufactured to certain tolerances, so a property specified on a fitness knife-edge would be impractical. For this reason, GAs can be refined so that they favor *robust* optimization, meaning that the fitness of the preferred solution is insensitive to slight deviations from its precise position in the fitness landscape (Lee and Rowlands 2005).

It is rarely necessary to code the optimization algorithms from scratch, as a variety of software packages are available, many free of charge. These packages provide nondomain-specific tools in the same way that an expert system shell does, often as libraries for linking to other software. A GA package typically provides data structures and operators that act on them so that the user has only to define the chromosome structure, fitness function, and a set of parameters covering:

- Population size
- Mutation rate
- Crossover rate
- Number of generations or other termination condition
- Selection method

The main problem in applying GAs to real optimization problems lies in finding a chromosome representation that remains valid after each generation. Nevertheless, the technique has been successfully employed in a wide variety of applications, including automatic scheduling of manufacturing operations (Yamada and Nakano 1992), design of power circuits (Zhang et al. 2006), biological gene expression profiling (To and Vohradsky 2007), protein structure prediction (Wong et al. 2010), and design of communications networks (Davis and Coombs 1987).

Further Reading

- Davis, L., ed. 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York.
- Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.
- Goldberg, D. E., and K. Sastry. 2010. *Genetic Algorithms: The Design of Innovation*. 2nd ed. Springer, Berlin, Germany.

- Haupt, R. L. 2004. *Practical Genetic Algorithms*. 2nd ed. Wiley-Blackwell, London, UK.
- Johnson, J. H., and P. D. Picton. 1995. *Concepts in Artificial Intelligence*. Butterworth-Heinemann, Oxford, UK.
- Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.
- Wirsansky, W. 2020. *Hands-On Genetic Algorithms With Python*. Packt Publishing, Birmingham, UK.

Chapter 8

Shallow Neural Networks

8.1 Introduction

The field of artificial intelligence has been through cycles of enthusiasm and popularity since the field began in the 1950s. Developments in artificial neural networks have been a major factor in those ebbs and flows. There was significant excitement created by the range of neural networks developed in the mid-1980s, which will loosely be described here as shallow networks. These networks remain an important part of the AI toolkit in their own right. They are also the precursor to the deep-learning algorithms that have driven much of the excitement around AI in the 21st century. Deep learning networks are described in Chapter 9. They are an extension of the ideas presented in this chapter, so it is important to understand how these shallow networks work first, before moving on the techniques in Chapter 9.

Artificial neural networks are a family of techniques for numerical learning, like the optimization algorithms reviewed in Chapters 6 and 7, but in contrast to the symbolic learning techniques reviewed in Chapter 5. They consist of many non-linear computational elements that form the network nodes or *neurons*, linked by weighted interconnections. They are analogous in structure to the neurological system in humans and animals, which is made up of real rather than artificial neural networks. Practical artificial neural networks are much simpler than biological ones, so it is unrealistic to expect them to produce the sophisticated behavior of humans or animals. Nevertheless, they are effective at a range of tasks based on pattern matching. Throughout the rest of this book, we will use the expression *neural network* to mean an artificial neural network. The technique of using neural networks is described as *connectionism*. Neural networks typically comprise artificial neurons arranged in layers. The networks described in this chapter have few layers—seldom more than three—and are therefore considered shallow.

Each node in a neural network may have several inputs, each of which has an associated weighting. The node performs a simple computation on its input values, which are single integers or real numbers, to produce a single numerical value as its output. The output from a node can either form an input to other nodes or be part of the output from the network as a whole. The overall effect is that a neural network generates a pattern of numbers at its outputs in response to a pattern of numbers at its inputs. These patterns of numbers are one-dimensional arrays known as *vectors*, for example, (0.1, 1.0, 0.2).

Each neuron performs its computation independently of the other neurons, except that the outputs from some neurons may form the inputs to others. Thus, neural networks have a highly parallel structure that allows them to take advantage of parallel processing computers. They can also run on conventional serial computers, they just take longer to run that way. Neural networks are tolerant of the failure of individual neurons or interconnections. The performance of the network is said to *degrade gracefully* if these localized failures within the network should occur.

The weights on the node interconnections, together with the overall topology, define the output vector that is derived by the network from a given input vector. The weights do not need to be known in advance, but can be learned by adjusting them automatically using a training algorithm. In the case of *supervised* learning, the weights are derived by repeatedly presenting to the network a set of example input vectors along with the corresponding desired output vector for each of them. The weights are adjusted with each iteration until the actual output for each input is close to the desired vector. In the case of *unsupervised* learning, the examples are presented without any corresponding desired output vectors. With a suitable training algorithm, the network adjusts its weights in accordance with naturally occurring patterns in the data. The output vector then represents the position of the input vector within the discovered patterns of the data.

Part of the appeal of neural networks is that, when presented with noisy or incomplete data, they will produce an approximate answer rather than one that is incorrect. This is another aspect of the graceful degradation of neural networks mentioned earlier. Similarly, when presented with unfamiliar data that lie within the range of its previously seen examples, the network will generally produce an output that is a reasonable interpolation between the example outputs. Neural networks are, however, unable to extrapolate reliably beyond the range of the previously seen examples. Interpolation can also be achieved by fuzzy logic (see Chapter 3). Thus, neural networks and fuzzy logic often represent alternative solutions to a particular engineering problem and may be combined in a hybrid system (see Chapter 10).

8.2 Neural Network Applications

Neural networks can be applied to a diversity of tasks, but they all have the common theme of *pattern recognition*. In general, the network associates a pattern in the form of an input vector (x_1, x_2, \dots, x_n) with a particular output vector (y_1, y_2, \dots, y_m),

although the function linking the two may be unknown and may be highly nonlinear. (A linear function is one that can be represented as $f(x) = ax + c$, where a and c are constants; a nonlinear one may include higher order terms for x , or trigonometric or logarithmic functions of x .) The ability to recognize patterns is useful for classification, prediction, clustering, and recall. These application areas are described in the following subsections.

8.2.1 Classification

Often the output vector from a neural network is used to represent one of a set of known possible outcomes, that is, the network acts as a *classifier*. For example, a speech recognition system could be devised to recognize three different words: *yes*, *no*, and *maybe*. The digitized sound of the words would be preprocessed in some way to form the input vector. The desired output vector would then be either $(0, 0, 1)$, $(0, 1, 0)$, or $(1, 0, 0)$, representing the three classes of the words.

Such a network would be trained using a set of examples known as the *training data*. Each example would comprise a digitized utterance of one of the words as the input vector, using a range of different voices, together with the corresponding desired output vector. During training, the network learns to associate similar input vectors with a particular output vector. When it is subsequently presented with a previously unseen input vector, the network selects the output vector that offers the closest match. This type of classification would not be straightforward using nonconnectionist techniques, as the input data rarely correspond exactly to any one example in the training data.

8.2.2 Nonlinear Estimation and Prediction

Neural networks provide a useful technique for estimating the values of variables that cannot be measured easily, but which are known to depend in some complex way on other more accessible variables. The measurable variables form the network input vector, and the unknown variables constitute the output vector. We can call this use *nonlinear estimation*, of which *prediction* is a specific case. The network is initially trained using a set of examples that comprise the training data. Supervised learning is used, so each example in the training data comprises two vectors: an input vector and its corresponding desired output vector. (Some values for the less accessible variable need to have been obtained to form the desired outputs.) During training, the network learns to associate the example input vectors with their desired output vectors. When it is subsequently presented with a previously unseen input vector, the network is able to interpolate between similar examples in the training data to generate an output vector.

The most common use of nonlinear estimation is *prediction*, where the training data are historical associations between inputs and outputs. The trained network can be used to predict an output for a new set of circumstances presented by the input vector.

8.2.3 Clustering

Clustering is a form of unsupervised learning, that is, the training data comprise a set of example input vectors without any corresponding desired output vectors. As successive input vectors are presented, they are clustered into N groups, where the integer N may be prespecified or may be allowed to grow according to the diversity of the data. For instance, digitized preprocessed spoken words could be presented to the network. The network would learn to cluster together the examples that it considered to be in some sense similar to each other. In this example, the clusters might correspond to different words or different voices. Clustering is useful for data compression and is an important aspect of *data mining*, that is, finding patterns in complex data.

Once the clusters have formed, a second neural network can be trained to associate each cluster with a particular desired output. The overall system then becomes a classifier, where the first network is unsupervised and the second one is supervised. As explained in more detail in Section 8.6.2, the combined process is *semi-supervised* learning. Its key advantage is that only a subset of the training data need to be labelled with their corresponding true classification.

8.2.4 Memory and Recall

A content-addressable memory (CAM) is a type of computer memory that can rapidly find stored content. A CAM model can be achieved in a neural network using a form of supervised learning. During training, each example input vector becomes stored in a dispersed form through the network. There are no separate desired output vectors associated with the training data, as the training data represent both the inputs and the desired outputs.

When a previously unseen vector is subsequently presented to the network, it is treated as though it were an incomplete or error-ridden version of one of the stored examples. So, the network regenerates the stored example that most closely resembles the presented vector. This application can be thought of as a type of classification, where each of the examples in the training data belongs to a separate class, and each represents the ideal vector for that class.

This approach is useful when classes can be characterized by an ideal or perfect example. For example, printed text that is subsequently scanned to form a digitized image will contain noisy and imperfect examples of printed characters. For a given font, an ideal version of each character can be stored in a CAM and produced as its output whenever an imperfect scanned version is presented as its input. Thus, a form of automatic character recognition is achieved. Chapter 9 will introduce *generative* networks, which extend these ideas by deliberately introducing variance from the inputs and from the stored information so as to generate new examples.

8.3 Nodes and Interconnections

Each node, or neuron, in a neural network is a simple computing element having an input side and an output side. Each node may have directional connections to many other nodes at both its input and output sides. Each input x_i is multiplied by its associated weight w_i . Typically, the node's role is to sum each of its weighted inputs and add a bias term w_0 to form an intermediate quantity called the *activation*, a . It then passes the activation through a nonlinear function f_t known as the *transfer function* or *activation function*. Figure 8.1 shows the function of a single neuron.

The behavior of a neural network depends on its topology, the weights, the bias terms, and the transfer function. The weights and biases can be learned, and the learning behavior of a network depends on the chosen training algorithm. Typically, a *sigmoid* function is used as the transfer function, as shown in Figure 8.2a. The sigmoid function (von Seggern 2007) is given by:

$$f_t(a) = \frac{1}{1 + e^{-a}} \quad (8.1)$$

For a single neuron, the activation a is given by:

$$a = \left(\sum_{i=1}^n w_i x_i \right) + w_0 \quad (8.2)$$

where n is the number of inputs, and the bias term w_0 is defined separately for each node. Figures 8.2b and 8.2c show the ramp and step functions, which are alternative nonlinear functions sometimes used as transfer functions.

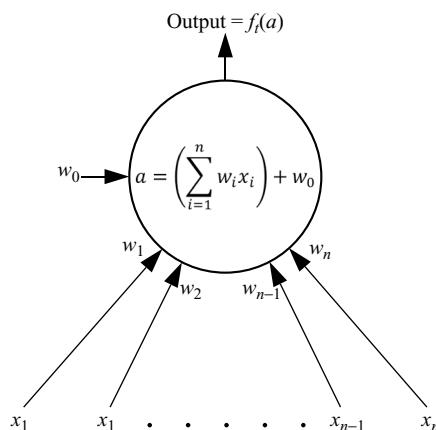


Figure 8.1 A single neuron.

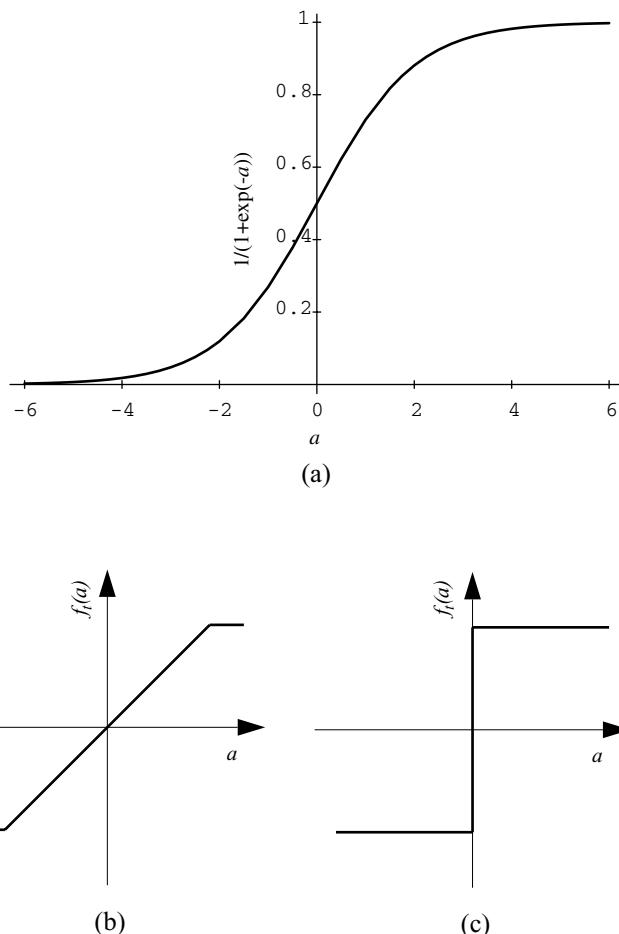


Figure 8.2 Nonlinear transfer functions: (a) a sigmoid function; (b) a ramp function; (c) a step function.

Many network topologies are possible, but we will concentrate on a selection that illustrates some of the different applications for neural networks. We will start by looking at single and multilayer perceptrons (SLPs and MLPs), which can be used for classification or, more generally, for nonlinear mapping. We will then consider the category of recurrent networks, which are used for tasks that include classification of data with a temporal context and acting as a content-addressable memory. The final category comprises unsupervised networks, which are used for clustering.

8.4 Single and Multilayer Perceptrons (SLPs and MLPs)

8.4.1 Network Topology

The topology of an MLP is shown in Figure 8.3. The neurons are organized in layers, such that each neuron is totally connected to the neurons in the layers above and below, but not to the neurons in the same layer. These networks are also called *feedforward networks*, although this term could be applied more generally to any network where the direction of data flow is always “forward,” that is, toward the output. MLPs can be used either for classification or as nonlinear estimators. The number of nodes in each layer and the number of layers are determined by the network builder, often on a trial-and-error basis.

There is always an input layer and an output layer; the number of nodes in each is determined by the number of inputs and outputs being considered. There may be any number of layers between these two layers. Unlike the input and output layers, the layers in between often have no obvious meaning associated with them, and they are known as *hidden layers*. If there are no hidden layers, the network is an SLP. The network shown in Figure 8.3 has three input nodes, two hidden layers with four nodes each and an output layer of two nodes. It can, therefore, be described as a 3–4–4–2 MLP.

An MLP operates by feeding data forward along the interconnections from the input layer, through the hidden layers, to the output layer. With the exception of the nodes in the input layer, the inputs to a node are the outputs from all the nodes in the previous layer. At each node apart from those in the input layer, the data

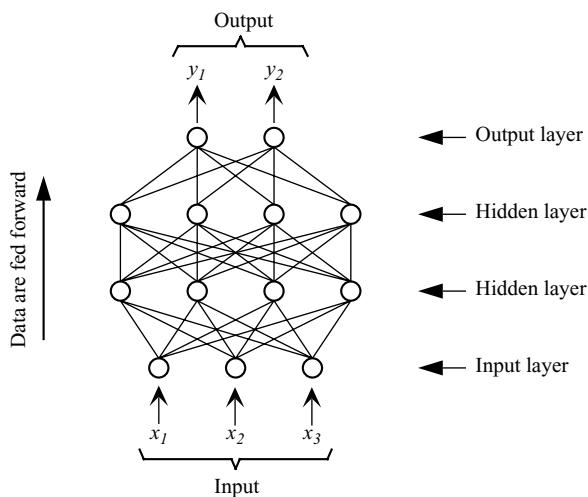


Figure 8.3 A 3–4–4–2 MLP (bias not shown).

are weighted, summed, added to the bias, and then passed through the transfer function.

There is some inconsistency in the literature over the counting of layers, arising from the fact that the input nodes do not perform any processing, but simply feed the input data into the nodes above. Thus, although the network in Figure 8.3 is clearly a four-layer network, it only has three *processing layers*. An SLP has two layers (the input and output layers) but only one processing layer, namely the output layer.

8.4.2 Perceptrons as Classifiers

In general, neural networks are designed so that there is one input node for each element of the input vector and one output node for each element of the output vector. Thus, in a classification application, each output node would usually represent a particular class. A typical representation for a class would be for a value close to 1 to appear at the corresponding output node, with the remaining output nodes generating a value close to 0. A simple decision rule is needed in conjunction with the network, for example, the “*winner takes all*” rule selects the class corresponding to the node with the highest output. If the input vector does not fall into any of the classes, none of the output values may be very high. For this reason, a more sophisticated decision rule might be used, for example, one that specifies that the output from the winning node must also exceed a predetermined threshold such as 0.5.

Instead of having one output node per class, more compact representations are also possible for classification problems. Hallam et al. (1990) have used just two output nodes to represent four classes. This was achieved by treating both outputs together, so that the four possibilities corresponding to four classes are (0, 0), (0, 1), (1, 0), and (1, 1). One drawback of this approach is that it is more difficult to interpret an output that does not closely match one of these possibilities. For example, what would an output of (0.5, 0.5) represent?

Let us now return to the more usual case where each output node represents a distinct class. Consider the practical example of an MLP for interpreting satellite images of the Earth in order to recognize different forms of land use (Hopgood 2005). Figure 8.4 shows a region of the Mississippi Delta, imaged at six different wavebands. The 6–10–5 MLP shown in Figure 8.5 was trained to associate the six waveband images with the corresponding land use. The pixels of the waveband images constitute the inputs and the five categories of land use (water, trees, cultivated, soil/rock, swamp) constitute the outputs. The network was trained pixel-by-pixel on just the top 1/16 of these images and tested against the whole images, 15/16 of which were previously unseen. The results are shown in Figure 8.6. The classification is mostly correct, although some differences between the results and the actual land use can be seen. The neural network performance could certainly be improved with a little refinement, but it has deliberately been left unrefined so

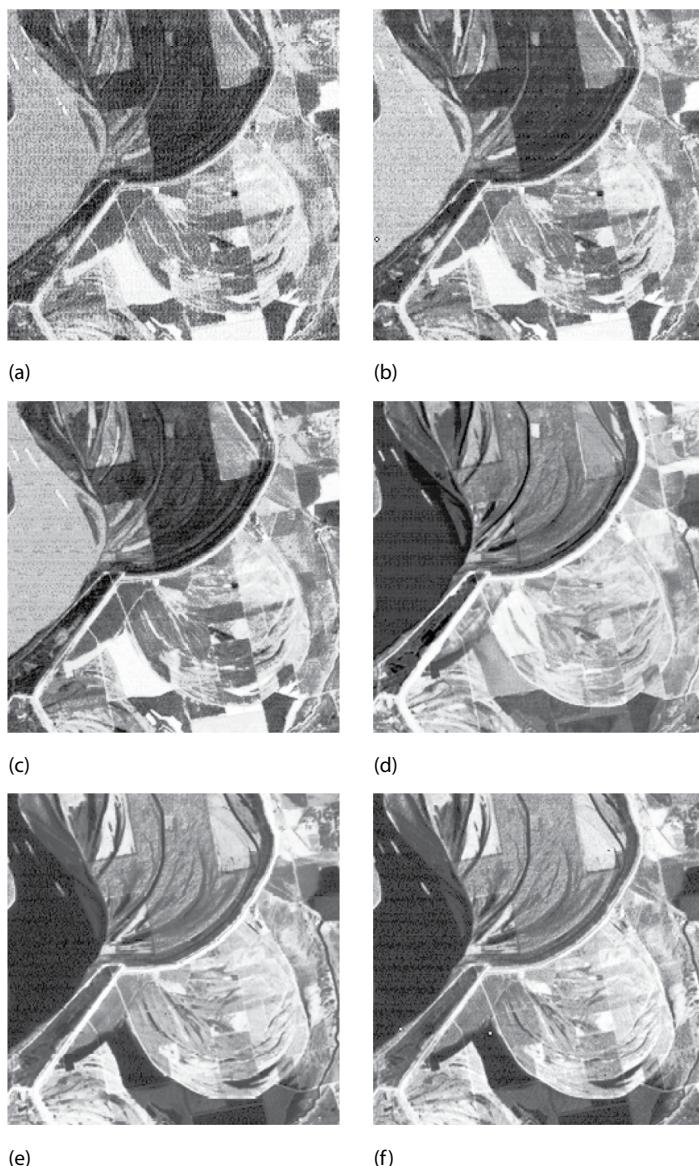


Figure 8.4 Portion of a Landsat-4™ satellite image of an area just to the south of Memphis, Tennessee, taken in six different wavebands. (Source: NASA.)

that these discrepancies can be seen. Nevertheless, the network's ability to generalize from a limited set of examples is clearly demonstrated.

In this example, each input vector has six elements, so it can be represented as a point in six-dimensional *state space*, sometimes called the *pattern space*. Six dimensions

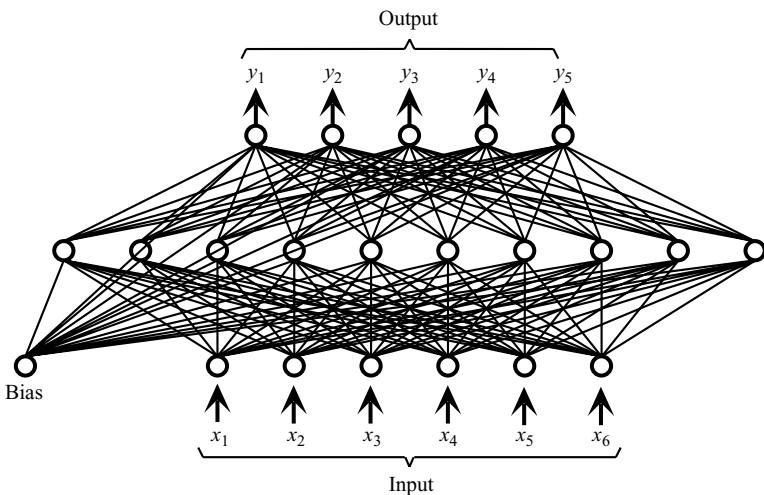


Figure 8.5 A 6–10–5 MLP, with bias represented as an extra node.

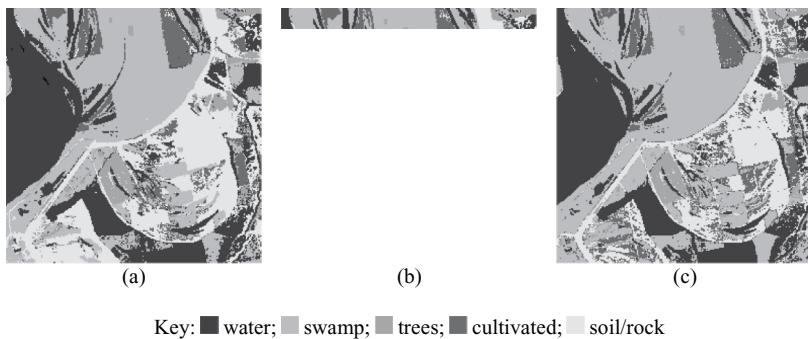


Figure 8.6 Classification results: (a) actual land use map; (b) portion used for training; (c) land use map from neural network outputs. (Derived from Hopgood 2005.)

are hard to visualize but, if the input vector has only two elements, it can be represented as a point in two-dimensional state space. The process of classification is then one of drawing dividing lines between regions. An SLP, with two neurons in the input layer and the same number of neurons in the output layer as there are classes, can associate with each class a single straight dividing line, as shown in Figure 8.7a. Classes that can be separated in this way are said to be *linearly separable*. More generally, n -dimensional input vectors are points in n -dimensional hyperspace. If the classes can be separated by $(n-1)$ -dimensional hyperplanes, they are linearly separable.

To see how an SLP divides up the pattern space with hyperplanes, consider a single processing neuron of an SLP. Its output, prior to application of the transfer function, is a real number given by Equation 8.2. Regions of the pattern space that

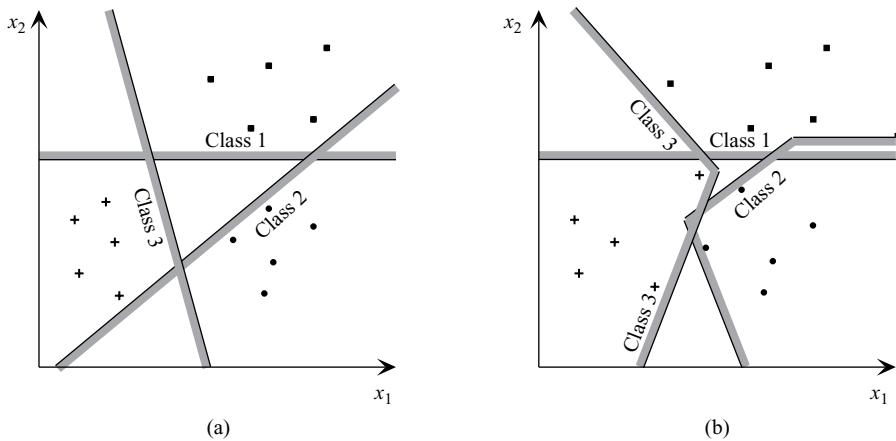


Figure 8.7 Dividing up the pattern space: (a) linearly separable classes; (b) nonlinearly separable classes. Data points belonging to classes 1, 2, and 3 are, respectively, represented by □, ●, and +.

clearly belong to the class represented by the neuron will produce a strong positive value, and regions that clearly do not belong to the class will produce a strong negative value. The classification becomes increasingly uncertain as the activation α becomes close to zero, and the dividing criterion is usually assumed to be $\alpha = 0$. A zero value for α would correspond to an output of 0.5 after the application of the sigmoid transfer function (Figure 8.2a). Thus, the hyperplane that separates the two regions is given by:

$$\left(\sum_{i=1}^n w_i x_i \right) + w_0 = 0 \quad (8.3)$$

In the case of two inputs, Equation 8.3 becomes simply the equation of a straight line, since it can be rearranged as follows:

$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{w_0}{w_2} \quad (8.4)$$

where $-w_1/w_2$ is the gradient and $-w_0/w_2$ is the intercept on the x_2 axis.

For problems that are not linearly separable, as in Figure 8.7b, regions of arbitrary complexity can be drawn in the state space by a multilayer perceptron with one hidden layer and a differentiable, that is, smooth, transfer function such as the sigmoid function (Figure 8.2a). The first processing layer of the MLP can be thought of as dividing up the state space with straight lines (or hyperplanes), and the second processing layer forms multifaceted regions by Boolean combinations (*and*, *or*, and *not*) of the linearly separated regions. It is therefore generally accepted that only

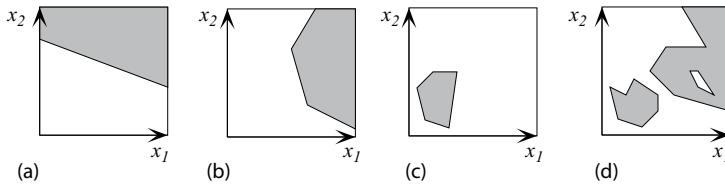


Figure 8.8 Regions in state space distinguished by a perceptron. A convex region has the property that a line joining points on the boundary passes only through that region. (a) No hidden layers: region is a half plane bounded by a hyperplane; (b) one hidden layer with step transfer function: convex open region; (c) one hidden layer with step transfer function: convex closed region; (d) two hidden layers with step transfer function or one hidden layer with smooth transfer function: regions of arbitrary complexity can be defined. (Derived from Rumelhart et al. 1986b.)

one hidden layer is necessary to perform any nonlinear mapping or classification with an MLP that uses a sigmoid transfer function (Figure 8.8). This is *Kolmogorov's Existence Theorem* (Hornick et al. 1989). Similarly, no more than two hidden layers are required if a step transfer function is used. However, the ability to learn from a set of examples cannot be guaranteed and, therefore, the detailed topology of a network inevitably involves a certain amount of trial and error. A pragmatic approach to network design is to start with a small network and expand the number of nodes or layers as necessary.

8.4.3 Training a Perceptron

During training, a multilayer perceptron learns to separate the regions in state space by adjusting its weights and bias terms. Appropriate values are learned from a set of examples comprising input vectors and their corresponding desired output vectors. An input vector is applied to the input layer, and the output vector produced at the output layer is compared with the desired output. For each neuron in the output layer, the difference between the generated value and the desired value is the *error*. The overall error for the neural network is expressed as the square root of the mean of the squares of the errors. This is the *root-mean-squared (RMS)* value, designed to take equal account of both negative and positive errors. The RMS error is minimized by altering the weights and bias terms, which may take many passes through the training data. The search for the combination of weights and biases that produces the minimum RMS error is an optimization problem like those considered in Chapters 6 and 7, where the cost function is the RMS error. When the RMS error has become acceptably low for each example vector, the network is said to have *converged*, and the weights and bias terms are retained for application of the network to new input data.

One of the most commonly used training algorithms is the *back-error propagation algorithm*, sometimes called the *generalized delta rule* (Rumelhart et al. 1986a, 1986b).

This algorithm is a gradient-proportional descent technique (see Chapter 6), and it relies upon the transfer function being continuous and differentiable. The sigmoid function (Figure 8.2a) is a particularly suitable choice, as will be shown in the following text, because its derivative $f'_t(a)$ is simply given by:

$$f'_t(a) = f_t(a)(1 - f_t(a)) \quad (8.5)$$

Remember that $f_t(a)$ is the output y from a neuron when the transfer function f_t is applied to its activation a .

The use of the back-error propagation algorithm for optimizing weights and bias terms can be made clearer by treating the biases as weights on the interconnections from dummy nodes, whose output is always 1, as shown in Figure 8.9. A flowchart describing the back-error propagation algorithm is presented in Figure 8.10, using the nomenclature shown in Figure 8.9.

At the core of the algorithm is the *delta rule* that determines the modifications to the weights, Δw_{Bij} :

$$\Delta w_{Bij} = \eta \delta_{Bi} y_{Aj} + \alpha (\Delta w_{Bij}) \quad (8.6)$$

for all nodes j in layer A and all nodes i in layer B , where $A = B - 1$. Neurons in the output layer and in the hidden layers have an associated error term, δ (pronounced *delta*). When the sigmoid transfer function is used, δ_{Aj} is given by:

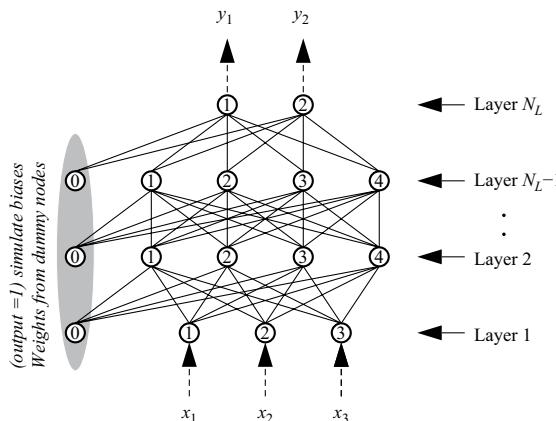


Figure 8.9 Nomenclature for the back-error propagation algorithm in Figure 8.10. N_L = number of layers (4 in this example); w_{Aij} = weight between node i on level A and node j on level $A-1$ ($N_L \geq A \geq 2$); a_{Ai} = activation at node i on level A ; y_{Ai} = output from node i on level A ; δ_{Ai} = an error term associated with node i on level A .

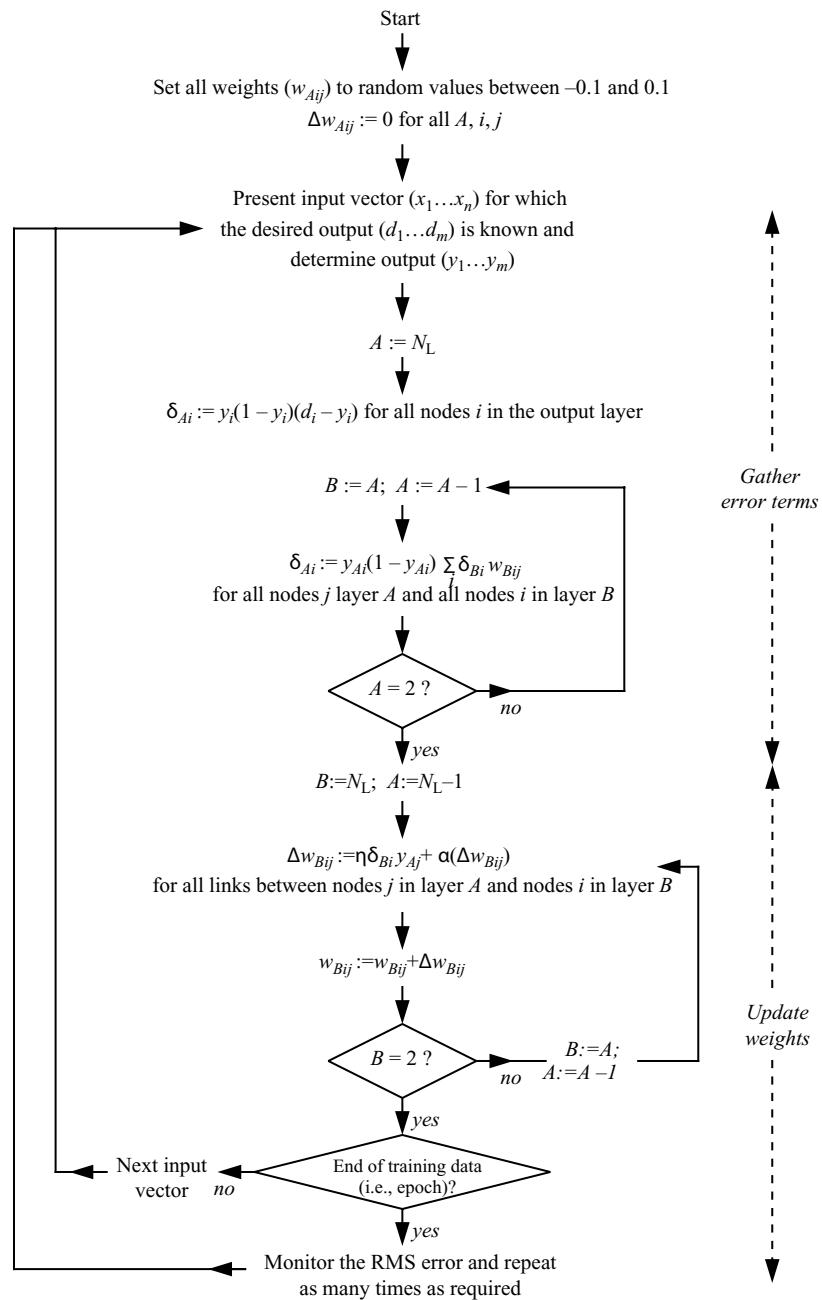


Figure 8.10 The back-error propagation algorithm.

$$\left. \begin{aligned}
 & \text{output layer:} \\
 & \delta_{Ai} = f_t'(a_{Ai})(d_i - y_{Ai}) = y_{Ai}(1 - y_{Ai})(d_i - y_{Ai}) \\
 & \text{hidden layers:} \\
 & \delta_{Aj} = f_t'(a_{Aj}) \sum_i \delta_{Bi} w_{Bij} = y_{Aj}(1 - y_{Aj}) \sum_i \delta_{Bi} w_{Bij}
 \end{aligned} \right\} \quad (8.7)$$

The *learning rate*, η , is applied to the calculated values for δA_j . Knight (1990) suggests a value for η of about 0.35. As written in Equation 8.6, the delta rule includes a *momentum* coefficient, α , although this term is sometimes omitted, that is, α is sometimes set to zero. Gradient-proportional descent techniques can be inefficient, especially close to a minimum in the cost function, which in this case is the RMS error of the output. To address this inefficiency, a momentum term forces changes in weight to be dependent on previous weight changes. The value of the momentum coefficient must be in the range 0–1. Knight (1990) suggests that α be set to 0.0 for the first few training passes and then increased to 0.9.

Other training algorithms have also been successfully applied to perceptrons. For instance, Willis et al. (1991) favor the chemotaxis algorithm, which incorporates a random statistical element in a similar fashion to simulated annealing (see Section 6.5). Other alternative learning algorithms have been developed, for example, Campolucci et al. (1997) and Moallem and Monadjemi (2010), but the back-propagation algorithm remains the most widely used approach.

8.4.4 Hierarchical Perceptrons

In complex problems involving many inputs, some researchers recommend dividing an MLP into several smaller MLPs arranged in a hierarchy, as shown in Figure 8.11. In this example, the hierarchy comprises two levels. The inputs are shared among the MLPs at level 1, and the outputs from these networks form the inputs to an MLP at level 2. This approach is often useful if meaningful intermediate variables can be identified as the outputs from the level 1 MLPs. For example, if the inputs are measurements from sensors for monitoring equipment, the level 1 outputs could represent diagnosis of any faults, and the level 2 outputs could represent the recommended control actions (Kim and Park 1993). In this example, a single large MLP could, in principle, be used to map directly from the sensor measurements to the recommended control actions. However, convergence of the smaller networks in the hierarchical MLP is likely to be achieved more easily. Furthermore, as the constituent networks in the hierarchical MLP are independent from each other, they can be trained either separately or in parallel.

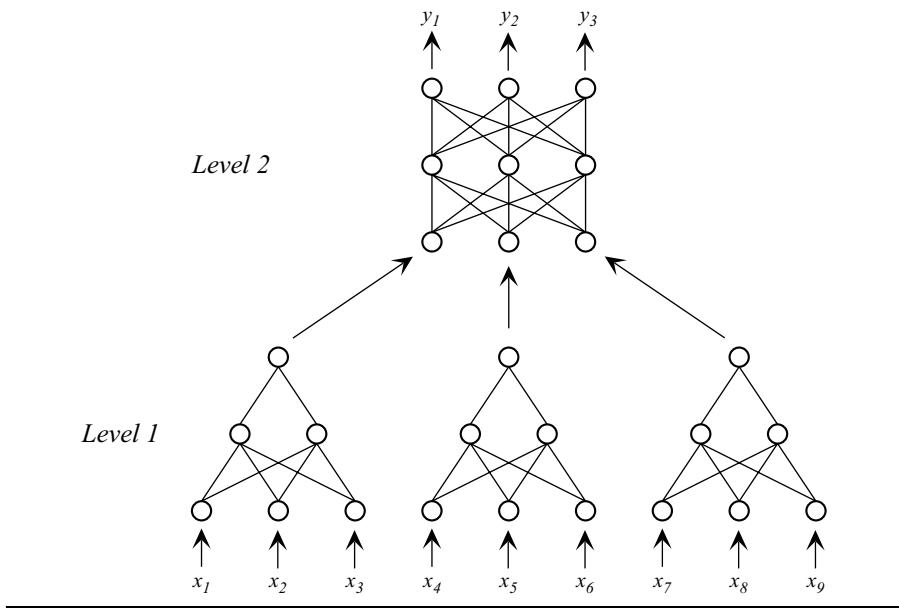


Figure 8.11 A hierarchical MLP.

8.4.5 Buffered Perceptrons

Many forms of classification require *contextual* information, that is, information that either precedes or follows the example being classified. Examples include weather forecasting, where an individual air pressure reading is much more meaningful in the context of whether it is increasing or decreasing, or at a peak or minimum. Similarly, neural networks for classifying speech require contextual information for individual words, and for syllables within words. Neural networks that take account of such time-dependent effects are said to be *dynamic*.

The family of recurrent neural networks (Section 8.5) is specifically designed to be dynamic. One approach to providing contextual information with an MLP is to provide a buffer as part of the input layer. At any one time, the input to the MLP comprises the current input values plus their context, that is, their values at the previous and subsequent time step. At the next time step, all inputs are advanced so that the input at time t becomes part of the context for the input at time $t+1$. This approach was used in the NETtalk system for converting written text to speech (Sejnowski and Rosenberg 1986).

8.4.6 Some Practical Considerations

8.4.6.1 Overtraining

Sometimes it is appropriate to stop the training process before the point where no further reductions in the RMS error are possible. This is because it is possible to

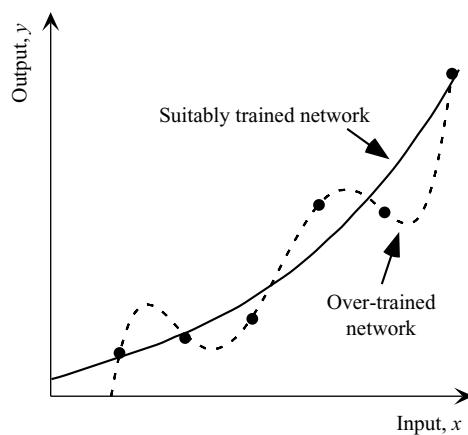
overtrain the network, so that it becomes expert at giving the correct output for the training data, but less expert at dealing with new data. This is likely to be a problem if the network has been trained for too many cycles or if the network is over-complex for the task in hand. For instance, the inclusion of additional hidden layers or large numbers of neurons within the hidden layers will tend to promote over-training. The effect of over-training is shown in Figure 8.12a for a nonlinear mapping of a single input parameter onto a single output parameter, and Figure 8.12b shows the effect of over-training using the nonlinearly separable classification data from Figure 8.7b.

One way of avoiding over-training is to divide the data into three sets, known as the *training*, *testing*, and *validation* data. Training takes place using the training data, and the RMS error with these data is monitored. Additionally, at predetermined intervals, the training is paused and the current weights saved. At these points, before training resumes, the network is presented with the test data and an RMS error calculated. The RMS error for the training data decreases steadily until it stabilizes. However, the RMS error for the test data may pass through a minimum and then start to increase again because of the effect of over-training, as shown in Figure 8.13. As soon as the RMS error for the test data starts to increase, the network is over-trained, but the previously stored set of weights would be close to the optimum. Finally, the performance of the network can be evaluated by testing it using the previously unseen validation data.

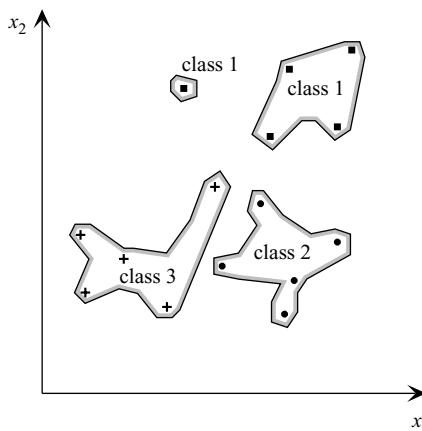
8.4.6.2 Leave-One-Out and K-Fold Cross-Validation

A problem that is frequently encountered in real applications is a shortage of suitable data for training and testing a neural network. Hopgood et al. (1993) describe a classification problem where there were only 20 suitable examples, which needed to be shared between the training and testing data. They used a technique called *leave-one-out* as a way of reducing the effect of this problem. The technique involves repeatedly training on all but one of the examples and testing on the missing one. So, in this case, the network would initially be trained on 19 of the examples and tested on the remaining one. This procedure is repeated a further 19 times, omitting a different example each time from the training data, resetting the weights to random values, retraining, and then testing on the omitted example. The leave-one-out technique is clearly time-consuming, as it involves resetting the weights, training, testing, and scoring the network many times, that is, 20 times in this example. Its advantage is that the performance of the network can be evaluated using every available example as though it were previously unseen test data.

The leave-one-out technique is a special case of the more general *k-fold cross-validation* method for building and evaluating models using sparse data (Wong 2015). The parameter k is the number of groups into which the data are divided. Each unique group is taken in turn as the test set, with the remainder used as the training set. In the case of leave-one-out, k is set equal to N_e , the number of examples in the entire data set.



(a)



(b)

Figure 8.12 The effect of over-training: (a) nonlinear estimation; (b) classification. The symbols •, ▀, and + represent the data points used for training.

8.4.6.3 Data Scaling

Neural networks that accept real numbers are only effective if the input values are constrained to suitable ranges, typically between 0 and 1 or between -1 and 1. The range of the outputs depends on the chosen transfer function, for example, the output range is between 0 and 1 if the sigmoid function is used. In real applications, the actual input and desired output values may fall outside these ranges or may be constrained to a narrow band within them. In either case, the input data and the

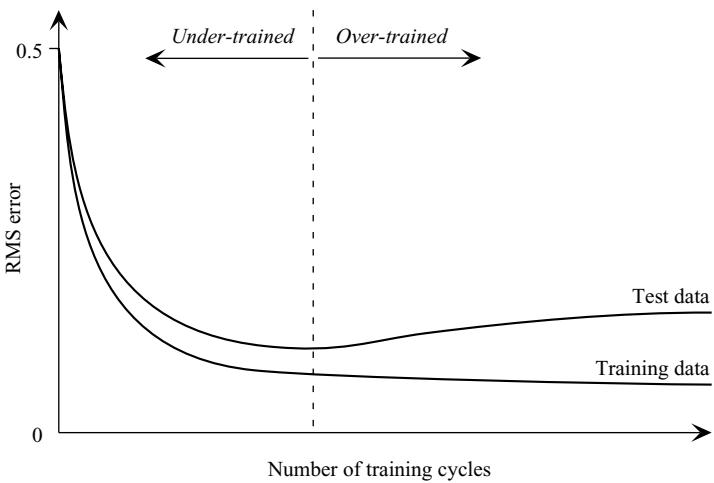


Figure 8.13 RMS error during training.

desired outputs will need to be scaled, usually linearly, before being presented to the neural network. Some neural network packages perform the scaling automatically.

8.5 Recurrent Networks

8.5.1 Simple Recurrent Network (SRN)

The buffered perceptrons, considered in Section 8.4.5, provide temporal context for a particular set of inputs. However, they do so at a cost, as this approach significantly increases the number of connections and requires a prior knowledge of the number of preceding and subsequent inputs that are required to provide the context. In order to reduce the need for an external buffer, several recurrent neural network architectures have been developed. Their distinctive feature is that they contain connections that feedback from the processing layers of the network (i.e., the hidden and/or output layers) to the input layer. These connections can be used to present to the network the recent input history, as the context for the current input.

The classic format is the simple recurrent network (SRN), developed by Elman (1991). The SRN is based on a three-layer perceptron, with additional feedback connections from the hidden layer to the input, as shown in Figure 8.14. These feedback connections link to additional input nodes, whose function is to provide a context, or immediate history, for the current inputs. The initial activations of the context neurons are set to zero, corresponding to an output of 0.5 if the sigmoid transfer function is used. Subsequently, the outputs of the context neurons at time t are copies of the outputs from the hidden layer neurons at time $t-1$. All of the feedback connections must, therefore, have weights equal to 1. As each hidden neuron connects to one context neuron, the SRN must contain the same number of context neurons as hidden neurons.

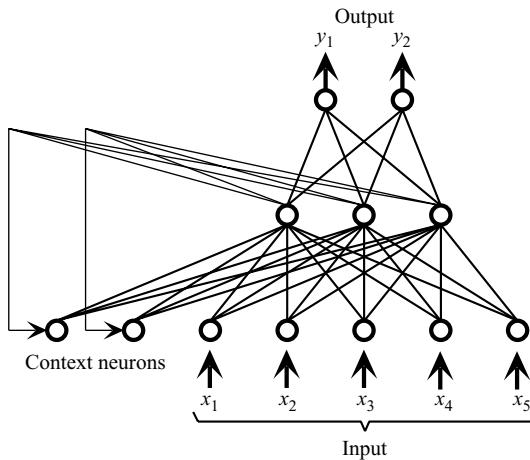


Figure 8.14 Simple recurrent network (SRN).

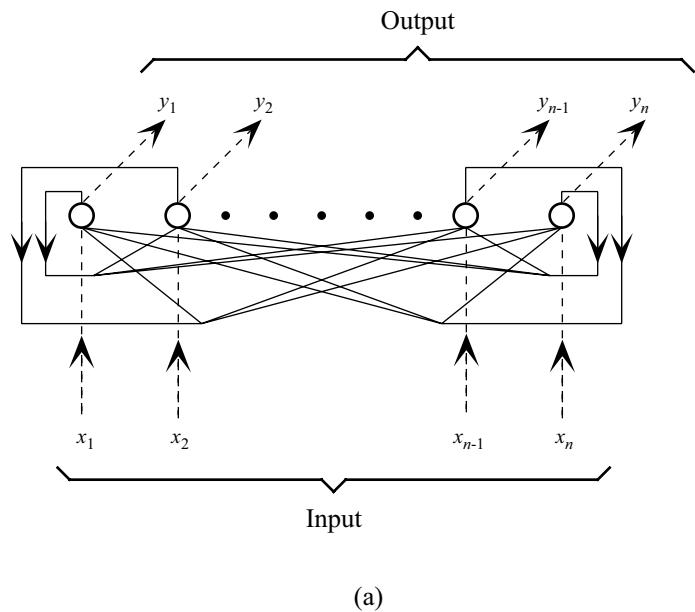
Training of the SRN proceeds in the same manner as an MLP, using the back-error propagation algorithm to adjust the weights on the feedforward connections, while the weights on the feedback connections are left set to 1. The training algorithm adjusts the weights on the connections into the hidden layer to produce the correct context for the next pattern in the sequence. It simultaneously adjusts the weights on the connections into the output layer to produce the desired output for the current input pattern.

SRNs have been applied to various time-dependent processing applications such as the generation of facial expression for robots (Matsui et al. 2009; Matsui et al. 2008) and modeling the intracranial pressure for patients in neurosurgical intensive care (Shieh et al. 2004). However, the majority of applications have concerned the understanding and generation of natural language (Cernanský et al. 2007; Chalup and Blair 2003; Frank 2006).

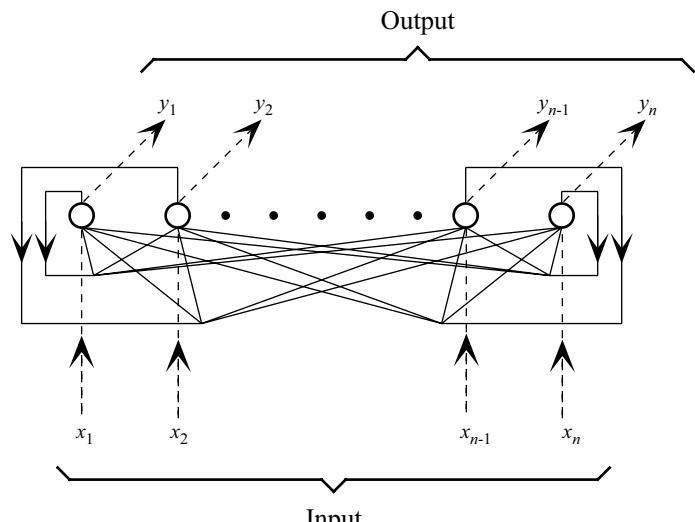
A major drawback with the SRN is that it can only take account of historical context. In the case of natural language and speech processing, this means that the network can take into account the prior utterances, but not the subsequent ones. This weakness was addressed in an alternative architecture by McQueen et al. (2005) called STORM (Spatio Temporal Self-Organizing Recurrent Map). It provides both forward and backward context to a self-organizing map (or SOM, described in Section 8.6.2).

8.5.2 Hopfield Network

Like the SRN, the Hopfield network also has connections from its output to its input, so it is another type of recurrent network. It has only one layer, and the nodes are used for both input and output. The network topology is shown in Figure 8.15a. The network is normally used as a content-addressable memory where each training



(a)



(b)

Figure 8.15 The topology of: (a) the Hopfield network; (b) the MAXNET. Circular connections from a node to itself are allowed in the MAXNET, but they are disallowed in the Hopfield network.

example is treated as a model vector or *exemplar*, to be stored by the network. The Hopfield network uses binary input values, typically 1 and -1. By using the step nonlinearity shown in Figure 8.2c as the transfer function f_i , the output is forced to remain binary too. If the activation a is on the step with a value of zero, the output is indeterminate, so a convention is needed to yield an output of either 1 or -1.

If the network has N_n nodes, then the input and output would both comprise a vector of N_n binary digits. If there are N_e exemplars to be stored, the network weights and biases are set according to the following equations:

$$w_{ij} = \begin{cases} \sum_{k=1}^{N_e} x_{ik} x_{jk} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases} \quad (8.8)$$

$$w_{i0} = \sum_{k=1}^{N_e} x_{ik} \quad (8.9)$$

where w_{ij} is the weighting on the connection from node i to node j , w_{i0} is the bias on node i , and x_{ik} is the i th digit of example k . There are no circular connections from a node to itself, hence $w_{ii} = 0$ where $i = j$.

Setting weights in this way constitutes the learning phase, and results in the exemplars being stored in a distributed fashion in the network. If a new vector is subsequently presented as the input, then this vector is initially the output, too, as nodes are used for both input and output. The node function (Figure 8.1) is then performed on each node in parallel. If this function is repeated many times, the output will be progressively modified and will converge on the exemplar that most closely resembles the initial input vector. In order to store reliably at least half the exemplars, Hopfield (1982) estimated that the number of exemplars, N_e , should not exceed approximately $0.15N_n$.

If the network is to be used for classification, a further stage is needed in which the result is compared with the exemplars to pick out the one that matches.

8.5.3 Maxnet

The MAXNET (Figure 8.15b) has an identical topology to the Hopfield network, except that the weights on the circular interconnections, w_{ii} , are not always zero as they are in the Hopfield network. The MAXNET is used to recognize which of its inputs has the highest value. In this role it is sometimes used in conjunction with other networks, such as a multilayer perceptron, to select the output node that generates the highest value. Suppose that the multilayer perceptron has four output nodes, corresponding to four different categories. A MAXNET to determine the maximum

output value (and, hence, the solution to the classification task) would have four nodes and four alternative output patterns after convergence, that is, four exemplars:

$$\begin{matrix} x & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & x \end{matrix}$$

where $x > 0$. The MAXNET would adjust its highest input value to x , and reduce the others to 0. Note that the MAXNET is constructed to have the same number of nodes (N_n) as the number of exemplars (N_e). Contrast this with the Hopfield network, which needs approximately seven times as many nodes as exemplars.

Using the same notation as Equations 8.8 and 8.9, the interconnection weights are set as follows:

$$w_{ij} = \begin{cases} -\varepsilon & \text{where } \varepsilon < \frac{1}{N_n} \quad \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \quad (8.10)$$

8.5.4 The Hamming Network

The Hamming network has two parts: a twin-layered feedforward network and a MAXNET, as shown in Figure 8.16. The feedforward network is used to compare the input vector with each of the examples, awarding a matching score

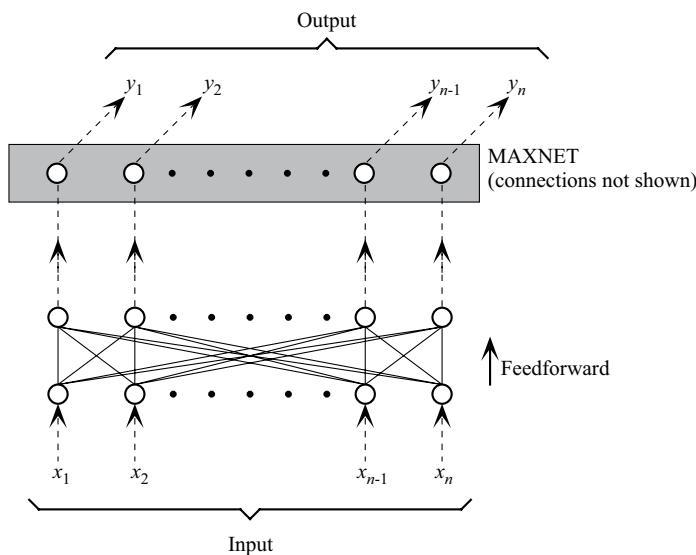


Figure 8.16 The Hamming network.

to each example. The MAXNET is then used to pick out the example that has attained the highest score. The overall effect is that the network can categorize its input vector.

8.6 Unsupervised Networks

8.6.1 Adaptive Resonance Theory (ART) Networks

All of the neural networks introduced so far have used supervised learning, that is, the network has been shown the desired output, and it learns by minimizing the error in relation to the desired outputs. The Adaptive Resonance Theory networks (ART1 and ART2) of Carpenter and Grossberg (1987) are worthy of mention because they are early examples of networks that learn without supervision. The ART1 network topology comprises bidirectional interconnections between a set of input nodes and a MAXNET, as shown in Figure 8.17.

The network classifies the incoming data into clusters. When the first example is presented to the network, it becomes stored as an exemplar or model pattern. The second example is then compared with the exemplar and is either considered sufficiently similar to belong to the same cluster or stored as a new exemplar. If an example is considered to belong to a previously defined cluster, the exemplar for that cluster is modified to take account of the new member. The performance of the

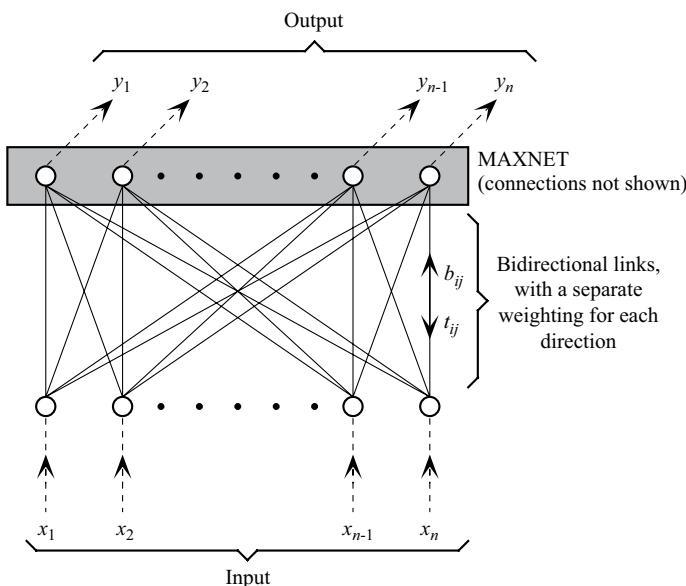


Figure 8.17 The ART1 network.

network is dependent on the way in which the differences are measured, that is, the *closeness* measure, and the threshold or *vigilance*, ρ , beyond which a new exemplar is stored. As each new vector is presented, it is compared with all of the current exemplars in parallel. The number of exemplars grows as the network is used, that is, the network learns new patterns. The operation of the ART1 network, which takes binary inputs, is summarized in Figure 8.18. ART2 is similar but takes continuously varying inputs.

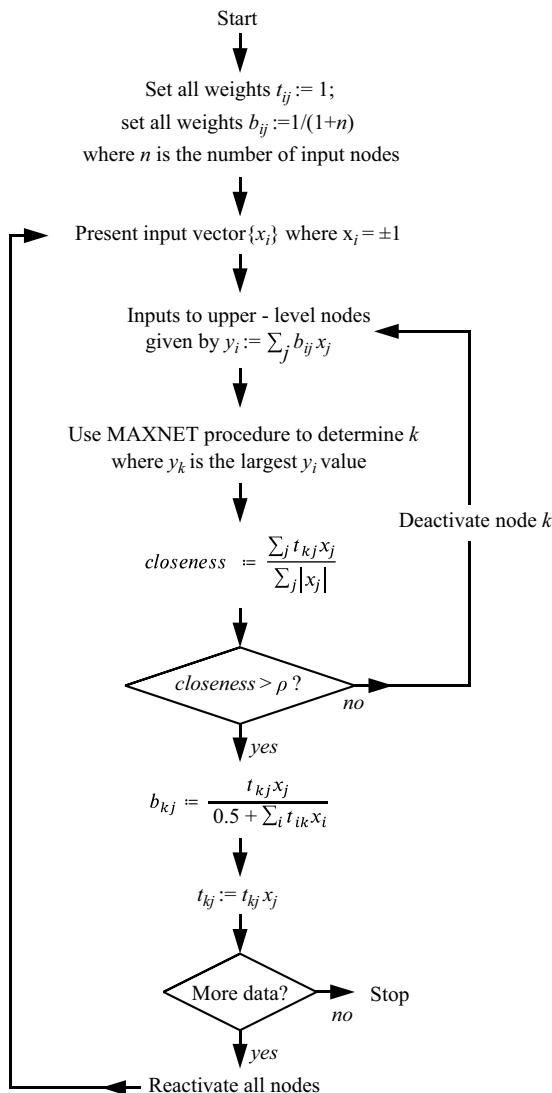


Figure 8.18 Unsupervised learning in ART1.

8.6.2 Kohonen Self-Organizing Networks

Kohonen self-organizing networks, sometimes called self-organizing maps (SOMs), provide another example of networks that can learn without supervision. The processing nodes can be imagined to be arranged in a two-dimensional array, known as the Kohonen layer (Figure 8.19). There is also a separate one-dimensional layer of input nodes, where each input node is connected to each node in the Kohonen layer. As in the MLP, the input neurons perform no processing but simply pass their input values to the processing neurons, with an applied weighting.

As in the ART networks, the Kohonen network learns to cluster together similar patterns. The learning mechanism involves competition between the neurons to respond to a particular input vector (Kohonen 1987, 1988; Hecht-Nielson 1990; Lippmann 1987). The “winner” has its weightings set so as to generate a high output, approaching 1. The weightings on nearby neurons are also adjusted so as to produce a high value, but the weights on the “losers” are left alone. The neurons that are nearby the winner constitute a *neighborhood*.

When the trained network is presented with an input pattern, one neuron in the Kohonen layer will produce an output larger than the others, and it is said to have fired. When a second similar pattern is presented, the same neuron or one in its neighborhood will fire. As similar patterns cause topologically close neurons to fire, clustering of similar patterns is achieved. The effect can be demonstrated by training the network using pairs of Cartesian coordinates. The trained network has the property that the distribution of the firing neurons corresponds with the Cartesian coordinates represented by the input vector. Thus, if the input elements fall in the range between -1 and 1, then an input vector of (-0.9, 0.9) will cause a neuron close to one corner of the Kohonen layer to fire, while an input vector of (0.9, -0.9) would cause a neuron close to the opposite corner to fire.

Although Kohonen self-organizing networks are unsupervised, they can form part of a hybrid network that contains additional layers for supervised learning.

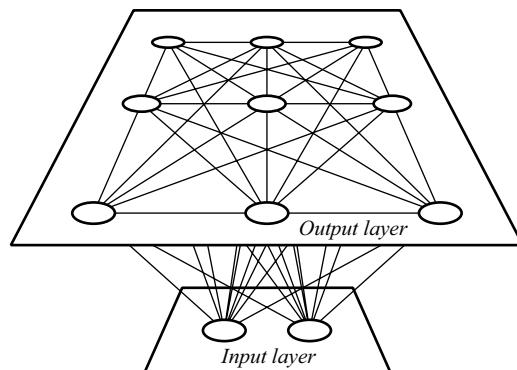


Figure 8.19 A Kohonen self-organizing network.

This structure can be achieved by passing the coordinates of the firing neuron to an MLP. In this arrangement, learning takes place in two distinct phases. First, the Kohonen self-organizing network learns, without supervision, to associate regions in the pattern space with clusters of neurons in the Kohonen layer. Second, an MLP learns to associate the coordinates of the firing neuron in the Kohonen layer with the desired class.

Crucially, the supervised learning phase does not need to use the whole training data set, but just exemplars that are typical of the cluster, such as those near the cluster center. The overall process is described as *semi-supervised learning*. It comprises unsupervised learning to form the clusters, followed by supervised learning on just the cluster exemplars. The benefit is that, in many domains, labeled data are sparse and therefore precious. Labeled, or tagged, data are those data that contain their desired classification, as required for supervised learning. For example, postal services require automated recognition of the characters in handwritten addresses. They may have access to many thousands of examples of scanned handwritten characters, but labeling each of them with the corresponding letter, digit, or special character would be laborious.

8.6.3 Radial Basis Function (RBF) Networks

Radial basis function (RBF) networks offer another alternative method of unsupervised learning. They are feedforward networks, the overall architecture of which is similar to that of a three-layered perceptron, that is, an MLP with one hidden layer. The RBF network architecture is shown in Figure 8.20. The input and output neurons are similar to those of a perceptron, but the neurons in the hidden layer, sometimes called the *prototype* layer, are different. The input neurons do not perform any processing, but simply feed the input data into the nodes above. The neurons in the output layer produce the

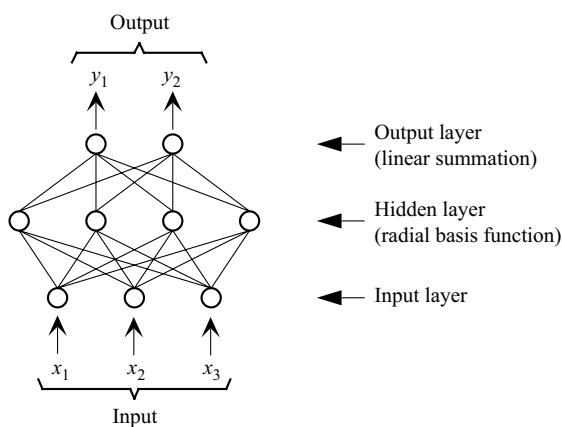


Figure 8.20 A radial basis function network.

weighted sum of their inputs, which is usually passed through a linear transfer function, in contrast to the nonlinear transfer functions used with perceptrons.

The processing neurons considered so far in this chapter produce an output that is the weighted sum of their inputs, passed through a transfer function. However, in an RBF network, the neurons in the hidden layer behave differently. For an input vector (x_1, x_2, \dots, x_n) , a neuron i in the hidden layer produces an output, y_i , given by:

$$y_i = f_r(r_i) \quad (8.11)$$

$$r_i = \sqrt{\sum_{j=1}^n (x_j - w_{ij})^2} \quad (8.12)$$

where w_{ij} are the weights on the inputs to neuron i , and f_r is a symmetrical function known as the RBF. The most commonly used RBF is a Gaussian function:

$$f_r(r_i) = \exp\left(\frac{-r_i^2}{2\sigma_i^2}\right) \quad (8.13)$$

where σ_i is the standard deviation of a distribution described by the function (Figure 8.21). Each neuron, i , in the hidden layer has its own separate value for σ_i .

The Euclidean distance between two points is the length of a line drawn between them. If the set of weights $(w_{i1}, w_{i2}, \dots, w_{in})$ on a given neuron i is treated as the coordinates of a point in pattern space, then r_i is the Euclidean distance from there to the point represented by the input vector (x_1, x_2, \dots, x_n) . During unsupervised learning, the network adjusts the weights—more correctly called *centers* in an RBF network—so

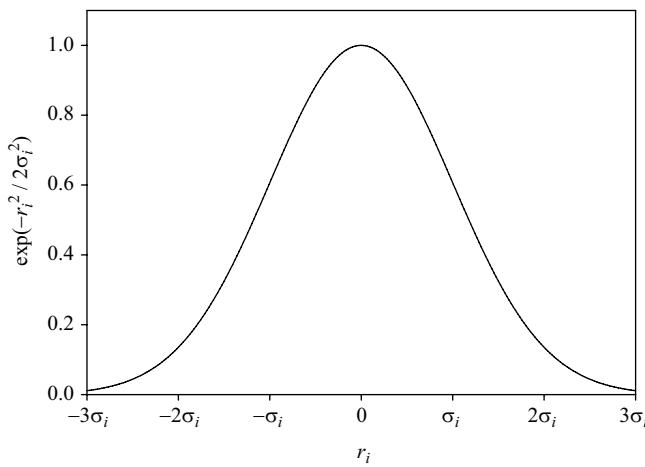


Figure 8.21 Gaussian RBF with standard deviation σ_i .

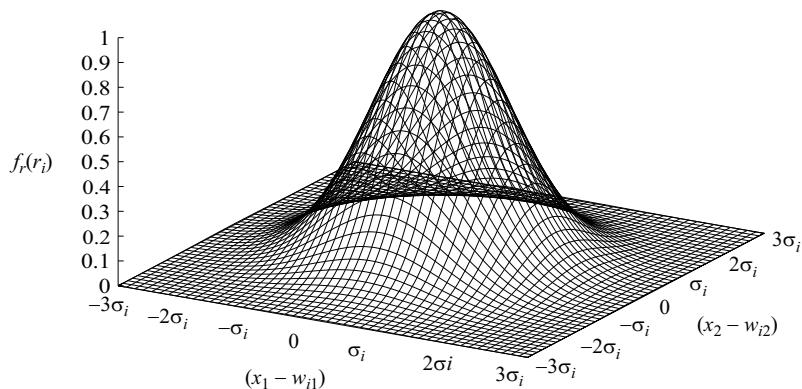


Figure 8.22 Gaussian RBF with standard deviation σ_i applied to two input variables.

that each point $(w_{i1}, w_{i2}, \dots, w_{in})$ represents the center of a cluster of data points in pattern space. Similarly, it defines the sizes of the clusters by adjusting the variables σ_i (or equivalent variables if an RBF other than the Gaussian is used). Data points within a certain range, for example, $2\sigma_i$, from a cluster center, might be deemed members of the cluster. Therefore, just as a single-layered perceptron can be thought of as dividing up two-dimensional pattern space by lines, or n -dimensional pattern space by hyperplanes, so the RBF network can be thought of as drawing circles around clusters in two-dimensional pattern space, or hyperspheres in n -dimensional pattern space. One such cluster can be identified for each neuron in the hidden layer. Figure 8.22 shows a Gaussian function in two-dimensional pattern space, from which it can be seen that a fixed output value (e.g., 0.5) defines a circle in the pattern space.

The unsupervised learning in the hidden layer is followed by a separate supervised learning phase in which the output neurons learn to associate each cluster with

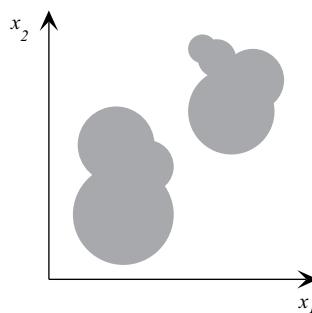


Figure 8.23 RBF networks can define arbitrary shapes for regions in the pattern space.

a particular class. By associating several circular clusters of varying center and size with a single class, arbitrary shapes for class regions can be defined (Figure 8.23).

8.7 Spiking Neural Networks (SNNs)

Maass (1997) has identified three generations of artificial neural networks, to which a fourth generation might be added in the form of the deep-learning networks presented in Chapter 9. The *first generation* of artificial neural networks relates to work in the 1940s and 1950s on simple neurons that fire (i.e., produce a high output value) if the weighted sum of their inputs exceeds a threshold. These neurons were effectively the same as the ones described in Section 8.3 and Figure 8.1, using a step transfer function. A simple learning approach was explored, known as Hebb's law. Weights were increased to make a neuron fire when the input associated with that weight was high.

The first generation of artificial neural networks lacked a feedback mechanism to compare the actual and desired outputs. This deficiency was addressed by the *delta rule* in the 1960s, but it was not until the introduction of the *generalized delta rule* in the 1980s that nonlinear separation of pattern space became possible.

All of the neural networks considered so far in this chapter have been characterized by their ability to generate an output vector from an input vector, whether previously seen or unseen. The outputs are therefore determined solely by the input *values*, and not by the *timing* of those inputs. These value-based networks might be classed as the *second generation* of artificial neural networks (Maass 1997). Recurrent neural networks are also in this group as their model is based on sequences of values rather than their timing.

Although second-generation neural networks are undoubtedly useful tools, they are far from an accurate biological model. These neural networks are biologically inspired, that is, they mimic the nervous system of animals, including humans at a superficial level. Biological neurons are cells connected together by synapses, as shown in Figure 8.24. A synapse produces a chemical response to an input. The strength of the response can vary, and it is this feature that is modeled by the weights that connect the units of an artificial neural network. The biological neuron "fires" if the sum of all the reactions from the synapses is sufficiently large, just as an artificial neuron is deemed to have fired if its output exceeds a given threshold. However, this is more or less where the analogy ceases to apply. Biological neurons are far more complex than the simple neurons presented so far and second-generation artificial neural networks bear little resemblance to a brain.

Spiking neural networks (SNNs) have been presented as a *third generation* of artificial neural networks that are intended as more biologically plausible models of neural processing. A key initiative has been the development of *spiking neurons* that communicate via the timing of spikes or a sequence of spikes (Ghosh-Dastidar and

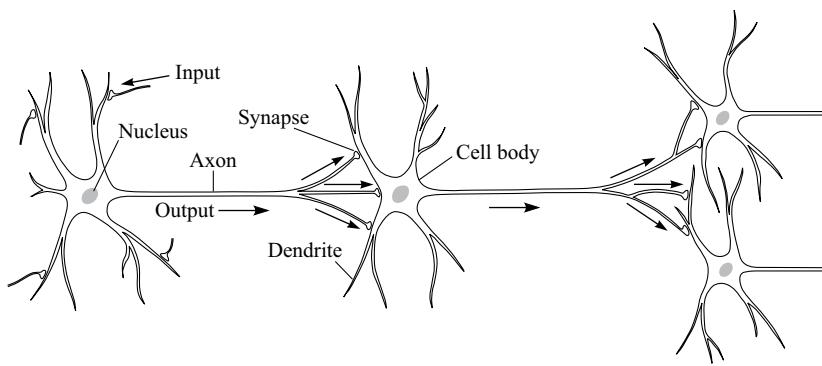


Figure 8.24 Biological neurons connected at synapses.

Adeli 2009). As with the first-generation neurons, spiking neurons produce only a binary output (0 or 1). The inputs and outputs are short-lived spikes whose timing conveys information, as in biological neurons, while the magnitude of the input and output spikes is immaterial.

The function of the neurons is determined by the *spiking response model*. The neurons can be connected as recurrent or feedforward networks, and a variety of learning algorithms have been proposed.

The claimed advantages of SNNs do not stop at biological plausibility. They also have the advantage of being able to process time-varying patterns of information because of the dynamic representation inherent in spiking neurons. Just as a three-layered perceptron with a sigmoid transfer function is theoretically capable of any nonlinear mapping, so SNNs have the theoretical capability to approximate any continuous function. Their main disadvantage is their large demand for computing power during training.

8.8 Summary

A neural network may be used to solve a problem in its own right, or as a component of a larger system. Neural networks are an important type of numerical learning technique. Numerical learning is based upon adapting numerical parameters in order to achieve a close match between a desired output and the actual output. Neural networks can be used to model any nonlinear mapping between variables, and they are frequently used in classification tasks. When presented with data that lie between previously encountered data, neural networks will generally interpolate to produce an output between those generated previously. Neural networks may therefore be a substitute for fuzzy logic in some applications. The parallelism of neural networks makes them ideally suited to parallel processing computers.

An often-stated drawback of neural networks is that their reasoning is opaque. The learned weights can rarely be understood in a meaningful way, although rules can be extracted from them, at least in principle, as discussed in Chapter 10. Thus, the neural network is often regarded as a “black box” that simply generates an output from a given input. For classification problems, the black-box metaphor contrasts with the more transparent approach of analytical models such as *support vector machines* (SVMs) (Abe 2010). SVMs use the statistical technique of regression to find a hyperplane that not only separates the classes, but that has the maximum margin from the exemplars at the boundaries, known as the support vectors. Nevertheless, by confining the use of the neural network to subtasks within a problem, they can play a key role in the design of an intelligent system and the overall problem-solving strategy can remain clear.

Further Reading

- Bishop, C. M. 1995. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, UK.
- Gurney, K. 1997. *An Introduction to Neural Networks*. CRC Press, Boca Raton, FL.
- Haykin, S. 2008. *Neural Networks and Learning Machines*. 3rd ed. Pearson Education, Upper Saddle River, NJ.
- Picton, P. D. 2000. *Neural Networks*. 2nd ed. Palgrave Macmillan, New York, NY.
- Taylor, M. 2017. *Neural Networks: A Visual Introduction for Beginners*, Blue Windmill Media, Sheffield, UK.

Chapter 9

Deep Neural Networks

9.1 Deep Learning

There are many forms of machine learning, including case-based reasoning (Chapter 5), genetic algorithms (Chapter 7), and neural networks (Chapter 8). Nevertheless, one particular form of machine learning has become especially popular since the 2010s, so-called *deep learning*. This term applies almost exclusively to large neural networks, so the terms *deep neural network* (DNN), *deep-learning neural network*, and *deep-learning algorithm* tend to be used interchangeably.

DNNs extend the ideas of the neural networks that were introduced in Chapter 8. Indeed, Section 8.7 highlighted the three generations of neural networks (Maass 1997), to which DNNs might be added as the fourth. They are “deep” in two ways: they have multiple layers of neurons and they break down their classification problem into component parts, or multiple layers of abstraction. Nevertheless, they remain pattern-spotters that have no conceptual understanding, deep or otherwise. So, they are still shallow when it comes to domain understanding.

The larger and more complex neural networks become, the greater is their power to model a problem space in detail. So, in principle, larger neural networks can solve more and more complex problems. However, simply building an enormous multilayer perceptron (Section 8.4) is unlikely to succeed because of two difficulties:

1. The computational demand scales exponentially with the number of nodes, since each node is fully connected to the nodes in any preceding or subsequent layers. Even though readily available computer power has improved substantially since the development of the backpropagation algorithm in the 1980s, the training of large networks would nevertheless soon become impractical.
2. A more and more complex neural network would be better and better in principle at modeling the training data. In practice, vastly complex networks either

learn no better than a shallow network or they become *overtrained*, that is, specialized at modeling the training data rather than generalized to produce an accurate classification from previously unseen data.

Deep-learning neural networks overcome these difficulties through carefully designed structures. The discussion here will start by consideration of convolutional neural networks (CNNs), which are specialized to image-processing tasks. Nevertheless, two general principles carry forward to all the other deep-learning networks: selective connections between neurons and a structured breakdown of the problem through the layers of the network.

9.2 Convolutional Neural Networks (CNNs) for Image Recognition

9.2.1 Origins

The two difficulties described in the previous section started to be overcome following a series of more and more successful large neural network designs during the 2000s and 2010s. A key trigger for this wave of neural network development was an article about extracting text from handwritten documents (LeCun et al. 1998). This work was a milestone, even though it built on other research and the concepts have been refined subsequently. The core contribution by LeCun et al. was the development of a CNN for image recognition. It overcame the two difficulties by:

1. Not being fully interconnected. As each interconnection represents a computation, the nodes were selectively interconnected. Some of the connections share the same weights, reducing the complexity further.
2. Breaking down the problem into layers of abstraction, i.e., individual features that combine to give an overall interpretation of the image.

9.2.2 Motivation for Convolutional Networks

To understand why the idea of layers of abstraction is important for image recognition, it is first necessary to understand some of the shortcomings of shallow networks. As an example, consider the problem of interpreting handwritten characters like the ones in Figure 9.1. These characters are taken from handwritten zip codes (or postcodes) and their automated recognition is an example of AI in routine use today. At the time of the LeCun et al. article in 1998, the recognition of such characters remained a technical challenge.

The images shown in Figure 9.1 are stored as 42×52 arrays of black/white pixels. Each image is therefore made up from 2184 bits. If the characters were presented to



Figure 9.1 Handwritten characters. (Courtesy of Andy Downton.)

a conventional neural network like an MLP in this raw form, a network with 2184 input nodes would be required. Not only is an MLP with such a large number of input nodes unwieldy, but it is also highly unlikely to succeed.

Simply supplying a conventional shallow neural network with the raw data and hoping it will find appropriate features rarely produces good results. Instead, some

form of transformation is required which picks out features of the data that can be used to distinguish between the patterns. Such features are called *primitives*. Using the primitives, a neural network can recognize or classify the corresponding input data.

A popular method used to transform the character data into its primitives relies on the fact that similar characters have similar numbers of curves, loops, and lines. Each image is scanned vertically and horizontally, and on each scan the number of times the image changes from white to black is counted. These counts provide an estimate of the number of curves, loops, and lines an image contains. Figure 9.2 shows an example. Those parts of the image that are all white end up with a count of 0. Scans that cross a single line produce a count of 1, and scans that cross loops produce a count of 2. For the image in Figure 9.2, the resulting counts are as follows:

Vertical scans from the left: 0 1 1 1 1 1 2 3 2 2 2 1 1 1 1 1 2 1 1 1 0

Horizontal scans from the bottom: 0 0 1 2 2 2 2 2 2 2 1 1 2 2 2 2 2 1 1 1 1
1 0 0

In this example, counts have been taken from every second row and every second column of pixels in order to reduce the data size. The counts are placed together

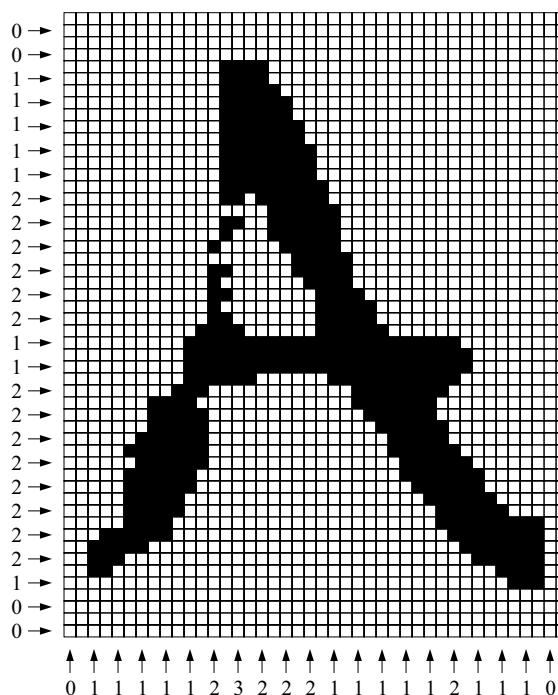


Figure 9.2 Scanning to count white-to-black transitions.

in a single vector for each character. These vectors should contain enough information for the characters to be distinguished from each other in an MLP, but there are still some additional complications to consider.

If a pattern, say the letter “A,” is used to train a neural network classifier, the aim would be for the trained network to recognize a similar-looking letter “A.” However, if the new character is a different size from the original one, or it is in a different part of the image, or it is rotated relative to the original character, then the neural network will often fail to identify it.

So, additional preprocessing is required to ensure that the characters are roughly the same size, are all moved to the same alignment within their frame, and there is little or no rotation between them. They are then said to be size invariant, translationally invariant, and rotationally invariant. So, in order to classify the handwritten characters with an MLP or any other type of shallow neural network, a lot of preliminary work is required in preprocessing (a) to ensure these three types of invariance and (b) to extract the primitives. Once these steps have been undertaken, a shallow neural network like an MLP can achieve a high success rate in recognizing or classifying the input data.

CNNs are specifically designed for image recognition. They represent a significant breakthrough as they are able to undertake the preprocessing steps as part of their classification function. So, good results can be achieved with almost no preprocessing at all.

9.2.3 CNN Structure

In a CNN, multiple layers precede a conventional fully connected classifier network, as shown in Figure 9.3. These preceding layers are the *convolution layers*, and their function is to extract the primitives automatically. Each layer transforms its input into a more abstract representation, before finally presenting the primitives to a conventional neural network classifier. In effect, the network learns the primitives for itself from a bitmap image. It has removed the painstaking work of preprocessing the image into a vector form that represents information about the primitives.

There are two principal types of convolution layers: feature maps and pooling layers. The design of the input layer is also novel, and all three types of layers are described in the following subsections.

9.2.3.1 Input Layer

We have seen in the previous section that preprocessing an image is necessary to extract primitives for presentation to an MLP. A CNN overcomes that difficulty through its pre-classification layers. Nevertheless, if we want to present a raw bitmap to a neural network, we immediately run up against a problem. The standard input to any of the network architectures considered so far is a one-dimensional (1D) vector, whereas a bitmap is a two-dimensional (2D) matrix. Of course,

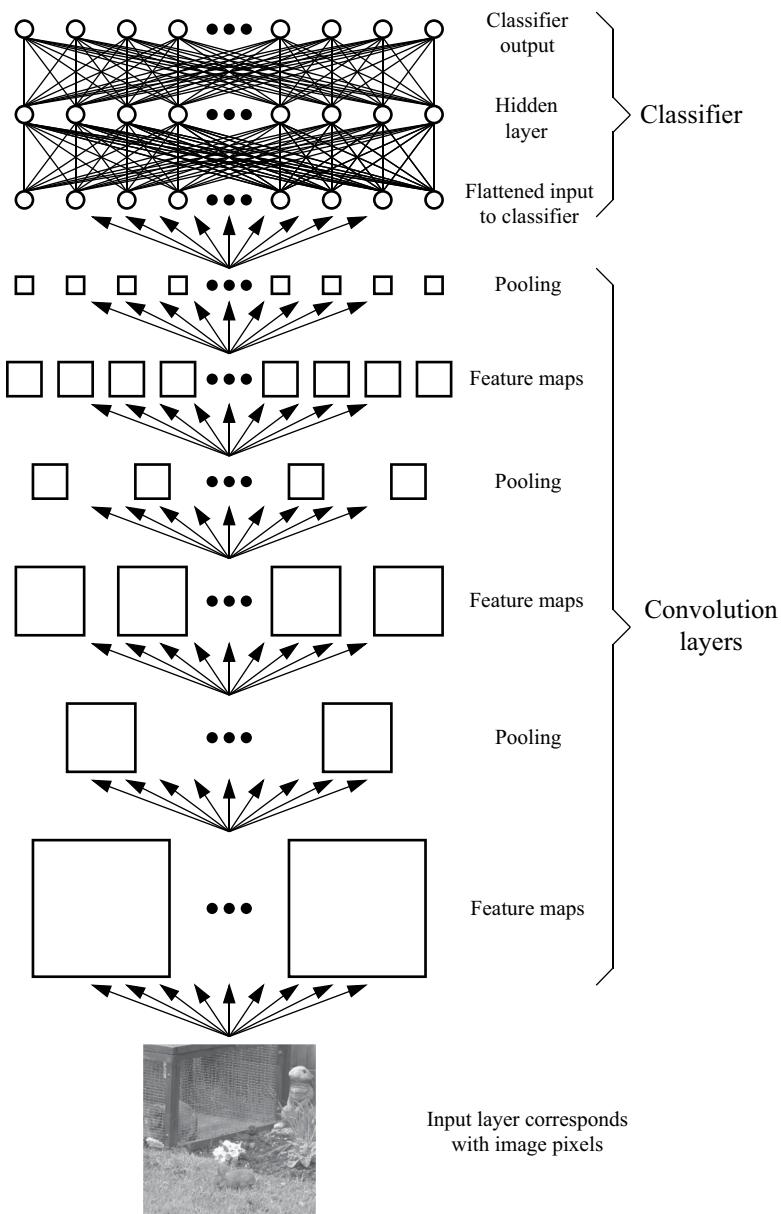


Figure 9.3 Structure of a CNN.

we could flatten out a bitmap by placing each row of pixels after the previous one, as shown in Figure 9.4 for a simple 5×5 bitmap of a number “3.” However, in the flattened out 1D vector, the information about the spatial proximity of the pixels has been lost.

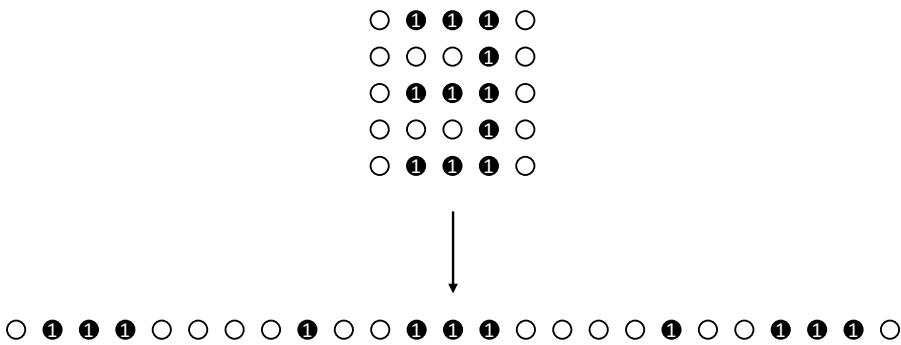


Figure 9.4 Loss of spatial information as a 2D bitmap is converted into a 1D vector.

To overcome this problem, the input layer to a CNN has the same 2D structure as the bitmap input. For each pixel, there is a corresponding input neuron. A cluster of input neurons, corresponding to a cluster of pixels in the input image, is called the *local receptive field* (LRF) for a hidden neuron in the next layer. That next layer comprises feature maps, described in the next subsection.

9.2.3.2 Feature Maps

Sitting above the input layer, the first hidden layer contains neurons in multiple panels called *feature maps*. Instead of each neuron in the input layer being connected to each neuron in the first hidden layer, as would be the case for an MLP, an LRF in the input layer is connected to a single corresponding neuron in a feature map. Just as in an MLP, the hidden neuron has an adjustable bias and an adjustable weight on each of its input connections. Through adjustments of these weights and bias, the neuron is trained to analyze its LRF.

The neighboring hidden neuron is connected to an LRF that is shifted by a number of pixels, called the *stride length*. Assuming that the stride length is set shorter than the width of the LRFs, then neighboring LRFs overlap each other. Figure 9.5 shows an example where the input image is 24×24 pixels, so the input layer must comprise 24×24 neurons. If the LRF is 5×5 neurons and the stride length is one pixel, then the feature map will contain 20×20 neurons. It is smaller than the input layer because of edge effects, that is, the central pixel of a cluster cannot be closer than two pixels from the edge of the image.

So, each LRF is connected to a single hidden neuron in a feature map, and there can be many feature maps that comprise the first hidden layer, as shown in Figure 9.3. As each feature map is a 2D array of neurons, the first hidden layer is effectively a three-dimensional (3D) array of neurons.

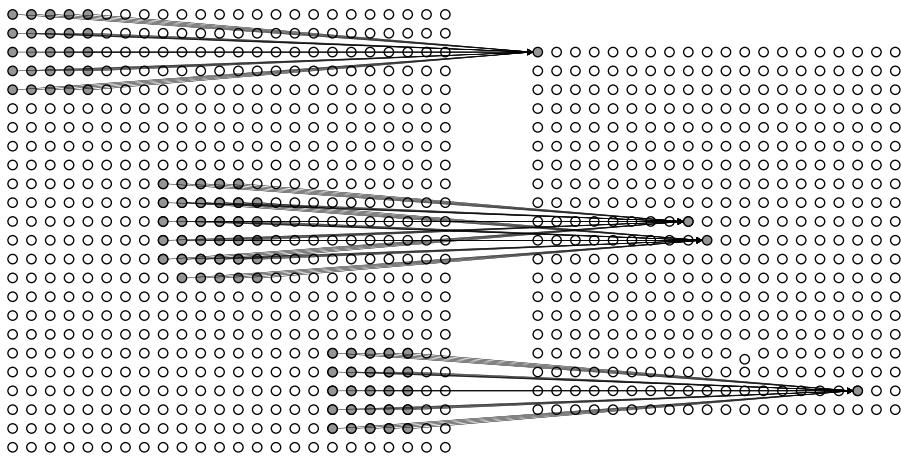


Figure 9.5 Local receptive fields are connected to single neurons in the feature map.

For each neuron in our example feature map, there are 5×5 weights plus a bias, connecting to its corresponding LRF in the input layer. A key characteristic of the CNN is that these weights are identical for each neuron across the feature map. In this way, each feature map can be trained to become specialized in detecting a specific type of feature in the image. In the example of character recognition considered earlier, these features might be curves, loops, lines, or combinations thereof. Because each feature map has connections to all the LRFs in the input layer, it doesn't matter where in the image the feature occurs. In this way, the problem of translational variance is overcome.

This collection of feature maps that makes up the first hidden layer is actually carrying out a function called a *convolution* (Hirschman and Widder 2005), which is where the network name originates. When one function is convolved with another, a third function is produced that has characteristics of each of the original functions. In our case, the set of outputs from a feature map, Y , are given by:

$$Y = f_t \{ b + (W * X) \} \quad (9.1)$$

where f_t is the transfer function used by the feature map, b is the bias, W is the set of weights on the inputs X , and $*$ is the convolution operator. More precisely, the output y_{ij} from each hidden neuron (i, j) in the feature map is given by:

$$y_{ij} = f_t \left(b + \sum_{k=0}^4 \sum_{l=0}^4 w_{kl} x_{(i+k)(j+l)} \right) \quad (9.2)$$

where w_{kl} are the weights on the links from the LRF to the hidden neuron, and $x_{(i+k)(j+l)}$ are the values on the neurons at the input layer. Just like the perceptrons in Chapter 8, the input layer carries out no processing, so it makes no difference whether we describe these values as its inputs, activations, or outputs.

Notice that there is no subscript on the bias, because the same value is shared across the feature map. Furthermore, the same set of weights is used for each set of links from an LRF to a hidden neuron. So, in our example, the same set of 5×5 weights are shared for each of the 20×20 hidden neurons in the feature map. This sharing of weights is one of the ways by which the CNN avoids an explosion in computation demand. For each feature map in our example, 26 weights (including one bias) must be learnt, rather than $20 \times 20 \times 26$ weights, i.e., 10,400. The set of shared weights and the shared bias are sometimes collectively called a *kernel* or *filter*.

The pattern of feature maps is repeated in the subsequent convolution layers. In this way, the features that are identified become progressively more abstracted from the raw bitmap data. For example, in character recognition, the first feature map layer might detect simple lines and curves. The second layer of feature maps might detect higher-level features like loops or combinations of lines and curves. At each subsequent level, the identified features become more and more characteristic of specific alphanumeric characters.

9.2.3.3 Pooling and Classification Layers

Each convolution stage multiplies the number of neurons. Each stage requires many feature maps, where each feature map has dimensions similar to the previous layer, with only a small reduction through the LRF edge effects. To reduce the number of neurons and thereby keep the computational demand under control, pooling layers are used to simplify and condense the outputs from the feature maps. Although there is some loss of information in the process, image recognition is not typically critically dependent on the precise location of features within images, but rather their positions relative to each other. In fact, a reduction in detailed positional information can improve generalization as well as reducing computational demand.

There are various ways in which pooling can be achieved. One approach is *max-pooling*, in which the feature map is compressed through a reduction in its resolution. For example, a 3×3 segment of the feature map can be mapped onto a single neuron in the pooling layer by simply taking the largest activation of the nine available. An alternative approach is to take the root-mean-square of the sum of the activations from the segment.

Pooling is also used at the final convolution stage before classification. At the final stage of a CNN, a classification is typically required, for which a multilayer perceptron or similar fully connected neural-network classifier is well suited. However, connecting the final feature maps directly to an MLP would be unwieldy and would, again, risk a computational explosion. Even if we consider a small example comprising 20 feature maps of 20×20 neurons each, there would be 8000 outputs to

connect to the classifying network. To reduce this complexity, another pooling layer is used. Its outputs are flattened into a 1D vector for presentation to a classification network akin to an MLP, described in Chapter 8, as shown in Figure 9.3.

9.2.4 Pretrained Networks and Transfer Learning

An underlying principle of artificial neural networks is *generalization*, that is, the ability to produce an accurate classification or prediction from input data that did not appear in the training set. For a network to generalize reliably, the training data must be broadly representative of the previously unseen test data. In the case of a classification task based on simple numerical inputs, that requirement means that the test data are interpolations rather than extrapolations of the training data. In the case of image recognition and other deep learning challenges, the phrase “broadly representative” may be more loosely defined, to the extent that the test domain may be different from the training domain.

The application of a trained artificial neural network in a domain different from the training domain is the basis of *transfer learning*. A pretrained network can sometimes be applied directly in a new domain or, more often, it provides an efficient starting point for training in the new domain. There is a saving in computational cost, since training from scratch for each new application is avoided. There may also be an improvement in performance arising from an overall combined training set that is larger and wider than the domain-specific set. Torrey and Shavlik (2009) have identified three mechanisms for these improvements:

- The pretrained network starts at a higher level of capability than an untrained network.
- The pretrained network may learn more rapidly than a network learning from scratch.
- After retraining in the new domain, the pretrained network may settle at a higher level of performance than a network trained from scratch.

Four groups of layers of a CNN have been described in the previous section: LRFs, feature maps, pooling, and classification. Of these groups of layers, only the classification layer is specific to the domain being addressed. The tasks of locating features in an image are not specific to any particular image type. So, a network trained on one type of image can be reused in a completely different domain if the image resolution is the same and the level of detail is broadly similar.

There are many varieties of CNN available as library functions in a range of software development environments. Some of the well-known variants include *Alexnet* (Krizhevsky et al. 2017), *GoogLeNet* (Szegedy et al. 2015), and *ShuffleNet* (Zhang et al. 2018). Networks like these are available pre-trained to categorize more than a million images from the ImageNet database into more than 1000 classes. As their training set is so large and their classification categories so diverse, they can be

applied directly to a range of problems without any further training. Alternatively, the trained network's convolutional layers can be left with their weights intact and just the classification layers trained in the domain of interest. The pretrained network has already learned to extract informative features from diverse images, so it can be the starting point for any specialized image classification.

The idea of a pretrained CNN is particularly appealing as it can avoid the need for an extensive training data set and can sidestep the computationally expensive training phase. In general, it is the training of a neural network that is computationally demanding, not the application of a trained network to tasks like classification, prediction, clustering, or mapping. Nevertheless, some authors have highlighted the possible pitfalls of using pretrained networks, for example, Yosinski et al. (2014). Whatever the problem domain, it always pays to understand the data and how a neural network is using it. Given that the CNN design was inspired by the cat's visual cortex, it is ironic that a popular demonstration is the use of CNNs to distinguish cats from dogs!

9.2.5 CNNs in Context

The CNN is a classic example of deep learning. It uses multiple layers of neurons to pick out features in images before classifying those features. There are too many different designs of deep learning neural networks to cover them all here, but they share many similar principles with the CNN. They are all concerned with recognizing patterns in data, whether or not those data are in the form of visual images. In achieving that goal, they all attempt to avoid computational overload by techniques such as the selective connectivity between nodes, the structured breakdown of the problem into layers of abstraction, and the re-use of trained networks.

9.3 Generative Networks

9.3.1 Generative Versus Discriminative Algorithms

All the neural networks described so far have been *discriminatory*. That is to say, they discriminate between input patterns to decide what label to attach to them. The purposes for that discrimination have been mostly either classification or prediction. (A fuller set of neural network applications is described in Section 8.2.)

We also saw that the networks can be supervised, with a target value attached to every item of training data. Alternatively, they may be unsupervised, where the network discovers clusters in the data. *Generative* networks take the concept of unsupervised learning further, by allowing the network to create new data sets. Generative and discriminatory networks can be seen as having inverse purposes. Whereas a discriminatory network learns to associate classifications with complex data vectors, a generative network learns to generate complex data vectors associated with

classifications. The applications are manifold, from the creation of art and music, to designing new molecules and computer code.

9.3.2 Autoencoder Networks

Autoencoders are a type of generative network that can be seen as “creative”, in the sense of producing something new at its output (Hinton and Salakhutdinov 2006). They have attracted particular interest for creating so-called *deepfake* images, which are convincing fake images that have been derived from genuine ones using deep learning (Güera and Delp 2018).

The prefix “auto” means “self” or “same”, so the basic autoencoder is designed to produce an output that matches its input. For that reason, we can assume that it has the same number of output nodes as input nodes. If there were sufficient nodes in each hidden layer, the network would quickly learn to propagate each input directly to its corresponding output by placing a chain of weights of value 1 on the connections, with 0 elsewhere. However, if there are insufficient nodes in a hidden layer, those layers form a bottleneck that constricts the information throughput, and the input values cannot be directly transmitted to the outputs. Instead, the network must learn to pick out key features from the input and to reproduce them at the output. A simple version of the network design is shown in Figure 9.6, although a practical autoencoder would have many more nodes and layers. The funneling down part of the network is the encoder, while the fanning out part is the decoder.

In this basic form, the autoencoder has two potential uses:

1. *Image compression.* The image representation stored at the hidden layers is a compressed representation of the input. Small details of the original image are likely to be lost in the process, as is typical of any form of lossy image compression. Such compression is of limited practical value, however, as more efficient techniques like JPEG are well established.
2. *Noise removal.* Any noise in the original image adds no useful information about the image content. The network may therefore remove the noise along with the other small details that are filtered out in the recreation of the output image. So, a cleaner version of the image, with the noise removed, can be produced at the output. This application is sometimes called *denoising*.

The creative applications of the autoencoder start to emerge when an autoencoder trained on one image is combined with another autoencoder trained on a different image. *Deepfake* techniques rely on using the same encoder on images that are of a similar type, such as faces. This similarity ensures that the encoder can generalize features that can be transferred between different decoders. Imagine a network that uses a single encoder trained on images *A* and *B*, but where two decoders are trained separately on *A* and on *B*. If image *A* is presented to the network and the decoder trained on that image is used, then a compressed version of *A* would be expected at

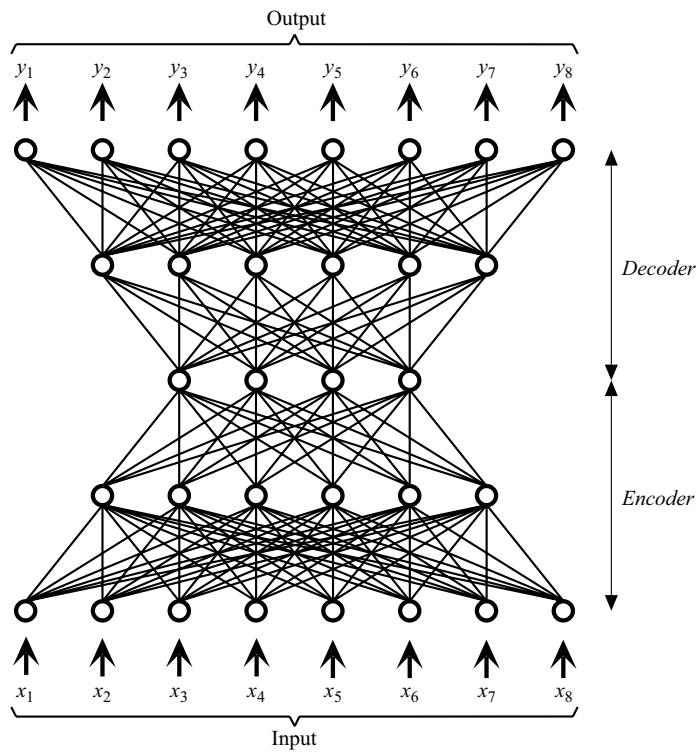


Figure 9.6 Structure of a simple autoencoder.

the output. On the other hand, if image A is presented again but the decoder trained on image B is used, we would see a version of B that contains features transferred from image A . In this way, new and realistic images can be generated.

9.3.3 Generative Adversarial Networks (GANs)

Generative adversarial networks (GANs) have proved to be one of the most effective designs for creative purposes (Goodfellow et al. 2014). In fact, they comprise two networks in opposition, hence the term *adversarial*. The generative network is the creative side, producing data items that are intended to be of high quality although, in fact, their quality may be quite variable. The discriminatory network is there to recognize the good ones. The roles of the generator and the discriminator are shown in Figure 9.7.

In the context of art, the generative network might be trying to produce an image that resembles the works of one of the great impressionists like Monet, while the discriminatory network might be trained to classify different genres of art. So, the generative network is like a fraudster trying to produce fake art, while

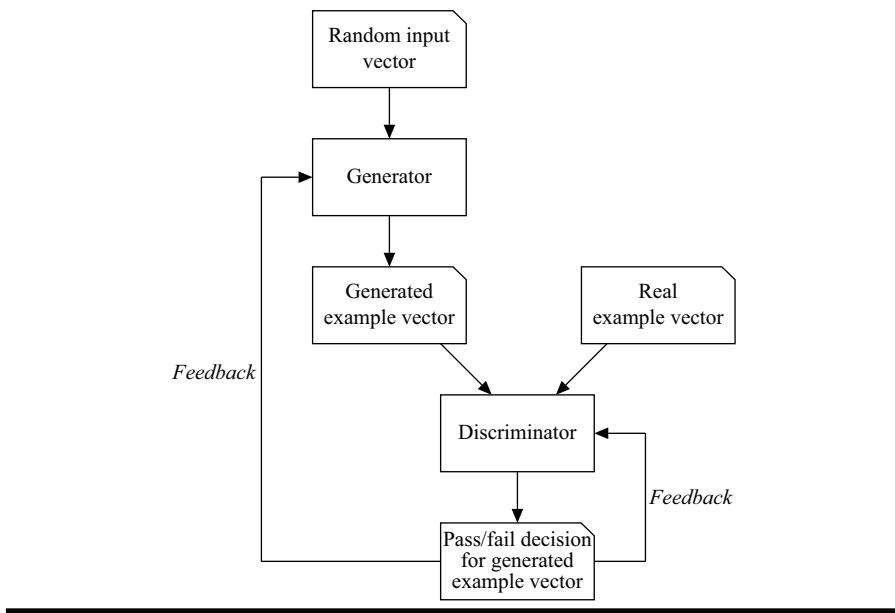


Figure 9.7 A GAN flowchart.

the discriminator is like a gallery curator, determined to accept only the finest and genuine works of art. The generator learns to produce better and better images, based on feedback from the discriminator. The discriminator, on the other hand, receives its feedback by comparison of the fakes with genuine art. It becomes more and more discerning through being trained on genuine images of fine art alongside the images produced by the generator. Once the generator produces an image that is good enough to “fool” the discriminator, the quality can be assumed to be high.

9.4 Long Short-Term Memory (LSTM) Networks

Recurrent neural networks (RNNs) were described in Section 8.5, while DNN designs have been introduced in this chapter. In fact, RNNs may be considered a type of DNN specifically adapted to sequential data. Although the standard representations do not look deep on the surface, the feedback from the outputs to the inputs creates depth as each time-step builds on the previous one.

Long Short-Term Memory (LSTM) networks, originally conceived by Hochreiter and Schmidhuber (1997), are probably the most practical and widely used of all RNNs. Jozefowicz et al. (2015) have compared a variety of RNNs, concluding that LSTMs are not necessarily optimal, but that they are nevertheless

practical and that their performance can be further improved with some small tweaks. Greff et al. (2017) have compared the performance of all of the popular variants of the LSTM, finding little difference between them.

The simple recurrent network (SRN) described in Section 8.5.1 contains feed-back connections from the hidden layer to additional input neurons called the context neurons. The accompanying description stated that this feedback provides the network with a context for the current input data in the form of the *immediate* history of inputs, typically a maximum of 5–10 iterations (Gers et al. 2000). In some applications, that immediate context is all that is needed, for example, for following a data trend in machine condition monitoring. In the case of natural language, the immediately preceding words in a sentence provide a context for the current word that is presented to the network. For example, any word followed by “the” will be a noun, and the type of noun will be given a context by a preceding verb, e.g., “throw the ...” or “tell the ...”

However, in many applications, that immediate data history is insufficient. The useful contextual information may lie further back in the data set and would be completely lost by any of the recurrent networks considered so far. In the case of natural language, important contextual clues may lie in a preceding sentence as well as in the immediately preceding few words. In the case of condition monitoring by following a measurement through time, the immediate prior measurements will provide a trend, but a longer history is required in order to recognize norms and boundaries. Two key problems are that the back-propagated context signals tend to either shrink or grow with every time-step, soon either disappearing or dominating. These two difficulties in the use of longer-term contextual information are known respectively as the *vanishing gradient problem* and the *exploding gradient problem*. LSTM networks address these specific limitations and are one of the most useful practical recurrent networks.

Figure 9.8a depicts the conventional representation of an RNN, showing the recursion from the output to the input, while Figure 9.8b shows the same network unraveled to show what is happening at each time-step. At the first time-step, the processing layer is presented with the input vector and produces an output vector. For every subsequent time-step, the processing layer is presented with both the current input vector and the previous output vector, to produce the new output vector. That concept from the simple RNN is retained in the LSTM network, but the neuron function is more complex. In the simple RNN, the processing layer sums its inputs, including the feedback from the previous time-step, and passes them through a transfer function. In the LSTM network, that single function is replaced with four functions called gates, shown in Figure 9.9. Collectively, the four gates that comprise the processing layer are described as a *memory cell*.

The core concept of the LSTM network is the *cell state*. The cell state is analogous to the traffic on a highway. The important memories are propelled forward along the highway, while less important ones depart via the slip lanes and are

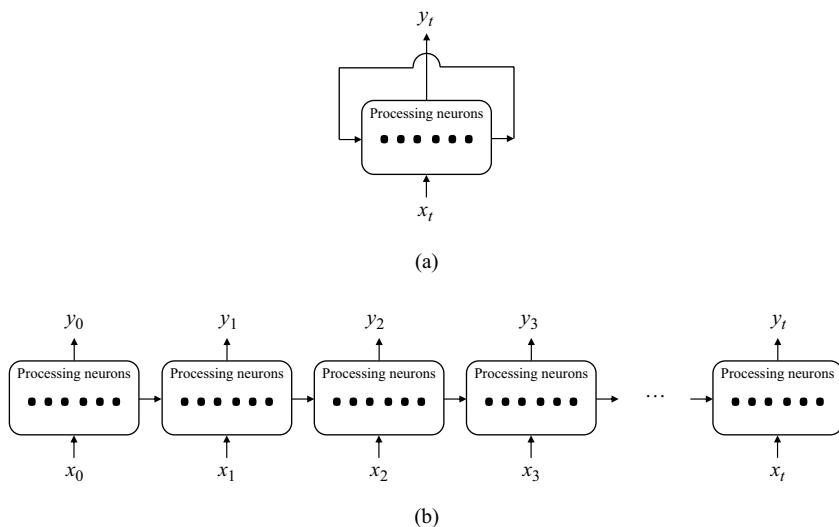


Figure 9.8 Recurrent neural network: (a) conventional representation; (b) unfurled to show the separate time-steps.

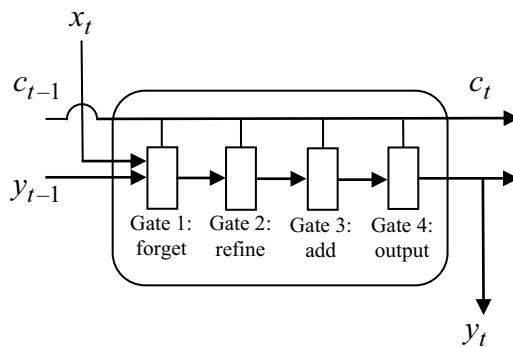


Figure 9.9 The four gates of an LSTM network memory cell.

discarded. Conversely, new memories may be added, feeding into the traffic. The importance of a past contextual observation is more influential than its recency. This persistence of important memories through time is achieved through a set of gates within the cell.

The first gate in the memory cell is the forget gate. It is responsible for filtering out the less important memories from the cell state, that is, it provides the slip lanes from the cell highway. The second and third gates regulate the input, determining the new information that might be added to the cell state. The second gate updates retained memories while the third adds new ones. Fourthly, the output gate

determines which new and updated information will be conveyed forward to form the new cell state.

9.5 Summary

The history of artificial neural networks can be traced back to when McCulloch and Pitts (1943) proposed a neuron design similar to that used in modern networks. The ADALINE network (Widrow and Hoff 1960) moved things along through the use of weighted neurons. It might be seen as a precursor for the multilayer perceptron and recurrent neural networks that appeared in the 1980s and provided the first practical neural networks as tools for classification and prediction. There have been numerous network designs since, with the CNN by LeCun et al. (1998) triggering a wave of designs of deeply layered networks since the start of the millennium. It has not been possible to cover them all here, but this chapter and the previous one have aimed to describe the most important designs of networks as practical tools for AI.

Further Reading

- Aggarwal, C. C. 2018. *Neural Networks and Deep Learning*. Springer, Cham, Switzerland.
- Alpaydin, E. 2016. *Machine Learning: The New AI*. MIT Press, Cambridge, MA.
- Chollet, F. 2018. *Deep Learning With Python*. Manning, Shelter Island, NY.
- Goodfellow, I., Y. Bengio, A. Courville, and F. Bach. 2017. *Deep Learning*. MIT Press, Cambridge, MA.
- Kelleher, J. D. 2019. *Deep Learning*. MIT Press, Cambridge, MA.
- Michelucci, U. 2018. *Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks*. Apress, Dübendorf, Switzerland.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 10

Hybrid Systems

10.1 Convergence of Techniques

The previous chapters of this book have demonstrated that there is a wide variety of computing techniques that can be applied to particular problems. These include knowledge-based systems, computational intelligence methods, and conventional programs. In many cases, the techniques need not be exclusive alternatives to each other but can be seen as complementary tools that can be brought together within a hybrid system. All of the techniques reviewed in this book can, in principle, be mixed with other techniques. This chapter is structured around four ways in which different computational techniques can be complementary:

- *Dealing with multifaceted problems.* Most real-life problems are complex and have many facets, where each facet may be best suited to a different technique. Therefore, many practical systems are designed as hybrids, incorporating several specialized modules, each of which uses the most suitable tools for its specific task. One way of allowing the modules to communicate is by designing the hybrid as a blackboard system, described in Section 10.2.
- *Parameter setting.* Several of the techniques described in this chapter, such as genetic–fuzzy and genetic–neural systems, are based on the idea of using one technique to set the parameters of another.
- *Capability enhancement.* One technique may be used within another to enhance the latter’s capabilities. For example, Lamarckian or Baldwinian inheritance involves the inclusion of a local-search step within a genetic algorithm. In this case, the aim is to raise the fitness of individual chromosomes and to speed convergence of the genetic algorithm toward an optimum.
- *Clarification and verification.* Neural networks have the ability to learn associations between input vectors and associated outputs. However, the underlying

reasons for the associations are opaque, as they are effectively encoded in the weightings on the interconnections between the neurons. Efforts have been made to extract equivalent rules from the network automatically. The extracted rules can be more readily understood than the interconnection weights from which they are derived. Verification rules can apply additional knowledge to check the validity of the network's output.

10.2 Blackboard Systems for Multifaceted Problems

The *blackboard model* or *blackboard architecture*, shown in Figure 10.1, provides a software structure that is well suited to multifaceted tasks. Systems that have this kind of structure are called *blackboard systems*. In a blackboard system, knowledge of the application domain is divided into modules. These modules were historically referred to as *knowledge sources* (or Ks) but, as they are independent and autonomous, they are now commonly regarded as agents. Each agent is designed to tackle a particular subtask. The agents are independent and can communicate only by reading from or writing to the *blackboard*, a globally accessible working memory where the current state of understanding is represented. The agents can also delete unwanted information from the blackboard.

The first published application of a blackboard system was Hearsay-II for speech understanding in 1975 (Erman et al. 1980; Lesser et al. 1975). Further developments took place during the 1980s (Nii 1986). In the late 1980s, the Algorithmic and Rule-Based Blackboard System (ARBS) was developed and subsequently applied to diverse problems including the interpretation of ultrasonic images (Hopgood et al. 1993), the management of telecommunications networks (Hopgood 1994), and the control of plasma deposition processes (Hopgood et al. 1998). Later, ARBS was redesigned as a

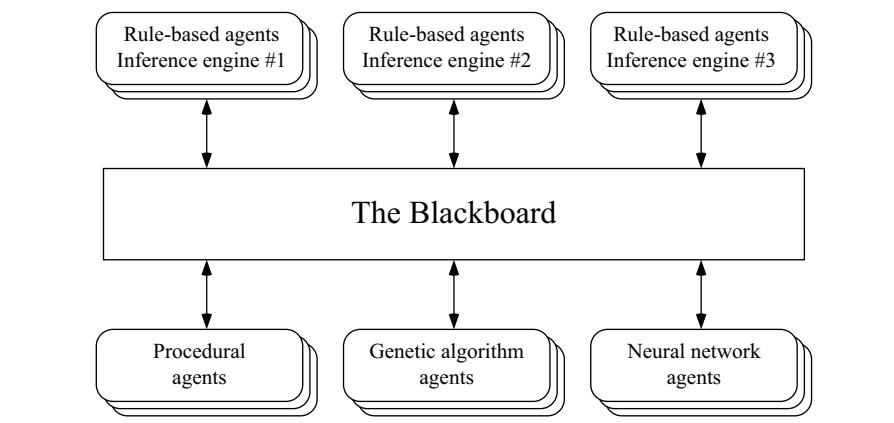


Figure 10.1 The blackboard model.

distributed system, DARBS, in which the software modules run in parallel, possibly on separate computers connected via the Internet (Nolle et al. 2002c; Tait et al. 2008). DARBS is freely available as open-source software.* The blackboard model is now widely seen as a key technology for multiagent systems (Brzykcy et al. 2001).

A blackboard system is analogous to a team of experts who communicate their ideas via a physical blackboard, by adding or deleting items in response to the information that they find there. Each agent represents such an expert having a specialized area of knowledge. As each agent can be encoded in the most suitable form for its particular task, blackboard systems offer a mechanism for the collaborative use of different computational techniques such as rules, neural networks, genetic algorithms, and fuzzy logic. The inherent modularity of a blackboard system is also helpful for maintenance. Each rule-based agent can use a suitable reasoning strategy for its particular task, for example, forward chaining or backward chaining, and can be thought of as a rule-based system in microcosm.

Agents are applied in response to information on the blackboard, when they have some contribution to make. This responsiveness leads to increased efficiency, since the detailed knowledge within an agent is only applied when that agent becomes relevant. The agents are said to be *opportunistic*, activating themselves whenever they can contribute to the global solution. In early, nondistributed blackboard systems, true opportunism was difficult to achieve as it required explicit scheduling and could involve interrupting an agent that was currently active. In modern blackboard systems, each agent is its own process, either in parallel with other agents on multiple processors or concurrently on a single processor, so no explicit scheduling is required.

In the interests of efficiency and clarity, some degree of structure is usually imposed on the blackboard by dividing it into partitions. An agent then only needs to look at a partition rather than the whole blackboard. Typically, the blackboard partitions correspond to different levels of analysis of the problem, progressing from detailed information to more abstract concepts. In the *Hearsay-II* blackboard system for computerized understanding of natural speech, the levels of analysis include those of syllable, word, and phrase (Erman et al. 1980). In ultrasonic image interpretation using ARBS, described in more detail in Chapter 12, the levels progress from raw signal data, via a description of the significant image features, to a description of the defects in the component (Hopgood et al. 1993).

The key advantages of the blackboard architecture can be summarized as follows (Feigenbaum 1988):

- Many and varied sources of knowledge can participate in the development of a solution to a problem.
- Since each agent has access to the blackboard, each can be applied as soon as it becomes appropriate. This is opportunism, that is, application of the right knowledge at the right time.

* Information on accessing DARBS is available at adrianhopgood.com.

- For many types of problems, especially those involving large amounts of numerical processing, the characteristic style of incremental solution development is particularly efficient.
- Different types of reasoning strategy (for example, data-driven and goal-driven) can be mixed as appropriate in order to reach a solution.
- Hypotheses can be posted onto the blackboard for testing by other agents. A complete test solution does not have to be built before deciding to modify or abandon the underlying hypothesis.
- In the event that the system is unable to arrive at a complete solution to a problem, the partial solutions appearing on the blackboard are available and may be of some value.

10.3 Parameter Setting

Designing a suitable solution for a given application can involve a large amount of trial and error. Although computational intelligence methods avoid the “knowledge acquisition bottleneck” that was mentioned in Section 5.4 in relation to knowledge-based systems, a “parameter-setting bottleneck” may be introduced instead. The techniques described here are intended to avoid this bottleneck by using one computational intelligence technique to set the parameters of another.

10.3.1 Genetic–Neural Systems

Part of the practical challenge in building a useful neural network is to choose the right architecture and the right learning parameters. Neural networks can suffer from a parameter-setting bottleneck as the developer struggles to configure a network for a particular problem (Woodcock et al. 1991b). Whether a network will converge, that is, learn suitable weightings, will depend on the topology, the transfer function of the nodes, the values of the parameters in the training algorithm, and the training data. The ability to converge may even depend on the order in which the training data are presented.

Kolmogorov’s Existence Theorem (see Section 8.4.2) leads to the conclusion that a three-layered perceptron, with the sigmoid transfer function, can perform any mapping from a set of inputs to the desired outputs. Unfortunately, the theorem tells us nothing about the learning parameters, the necessary number of neurons, or whether additional layers would be beneficial. It is, however, possible to use a genetic algorithm to optimize the network configuration (Yao 1999). A suitable cost function might combine the RMS error with duration of training.

Supervised training of a neural network involves adjusting its weights until the output patterns obtained for a range of input patterns are as close as possible to the desired patterns. The different network topologies use different training algorithms

for achieving this weight adjustment, typically through back-propagation of errors. Just as it is possible for a genetic algorithm to configure a network, it is also possible to use a genetic algorithm to train the network (Yao 1999). The training can be achieved by letting each gene represent a network weight, so that a complete set of network weights is mapped onto an individual chromosome. Each chromosome can be evaluated by testing a neural network with the corresponding weights against a series of test patterns. A fitness value can be assigned according to the error, so that the weights represented by the fittest generated individual correspond to a trained neural network.

10.3.2 Genetic–Fuzzy Systems

The performance of a fuzzy system depends on the definitions of the fuzzy sets and on the fuzzy rules. As these parameters can all be expressed numerically, it is possible to devise a system whereby they are learned automatically using genetic algorithms. A chromosome can be devised that represents the complete set of parameters for a given fuzzy system. The cost function could then be defined as the total error when the fuzzy system is presented with a number of different inputs with known desired outputs.

Often, a set of fuzzy rules for a given problem can be drawn up fairly easily, but defining the most suitable membership functions remains a difficult task. Karr (1991a, 1991b) has performed a series of experiments to demonstrate the viability of using genetic algorithms to specify the membership functions. In Karr's scheme, all membership functions are triangular. The variables are constrained to lie within a fixed range, so the fuzzy sets *low* and *high* are both right-angle triangles (Figure 10.2). The slope of these triangles can be altered by moving their intercepts on the abscissa, marked \max_i and \min_i in Figure 10.2. All intermediate fuzzy sets are assumed to have membership functions that are isosceles triangles. Each is defined by two points, \max_i and \min_i , where i is the label of the fuzzy set. The chromosome

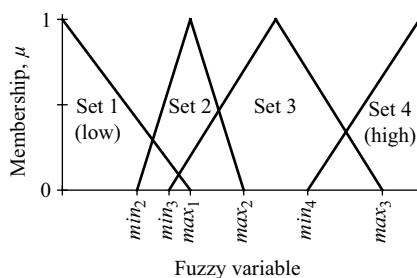


Figure 10.2 Defining triangular membership functions by their intercepts on the abscissa.

is then a list of all the points \max_i and \min_i that determine the complete set of membership functions. In several demonstrator systems, Karr's GA-modified fuzzy controller outperformed a fuzzy controller whose membership functions had been set manually. This success demonstrates that tuning the fuzzy sets is a numerical optimization problem.

10.3.3 Fuzzy–Genetic Systems

Fuzzy–genetic systems are the converse to the genetic–fuzzy systems introduced in Section 10.3.2. Here, fuzzy logic is used to set the parameters of a genetic algorithm. Herrera and Lozano (1996) have proposed a set of fuzzy rules that adjust the crossover and mutation probabilities, p_c and p_m , of a genetic algorithm depending on how well the algorithm is converging toward an optimum. Their approach was adapted by Khmeleva et al. (2018), who defined three measures of the GA performance, derived from the mean cost, $\overline{\text{cost}}_{(t)}$, and the lowest cost, $\text{cost}_{\text{best}(t)}$, at time step t :

- Convergence factor, $CF = \left(\frac{\text{cost}_{\text{best}(t-1)}}{\text{cost}_{\text{best}(t)}} - 1 \right) \times 100\%$
- Unimprovement factor, $UF = \text{number of cycles since a reduction in cost}$
- Variance factor, $VF = \frac{\overline{\text{cost}}_{(t)} - \text{cost}_{\text{best}(t)}}{\text{cost}_{\text{best}(t)}}$

They have applied these three measures to the crew-scheduling problem (CSP), specifically scheduling drivers for freight trains on busy rail networks, using the following set of fuzzy rules:

- If CF is high and UF is low, then p_m becomes low and p_c becomes high.
- If CF is medium and UF is low, then p_m becomes low and p_c becomes high.
- If CF is low and UF is high, then p_m becomes high and p_c becomes low.
- If CF is low and UF is medium, then p_m and p_c become medium.
- If CF is low and UF is low, then p_m becomes low and p_c becomes high.
- If UF is low and VF is low, then p_m becomes high and p_c becomes medium.
- If UF is low and VF is medium, then p_m and p_c become medium.

Khmeleva et al. (2018) have shown that the fuzzy-logic adjustment led to improved schedules in fewer generations, as shown in Figure 10.3.

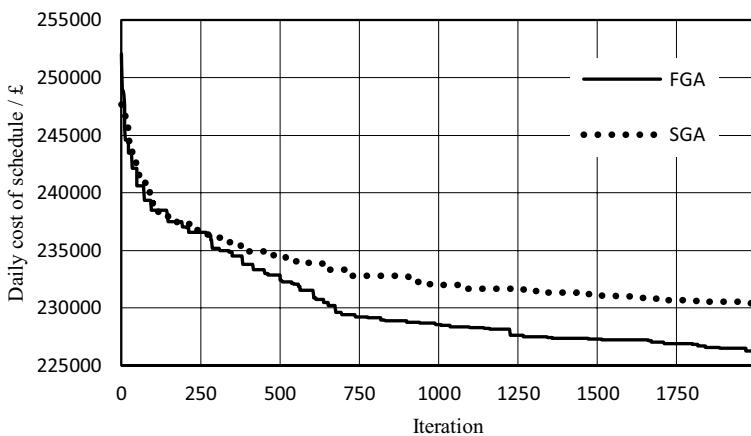


Figure 10.3 Performance of a standard GA (SGA) for the crew-scheduling problem compared with a fuzzy GA (FGA) in which fuzzy logic dynamically adjusts the GA parameters. (Derived from Khmeleva, E. et al. 2018.)

10.4 Capability Enhancement

One technique may be used within another to enhance the latter's capabilities. Here, three examples are described: neuro-fuzzy systems, which combine the benefits of neural networks and fuzzy logic; Lamarckian and Baldwinian inheritance for enhancing the performance of a genetic algorithm with local search around individuals in the population; and learning classifier systems (LCSs) that use genetic algorithms to discover rules.

10.4.1 Neuro-Fuzzy Systems

Section 10.3.2 showed how a genetic algorithm can be used to optimize the parameters of a fuzzy system. In such a scheme, the genetic algorithm for parameter setting and the fuzzy system that uses those parameters are distinct and separate. The parameters for a fuzzy system can also be learned using neural networks, but here much closer integration is possible between the neural network and the fuzzy system that it represents. A neuro-fuzzy system is a fuzzy system, the parameters of which are derived by a neural network learning technique. It can equally be viewed as a neural network that represents a fuzzy system. The two views are equivalent, and it is possible to express a neuro-fuzzy system in either form.

Consider the following fuzzy rules, based on the example used in Chapter 3:

```
fuzzy_rule r9_1f
if temperature is high
```

or water_level is high
then pressure becomes high.

```
fuzzy_rule r9_2f
  if temperature is medium
  or water_level is medium
  then pressure becomes medium.
```

```
fuzzy_rule r9_3f
  if temperature is low
  or water_level is low
  then pressure becomes low.
```

These fuzzy rules and the corresponding membership functions can be represented by the neural network shown in Figure 10.4. The first stage is fuzzification, in which any given input value for temperature is given a membership value for *low*, *medium*, and *high*. A single-layer perceptron, designated level 1 in Figure 10.4, can achieve this goal because it is a linear classification task. The only difference from other classifications met previously is that the target output values are not just 0 and 1, but any value in the range 0–1. A similar network is required at level 1 for the other input variable, *water_level*. The neurons whose outputs correspond to the *low*, *medium*, and *high* memberships are marked L, M, and H, respectively, in Figure 10.4.

Level 2 of the neuro–fuzzy network performs the role of the fuzzy rules, taking the six membership values as its inputs and generating as its outputs the memberships for *low*, *medium*, and *high* of the fuzzy variable *pressure*. The final stage, at level 3, involves combining these membership values to produce a defuzzified value for the output variable.

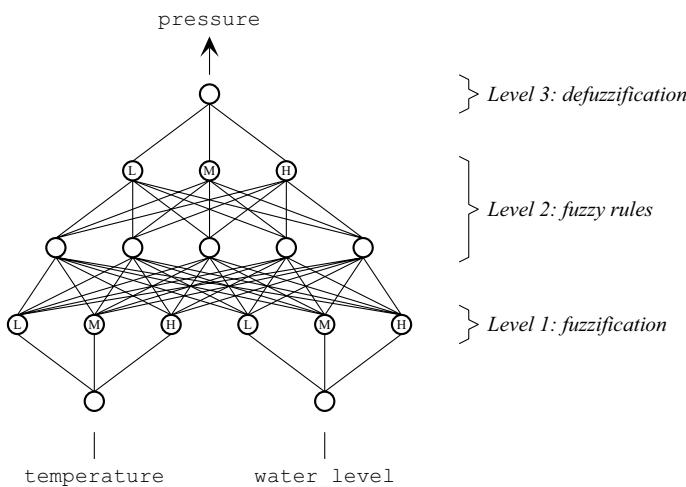


Figure 10.4 A neuro–fuzzy network.

The definitions of the fuzzy sets and the fuzzy rules are implicit in the connections and weights of the neuro–fuzzy network. Using a suitable learning mechanism, the weights can be learned from a series of examples. The network can then be used on previously unseen inputs to generate defuzzified output values. In principle, the fuzzy sets and rules can be inferred from the network and run as a fuzzy rule-based system, of the type described in Chapter 3, to produce identical results (Altug et al. 1999; Chow et al. 1999). The popular *ANFIS (adaptive-network-based fuzzy inference system)* is based on similar principles. It uses a multilayered feedforward neural network known as an adaptive network, in which each node performs a specific function dependent on its input data and individual parameters (Jang 1993).

10.4.2 Memetic Algorithms: Genetic Algorithms with Local Search

Capability enhancement of a genetic algorithm can be achieved by hybridizing it with local search procedures to produce what is known as a *memetic algorithm*. This form of hybridization can help the genetic algorithm to optimize individuals within the population, while maintaining the GA's ability to explore the search space. The aim would be either to increase the quality of the solutions, so they are closer to the global optimum, or to increase the efficiency, that is, the speed at which the optimum is found. One approach is to introduce an extra step involving local search in the immediate vicinity of each chromosome in the population, to see whether any of its neighbors offers a fitter solution.

A commonly used, if rather artificial, example is to suppose that the task is to find the maximum denary integer represented by a seven-bit binary-encoded chromosome. The optimum is clearly 1111111. A chromosome's fitness could be taken as the highest integer represented either by itself or by any of its near neighbors found by local search. A near neighbor of an individual might typically be defined as one that has a Hamming separation of 1 from it, that is, one bit is different. For example, the chromosome 0101100 (denary value 44) would have a fitness of 108, since it has a nearest neighbor 1101100 (denary value 108). The chromosome could be either:

- a. replaced by the higher scoring neighbor, in this case 1101100 (this is *Lamarckian inheritance*) or
- b. left unchanged while retaining the fitness value of its fittest neighbor (this is *Baldwinian inheritance*).

The first scheme, in which the chromosome is replaced by its fitter neighbor, is named after Lamarck (Lamarck 1809). Lamarck believed that, in nature, offspring can genetically inherit their parents' learned characteristics and behaviors. Although this concept is a controversial view of nature, it is nevertheless a useful technique for

genetic algorithms, which are inspired by nature but do not need to be an accurate biological simulation.

In the alternative scheme, Baldwinian inheritance, the chromosome is unchanged but a higher fitness is ascribed to it to acknowledge its proximity to a fitter solution. This model is named after Baldwin's view of biological inheritance (Baldwin 1896), which is now widely accepted in preference to Lamarck's. Baldwin argued that, although learned characteristics and behaviors cannot be inherited genetically, the capacity to learn such characteristics and behaviors can be.

El-Mihoub et al. (2004, 2006b) have set out to establish the relative benefits of the two forms of inheritance. In a series of experiments, they have adjusted the relative proportions of Lamarckian and Baldwinian inheritance, and the probability of applying a local steepest-gradient descent search to any individual in the population. Their results for a four-dimensional Schwefel function (Schwefel 1981) are shown in Figure 10.5. They have shown that the optimal balance between the two forms of inheritance depends on the population size, the probability of local search and the nature of the fitness landscape (El-Mihoub et al. 2006a; Hopgood 2005). With the goal of finding this balance automatically, they have also shown that these parameters can be made self-adaptive by coding them within the chromosomes (El-Mihoub et al. 2020). Taking this idea to its extreme conclusion, a completely self-adapting genetic algorithm could be envisaged where no parameters at all need to be specified by the user.

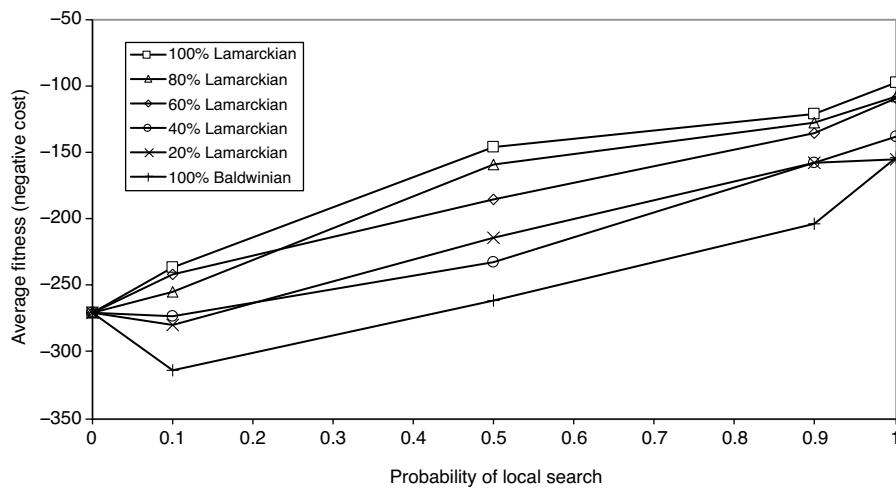


Figure 10.5 Effects of probability of local search and relative proportions of Lamarckian and Baldwinian inheritance for a four-dimensional Schwefel test function (Reprinted from Hopgood, A. A. 2005. The state of artificial intelligence. In *Advances in Computers* 65, edited by M. V. Zelkowitz. Elsevier, Oxford, UK, 1–75. Copyright (2005). With permission from Elsevier.)

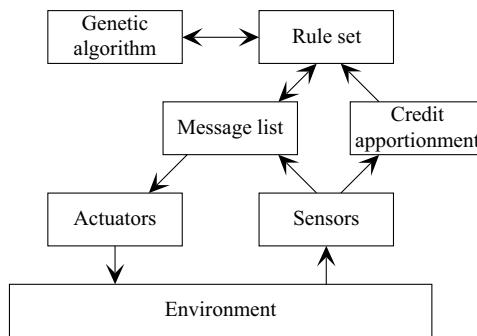


Figure 10.6 Learning classifier system.

10.4.3 Learning Classifier Systems (LCSs)

Holland's LCSs combine genetic algorithms with rule-based systems to provide a mechanism for rule discovery (Holland and Reitman 1978; Goldberg 1989). The rules are simple production rules (see Section 2.1), coded as a fixed-length mixture of binary numbers and wild-card, that is, “don’t care” characters. Their simple structure makes it possible to generate new rules by means of a genetic algorithm.

The overall LCS is illustrated in Figure 10.6. At the heart of the system is the message list, which fulfills a similar role to the blackboard in a blackboard system. Information from the environment is placed there, along with rule deductions and instructions for the actuators that act on the environment.

A *credit-apportionment* system, known as the *bucket-brigade algorithm*, is used to maintain a credit balance for each rule. The genetic algorithm uses a rule’s credit balance as the measure of its fitness. Conflict resolution (see Section 2.8) between rules in the conflict set is achieved via an auction, in which the rule with the most credits is chosen to fire. In doing so, it must pay some of its credits to the rules that led to its conditions being satisfied. If the fired rule leads to some benefit in the environment, it receives additional credits.

10.5 Clarification and Verification of Neural Network Outputs

Neural networks have the ability to learn associations between input vectors and their associated outputs. However, the underlying reasons for the associations may be opaque, as the associations are effectively encoded in the weightings on the interconnections between the neurons. A neural network is therefore often described as a “black box,” with an internal state that conveys no readily useful information to an observer. This metaphor implies that the inputs and outputs have significance for an observer, but the weights on the interconnections between the neurons do not.

This analogy contrasts with a transparent system, such as a KBS, where the internal state, such as the value of a variable, does have meaning for an observer.

There has been considerable research into *rule extraction*. Here, the aim is to produce automatically a set of intelligible rules that are equivalent to the trained neural network from which they have been extracted (Tsukimoto 2000). A variety of methods have been reported for extracting different types of rules, including production rules and fuzzy rules.

In safety-critical systems, reliance on the output from a neural network without any means of verification is not acceptable. It has, therefore, been proposed that rules be used to verify that the neural network output is consistent with its input (Picton et al. 1991). The use of rules for verification implies that at least some of the domain knowledge can be expressed in rule form. Johnson et al. (1990) suggest that an *adjudicator* module be used to decide whether a set of rules or a neural network is likely to provide the more reliable output for a given input. The adjudicator would have access to information relating to the extent of the neural network's training data and could determine whether a neural network would have to interpolate between, or extrapolate from, examples in the training set. This is important, since neural networks are good at interpolation but poor at extrapolation. The adjudicator may, for example, call upon rules to handle the exceptional cases that would otherwise require a neural network to extrapolate from its training data. If heuristic rules are also available for the less exceptional cases, then they could be used to provide an explanation for the neural network's findings. A supervisory rule-based module could dictate the training of a neural network, deciding how many nodes are required, adjusting the learning rate as training proceeds, and deciding when training should terminate.

10.6 Summary

This section has looked at just some of the ways in which different computation techniques—including intelligent systems and conventional programs—can work together within hybrid systems. Chapters 12 through 15 deal with applications of intelligent systems in engineering and science, most of which involve hybrids of some sort.

Further Reading

- Castillo, O., and P. Melin, eds. 2020. *Hybrid Intelligent Systems in Control, Pattern Recognition and Medicine*. Springer Nature, Cham, Switzerland.
- Zgurovsky, M., V. Sineglazov, and E. Chumachenko. 2020. *Artificial Intelligence Systems Based on Hybrid Neural Networks: Theory and Applications*. Springer Nature, Cham, Switzerland.

Chapter 11

AI Programming Languages and Tools

11.1 A Range of Intelligent Systems Tools

The previous chapters have introduced a range of intelligent systems techniques, covering both knowledge-based systems (KBSs) and computational intelligence. The tools available to assist in constructing intelligent systems can be roughly divided into the following categories:

- Expert-system shells, for example, Drools, CLIPS, and Jess.
- AI toolkits, for example, Flex/VisiRule for KBSs (see Chapter 2), the FLINT toolkit for modeling uncertainty (see Chapter 3), neural network packages such as TensorFlow, and multiagent tools such as DARBS (see Chapter 10).
- Libraries that extend the capabilities of programming environments like MATLAB, C++, Java, R, and Python.
- Object-oriented programming languages, for example, C++, Java, Smalltalk, and CLOS.
- Traditional procedural programming languages, for example, C, Pascal, and Fortran.
- AI programming languages for processing words, symbols, and relations, for example, Lisp and Prolog. Python includes these features and, as it has become the de facto language for AI, it will also be included in this group.

An expert-system shell is an expert system that is complete except for the knowledge base. Expert-system shells were an early form of AI toolkit. They were designed to allow the rapid design and implementation of rule-based expert systems, but often

lacked flexibility. Typically, an expert-system shell includes an inference engine, a user interface for programming, and a user interface for running the system. The programming interface might comprise a specialized editor for creating rules in a predetermined format, and some debugging tools. The shell user enters rules in a declarative fashion and ideally should not need to be concerned with the workings of the inference engine. In practice, this ideal is rarely met, and a typical difficulty in using a shell is ensuring that rules are applied when expected. As the user has no direct control over the inference engine, it is usually necessary to gain some insight into its workings and to tailor the rules to achieve the desired effect. This is not necessarily easy and detracts from the advantages of having a separate knowledge base. Nevertheless, shells are easy to use in other respects and allow a simple KBS to be constructed quickly. They are, therefore, useful for building prototype expert systems and for educational purposes. However, their inflexible facilities for knowledge representation and inference tend to limit their general applicability.

The programming languages offer much greater flexibility and can be used to build customized tools. Most programming languages are procedural rather than declarative, although the Prolog language incorporates both programming styles. Some KBS toolkits such as Flex and its VisiRule interface aim to provide the best of all worlds. They offer the programmer access to the underlying language and to tools for constructing rules, agents, and other knowledge-based entities. Figure 11.1 shows the example of a rule-based system for diagnosing faults in the braking system for a car. The relationships between the conditions, conclusions, and facts can

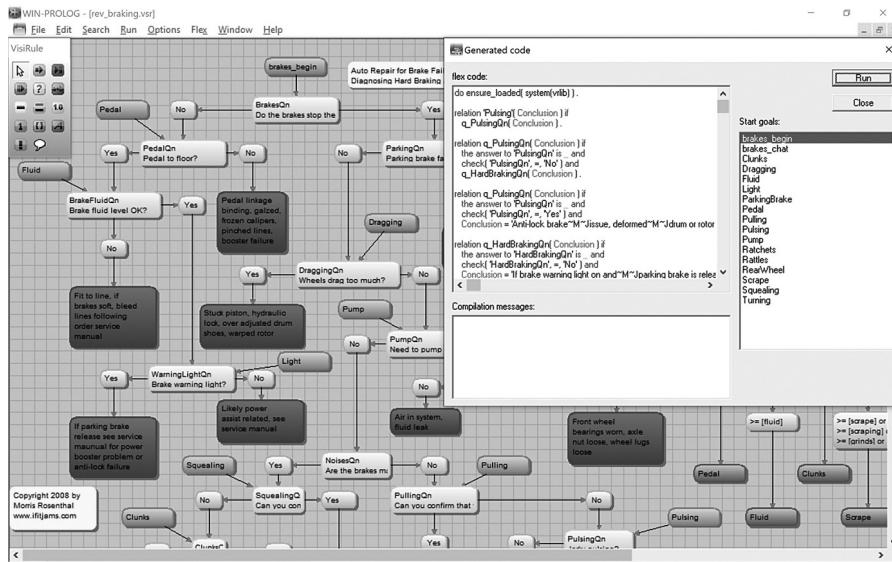


Figure 11.1 VisiRule representation of a brakes diagnosis system, showing the automated generation of Flex code. Original flowchart by Morris Rosenthal.

be visualized and edited graphically, while the corresponding Flex code can also be generated, viewed, and edited. Any changes in the visual representation are immediately reflected in the code. The provision of the visual tools can save considerable programming effort, while the ability to access the underlying language gives the freedom to build additional capabilities. Flex itself is built on the AI programming language, Prolog, introduced in Section 11.4, providing a further level of flexibility.

Libraries offer similar advantages to toolkits. Instead of supplying the complete programming environment, libraries provide specific functionality on the assumption that the underlying programming environment already exists. There is a range of KBS and computational intelligence libraries available, for example, for C++, MATLAB, Java, and Python.

11.2 Features of AI Languages

The remainder of this chapter will focus on the three main AI programming languages: Lisp, Prolog, and Python. A key feature of all three languages is the ability to manipulate symbolic data, that is, characters and words, as well as numerical data. One of the most important structures for this manipulation is lists, introduced in the following section.

11.2.1 Lists

A knowledge base may contain a mixture of numbers, letters, words, punctuation, and complete sentences. Most computer languages can handle such a mixture of characters, provided the general format can be anticipated in advance. Suppose we wanted to represent a simple fact such as:

```
pressure in valve_2 is 12.8 MPa.
```

where $1\text{MPa} = 1\text{MNm}^{-2}$. One possible C++ implementation (see Chapter 4) of this fact is to define a `Valve` class and the instance `valve2`, as follows:

```
class Valve
{
public:
    Valve(int id, float pres, const char *u); // constructor
    ~Valve(); // destructor
private:
    int identity_number;
    float pressure;
    char *units;
}
Valve::Valve(int id, float pres, const char *u)
{ //constructor definition
```

```

identity_number = id;
pressure = pres;
units = (char*) u;
}

Valve::~Valve()
{ //destructor definition
    cout << "instance of Valve destroyed" << endl;
}

main()
{
    Valve valve2(2, 12.8, "MPa"); // create an instance of Valve
}

```

The class definition could be specialized to handle different types of valves, but a completely new construction would be needed for a fact or rule with a different format, such as:

```
if pressure is above 10 MPa then close valve.
```

Lists are data structures that allow words, numbers, and symbols to be combined in a wide variety of ways. They are useful for symbol manipulation and are a key feature of AI programming languages. The preceding example could be represented as a list in Lisp, Prolog, or Python, respectively, as follows:

```
(close_valve (exceeds pressure 10 mpa))
[if, pressure, exceeds, 10, MPa, then, close, valve]
['if', 'pressure', 'exceeds', 10, 'MPa', 'then', 'close',
'velve']
```

Lisp uses round brackets with elements separated by spaces, whereas Prolog and Python use square brackets with elements separated by commas. The Lisp example includes a list within a list, that is, a *nested list*. Although the Prolog and Python examples look like a rule, each list is really just a list of words and numbers. Separate code would be required to interpret it as a rule.

Other languages such as C++ are sufficiently versatile to allow lists to be implemented by creating pairs of values and pointers, forming a so-called linked list. However, one strength of the AI languages is the integration of lists into the language, together with the necessary facilities for manipulating those lists.

11.2.2 Other Data Types

As well as lists, there are a number of other data types available in the AI languages. Unlike traditional programming languages, variables in the AI languages can be used to store any type of data. A list could be assigned to a given variable immediately

after assigning a real number to the same variable. The declaration of variables is not always necessary. However, declarations are normally made in Lisp to specify explicitly the scope of a variable. In Lisp, undeclared variables are assumed to be global, that is, memory is allocated for the whole duration of a program or interactive session.

Although the assignment of different types of data to variables is transparent to the programmer, the computer needs to know the type of data in order to handle it correctly. There are various techniques for meeting this requirement. Typically, the value associated with a variable includes a tag, hidden from the programmer, which labels its type. Some commonly used data types in the AI languages are shown in Table 11.1.

Lisp and Python also allow the creation of arrays and structures, similar to those used in C. Strings can be made up of any printable characters, including numbers and spaces, enclosed in quotation marks. Words can generally be regarded as a sequence of one or more characters without any spaces, as spaces and certain other characters may act as separators for words, depending on the language. Examples of words include variable names and the elements of the lists shown in Table 11.1. List elements are not always words, as lists can contain nested lists or numbers. The term *atom* is used to denote a fundamental data type that cannot be made up from other data types. For example, numbers and words are atoms, but lists are not.

During the execution of a program, various data structures may be created. There is, therefore, a need for management of the computer memory, that is, memory must be allocated when needed and subsequently reclaimed. In languages such as C, the responsibility for memory management rests with the programmer, who must allocate and reclaim memory at appropriate places in the program using the

Table 11.1 Some Data Types

Type	Examples
Integer	0, 23, -15
Real (floating point) number	3.1415927, -1.24
String	"a string in Lisp, Prolog, or Python" 'another Python string'
Word (Lisp and Prolog)	myword, x, z34
Word (Python)	'myword', 'x', 'z34'
List	(a list of words in Lisp) [a, list, in, Prolog] ['list', 'in', 'Python']

`malloc` and `free` commands, or their equivalent. In the AI languages and some object-oriented languages like Smalltalk, the memory is managed automatically. The programming environment must be capable of both dynamically allocating memory and freeing memory that is no longer required. The latter process is called *garbage collection*. This process is useful, although it can result in a momentary pause in the computer's response while the garbage collection takes place.

11.2.3 Programming Environments

In general, all of the AI languages considered here form part of their own interactive programming environment. Typically, there is a console window into which code can be typed. Such instructions are interpreted and obeyed immediately, and the output printed on the screen. They are often described as *interpreted* languages, that is, the program instructions are interpreted directly in the programming environment. Unlike a compiled language such as C++, there is no requirement to separately compile the source code into a machine-executable file that is run separately. In fact, this distinction between interpreted and compiled languages is not categorical, but it can at least be said that most implementations of the AI programming languages behave *as though* they are interpreted.

Typing commands into a console window is a useful way of inspecting the values of variables and testing ideas, but it is not a practical way of writing a program. Instead, a text editor is used to enter code so that it can subsequently be modified, saved, checked, and run. In an integrated programming environment, the code can be evaluated or compiled from within the editor. Debugging tools can stop the program at a selected point or when an error occurs, so the assignments up to that point can be examined.

Code that has been written in an AI language will, in general, run more slowly than a compiled C or C++ program. However, the appeal of AI languages is their power in terms of flexibility and clarity for the programmer, rather than their computational or memory efficiency.

11.3 Lisp

11.3.1 Background

It has already been noted that a feature of the AI languages is the integration of lists and list manipulation into the language. This is particularly so in the case of Lisp, as a Lisp program is itself a list made up of many lists. Indeed, the name *Lisp* is derived from the phrase *list processing*.

Historically, Lisp has developed in an unregulated fashion. Different syntax and features were introduced into different implementations, partly as a consequence of the existing hardware and software environment. As a result, many different Lisp

dialects such as Interlisp, Franzlisp, Maclisp, Zetalisp, and Scheme were developed. A standard was subsequently produced—Common Lisp—that was intended to combine the most useful and portable features of the dialects into one machine-independent language (Steele 1990). This standard form of the language is also the basis of CLOS (Common Lisp Object Standard), an object-oriented extension of Common Lisp. All the examples introduced here are based upon the definition of Common Lisp and should work on any Common Lisp system.

A list is a collection of words, numbers, strings, functions, and further lists, enclosed in parentheses. The following are all examples of lists:

```
(a b c d)
(My car is 10 years old)
(a list (a b c) followed by an empty list ())
```

Lisp uses the keyword `nil` to denote an empty list, so that `()` and `nil` are equivalent. Lisp extends the idea of a language based upon list manipulation to the point where everything is either a list or an element of a list. There are only a few basic rules to remember in order to understand how Lisp works. However, because its structure is so different from other languages, Lisp may seem rather strange to the novice.

11.3.2 Lisp Functions

It would be valid to describe Lisp as a procedural language, that is, the computer is told exactly what to do and in what order to do it. However, a more precise description would be that it is a *functional* language. That is because a Lisp program is made up of lists that are interpreted as functions that, by definition, return a single value.

The three key rules to understanding Lisp are as follows:

- Each item of a list is evaluated.
- The first item is interpreted as a function name.
- The remaining items are the parameters (or arguments) of the function.

Some of the functions with which we will be dealing are strictly speaking *macros*, which are predefined combinations of functions. However, the distinction need not concern us here. With a few exceptions, the parameters to a function are always evaluated before the function itself. The parameters themselves may also be functions and can even be the same function (thereby permitting recursion). The following example would be a valid Lisp call to the function `print`. The computer's prompt, which precedes user input, varies between implementations but is shown here as `<c1>`, for Common Lisp:

```
<c1> (print "hello world")
hello world
hello world
```

The first element in the list was interpreted as a function name, and the second item as its argument. The argument evaluates to the string “hello world.” It might seem surprising that `hello world` is printed twice. It is first printed because we instructed Lisp to do so. It is then printed again because Lisp always prints out the value of the function that it is given. In this example, the function `print` returned as its value the item that it had printed. Consider what will happen if we type the following:

```
<cl> (print hello)
Error: Unbound variable: HELLO.
```

This error message has come about because Lisp evaluates every item in the list. In this example, the interpreter tried to evaluate `hello`, but found it to be undefined. A defined variable is one that has a value (perhaps another function) assigned to it. However, in this case we did not really want `hello` to be evaluated. There are many instances in writing Lisp code when we would like to suppress Lisp’s habit of evaluating every argument. A special function, `quote`, is provided for this specific purpose.

The `quote` function takes only one argument, which it does not evaluate but simply returns in the same form that it is typed:

```
<cl> (quote hello)
hello
<cl> (print (quote hello))
hello
hello
```

The `quote` function is used so often that a shorthand form has been made available. The following are equivalent:

```
(quote hello)
```

and

```
'hello
```

We would also wish to suppress the habit of functions evaluating their arguments when making assignments. Here is an assignment in C++:

```
/* assign value 2.5 to my_variable in C++ */
my_variable = 2.5;
```

A value of `2.5` is assigned to `my_variable`, which would need to have been declared as type `float`. Lisp provides various functions for achieving this, one of which is `setf`:

```
<cl> (setf my_variable 2.5)
2.5
```

Since the first argument is a variable name, only the second argument to `setf` is evaluated. This is because we want to make an assignment to the variable name and not to whatever was previously assigned to it. The second parameter of `setf`, namely 2.5, evaluates to itself, and this value is then assigned to `my_variable`. Like all Lisp functions, `setf` returns a value, in this case 2.5, and this value is printed on the screen by the Lisp interpreter.

As we noted earlier, Lisp usually tries to evaluate the parameters of a function before attempting to evaluate the function itself. The parameters may be further functions with their own parameters. There is no prescribed limit to the embedding of functions within functions in this manner, so complex composite functions can easily be built up and perhaps given names so that they can be reused. The important rules for reading or writing Lisp code are as follows:

- List elements are interpreted as a function name and its parameters (unless the list is the argument to `quote`, `setf`, or a similar function).
- A function always returns a value.

Lisp code is constructed entirely from lists, while lists also represent an important form of data. Therefore, it is hardly surprising that built-in functions for manipulating lists form an important part of the Lisp language. Two basic functions for list manipulation are `first` and `rest`. For historical reasons, the functions `car` and `cdr` are also available to perform the same tasks. The `first` function returns the first item of that list, while `rest` returns the list but with the first item removed. Here are some examples:

```
<cl>(first '(a b c d))
a
<cl>(rest '(a b c d))
(b c d)
<cl>(first (rest '(a b c d)))
b
<cl>(first '((a b) (c d)))
(a b)
```

Used together, `first` and `rest` can find any element in a list. However, two convenient functions for finding parts of a list are `nth` and `nthcdr`:

```
(nth 0 x) finds the 1st element of list x
(nth 1 x) finds the 2nd element of list x
(nth 2 x) finds the 3rd element of list x
(nthcdr 2 x) is the same as (rest (rest x))
(nthcdr 3 x) is the same as (rest (rest (rest x)))
```

Note that `rest` and `nthcdr` both return a list, while `first` and `nth` may return a list or an atom. When `rest` is applied to a list that contains only one element, an

empty list is returned, which is written as () or nil. When either `first` or `rest` is applied to an empty list, `nil` is also returned.

There are many other functions provided in Lisp for list manipulation and program control. It is not intended that this overview of Lisp should introduce them all. The purpose of this section is not to replace the many texts on Lisp, but to give the reader a feel for the unusual syntax and structure of Lisp programs. We will see, by means of a worked example, how these programs can be constructed and, in so doing, we will meet some of the important functions in Lisp.

11.3.3 A Worked Example

We will discuss in Chapter 13 the application of intelligent systems to problems of selection. One of the selection tasks that will be discussed in some detail is the selection of an appropriate material from which to manufacture a product or component. For the purposes of this worked example, let us assume that we wish to build a shortlist of polymers that meet some numerical specifications. We will endeavor to solve this problem using Lisp, and later in this chapter will encode the same example in Prolog and Python. The problem is specified in detail in Box 11.1.

BOX 11.1 PROBLEM DEFINITION: FINDING MATERIALS THAT MEET SOME SPECIFICATIONS

In Chapter 13, the problem of selecting materials will be discussed. One approach involves (among other things) finding those materials that meet a specification or list of specifications. This task is used here as an illustration of the features of the three main AI languages. The problem is to define a Lisp or Python function, or Prolog relation, called `accept`. When a materials specification or set of specifications is passed as parameters to `accept`, it should return a list of materials in the database that meet the specification(s). The list returned should contain pairs of material names and types. Thus, in Lisp syntax, the following would be a valid result from `accept`:

```
( (POLYPROPYLENE . THERMOPLASTIC)
  (POLYURETHANE_FOAM . THERMOSET) )
```

In this case, the list returned contains two elements, corresponding to the two materials that meet the specification. Each of the two elements is itself a list of two elements: a material name and its type. Here, each sublist is a special kind of list in Lisp called a *dotted pair* (see Section 11.3.3).

The specification that is passed to accept as a parameter will have a pre-defined format. If just one specification is to be applied, it will be expressed as a list of the form:

```
(property_name minimum_value tolerance)
```

An example specification in Lisp would be:

```
(flexural_modulus 1.0 0.1)
```

The units of flexural modulus are assumed to be GNm⁻². For a material to meet this specification, its flexural modulus must be at least 1.0 minus 0.1, that is, 0.9 GNm⁻². If more than one specification is to be applied, the specifications will be grouped together into a list of the form:

```
((property1 minimum_value1 tolerance1)
 (property2 minimum_value2 tolerance2)
 (property3 minimum_value3 tolerance3))
```

When presented with a list of this type, accept should find those materials in the database that meet all of the specifications.

The first part of our program will be the setting up of some appropriate materials data. There are a number of ways of doing this, including the creation of a list called materials_database:

```
(defvar materials_database nil)
(setf materials_database '(
  (abs thermoplastic
    (impact_resistance 0.2)
    (flexural_modulus 2.7)
    (maximum_temperature 70))
  (polypropylene thermoplastic
    (impact_resistance 0.07)
    (flexural_modulus 1.5)
    (maximum_temperature 100)))
  (polystyrene thermoplastic
    (impact_resistance 0.02)
    (flexural_modulus 3.0)
    (maximum_temperature 50)))
  (polyurethane_foam thermoset
    (impact_resistance 1.06)
    (flexural_modulus 0.9)
    (maximum_temperature 80)))
```

```
(pvc thermoplastic
  (impact_resistance 1.06)
  (flexural_modulus 0.007)
  (maximum_temperature 50))
(silicone thermoset
  (impact_resistance 0.02)
  (flexural_modulus 3.5)
  (maximum_temperature 240))))
```

The function `defvar` is used to declare the variable `materials_database` and to assign to it an initial value, in this case `nil`. The function `setf` is used to assign the list of materials properties to `materials_database`. In a real application we would be more likely to read this information from a file than to have it “hard coded” into our program, but this representation will suffice for the moment. The variable `materials_database` is used to store a list, each element of which is also a list. Each of these nested lists contains information about one polymer. The information is in the form of a name and category, followed by further lists in which property names and values are stored. Since `materials_database` does not have a predeclared size, materials and materials properties can be added or removed with ease.

Our task is to define a Lisp function, to be called `accept`, that takes a set of specifications as its parameters and returns a list of polymers (both their names and types) that meet the specifications. To define `accept`, we will use the function `defun`, whose purpose is to define a function. Like `quote`, `defun` does not evaluate its arguments. We will define `accept` as taking a single parameter, `spec_list`, that is a list of specifications to be met. Comments are indicated by a semi-colon:

```
(defun accept (spec_list)
  (let ((shortlist (setup)))
    (if (atom (first spec_list))
;; if the first element of spec_list is an atom,
;; then consider the specification
        (setf shortlist (meets_one spec_list shortlist))
;; else consider each specification in turn
        (dolist (each_spec spec_list)
          (setf shortlist (meets_one each_spec shortlist))))
    shortlist)) ;return shortlist
```

In defining `accept`, we have assumed the form that the specification `spec_list` will take. It will be either a list of the form:

```
(property minimum_value tolerance)
```

or a list made up of several such specifications:

```
((property1 minimum_value1 tolerance1)
 (property2 minimum_value2 tolerance2)
 (property3 minimum_value3 tolerance3))
```

Where more than one specification is given within `spec_list`, they must all be met.

The first function of `accept` is `let`, a function that allows us to declare and initialize local variables. In our example, the variable `shortlist` is declared and assigned the value returned by the function `setup`, which we have still to define. The variable `shortlist` is local in the sense that it exists only between `(let` and the matching closing bracket.

There then follows a Lisp conditional statement. The `if` function is used to test whether `spec_list` comprises one or many specifications. It does this by ascertaining whether the `first` element of `spec_list` is an atom. If it is, then `spec_list` contains only one specification, and the `first` element of `spec_list` is expected to be the name of the property to be considered. If, on the other hand, `spec_list` contains more than one specification, then its `first` element will be a list representing the first specification, so the test for whether the `first` element of `spec_list` is an atom will return `nil` (meaning “false”).

The general form of the `if` function is:

```
(if condition function1 function2)
```

which is interpreted as:

```
if <condition> then <do function1> else <do function2>.
```

In our example, `function1` involves setting the value of `shortlist` to the value returned by the function `meets_one` (yet to be defined), which is passed the single specification and the current `shortlist` of materials as its parameters. The “else” part of the conditional function (`function2`) is more complicated, comprising the `dolist` control function and its parameters. In our example, `dolist` initially assigns the first element of `spec_list` to the local variable `each_spec`. It then evaluates all functions between `(dolist` and its associated closing bracket. In our example, there is only one function, which sets `shortlist` to the value returned from the function `meets_one`. Since `dolist` is an iterative control function, it will repeat the process with the second element of `spec_list`, and so on until there are no elements left.

It is our intention that the function `accept` should return a list of polymers that meet the specification or specifications. When a function is made up of many

functions, the value returned is the last value to be evaluated. To ensure that the value returned by `accept` is the final shortlist of polymers, `shortlist` is evaluated as the last line of the definition of `accept`. Note that the bracketing is such that this evaluation takes place within the scope of the `let` function, within which `shortlist` is a local variable.

We have seen that the first task performed within the function `accept` was to establish the initial shortlist of polymers by evaluating the function `setup`. This function produces a list of all polymers (and their types) that are known to the system through the definition of `materials_database`. The `setup` function is defined as follows:

```
(defun setup ()
  (let ((shortlist nil))
    (dolist (material materials_database)
      (setf shortlist (cons (cons (first material)
        (nth 1 material)) shortlist)))
    shortlist) ;return shortlist
```

The empty brackets at the end of the first line of the function definition signify that `setup` takes no parameters. As in the definition of `accept`, `let` is used to declare a local variable and to give it an initial value of `nil`, that is, the empty list. Then `dolist` is used to consider each element of `materials_database` in turn and to assign it to the local variable `material`. Each successive value of `material` is a list comprising the polymer name, its type, and lists of properties and values. Our aim is to extract from it a list comprising a name and type only, and to collect all such two-element lists together into the list `shortlist`. The Lisp function `cons` adds an element to a list, and it can therefore be used to build up `shortlist`.

Assuming that its second parameter is a list, `cons` will return the result of adding its first parameter to the front of that list. The function can be illustrated by the following example:

```
<c1>(cons 'a '(b c d))
(a b c d)
```

However, the use of `cons` is still valid even if the second parameter is not a list. In such circumstances, a special type of two-element list, called a *dotted pair*, is produced:

```
<c1>(cons 'a 'b)
(a . b)
```

In our definition of `setup` we have used two embedded calls to `cons`, one of which produces a dotted pair while the other produces an ordinary list. The most deeply embedded (or *nested*) call to `cons` is called first, as Lisp always evaluates parameters to a function before evaluating the function itself. So, the first `cons` to be evaluated

returns a dotted pair comprising the first element of material (which is a polymer name) and the second element of material (which is the polymer type). The second cons to be evaluated returns the result of adding the dotted pair to the front of shortlist. Then shortlist is updated to this value by the call of setf. As in the definition of accept, shortlist is evaluated after the dolist loop has terminated, to ensure that setup returns the last value of shortlist.

As we have already seen, accept passes a single specification, together with the current shortlist, as parameters to the function meets_one. It is this function that performs the most important task in our program, namely, deciding which materials in the shortlist meet the specification:

```
(defun meets_one (spec shortlist)
  (dolist (material shortlist)
    (let ((actual (get_database_value (first spec)
                                      (first material))))
      (if (< actual (- (nth 1 spec) (nth 2 spec)))
          (setf shortlist (remove material shortlist)))))

    ;;pseudocode equivalent:
    ;;if actual < (value - tolerance)
    ;;then shortlist = shortlist without material
  shortlist) ;return shortlist
```

We have already met most of the Lisp functions that make up meets_one. In order to consider each material-type pair in the shortlist, dolist is used. The actual value of a property (such as maximum operating temperature) for a given material is found by passing the property name and the material name as arguments to the function get_database_value, which has yet to be defined. The value returned from get_database_value is stored in the local variable actual. The value of actual is then compared with the result of subtracting the specified tolerance from the specified target value. In our example, the tolerance is used simply to make the specification less severe. It may be surprising at first to see that subtraction and arithmetic comparison are both handled as functions within Lisp. Nevertheless, this treatment is consistent with other Lisp operations. The code includes a comment with some pseudocode that shows the conventional positioning of the operators in a traditional programming language. Note that nth is used in order to extract the second and third elements of spec, which represent the specification value and tolerance, respectively.

If the database value, actual, of the property in question is less than the specification value minus the tolerance, then the material is removed from the shortlist. The Lisp function remove is provided for this purpose. Because the arguments to the function are evaluated before the function itself, it follows that remove is not able to alter shortlist itself. Rather, it returns the list that would be produced if material were removed from a copy of shortlist. Therefore, setf has to be used to set shortlist to this new value.

As in our other functions, `shortlist` is evaluated, so its value will be returned as the value of the function `meets_one`. At a glance, it may appear that this step is unnecessary, as `setf` would return the value of `shortlist`. However, it is important to remember that this function is embedded within other functions. The last function to be evaluated is in fact `dolist`. When `dolist` terminates by reaching the end of `shortlist`, it returns the empty list, `()`, which is clearly not the desired result.

There now remains only one more function to define in order to make our Lisp program work, namely, `get_database_value`. As mentioned previously, `get_database_value` should return the actual value of a property for a material, when the property and material names are passed as arguments. Common Lisp provides us with the function `find`, which is ideally suited to this task. The syntax of `find` is best illustrated by example. The following function call will search the list `materials_database` until it finds a list whose `first` element is identically equal (`eq`) to the value of `material`:

```
(find material materials_database :test #'eq :key #'first)
```

Other tests and keys can be used in conjunction with `find`, as defined by Steele (1990). Having found the data corresponding to the material of interest, the name and type can be removed using `nthcdr`, and `find` can be called again in order to find the list corresponding to the property of interest. The function `get_database_value` can therefore be written as follows:

```
(defun get_database_value (prop_name material)
  (nth 1 (find prop_name
    (nthcdr 2
      (find material materials_database :test #'eq :key #'first))
    :test #'eq :key #'first)))
```

We are now in a position to put our program together, as shown in Box 11.2, and to ask Lisp some questions about polymers:

```
<cl> (accept '(maximum_temperature 100 5))
((SILICONE . THERMOSET) (POLYPROPYLENE . THERMOPLASTIC))
<cl> (accept '((maximum_temperature 100 5) (impact_resistance
      0.05 0)))
((POLYPROPYLENE . THERMOPLASTIC))
```

Once the function definitions have been evaluated, each function (such as `accept`) becomes known to Lisp in the same way as all of the predefined functions such as `setf` and `dolist`. Therefore, we have extended the Lisp language so that it has become specialized for our own specific purposes. It is this ability that makes Lisp such a powerful and flexible language. If a programmer does not like the facilities

BOX 11.2 A WORKED EXAMPLE IN LISP

```
(defvar materials_database nil)
(setf materials_database '(

  (abs thermoplastic
    (impact_resistance 0.2)
    (flexural_modulus 2.7)
    (maximum_temperature 70))
  (polypropylene thermoplastic
    (impact_resistance 0.07)
    (flexural_modulus 1.5)
    (maximum_temperature 100))
  (polystyrene thermoplastic
    (impact_resistance 0.02)
    (flexural_modulus 3.0)
    (maximum_temperature 50))
  (polyurethane_foam thermoset
    (impact_resistance 1.06)
    (flexural_modulus 0.9)
    (maximum_temperature 80))
  (pvc thermoplastic
    (impact_resistance 1.06)
    (flexural_modulus 0.007)
    (maximum_temperature 50))
  (silicone thermoset
    (impact_resistance 0.02)
    (flexural_modulus 3.5)
    (maximum_temperature 240)))))

(defun accept (spec_list)
  (let ((shortlist (setup)))
    (if (atom (first spec_list))
        ;;if the first element of spec_list is an atom,
        ;;then consider the specification
        (setf shortlist (meets_one spec_list shortlist))
        ;;else consider each specification in turn
        (dolist (each_spec spec_list)
          (setf shortlist (meets_one each_spec shortlist))))
    shortlist)) ;return shortlist

(defun setup ()
  (let ((shortlist nil))
    (dolist (material materials_database)
      (setf shortlist (cons (cons (first material)
        (nth 1 material)) shortlist)))
    shortlist)) ;return shortlist
```

(Continued)

BOX 11.2 (Continued)

```
(defun meets_one (spec shortlist)
  (dolist (material shortlist)
    (let ((actual (get_database_value (first spec)
                                      (first material))))
      (if (< actual (- (nth 1 spec) (nth 2 spec)))
          (setf shortlist (remove material shortlist)))))

    ;;pseudocode equivalent:
    ;;if actual < (value - tolerance)
    ;;then shortlist = shortlist without material
    shortlist) ;return shortlist

(defun get_database_value (prop_name material)
  (nth 1 (find prop_name
                (nthcdr 2
                  (find material materials_database :test #'eq :key
#'(first))
                  :test #'eq :key #'first))))
```

that Lisp offers, he or she can alter the syntax by defining his or her own functions, thereby producing an alternative language. Lisp-based KBS toolkits can be built on this basis.

11.4 Prolog

11.4.1 Background

Prolog is an AI language that can be programmed declaratively. It is, therefore, very different from Lisp, which is a procedural (or, more precisely, functional) language that can nevertheless be used to build declarative applications. As we will see, although Prolog can be used declaratively, an appreciation of the procedural behavior of the language is needed. In other words, programmers need to understand how Prolog uses the declarative information that they supply.

Prolog is suited to symbolic (rather than numerical) problems, particularly logical problems involving relationships between items. It is also suitable for tasks that involve data lookup and retrieval, as pattern-matching is fundamental to the functionality of the language. Because Prolog is so different from other languages in its underlying concepts, many newcomers find it a difficult language. Whereas most languages can be learned rapidly by someone with computing experience, Prolog is perhaps more easily learned by someone who has never programmed.

11.4.2 A Worked Example

The main building blocks of Prolog are lists (as in Lisp) and *relations*, which can be used to construct *clauses*. We will demonstrate the declarative nature of Prolog programs by constructing a small program for selecting polymers from a database of polymer properties. The task will be identical to that used to illustrate Lisp, namely selecting from a database those polymers that meet a numerical specification or set of specifications. The problem is described in more detail in Box 11.1. We have already said that Prolog is good for data lookup, so let us begin by creating a small database containing some properties of materials. Our database will comprise a number of clauses such as this one involving the relation `materials_database`:

```
materials_database(polypropylene, thermoplastic,
[maximum_temperature, 100]).
```

This clause means that the three items in parentheses are related through the relation called `materials_database`. The third argument of the clause is a list (denoted by square brackets), while the first two arguments are atoms. The clause is our first piece of Prolog code, and it is purely declarative. We have given the computer some information about polypropylene, and this is sufficient to produce a working (though trivial) program. Even though we have not given Prolog any procedural information, that is, we have not told it how to use the information about polypropylene, we can still ask it some questions. Having typed the previous line of Prolog code, not forgetting the period, we can ask Prolog the question:

What type of material is polypropylene?

Depending on the Prolog implementation, the screen prompt is usually `?-`, indicating that what follows is treated as a query. Our query to Prolog could be expressed as:

```
?- materials_database(polypropylene, Family, _).
```

Prolog would respond:

```
Family = thermoplastic
```

This simple example illustrates several features of Prolog. Our program is a single line of code that states that polypropylene is a material of type thermoplastic and has a maximum operating temperature of 100 ($^{\circ}\text{C}$ assumed). Thus, the program is purely declarative. We have told Prolog what we know about polypropylene, but we have given Prolog no procedural instructions about what to do with that information. Nevertheless, we were able to ask a sensible question and receive a sensible reply. Our query includes some distinct data types. As `polypropylene` began with a lowercase letter, it was recognized as a constant, whereas `Family` was recognized

as a local variable by virtue of beginning with an uppercase letter. These distinctions stem from the following rules, which are always observed:

- local variables in Prolog can begin either with an uppercase letter or with an underscore character (for example, `x`, `My_variable`, `_another` are all valid variable names); and
- constants begin with a lowercase letter (for example, `adrian`, `polypropylene`, `pi` are all valid names for constants).

When presented with our query, Prolog has attempted to *match* the query to the relations (only one relation in our example) that it has stored. In order for any two terms to match, either:

- the two terms must be identical; or
- it must be possible to set (or *instantiate*) any local variables in such a way that the two terms become identical.

If Prolog is trying to match two or more clauses and comes across multiple occurrences of the same local variable name, it will always instantiate them identically. The only exception to this rule is the underscore character, which has a special meaning when used on its own. Each occurrence of the underscore character's appearing alone means:

I don't care what “_” matches so long as it matches something.

Multiple occurrences of the character can be matched to different values. The ‘_’ character is used when the value of a variable is not needed in the evaluation of a clause. Thus:

```
materials_database(polypropylene, thermoplastic,
[maximum_temperature, 100]).
```

matches:

```
materials_database(polypropylene, Family, _).
```

The relation name, `materials_database`, and its number of arguments (or its *arity*) are the same in each case. The first argument to `materials_database` is identical in each case, and the remaining two can be made identical by instantiating the variable `Family` to `thermoplastic` and the underscore variable to the list `[maximum_temperature, 100]`. We don't care what the underscore variable matches, so long as it matches something.

Now let us see if we can extend our example into a useful program. First, we will make our database more useful by adding some more data:

```
materials_database(abs, thermoplastic,
[[impact_resistance, 0.2],
```

```

[flexural_modulus, 2.7],
[maximum_temperature, 70]]).
materials_database(polypropylene, thermoplastic,
[[impact_resistance, 0.07],
[flexural_modulus, 1.5],
[maximum_temperature, 100]]).
materials_database(polystyrene, thermoplastic,
[[impact_resistance, 0.02],
[flexural_modulus, 3.0],
[maximum_temperature, 50]]).
materials_database(polyurethane_foam, thermoset,
[[impact_resistance, 1.06],
[flexural_modulus, 0.9],
[maximum_temperature, 80]]).
materials_database(pvc, thermoplastic,
[[impact_resistance, 1.06],
[flexural_modulus, 0.007],
[maximum_temperature, 50]]).
materials_database(silicone, thermoset,
[[impact_resistance, 0.02],
[flexural_modulus, 3.5],
[maximum_temperature, 240]]).

```

Our aim is to build a program that can select from the database those materials that meet a set of specifications. This requirement can be translated directly into a Prolog rule:

```

accept(Material,Type,Spec_list) :-
    materials_database(Material,Type,Stored_data),
    meets_all_specs(Spec_list,Stored_data).

```

The :- symbol stands for the word “if” in the rule. Thus, the rule just stated means:

accept a material, given a list of specifications, if that material is in the database and if the stored data about the material meet the specifications.

We now have to let Prolog know what we mean by a material meeting all of the specifications in the user’s specification list. The simplest case is when there are no specifications at all, in other words, the specification list is empty. In this case, the (nonexistent) specifications will be met regardless of the stored data. This fact can be simply coded in Prolog as:

```
meets_all_specs([],_).
```

The next most straightforward case to deal with is when there is only one specification, which we can code as follows:

```

meets_all_specs(Spec_list, Data) :-
    Spec_list= [Spec1|Rest],

```

```
atom(Spec1),
meets_one_spec([Spec1|Rest],Data).
```

This rule introduces the list separator `|`, which is used to separate the first element of a list from the rest of the list. As an example, consider the following Prolog query and its response:

```
?- [Spec1|Rest] = [flexural_modulus, 1.0, 0.1].
Spec1 = flexural_modulus, Rest = [1.0, 0.1]
```

The assignments to the variables immediately before and after the list separator are equivalent to taking the `first` and `rest` of a list in Lisp. Consistent with this comparison, the item immediately following a `|` symbol will always be instantiated to a list. Returning now to our rule, the first condition requires Prolog to try to match `Spec_list` to the template `[Spec1|Rest]`. If the match is successful, `Spec1` will become instantiated to the first element of `Spec_list` and `Rest` instantiated to `Spec_list` with its first element removed.

We could make our rule more compact by combining the first condition of the rule with the arguments to the goal:

```
meets_all_specs([Spec1|Rest],Data) :-
    atom(Spec1),
    meets_one_spec([Spec1|Rest],Data).
```

If the match is successful, we need to establish whether `Spec_list` contains one or many specifications. This goal can be achieved by testing the type of its first element. If the first element is an atom, the user has supplied a single specification, whereas a list indicates that more than one specification has been supplied. All this assumes that the intended format was used for the query. The built-in Prolog relation `atom` succeeds if its argument is an atom, and otherwise it fails.

We have not yet told Prolog what is meant by the relation called `meets_one_spec`, but we will do so shortly. Next, we will consider the general case where the user has supplied several specifications:

```
meets_all_specs([Spec1|Rest],Data) :-
    not atom(Spec1),
    meets_one_spec(Spec1,Data),
    meets_all_specs(Rest,Data).
```

An important feature demonstrated by this rule is the use of recursion, that is, the reuse of `meets_all_specs` within its own definition. Our rule says that the stored data meets the user's specification if each of the following is satisfied:

- We can separate the first specification from the remainder.
- The first specification is not an atom.

- The stored data meet the first specification.
- The stored data meet all of the remaining specifications.

When presented with a list of specifications, individual specifications are stripped off the list one at a time, and the rule is deemed to have been satisfied if the stored data satisfy each of them.

Having dealt with multiple specifications by breaking down the list into a set of single specifications, it now remains for us to define what we mean by a specification being met. This requirement is coded as follows:

```
meets_one_spec([Property, Spec_value, Tolerance], List) :-
    member([Property, Actual_value], List),
    Actual_value > Spec_value - Tolerance.
```

As in the Lisp example, we explicitly state that a user's specification must be in a fixed format, that is, the material property name, its target value, and the tolerance of that value must appear in sequence in a list. A new relation called `member` is introduced in order to check that the stored data for a given material include the property being specified, and to assign the stored value for that property to `Actual_value`. If the relation `member` is not built into our particular Prolog implementation, we will have to define it ourselves. Finally, the stored value is deemed to meet the specification if it is greater than the specification minus the tolerance.

The definition of `member`, taken from Bratko (2011), is similar in concept to our definition of `meets_all_specs`. The definition is that an item is a member of a list if that item is the first member of the list or if the list may be split so that the item is the first member of the second part of the list. This can be expressed more concisely and elegantly in Prolog than it can in English:

```
member(A, [A|L]) .
member(A, [_|L]) :- member(A, L).
```

Our program is now complete and ready to be interrogated. The program is shown in full in Box 11.3. In order to run a Prolog program, Prolog must be set a goal that it can try to prove. If successful, it will return all of the sets of instantiations necessary to satisfy that goal. In our case the goal is to find the materials that meet some specifications.

Now let us test our program with some example queries (or goals). First, we will determine which materials have a maximum operating temperature of at least 100°C, with a 5°C tolerance:

```
?- accept(M, T, [maximum_temperature, 100, 5]).  
M = polypropylene, T = thermoplastic;  
M = silicone, T = thermoset;  
no
```

BOX 11.3 A WORKED EXAMPLE IN PROLOG

```

materials_database(abs, thermoplastic,
  [[impact_resistance, 0.2],
   [flexural_modulus, 2.7],
   [maximum_temperature, 70]]).
materials_database(polypropylene, thermoplastic,
  [[impact_resistance, 0.07],
   [flexural_modulus, 1.5],
   [maximum_temperature, 100]]).
materials_database(polystyrene, thermoplastic,
  [[impact_resistance, 0.02],
   [flexural_modulus, 3.0],
   [maximum_temperature, 50]]).
materials_database(polyurethane_foam, thermoset,
  [[impact_resistance, 1.06],
   [flexural_modulus, 0.9],
   [maximum_temperature, 80]]).
materials_database(pvc, thermoplastic,
  [[impact_resistance, 1.06],
   [flexural_modulus, 0.007],
   [maximum_temperature, 50]]).
materials_database(silicone, thermoset,
  [[impact_resistance, 0.02],
   [flexural_modulus, 3.5],
   [maximum_temperature, 240]]).

accept(Material,Type,Spec_list) :-
  materials_database(Material,Type,Stored_data),
  meets_all_specs(Spec_list,Stored_data).

meets_all_specs([],_).
meets_all_specs([Spec1|Rest],Data) :-
  atom(Spec1),
  meets_one_spec([Spec1|Rest],Data).
meets_all_specs([Spec1|Rest],Data) :-
  not atom(Spec1),
  meets_one_spec(Spec1,Data),
  meets_all_specs(Rest,Data).

meets_one_spec([Property, Spec_value, Tolerance], List) :-
  member([Property, Actual_value], List),
  Actual_value > Spec_value - Tolerance.

member(A, [A|L]).
member(A, [_|L]) :- member(A, L).

```

The word no at the end indicates that Prolog's final attempt to find a match to the specification, after it had already found two such matches, was unsuccessful. We can now extend our query to find all materials that, as well as meeting the temperature requirement, have an impact resistance of at least 0.05 kJ/m:

```
?- accept(M,T, [[maximum_temperature, 100, 5],
   [impact_resistance, 0.05, 0]]).
M = polypropylene, T = thermoplastic;
no
```

11.4.3 Backtracking in Prolog

So far, we have seen how to program declaratively in Prolog, without giving any thought to how Prolog uses the declarative program to decide upon a sequential series of actions. In the example shown in the previous section, it was not necessary to know how Prolog used the information supplied to arrive at the correct answer. This situation represents the ideal of declarative programming in Prolog. However, the Prolog programmer invariably needs to have an idea of the procedural behavior of Prolog in order to ensure that a program performs correctly and efficiently. In many circumstances, it is possible to type a valid declarative program, but for the program to fail to work as anticipated because the programmer has failed to take into account how Prolog works.

Let us start by considering our last example query to Prolog:

```
?- accept(M,T, [[maximum_temperature, 100, 5],
   [impact_resistance, 0.05, 0]]).
```

Prolog treats this query as a goal, whose truth it attempts to establish. As the goal contains some variables (M and T), these will need to be instantiated in order to achieve the goal. Prolog's first attempt at achieving the goal is to see whether the program contains any clauses that directly match the query. In our example it does not, but it does find a rule with the `accept` relation as its conclusion:

```
accept(Material,Type,Spec_list) :-
  materials_database(Material,Type,Stored_data),
  meets_all_specs(Spec_list,Stored_data).
```

Prolog now knows that the goal will be achieved if it can establish the two conditions of this rule with M matched to Material, T matched to Type, and Spec_list instantiated to [[maximum_temperature, 100, 5], [impact_resistance, 0.05, 0]]. The two conditions then become goals in their own right. The first one, involving the relation `materials_database`, is easily achieved, and the second condition:

```
meets_all_specs(Spec_list,Stored_data).
```

becomes the new goal. Prolog's first attempt at satisfying this goal is to look at the relation:

```
meets_all_specs([], _).
```

However, this does not help as `Spec_list` is not instantiated to an empty list. Prolog must at this point *backtrack* to try another way of achieving the current subgoal. In other words, Prolog remembers the stage where it was before the failed attempt and resumes its reasoning along another path from there. Figure 11.2 shows the reasoning followed by Prolog when presented with the goal:

```
?- accept(M, T, [[maximum_temperature, 100, 5],  
[impact_resistance, 0.05, 0]]).
```

The illustration shows Prolog's first attempt at a solution, namely, `M=abs` and `T=thermoplastic`, and the steps that are followed before rejecting these particular instantiations as a solution. The use of backtracking is sensible up until the point where it is discovered that the maximum operating temperature of ABS (acrylonitrile-butadiene-styrene) is too low. When this has been determined, we would ideally like the program to reject ABS as a candidate material and move on to the next contender. However, Prolog does not give up so easily. Instead, it backtracks through every step that it has taken, checking to see whether there may be an alternative solution (or set of instantiations) that could be used. Ultimately it arrives back at the `materials_database` relation, and `Material` and `Type` become reinstated.

Prolog provides two facilities for controlling backtracking. They can be used to increase efficiency and to alter the meaning of a program. These facilities are:

- the *order* of Prolog code;
- the use of the *cut* operator.

Prolog tries out possible solutions to a problem in the order in which they are presented. Thus, in our example, Prolog always starts by assuming that the user has supplied a single materials specification. Only when it discovers that this is not the case does Prolog consider that the user may have submitted a list of several specifications. This is an appropriate ordering, as it is sensible to try the simplest solution first. In general, the ordering of code will affect the procedural meaning of a Prolog program (i.e., *how* the problem will be solved), but not its declarative meaning. However, as soon as the Prolog programmer starts to use the second facility, namely, the *cut* operator, the order of Prolog clauses can affect both the procedural and the declarative meaning of programs.

In order to prevent Prolog from carrying out unwanted backtracking, the *cut* symbol (!) can be used. Cuts can be inserted as though they were goals in their own

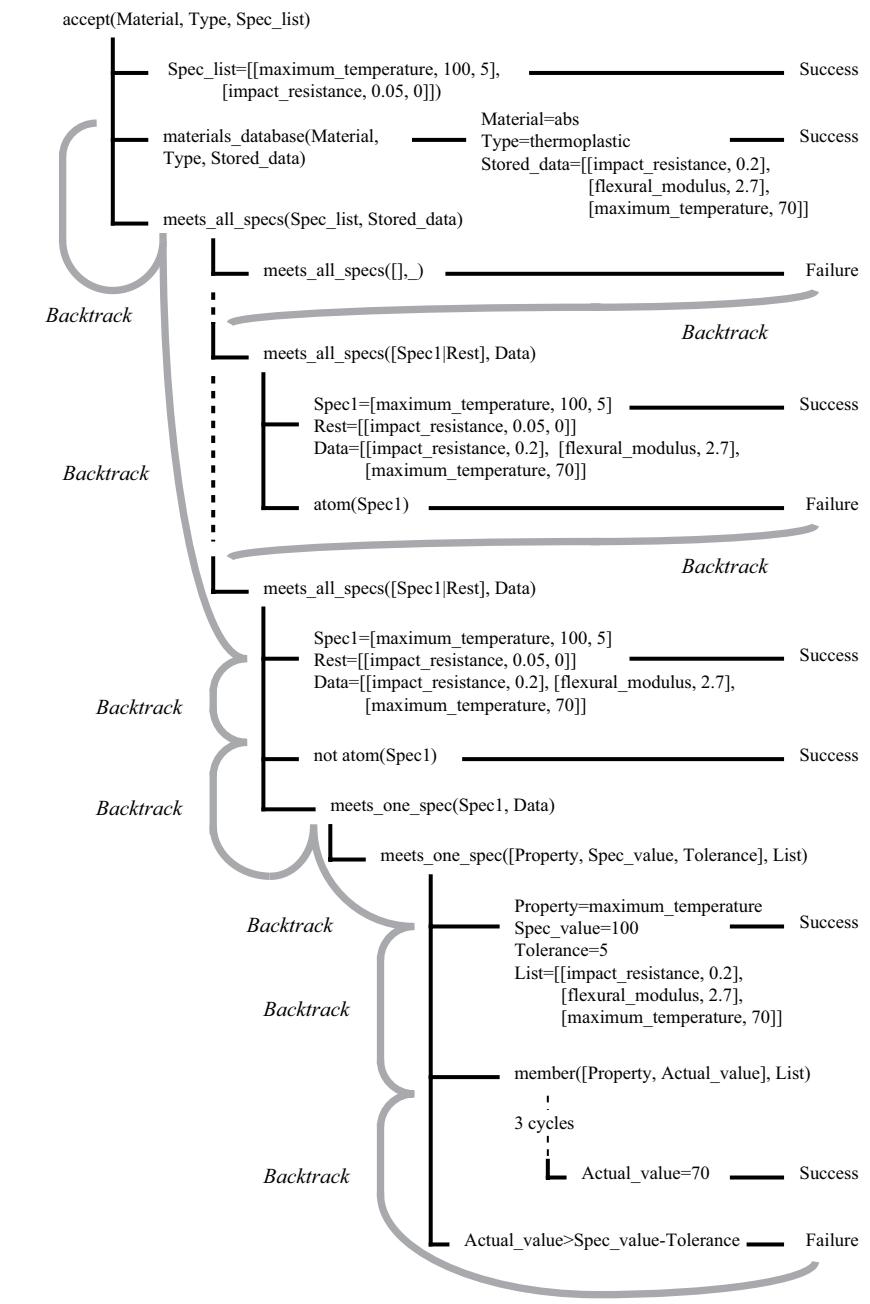


Figure 11.2 Backtracking in Prolog.

right. When Prolog comes across a cut, backtracking is prevented. Cuts can be used to make our example program more efficient by forcing Prolog immediately to try a new material once it has established whether or not a given material meets the specification. The revised program, with cuts included, is shown in Box 11.4 (the setting up of the `materials_database` relations is unchanged and has been omitted).

Although the discussion so far has regarded the cut as a means of improving efficiency by eliminating unwanted backtracking, cuts can also alter the declarative meaning of programs. This can be illustrated by referring once again to our materials selection program. The program contains three alternative means of achieving the `meets_all_specs` goal. The first deals with the case where the first argument is the empty list. The two others take identical arguments, and a distinction is made based upon whether the first element of the first argument (a list) is an atom. If the element is an atom, then the alternative case need not be considered, and this saving can be achieved using a cut (note that % indicates a comment):

```
meets_all_specs([Spec1|Rest],Data) :-
    atom(Spec1),!,% cut placed here
    meets_one_spec([Spec1|Rest],Data).

meets_all_specs([Spec1|Rest],Data) :-
    not atom(Spec1),% this test is now redundant
```

BOX 11.4 A PROLOG PROGRAM WITH CUTS

```
accept(Material,Type,Spec_list) :-
    materials_database(Material,Type,Stored_data),
    meets_all_specs(Spec_list,Stored_data).

meets_all_specs([],_):-!.% cut placed here

meets_all_specs([Spec1|Rest],Data) :-
    atom(Spec1),!,% cut placed here
    meets_one_spec([Spec1|Rest],Data).

meets_all_specs([Spec1|Rest],Data) :-
    meets_one_spec(Spec1,Data),
    meets_all_specs(Rest,Data).

meets_one_spec([Property, Spec_value, Tolerance], List) :-
    member([Property, Actual_value], List),!,% cut here
    Actual_value>Spec_value-Tolerance.

member(A, [A|L]).  
member(A, [_|L]):-member(A,L).
```

```
meets_one_spec(Spec1, Data),
meets_all_specs(Rest, Data).
```

Because of the positioning of the cut, if:

```
atom(Spec1)
```

is successful, then the alternative rule will not be considered. Therefore, the test:

```
not atom(Spec1)
```

is now redundant and can be removed. However, this test can only be removed provided that the cut is included in the previous rule. This example shows that a cut can be used to create rules of the form:

```
if <condition> then <conclusion1> else <conclusion2>.
```

The code in Box 11.4 includes two further cuts that prevent unnecessary backtracking.

While much of this discussion has concentrated on overcoming the inefficiencies that backtracking can introduce, it is important to remember that backtracking is essential for searching out a solution, and the elegance of Prolog in many applications lies in its ability to backtrack without the programmer needing to program this behavior explicitly.

11.5 Python

11.5.1 Background

Lisp was extensively used for list processing in the early years of AI, but Python has taken over as a general-purpose language that is well-suited to AI. Python has a Pascal-like syntax and can perform the tasks of other structured languages, as well as incorporating an extensive library of functions for manipulating symbols and lists. The language is very expressive, so that the meaning of Python code is generally quite clear and explicit. This transparency can be contrasted with Lisp, which is elegant in its compact representation, but can be difficult to follow.

Python also achieves a level of conciseness through its use of indenting to define the structure of the program, without relying on a closing symbol at the end of each block of code. To facilitate the correct formatting, Python is usually written using an interactive editor that assists with the indentation, the matching of brackets, etc.

Python has the flexibility to embrace all of the programming styles discussed so far. Although it is not a functional language like Lisp, it does nevertheless permit the functional style of programming. That is to say that Python functions can be defined, which always return a value, even if that value is `None`. Likewise, in practical terms,

Python is an object-oriented programming language. It doesn't strictly meet the requirements of object-orientation listed in Section 4.6.1, as encapsulation is not enforced. So, information in class members is visible unless steps are taken to hide it. In that respect, it is equivalent to C++ in providing access controls (Section 4.6.5). In Python, protected and private member classes are designated with names that begin with a single or double underscore, respectively. In practice, anything that can be done in C++ can be done in Python.

A great strength of Python is its huge user base who share libraries of code through open-source communities. This wealth of resources has been boosted further through software offered by large corporations.

11.5.2 A Worked Example

The syntax and semantics of Python can be illustrated by programming the same example that we used with Lisp and Prolog, that is, selecting from a database those polymers that meet a set of specifications, as defined in Box 11.1. As before, the first task will be to create the materials database, based on a list of polymer names with their corresponding types and a selection of physical properties. In Python, a list is a collection of data instances, of any type, separated by commas. The names of the polymers and their properties need to be treated as text strings, denoted by quotes. Without the quotes, the names would be interpreted as the names of variables.

Apart from these syntactical differences, the structure of lists in Python is similar to those in Lisp. So, we can start by creating a list called `materials_database`, just as we did in the Lisp example:

```
materials_database = [
    ['abs', 'thermoplastic',
     ['impact_resistance', 0.2],
     ['flexural_modulus', 2.7],
     ['maximum_temperature', 70]],
    ['polypropylene', 'thermoplastic',
     ['impact_resistance', 0.07],
     ['flexural_modulus', 1.5],
     ['maximum_temperature', 100]],
    ['polystyrene', 'thermoplastic',
     ['impact_resistance', 0.02],
     ['flexural_modulus', 3.0],
     ['maximum_temperature', 50]],
    ['polyurethane_foam', 'thermoset',
     ['impact_resistance', 1.06],
     ['flexural_modulus', 0.9],
     ['maximum_temperature', 80]],
    ['pvc', 'thermoplastic',
     ['impact_resistance', 1.06],
     ['flexural_modulus', 0.007],
     ['maximum_temperature', 50]],
```

```
[‘silicone’, ‘thermoset’,
 [‘impact_resistance’, 0.02],
 [‘flexural_modulus’, 3.5],
 [‘maximum_temperature’, 240]]]
```

As noted in Section 11.2.3, Python is an interpreted language, so the source code is interpreted and executed directly in the programming environment, without compilation into a separate file of machine-level instructions to be run separately. There is no need to declare the variables in advance. In our example, the variable `materials_database` is created automatically, and its data type becomes “list” as soon as we have allocated a list to it. When a variable is created outside of a function definition, it is assumed to be global, i.e., it is available anywhere in the program.

As with Prolog, the one assignment that we have coded so far is sufficient for us to ask Python some questions and in so doing to illustrate some of the features of the language. Lists in Python are indexed from a starting position of 0. The first element is accessed by the index 0, the second element by the index 1, etc. So, we can ask Python to print on the computer screen the value of the first item in `materials_database` as follows:

```
python> print(materials_database[0])
[‘abs’, ‘thermoplastic’, [‘impact_resistance’, 0.2],
 [‘flexural_modulus’, 2.7], [‘maximum_temperature’, 70]]
```

Here, the prompt in the computer console window is shown as `python>`, although a triple arrowhead (`>>>`) is more typical. The response from the interpreter is printed underneath. The list `materials_database` contains nested lists. In the `print` command, we have requested the first element in the list to be printed, which is itself a list. Any element of a list can be accessed simply by specifying its position in the list as an integer in square brackets after the list name. Accessing the first element of `spec_list` is equivalent to using `first` in Lisp.

Although Python is case-sensitive (`my_variable` is different from `My_variable`), it does not use case for any special purpose, unlike Prolog. Because Python can replicate the functionality of Lisp, we will build our Python program by defining functions that perform identical tasks to the Lisp functions described in Section 11.3.3. We will define a function called `accept`, which, like its Lisp equivalent, takes as its parameter a list (`spec_list`) containing one or more specifications, and returns a list of polymers that meet the specifications:

```
def accept(spec_list):
    shortlist = setup()
    if not(any(isinstance(i, list) for i in spec_list)):
        # no nested lists, so a single specification
        shortlist = meets_one(spec_list, shortlist)
    else:
        #nested lists, where each one is a specification
```

```

for each_spec in spec_list:
    if type(each_spec) == list:
        shortlist = meets_one(each_spec, shortlist)
    else: print ("Wrong specification")
return shortlist

```

The fact that `accept` returns a value is made explicit by the use of `return` in the final line of the definition to return the final value of `shortlist`. The variables `shortlist` and `each_spec` are introduced within the definition of the function, which means that they are local to it. Any values that are assigned to local variables within a function will be lost when its execution finishes.

The variable `shortlist` is initially assigned the value returned by the function `setup`, which has yet to be defined. The purpose of `setup` is to create an initial `shortlist` containing all the polymers and their type (thermoset or thermoplastic). The call to `setup` is made explicit by the empty brackets after the function name, indicating that it should be called with no arguments. In the absence of those empty brackets, the definition of the function called `setup` would merely be assigned to `shortlist`, but the function would not be made to run.

As in the Lisp program, we need to distinguish between the cases where `spec_list` is a single specification and when it is a list of several specifications each of which must be met. There are several ways that this distinction can be made. One approach is to check for the presence of embedded lists, corresponding to multiple specifications. This test is achieved with a “for” loop that uses the inbuilt function `isinstance` to test whether any element of `spec_list` is of the type `list`.

If `spec_list` contains only one specification, then a final list of polymers that meet the specification is returned by a single call of `meets_one`, which has yet to be defined. Otherwise `meets_one` must be called once for every specification in `spec_list`, and this aim is achieved by the use of another “for” loop:

```

for each_spec in spec_list:
    if type(each_spec) == list:
        shortlist = meets_one(each_spec, shortlist)

```

The code that constitutes the body of the loop is executed with `each_spec` set to each element of the list `spec_list` in turn. The final value of `shortlist`, after all polymers which fail to meet the specification have been removed, is the value returned by `accept`.

Consider the general case of a “for” loop of the type `for myelement in mylist`. If some of the elements of `mylist` are themselves lists, then these embedded lists will, in their turn, be assigned to `myelement`. The elements of the embedded lists will not be extracted by the “for” loop. This point is illustrated by loading the following procedure:

```

def example():
    mylist = ['a', ['embedded', 'list'], 'b', 'c']

```

```
for myelement in mylist:
    print(myelement)
```

If this procedure is called, the four elements of `mylist` will be printed, as shown:

```
python> example();
a
['embedded', 'list']
b
c
```

We will now define the procedure `setup`, which creates a list of all polymers in the materials database, to serve as the initial shortlist:

```
def setup():
    shortlist = []
    for material in materials_database:
        # put all materials on the initial shortlist
        shortlist.append(material[0:2])
    return shortlist
```

We start off by assigning the empty list to the local variable `shortlist`. We then step through the elements of `materials_database` using a “for” loop and add the polymer names and types to `shortlist`. There are a number of alternative ways of adding the polymer names and types to `shortlist`. The method shown here exploits the inbuilt `append` function. In our example, a two-element list containing a polymer name and its type is added to `shortlist` by using the index range `0:2`, which includes the first element (index 0) and the second element (index 1), but not the third (index 2).

An alternative means of achieving the same result would be through the use of the `+` symbol, which can be used to join, or concatenate, two lists:

```
shortlist = shortlist + [material[0:2]]
```

With the definition of `setup` complete, the next task is to define the procedure `meets_one`, which takes as its parameters a single property specification and the current shortlist, and returns a new shortlist of all polymers that meet the property specification:

```
def meets_one(spec, shortlist):
    # check specification is in the right format
    if (len(spec) == 3 and type(spec[1]) in (int, float)
        and type(spec[2]) in (int, float)):
        for polymer in materials_database:
            if polymer[0:2] in shortlist:
                # look up value of materials property for polymer
                # in shortlist
```

```

actual = get_database_value(spec[0], polymer[0])
if (type(actual) in (int, float)
    and actual < spec[1] - spec[2]):
    # outside specification so remove polymer
    # from shortlist
    shortlist.remove(polymer[0:2])
return shortlist
else:
    print("Wrong specification")

```

Two local variables are defined: `material` and `actual`. A “for” loop is used again, this time to extract each polymer in turn from the current shortlist, and to assign it to the local variable `material`. The value assigned to `material` is in fact a list, the first element of which is the polymer name. This polymer name is passed together with the property name to a function called `get_database_value`, which will have to be defined so that it returns the property value for the polymer.

The database value returned is then compared with the specified value less the tolerance. If the database value is found to be the smaller of the two, then the polymer in question is removed from the local copy of the shortlist by means of the inbuilt `remove` function. The fact that it is only a local copy of `shortlist` that changes is an important point. The `shortlist` referred to in `accept`, and the `shortlist` referred to in `meets_one` are distinct, and local to the respective functions in which they appear. Thus, the `shortlist` in the function `accept` only becomes updated when `meets_one` has finished its task, and the value returned is assigned to the copy in `accept`.

The only function now remaining to be defined is `get_database_value`. As the name implies, it looks up the value of a given property for a given polymer. This job is quite straightforward using Python’s list indexing. As defined, the return value of the function requires a match to the polymer name and property name and, for that reason, is embedded in two “if” statements. In the event of no match, the return statement will never be reached, and Python will return the special value `None`. The definition of `get_database_value` is as follows:

```

def get_database_value(prop_name, material):
    for polymer in materials_database:
        if polymer[0] == material:
            # found the target polymer in the materials database
            for property in polymer:
                if property[0] == prop_name:
                    # found the target material property in the database
                    return property[1]

```

The complete Python program is shown in Box 11.5, and it can be tested with the same examples that we set for our Lisp and Prolog programs. First, we can ask for

BOX 11.5 A WORKED EXAMPLE IN PYTHON

```

materials_database = [
    ['abs', 'thermoplastic',
     ['impact_resistance', 0.2],
     ['flexural_modulus', 2.7],
     ['maximum_temperature', 70]],
    ['polypropylene', 'thermoplastic',
     ['impact_resistance', 0.07],
     ['flexural_modulus', 1.5],
     ['maximum_temperature', 100]],
    ['polystyrene', 'thermoplastic',
     ['impact_resistance', 0.02],
     ['flexural_modulus', 3.0],
     ['maximum_temperature', 50]],
    ['polyurethane_foam', 'thermoset',
     ['impact_resistance', 1.06],
     ['flexural_modulus', 0.9],
     ['maximum_temperature', 80]],
    ['pvc', 'thermoplastic',
     ['impact_resistance', 1.06],
     ['flexural_modulus', 0.007],
     ['maximum_temperature', 50]],
    ['silicone', 'thermoset',
     ['impact_resistance', 0.02],
     ['flexural_modulus', 3.5],
     ['maximum_temperature', 240]]]

def accept(spec_list):
    shortlist = setup()
    if not(any(isinstance(i, list) for i in spec_list)):
        # no nested lists, so a single specification
        shortlist = meets_one(spec_list, shortlist)
    else:
        # nested lists, where each one is a specification
        for each_spec in spec_list:
            if type(each_spec) == list:
                shortlist = meets_one(each_spec, shortlist)
            else: print ("Wrong specification")
    return shortlist

def setup():
    shortlist = []
    for material in materials_database:
        # put all materials on the initial shortlist
        shortlist.append(material[0:2])
    return shortlist

```

(Continued)

BOX 11.5 (Continued)

```

def meets_one(spec, shortlist):
    # check specification is in the right format
    if (len(spec) == 3 and type(spec[1]) in (int, float)
        and type(spec[2]) in (int, float)):
        for polymer in materials_database:
            if polymer[0:2] in shortlist:
                # look up value of materials property for polymer
                # in shortlist
                actual = get_database_value(spec[0], polymer[0])
                if (type(actual) in (int, float)
                    and actual < spec[1] - spec[2]):
                    # outside specification so remove polymer
                    # from shortlist
                    shortlist.remove(polymer[0:2])
    return shortlist
else:
    print("Wrong specification")

def get_database_value(prop_name, material):
    for polymer in materials_database:
        if polymer[0] == material:
            # found the target polymer in the materials database
            for property in polymer:
                if property[0] == prop_name:
                    # found the target material property in the
                    # database
                    return property[1]

```

all polymers that have a maximum operating temperature of 100°C, with a 5°C tolerance:

```
python> print(accept(['maximum_temperature', 100, 5]))
[['polypropylene', 'thermoplastic'], ['silicone',
'thermoset']]
```

Second, we can add the requirement for an impact resistance of 0.05 kJ/m:

```
python> print(accept([[['maximum_temperature', 100,
5], ['impact_resistance', 0.05, 0]]])
[['polypropylene', 'thermoplastic']])
```

11.6 Comparison of AI Languages

For each AI language, the worked example gives some feel for the language structure and syntax. However, it does not form the basis for a fair comparison of their merit. The Prolog code is the most compact and elegant solution to the problem of choosing materials that meet a specification, because Prolog is particularly good at tasks that involve pattern matching and retrieval of data. However, the language places a number of constraints on the programmer, particularly in committing him or her to one particular search strategy. As we have seen, the programmer can control this strategy to some extent by judicious ordering of clauses and use of the cut mechanism. Prolog does not need a structure for iteration, for example, `for x from 1 to 10`, as recursion can be used to achieve the same effect.

Our Lisp and Python programs have a completely different structure from the Prolog example, as they have been programmed procedurally (or, more precisely, functionally). These languages include excellent facilities for manipulating symbols and lists of symbols. They provide the power to implement practically any reasoning strategy. Lisp is so flexible that it can be reconfigured by the programmer. Both Python and Lisp can be extended by defining or linking functions, while Python is also extensible by allowing code in other languages like C++ to be embedded within it. Conversely, Python code can be embedded in programs written in C++ and other languages. The worked example does not do justice to this flexibility. In particular, the materials database took the form of a long, flat list, whereas there are more structured ways of representing the data. As we have seen in Chapter 4, and we will see again in Chapter 13, object-oriented and frame-based programming allow a hierarchical representation of materials properties.

11.7 Summary

The three AI languages introduced here, Lisp, Prolog, and Python, are all well-suited to problems involving the manipulation of symbols. They can, therefore, form a good basis for the development of KBSs, while AI toolkits and extensions to conventional programming languages are a practical alternative. Python also excels at numerical problems and is often used for computational intelligence implementations, including neural networks and optimization algorithms.

The AI languages are flexible, elegant, and concise. In contrast with traditional programming languages, the same variable in an AI language can be used to hold a variety of data types. All three AI languages allow various types of data to be combined into lists, and they provide facilities for list manipulation.

Like most programming languages, Lisp and Python are procedural. Nevertheless, they can be used to support declarative programming. In contrast, the Prolog language is declarative, but can also be used procedurally. To enable declarative programming, Prolog includes a backtracking mechanism that allows it to explore all

possible ways in which a goal might be achieved. The programmer can exercise control over Prolog's backtracking by careful ordering of clauses and through the use of cuts.

In Lisp, lists are not only used for data, but also constitute the programs themselves. Lisp functions are represented as lists containing a function name and its arguments. As every Lisp function returns a value, the arguments themselves can be functions. Python is also a functional language, but it is not built around lists. Its more conventional structure provides transparency and flexibility.

Further Reading

Lisp

- Barski, C. 2010. *Land of Lisp*. No Starch Press, San Francisco, CA.
 Kreiker, P. 2000. *Visual Lisp: Guide to Artful Programming*. Delmar, Albany, NY.
 Seibel, P. 2011. *Practical Common Lisp*. Apress, New York.
 Touretzky, D. S. 2013. *Common Lisp: A Gentle Introduction to Symbolic Computation*. Dover Publications, New York.
 Watson, M. 2020. *Loving Common Lisp, or the Savvy Programmer's Secret Weapon*. 6th ed. Leanpub, Victoria, Canada.

Prolog

- Blackburn, P., Bos, J., and Striegnitz, K. 2006. *Learn Prolog Now!* College Publications, London.
 Bramer, M. 2013. *Logic Programming with Prolog*. 2nd ed. Springer, New York.
 Bratko, I. 2011. *Prolog Programming for Artificial Intelligence*. 4th ed. Addison-Wesley, Reading, MA.
 Clocksin, W. F. and Mellish, C. S. 2013. *Programming in Prolog: Using the ISO Standard*. 5th ed. Springer, New York.
 Scott, R. 2010. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, Denver, CO.

Python

- Barry, P. 2016. *Head First Python*, 2nd ed. O'Reilly, Sebastopol, CA.
 Cannon, J. 2014. *Python Programming for Beginners*. CreateSpace, Scotts Valley, CA.
 Downey, A. B. 2015. *Think Python: How to Think Like a Computer Scientist*, O'Reilly, Sebastopol, CA.
 Matthes, E. 2019. *Python Crash Course*, 2nd ed. No Starch Press, San Francisco, CA.
 Shaw, Z. A. 2017. *Learn Python 3 the Hard Way*. Addison-Wesley, Boston, MA.

Chapter 12

Systems for Interpretation and Diagnosis

12.1 Introduction

Diagnosis is the process of determining the nature of a fault or malfunction, based on a set of symptoms. So, the symptoms form the input data. They are interpreted and the underlying cause of these symptoms is the output. Diagnosis is, therefore, a special case of the more general problem of interpretation. There are many circumstances in which we may wish to interpret data, other than diagnosing problems. Examples include the interpretation of images (e.g., optical, x-ray, ultrasonic, electron microscopic), meters, gauges, and statistical data (e.g., from surveys of people or from a radiation counter). This chapter will examine some of the intelligent systems techniques that are used for diagnosis and for more general interpretation problems. The diagnosis of faults in a refrigerator will be used as an illustrative example, and the interpretation of ultrasonic images from welds in steel plates will be used as a detailed case study.

Since the inception of expert systems in the late 1960s and early 1970s, diagnosis and interpretation have been favorite application areas. Some of those early expert systems were remarkably successful and became regarded as “classics.” Three examples of these early successes are outlined as follows.

1. *PROSPECTOR*

PROSPECTOR (Duda et al. 1979; Hart et al. 1978) interpreted geological data and made recommendations of suitable sites for mineral prospecting. The

system made use of Bayesian updating as a means of handling uncertainty. (See Section 3.2.)

2. MYCIN

MYCIN (Shortliffe 1976) was a medical system for diagnosing infectious diseases and for selecting an antibiotic drug treatment. It is frequently referenced because of its novel (at the time) use of certainty theory. (See Section 3.3.)

3. DENDRAL

DENDRAL (Buchanan et al. 1969) interpreted mass-spectrometry data, and was notable for its use of a three-phase approach to the problem:

- Phase 1: *Plan*

Suggest molecular substructures that may be present to guide Phase 2.

- Phase 2: *Generate hypotheses*

Generate all plausible molecular structures.

- Phase 3: *Test*

For each generated structure, compare the predicted and actual data. Reject poorly matching structures and place the remainder in rank order.

A large number of intelligent systems have been produced subsequently, using many different techniques, to tackle a range of diagnosis and interpretation problems in science, technology, and engineering. Rule-based diagnostic systems have been applied to power plants (Arroyo-Figueroa et al. 2000), electronic circuits (Dague et al. 1991), furnaces (Dash 1990), an oxygen generator for use on Mars (Huang et al. 1992), and batteries in the Hubble space telescope (Bykat 1990). Bayesian updating has been used in nuclear power generation (Kang and Golay 1999), while fuzzy logic has been used in automobile assembly (Lu et al. 1998) and medical data interpretation (Kumar et al. 2007). Neural networks have been used for pump diagnosis (Yamashita et al. 2000) and magnetotelluric data for geology (Manoj and Nagarajan 2003), while a neural network–rule-based system hybrid has been applied to the diagnosis of electrical discharge machines (Huang and Liao 2000).

One of the most important techniques for diagnosis is case-based reasoning (CBR), described in Section 5.3. CBR has been used to diagnose cardiological diseases (Chakravarthy 2007), as well as faults in electronic circuits (Balakrishnan and Semmelbauer 1999), emergency battery backup systems (Netten 1998), and software (Hunt 1997). We will also see in this chapter the importance of models of physical systems such as power generation (Prang et al. 1996) and power distribution (Davidson et al. 2003). Applications of intelligent systems for the more general problem of interpretation include the interpretation of drawings (Maderlechner et al. 1987), seismic data (Zhang and Simaan 1987), optical spectra (Ampratwum et al. 1997), power-system alarms (Hossack et al. 2002), and nondestructive tests (Yella et al. 2006) including ultrasonic images (Hopgood et al. 1993). The last is a hybrid system, described in a detailed case study in Section 12.5.

12.2 Deduction and Abduction for Diagnosis

Given some information about the state of the world, we can often infer additional information. If this inference is logically correct (i.e., guaranteed to be true given that the starting information is true), then this process is termed *deduction*. Deduction is used to predict an effect from a given cause (see Chapter 1). Consider, for instance, a domestic refrigerator (Figure 12.1). If a refrigerator is unplugged from the electricity supply, we can confidently predict that, after a few hours, the ice in the freezer will melt and the food in the main compartment will no longer be chilled. The new assertion (that the ice melts) follows logically from the given facts and assertions (that the power is disconnected).

Imagine that we have a frame-based representation of a refrigerator (see Section 4.8), with a slot `state` that contains facets describing the current state of the refrigerator. If my refrigerator is represented by the frame instance `my_fridge`, then a simple deductive rule might be as follows, where capitalization denotes a local variable, as in Chapter 2:

```
rule r12_1
  if state of my_fridge is unplugged
  and transition_time of state of my_fridge is T
  and (time_now - T) > 5 /* hours assumed */
  then ice of my_fridge becomes melted
  and food_temperature of my_fridge becomes ambient.
```

While deductive rules have an important role to play, they are inadequate on their own for problems of diagnosis and interpretation. Instead of determining an effect given a cause, diagnosis and interpretation involve finding a cause given an effect.

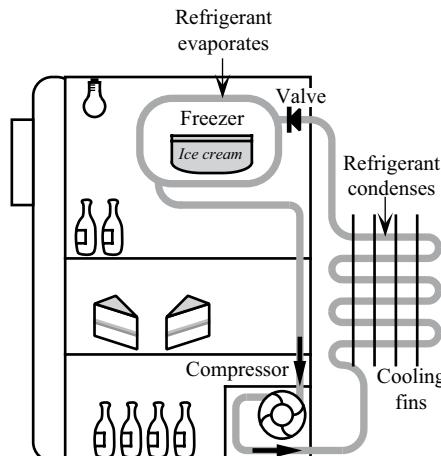


Figure 12.1 A domestic refrigerator.

This is termed *abduction* (see also Chapters 1 and 3) and it normally involves drawing a plausible inference rather than a certain one. Thus, if we observe that the ice in our freezer has melted and the food has warmed to room temperature, we could infer from rule r12_1 that our refrigerator is unplugged. However, this is clearly an unsound inference as there may be several other reasons for the observed effects. For instance, by reference to rules r12_2 and r12_3 in the following text, we might, respectively, infer that the fuse has blown or that there is a power blackout:

```
rule r12_2
  if fuse of my_fridge is blown
  and transition_time of fuse of my_fridge is T
  and (time_now - T) > 5 /* hours assumed */
  then ice of my_fridge becomes melted
  and food_temperature of my_fridge becomes ambient.

rule r12_3
  if power of my_fridge is blackout
  and transition_time of power of my_fridge is T
  and (time_now - T) > 5 /* hours assumed */
  then ice of my_fridge becomes melted
  and food_temperature of my_fridge becomes ambient.
```

So, given the observed symptoms about the temperature within the refrigerator, the cause might be that the refrigerator is unplugged, or it might be that the fuse has blown, or there might be a power blackout. Three different approaches to tackling the uncertainty of abduction are outlined in the following subsections: exhaustive testing, explicit modeling of uncertainty, and hypothesize-and-test.

12.2.1 Exhaustive Testing

We could use rules where the condition parts exhaustively test for every eventuality. Rules are, therefore, required of the form:

```
rule exhaustive
  if state of my_fridge is unplugged
  and transition_time of state of my_fridge is T
  and (time_now - T) > 5 /* hours assumed */
  and fuse of my_fridge is not blown
  and power of my_fridge is not blackout
  and ...
  and ...
  then ice of my_fridge becomes melted
  and food_temperature of my_fridge becomes ambient.
```

This is not a practical solution, except for trivially simple domains, and the resulting rule base would be difficult to modify or maintain. In the RESCU system (Leitch et al. 1991), those rules that can be used with confidence for

both deduction and abduction are labeled as *reversible*. Such rules describe a one-to-one mapping between cause and effect, such that no other causes can lead to the same effect.

12.2.2 Explicit Modeling of Uncertainty

The uncertainty can be explicitly represented using techniques such as those described in Chapter 3. This approach was adopted in MYCIN and PROSPECTOR. As noted in Chapter 3, many of the techniques for representing uncertainty are founded on assumptions that are not necessarily valid.

12.2.3 Hypothesize-and-Test

A tentative hypothesis can be put forward for further investigation. The hypothesis may be subsequently confirmed or abandoned. This hypothesize-and-test approach, which was used in DENDRAL, avoids the pitfalls of finding a valid means of representing and propagating uncertainty.

There may be additional sources of uncertainty, as well as the intrinsic uncertainty associated with abduction. For example, the evidence itself may be uncertain (e.g., we may not be sure that the food isn't cool) or vague (e.g., just what does "cool" or "chilled" mean precisely?).

The production of a hypothesis that may be subsequently accepted, refined, or rejected is similar, but not identical, to *nonmonotonic logic*. Under nonmonotonic logic, an earlier conclusion may be withdrawn in the light of new evidence. Conclusions that can be withdrawn in this way are termed *defeasible*, and a defeasible conclusion is assumed to be valid until such time as it is withdrawn.

In contrast, the hypothesize-and-test approach involves an active search for supporting evidence once a hypothesis has been drawn. If sufficient evidence is found, the hypothesis becomes held as a conclusion. If contrary evidence is found, then the hypothesis is rejected. If insufficient evidence is found, the hypothesis remains unconfirmed.

This distinction between nonmonotonic logic and the hypothesize-and-test approach can be illustrated by considering the case of our nonworking refrigerator. Suppose that the refrigerator is plugged in, the compressor is silent, and the light does not come on when the door is opened. Using the hypothesize-and-test approach, we might produce the hypothesis that there is a power blackout. We would then look for supporting evidence by testing whether another appliance is also inoperable. If this supporting evidence were found, the hypothesis would be confirmed. If other appliances were found to be working, the hypothesis would be withdrawn. Under nonmonotonic logic, reasoning would continue based on the assumption of a power blackout until the conclusion is defeated, if it is defeated at

all. Confirmation is neither required nor sought in the case of nonmonotonic logic. Implicitly, the following default assumption is made:

```
rule power_assumption
  if state of appliance is dead
  and power is not on
  /* power could be off or unknown */
then power becomes off.
/* assume for now that there is a power failure */
```

The term *default reasoning* describes this kind of implicit rule in nonmonotonic logic.

12.3 Depth of Knowledge

12.3.1 Shallow Knowledge

The early successes of diagnostic expert systems are largely attributable to the use of shallow knowledge, expressed as rules. This knowledge is the sort that a human expert might acquire by experience, without regard to the underlying reasons. For instance, a mechanic looking at a broken refrigerator might hypothesize that, if the refrigerator makes a humming noise but does not get cold, then it has lost coolant. While he or she may also know the detailed workings of a refrigerator, this detailed knowledge need not be used. Shallow knowledge can be easily represented:

```
rule r12_4
  if sound of my_fridge is hum
  and temperature of my_fridge is ambient
then hypothesis becomes loss_of_coolant.
```

Note that we are using the hypothesize-and-test approach to deal with uncertainty in this example. With the coolant as its focus of attention, an expert system may then progress by seeking further evidence in support of its hypothesis, for example, the presence of a leak in the pipes. Shallow knowledge is given a variety of names in the literature, including *heuristic*, *experiential*, *empirical*, *compiled*, *surface*, and *low-road*.

Expert systems built upon shallow knowledge may look impressive since they can move rapidly from a set of input data (the symptoms) to some plausible conclusions with a minimal number of intermediate steps, just as a human expert might. However, the limitations of such an expert system are easily exposed by presenting it with a situation that is outside its narrow area of expertise. When it is confronted

with a set of data about which it has no explicit rules, the system cannot respond or, worse still, may give wrong answers.

There is a second important deficiency of shallow-reasoning expert systems. Because the knowledge bypasses the causal links between an observation and a deduction, the system has no understanding of its knowledge. Therefore, it is unable to give helpful explanations of its reasoning. The best it can do is to regurgitate the chain of heuristic rules that led from the observations to the conclusions.

12.3.2 Deep Knowledge

Deep knowledge is the fundamental building block of understanding. A number of deep rules might make up the causal links underlying a shallow rule. For instance, the effect of the previous shallow rule r12_4 may be achieved by the following informally stated deep rules:

```
rule r12_5 /* not flex format */
  if current flows in the windings of the compressor
  then there is an induced rotational force on the windings.

rule r12_6 /* not flex format */
  if motor windings are rotating
  and axle is attached to windings and compressor vanes
  then compressor axle and vanes are rotating.

rule r12_7 /* not flex format */
  if a part is moving
  then it may vibrate or rub against its mounting.

rule r12_8 /* not flex format */
  if two surfaces are vibrating or rubbing against each other
  then mechanical energy is converted to heat and sound.

rule r12_9 /* not flex format */
  if compressor vanes rotate
  and coolant is present as a gas at the compressor inlet
  then coolant is drawn through the compressor and
  pressurized.

rule r12_10 /* not flex format */
  if pressure on a gas exceeds its vapor pressure
  then the gas will condense to form a liquid.

rule r12_11 /* not flex format */
  if a gas is condensing
  then it will release its latent heat of vaporization.
```

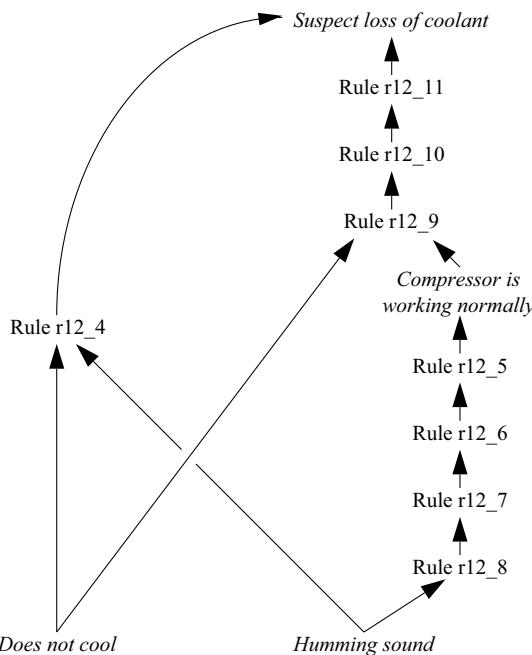


Figure 12.2 Comparing deep and shallow knowledge about a refrigerator.

Figure 12.2 shows how these deep rules might be used to draw the same hypothesis as the shallow rule r12_4.

There is no clear distinction between deep and shallow knowledge, but some rules are deeper than others. Thus, while rule r12_5 is deep in relation to the shallow rule r12_4, it could be considered shallow compared with knowledge of the flow of electrons in a magnetic field and the origins of the electromotive force that gives rise to the movement of the windings. In recognition of this limitation, Fink and Lusth (1987) distinguish *fundamental knowledge*, which is the deepest knowledge that has relevance to the domain. Thus, “unsupported items fall to the ground” may be considered a fundamental rule in a domain where details of Newton’s laws of gravitation are unnecessary. Similarly, Kirchoff’s first law, which states that the sum of the input current is equal to the sum of the output current at any point in a circuit, would be considered a deep rule for most electrical or electronic diagnosis problems. Nevertheless, it is rather shallow compared with detailed knowledge of the behavior of electrons under the influence of an electric field. Thus, Kirchoff’s first law is not fundamental in the broad domain of physics, since it can be derived from deeper knowledge. However, it may be treated as fundamental for most practical problems.

12.3.3 Combining Shallow and Deep Knowledge

There are merits and disadvantages to both deep and shallow knowledge. A system based on shallow knowledge can be very efficient, but it possesses no understanding and, therefore, has no ability to explain its reasoning. It will also fail dismally when confronted with a problem that lies beyond its expertise. The use of deep knowledge can alleviate these limitations, but the knowledge base will be much larger and less efficient. There is a trade-off between the two approaches.

The Integrated Diagnostic Model (IDM) (Fink and Lusth 1987) is a system that attempts to integrate deep and shallow knowledge. Knowledge is partitioned into deep and shallow knowledge bases, either one of which is in control at any one time. The controlling knowledge base decides which data to gather, either directly or through dialogue with the user. However, the other knowledge base remains active and is still able to make deductions. For instance, if the shallow knowledge base is in control and has determined that the light comes on when the refrigerator door is opened (as it should), the deep knowledge base can be used to deduce that the power supply is working, that the fuses are OK, that the wires from the power supply to the bulb are OK, and that the bulb is OK. All of this knowledge then becomes available to both knowledge bases.

In IDM, the shallow knowledge base is given control first, and it passes control to the deep knowledge base if it fails to find a solution. The rationale for this approach is that a quick solution is worth trying first. The shallow knowledge base can be much more efficient at obtaining a solution, but it is more limited in the range of problems that it can handle. If it fails, the information that has been gathered is still of use to the deep knowledge base.

A shallow knowledge base in IDM can be expanded through experience in two ways:

1. Each solved case can be stored and used to assist in solving similar cases. This is the basis of case-based reasoning (see Chapter 5).
2. If a common pattern between symptom and conclusions has emerged over a large number of cases, then a new shallow rule can be created. This is an example of rule induction (see Chapter 5).

12.4 Model-Based Reasoning

12.4.1 The Limitations of Rules

The amount of knowledge about a device that can be represented in rules alone is somewhat limited. A deep understanding of how a device works, and what can go wrong with it, is facilitated by a physical model of the device being examined.

Fulton and Pepe (1990) have highlighted three major inadequacies of a purely rule-based diagnostic system:

1. Building a complete rule set is a massive task. For every possible failure, the rule-writer must predict a complete set of symptoms. In many cases, this information may not even be available because the failure may never have happened before. It may be possible to overcome the latter hurdle by deliberately causing a fault and observing the sensors. However, this action would be inappropriate in some circumstances, such as an overheated core in a nuclear power station.
2. Symptoms are often in the form of sensor readings, and a large number of rules are needed solely to verify the sensor data (Scarl et al. 1987). Before an alarm signal can be believed at face value, related sensors must be checked to see whether they are consistent with the alarm. Without this measure, there is no reason to assume that a physical failure has occurred rather than a sensor failure.
3. Even supposing that a complete rule set could be built, it would rapidly become obsolete. As there is frequently an interdependence between rules, updating the rule base may require more thought than simply adding new rules.

These difficulties can be circumvented by means of a model of the physical system. Rather than storing a huge collection of rules where each rule represents the pairing of a symptom and its cause, these pairs can be *generated* by applying physical principles to the model. The model, which is often frame-based, may describe any kind of system, including physical (Fenton et al. 2001; Wotawa 2000), software (Mateis et al. 2000), medical (Montani et al. 2003), legal (Bruninghaus and Ashley 2003), and behavioral (de Koning et al. 2000).

12.4.2 Modeling Function, Structure, and State

Practically all physical devices are made up of fundamental components such as tubes, wires, batteries, and valves. As each of these performs a fairly simple role, it also has a simple failure mode. For example, a wire may break and fail to conduct electricity, a tube can spring a leak, a battery can lose its charge, and a valve may be blocked. Given a model of how these components operate and interact to form a device, faults can be diagnosed by determining the effects of local malfunctions on the global view, that is, on the overall device. Reasoning through consideration of the behavior of the components is sometimes termed reasoning from *second principles*. *First principles* are the basic laws of physics that determine component behavior.

Numerous different techniques and representations have been devised for modeling physical devices. Examples include IDM (Fink and Lusth 1987), Knowledge Engineer's Assistant (KEATS; Motta et al. 1988), DEDALE (Dague et al. 1987),

FLAME (Price 1998), NODAL (Cunningham 1998), GDE (Dekleer and Williams 1987), and ADAPtER (Portinale et al. 2004). These representations are generally based on agents, objects, or frames (see Chapter 4). ADAPtER combines elements of model-based and case-based reasoning.

The device is made up of a number of components, each of which may be represented as an instance of a class of components. The *function* of each component is defined within its class definition. The *structure* of a device is defined by links between the instances of components that make up the device. The device may be in one of several *states*, for example, a refrigerator door may be open or closed, and the thermostat may have switched the compressor on or off. These states are defined by setting the values of instance variables.

Object-oriented programming is particularly suited to device modeling because of the clear separation of function, structure, and state. These three aspects of a device are fundamental to understanding its operation and possible malfunctions. A contrast can be drawn with mathematical modeling, where a device can often be modeled more easily by considering its overall activity than by analyzing the component processes.

Model-based reasoning can even be used to diagnose faults in computer models of physical systems, so that the diagnostic model is two layers of abstraction away from the physical world. For example, model-based reasoning has been used to diagnose faults in electronic system designs represented in VHDL (VHSIC hardware description language, where VHSIC stands for “very-high-speed integrated circuit”; Wotawa 2000). It can be regarded as a debugger for the *documentation* of a complex electronic system rather than a debugger for the system itself.

12.4.2.1 Function

The function of a component is defined by the methods and attributes of its class. Fink and Lusth (1987) define four *functional primitives* that are classes of components. All components are considered to be specializations of one of these four classes, although Fink and Lusth hint at the possible need to add further functional primitives in some applications. The four functional primitives are as follows:

- *Transformer*—transforms one substance into another.
- *Regulator*—alters the output of substance *B*, based upon changes in the input of substance *A*.
- *Reservoir*—stores a substance for later output.
- *Conduit*—transports substances between other functional primitives.

A fifth class, *sensor*, may be added to this list. A sensor object simply displays the value of its input.

The word “substance” is intended to be interpreted loosely. Thus, water and electricity would both be treated as substances. The scheme was not intended to be completely general purpose, and Fink and Lusth acknowledge that it would need

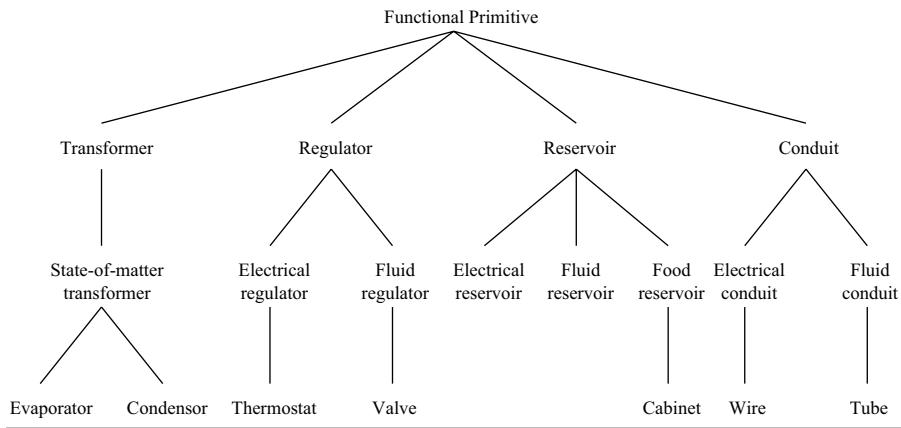


Figure 12.3 Functional hierarchy for some components of a refrigerator.

modifications in different domains. However, such modifications may be impractical in many domains, where specialized modeling may be more appropriate. As an illustration of the kind of modification required, Fink and Lusth point out that the behavior of an electrical conduit (a wire) is very different from a water conduit (a pipe), since a break in a wire will stop the flow of electricity, while a broken pipe will cause an out-gush of water.

These differences can be recognized by making pipes and wires specializations of the class *conduit* in an object-oriented representation. The *pipe* class requires that a substance be pushed through the conduit, whereas the *wire* class requires that a “substance” (i.e., electricity) be both pushed and pulled through the conduit. Gas pipes and water pipes would then be two of many possible specializations of the class *pipe*. Figure 12.3 shows a functional hierarchy of classes for some of the components of a refrigerator.

12.4.2.2 Structure

Links between component instances can be used to represent their physical associations, thereby defining the structure of a device. For example, two resistors connected in series might be represented as two instances of the class *resistor* and one instance of the class *wire*. Each instance of *resistor* would have a link to the instance of *wire*, to represent the electrical contact between them.

Figure 12.4 shows the instances and links that define some of the structure of a refrigerator. This figure illustrates that a compressor can be regarded either as a device made up from several components or as a component of a refrigerator. It is, therefore, an example of a *functional group*. The compressor forms a distinct module in the physical system. However, functional groups in general need not be modular in the physical sense. Thus, the evaporation section of the refrigerator may be

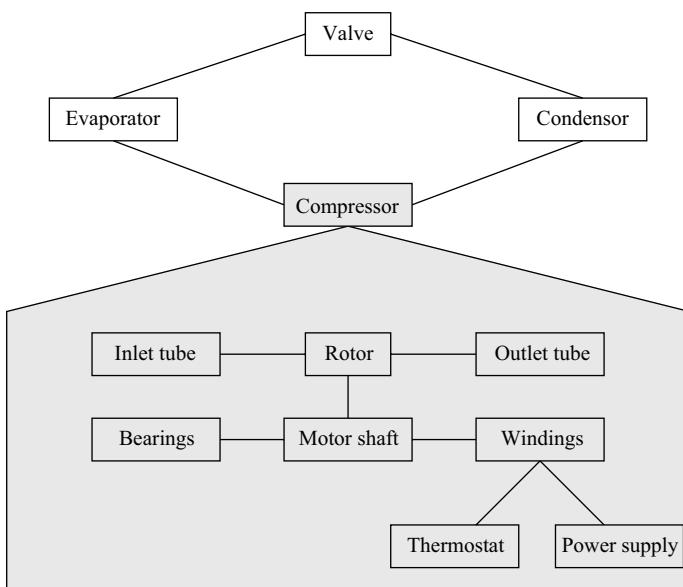
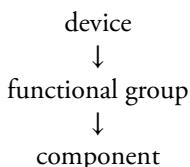


Figure 12.4 Visual display of the instances and links that define the structure of a refrigerator.

regarded as a functional group even though it is not physically separate from the condensation section, and the light system forms a functional group even though the switch is physically removed from the bulb.

Many device-modeling systems can produce a graphical display of the structural layout of a device, similar to Figure 12.4, given a definition of the instances and the links between them. Some systems, such as KEATS and IDM, allow the reverse process as well. With these systems, the user draws on the computer screen a structural diagram like the one in Figure 12.4, and the instances and links are generated automatically.

In devices where functional groups exist, the device structure is hierarchical. The hierarchical relationship can be represented by means of the composition relationship between objects (see Chapter 4). It is often adequate to consider just three levels of the structural hierarchy:



The application of a three-level hierarchy to the structure of a refrigerator is shown in Figure 12.5.

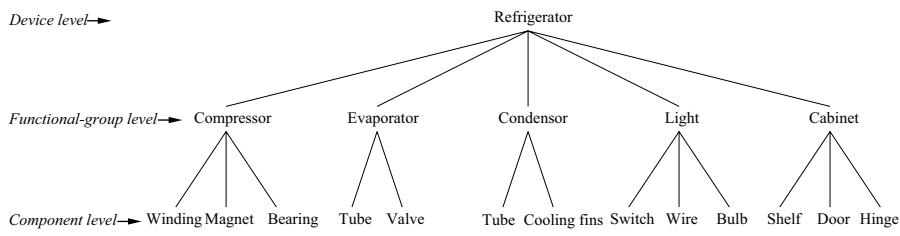


Figure 12.5 Structural hierarchy for some components of a refrigerator.

12.4.2.3 State

As already noted, a device may be in one of many alternative states. For example, a refrigerator door may be open or closed, and the compressor may be running or stopped. A state can be represented by setting appropriate instance variables on the components or functional groups, and transitions between states can be represented on a state map (Harel 1988) as shown in Figures 12.6 and 12.7. This representation is an example of a *finite-state machine* (FSM), which is a general term for a system model based on possible states and the transitions between them. FSMs have a wide variety of applications including the design of computer games (Wagner et al. 2006).

The state of some components and functional groups will be dependent on other functional groups or on external factors. Let us consider a refrigerator that is working correctly. The compressor will be in the state *running* only if the thermostat is in the state *closed circuit*. The state of the thermostat will alter according to the cabinet temperature. The cabinet temperature is partially dependent on an external factor, namely, the room temperature, particularly if the refrigerator door is open.

The thermostat behavior can be modeled by making the attribute `temperature` of the `cabinet` object an active value (see Section 4.6.13). The `thermostat` object is updated every time the registered temperature alters by more than a particular amount (say, 0.5°C). A method attached to the thermostat would toggle its state between *open circuit* and *closed circuit* depending on a comparison between the registered temperature and the set temperature. If the thermostat changes its state, it would send a message to the compressor, which in turn would change its state.

A map of possible states can be drawn up, with links indicating the transitions, which are the ways in which one state can be changed into another. A simple state map for a correctly functioning refrigerator is shown in Figure 12.6. A faulty refrigerator will have a different state map. Figure 12.7 shows the case of a thermostat that is stuck in the “*open circuit*” position.

Price and Hunt (1989) have modeled various mechanical devices using object-oriented programming techniques. They created an instance of the class

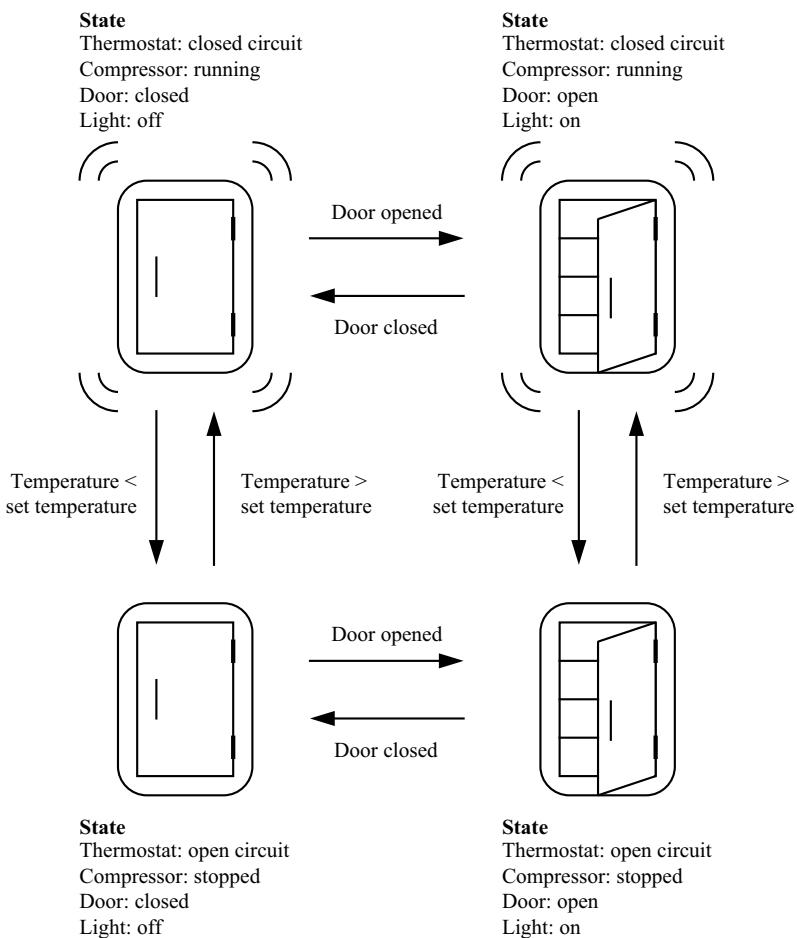


Figure 12.6 State map for a refrigerator working normally.

`device_state` for every state of a given device. In their system, each `device_state` instance is made up of instance variables (for example, recording external factors such as temperature) and links to components and functional groups that make up the device. This information is sufficient to completely restore a state. Price and Hunt found advantages in the ability to treat a device state as an object in its own right. In particular, the process of manipulating a state was simplified and kept separate from the other objects in the system. They were, thus, able to construct state maps similar to those shown in Figures 12.6 and 12.7, where each state was an instance of `device_state`. Instance links were used to represent transitions, such as opening the door, that could take one state to another.

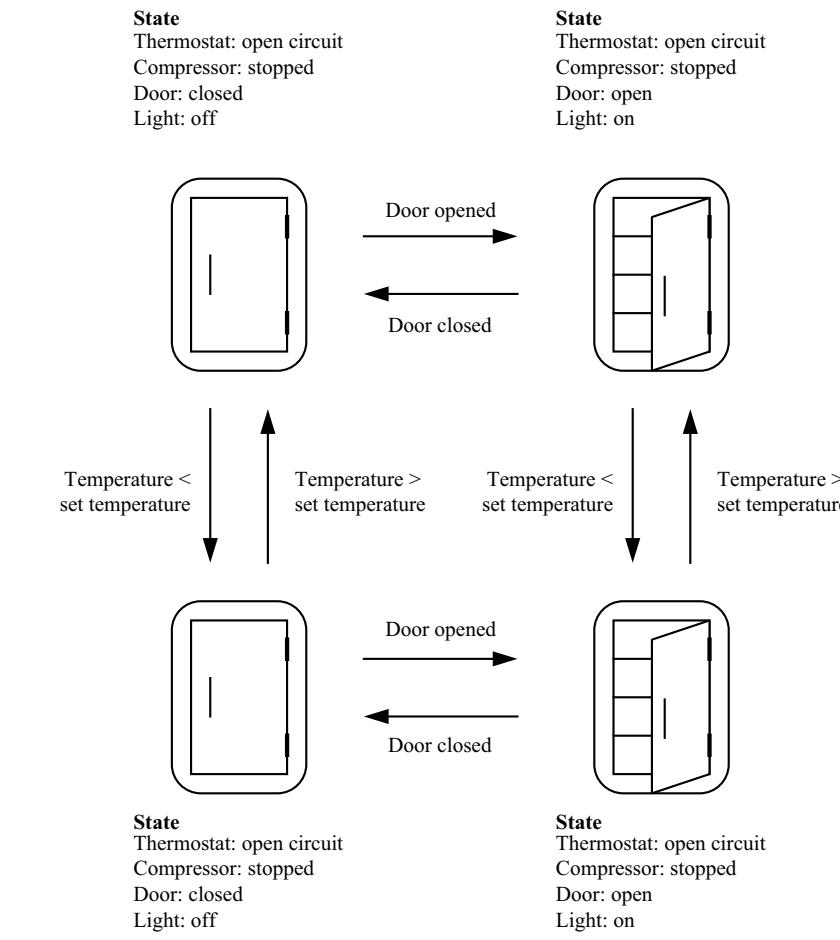


Figure 12.7 State map for a refrigerator with a faulty thermostat.

12.4.3 Using the Model

The details of how a model can assist the diagnostic task vary according to the specific device and the method of modeling it. In general, three potential uses can be identified:

- Monitoring the device to check for malfunctions
- Finding a suspect component, thereby forming a tentative diagnosis
- Confirming or refuting the tentative diagnosis by simulation

The diagnostic task is to determine which nonstandard component behavior in the model could make the output values of the model match those of the physical

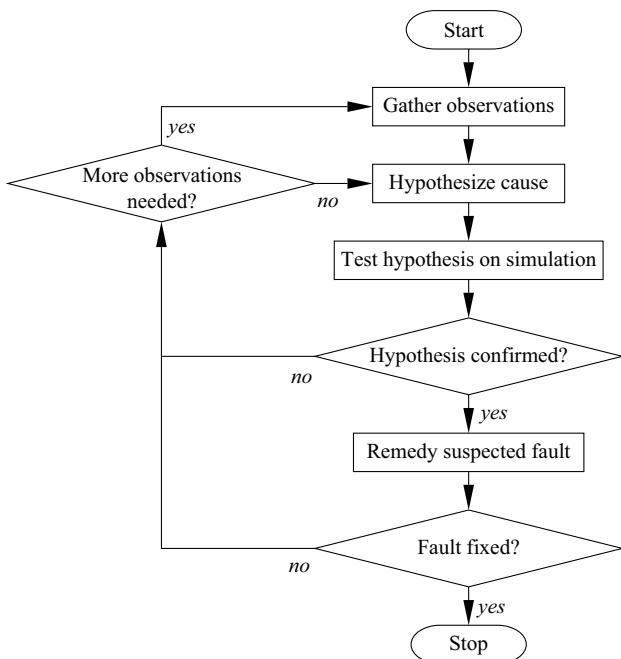


Figure 12.8 A strategy for model-based diagnosis.

system. An overall strategy is shown in Figure 12.8. A modification of this strategy is to place a weighting on the forward links between evidence and hypotheses. As more evidence is gathered, the weightings can be updated using the techniques described in Chapter 3 for handling uncertainty. The hypothesis with the highest weighting is tried first. If it fails, the next highest is considered, and so on. The weighting may be based solely on perceived likelihood, or it may include factors such as the cost or difficulty of fixing the fault. It is often worth trying a quick and cheap repair before resorting to a more expensive solution.

When a malfunction has been detected, the *single point of failure* assumption is often made. This assumption is that the malfunction has only one root cause. Such an approach is justified by Fulton and Pepe (1990) on the basis that no two failures are truly simultaneous. They argue that one failure will always follow the other, either independently or as a direct result of it.

A model can assist a diagnostic system that is confronted with a problem that lies outside its expertise. Since the function of a component is contained within the class definition, its behavior in novel circumstances may be predicted. If details of a specific type of component are lacking, comparisons can be drawn with sibling components in the class hierarchy.

12.4.4 Monitoring

A model can be used to simulate the behavior of a device. The output (such as data, a substance, or a signal) from one component forms the input to another component. If we alter one input to a component, the corresponding output may change, which may alter another input, and so on, resulting in a new set of sensor readings being recorded. Comparison with real-world sensors provides a monitoring facility.

The RESCU system (Leitch et al. 1991) uses model-based reasoning for monitoring inaccessible plant parameters. In a chemical plant, for instance, a critical parameter to monitor might be the temperature within the reaction chamber. However, it may not be possible to measure the temperature directly, as no type of thermometer would be able to survive the highly corrosive environment. Therefore, the temperature has to be inferred from temperature measurements at the chamber walls, and from gas pressure and flow measurements.

Rules or algorithms that can translate the available measurements (M_i) into the inaccessible parameters (P_i) are described by Leitch et al. (1991) as an *implicit model* (Figure 12.9a). Obtaining a reliable implicit model is often difficult, and *model-based reasoning* normally refers to the use of *explicit models* such as those described in Section 12.4.2. The real system and the explicit model are operated in parallel, the model generating values for the available measurements (M_i) and the inaccessible parameters (P_i). The model parameters (possibly including the parameters P_i) are adjusted to minimize the difference between the values of M_i generated by the model and the real system. This difference is called the *error*. By modifying the model in response to the error we have provided a feedback mechanism (Figure 12.9b),

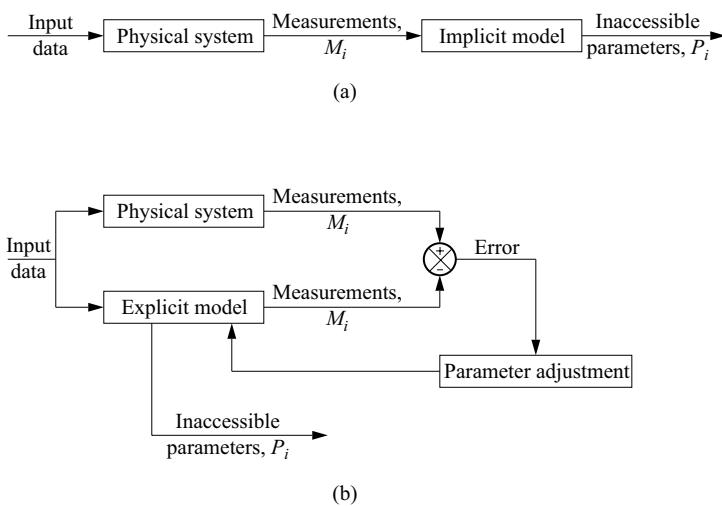


Figure 12.9 Monitoring inaccessible parameters by process modeling: (a) using an implicit model, (b) using an explicit model. (Derived from Leitch et al. 1991.)

discussed further in Chapter 15. This mechanism ensures that the model accurately mimics the behavior of the physical system. If a critical parameter P_i in the model deviates from its expected value, it is assumed that the same has occurred in the physical system and an alarm is triggered.

Analog devices (electrical or mechanical) can fail by varying degrees. When comparing a parameter in the physical system with the modeled value, we must decide how far apart the values have to be before we conclude that a discrepancy exists. Two ways of dealing with this problem are:

- to apply a tolerance to the expected value, so that an actual value lying beyond the tolerance limit is treated as a discrepancy; and
- to give the value a degree of membership of fuzzy sets (e.g., *much too high*, *too high*, *just right*, and *too low*). The degree of membership of these fuzzy sets determines the extent of the response needed. This approach is the essence of fuzzy control (see Chapters 3 and 15).

It is sometimes possible to anticipate a malfunction before it actually happens. This approach is adopted in EXPRES (Jennings 1989), a system for anticipating faults in the customer distribution part of a telephone network. The network is routinely subjected to electrical tests that measure characteristics such as resistance, capacitance, and inductance between particular points. A broken circuit, for example, would show up as a very high resistance. Failures can be anticipated and avoided by monitoring changes in the electrical characteristics with reference to the fault histories of the specific components under test and of similar components.

12.4.5 Tentative Diagnosis

The strategy for diagnosis shown in Figure 12.8 requires the generation of a hypothesis, that is, a tentative diagnosis. The method of forming a tentative diagnosis depends on the nature of the available data. In some circumstances, a complete set of symptoms and sensor measurements is immediately available, and the problem is one of interpreting them. More commonly in diagnosis, a few symptoms are immediately known, and additional measurements must be taken as part of the diagnostic process. The tentative diagnosis is a best guess, or hypothesis, of the actual cause of the observed symptoms. We will now consider three ways of generating a tentative diagnosis.

12.4.5.1 The Shotgun Approach

Fulton and Pepe (1990) advocate collecting a list of all objects that are upstream of the unexpected sensor reading. All of these objects are initially under suspicion. If several sensors have recorded unexpected measurements, only one sensor needs to be considered as it is assumed that there is only one cause of the problem. The process initially involves diagnosis at the functional-grouping level. Identification

of the faulty functional group may be sufficient. In that case, the diagnosis can stop and the functional group can simply be replaced in the real system. Alternatively, the process may be repeated by examining individual components within the failed functional group to locate the failed component.

12.4.5.2 Structural Isolation

KEATS (Motta et al. 1988) has been used for diagnosing faults in analog electronic circuits, and makes use of the *binary chop* or *structural isolation* strategy of Milne (1987). Initially the electrical signal is sampled in the middle of the signal chain and compared with the value predicted by the model. If the two values correspond, then the fault must be downstream of the sampling point, and a measurement is then taken halfway downstream. Conversely, if there is a discrepancy between the expected and actual values, the measurement is taken halfway upstream. This process is repeated until the defective functional grouping has been isolated. If the circuit layout permits, the process can be repeated within the functional grouping in order to isolate a specific component. Motta et al. (1988) found in their experiments that the structural isolation strategy closely resembles the approach adopted by human experts. They have also pointed out that the technique can be made more sophisticated by modifying the binary chop strategy to reflect heuristics concerning the most likely location of the fault.

12.4.5.3 The Heuristic Approach

Either of the preceding two strategies can be refined by the application of shallow (heuristic) knowledge. As noted in Section 12.3.3, IDM (Fink and Lusth 1987) has two knowledge bases, containing deep and shallow knowledge. If the shallow knowledge base has failed to find a quick solution, the deep knowledge base seeks a tentative solution using a set of five guiding heuristics:

1. If an output from a functional unit is unknown, find out its value by testing or by interaction with a human operator.
2. If an output from a functional unit appears incorrect, check its input.
3. If an input to a functional unit appears incorrect, check the source of the input.
4. If the input to a functional unit appears correct but the output is not, assume that the functional unit is faulty.
5. Examine components that are *nodes* before those that are *conduits*, as the former are more likely to fail in service.

12.4.6 Fault Simulation

Both the correct and the malfunctioning behavior of a device can be simulated using a model. The correct behavior is simulated during the monitoring of a device (Section

12.4.4). Simulation of a malfunction is used to confirm or refute a tentative diagnosis. A model allows the effects of changes in a device or in its input data to be tested. A diagnosis can be tested by changing the behavior of the suspected component within the model and checking that the model produces the symptoms that are observed in the real system. Such tests cannot be conclusive, as other faults might also be capable of producing the same symptoms, as noted in Section 12.2. Suppose that a real refrigerator is not working and makes no noise. If the thermostat on the refrigerator is suspected of being permanently open-circuit, this malfunction can be incorporated into the model and the effects noted. The model would show the same symptoms that are observed in the real system. The hypothesis is then confirmed as the most likely diagnosis.

Most device simulations proceed in a step-by-step manner, where the output of one component (component *A*) becomes the input of another (component *B*). Component *A* can be thought of as being “upstream” of component *B*. An input is initially supplied to the components that are furthest upstream. For instance, the thermostat is given a cabinet temperature, and the power cord is given a simulated supply of electricity. These components produce outputs that become the inputs to other components, and so on. This sort of simulation can run into difficulties if the model includes a feedback loop. In these circumstances, a defective component not only produces an unexpected output, but also has an unexpected input. The output can be predicted only if the input is known, and the input can be predicted only if the output is known. One approach to this problem is to supply initially all of the components with their correct input values. If the defective behavior of the feedback component is modeled, its output and input values would be expected to converge on a failure value after a number of iterations. Fink and Lusth (1987) found that convergence was achieved in all of their tests, but they acknowledge that there might be cases where convergence does not happen.

12.4.7 Fault Repair

Once a fault has been diagnosed, the next step is normally to fix the fault. Most fault diagnosis systems offer some advice on how a fault should be fixed. In many cases this recommendation is trivial, given the diagnosis. For example, the diagnosis worn bearings might be accompanied by the recommendation replace worn bearings, while the diagnosis leaking pipe might lead to the recommendation fix leak. A successful repair provides definite confirmation of a diagnosis. If a repair fails to cure a problem, then the diagnostic process must recommence. A failed repair may not mean that a diagnosis was incorrect. It is possible that the fault that has now been fixed had caused a second fault that also needs to be diagnosed and repaired.

12.4.8 Using Problem Trees

Some researchers (Dash 1990; Steels 1989) favor the explicit modeling of faults, rather than inferring faults from a model of the physical system. Dash (1990) builds

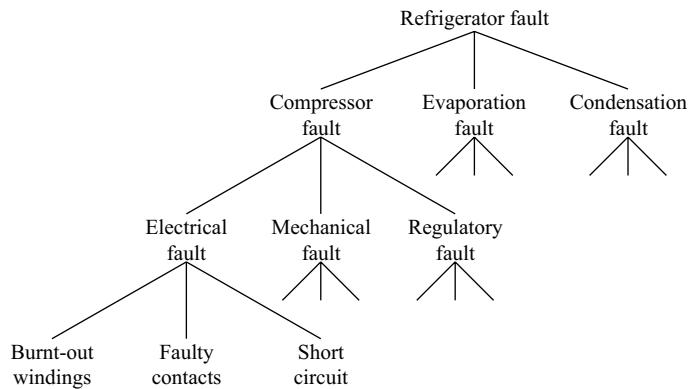


Figure 12.10 A problem tree for a refrigerator.

a hierarchical tree of possible faults (a *problem tree*, see Figure 12.10), similar to those used for classifying case histories (Figure 5.3a). Unlike case-based reasoning, problem trees cover all anticipated faults, whether or not they have occurred previously. By applying deep or shallow rules to the observed symptoms, progress can be made from the root (a general problem description) to the leaves of the tree (a specific diagnosis). The tree is, therefore, used as a means of steering the search for a diagnosis. An advantage of this approach is that a partial solution is formed at each stage in the reasoning process. A partial solution is generated even if there is insufficient knowledge or data to produce a complete diagnosis.

12.4.9 Summary of Model-Based Reasoning

Some of the advantages of model-based reasoning are as follows.

- A model is less cumbersome to maintain than a rule base. Real-world changes are easily reflected in changes in the model.
- The model need not waste effort looking for sensor verification. Sensors are treated identically to other components, and therefore a faulty sensor is as likely to be detected as any other fault.
- Unusual failures are just as easy to diagnose as common ones. This is not the case in a rule-based system, which is likely to be most comprehensive in the case of common faults.
- The separation of function, structure, and state may help a diagnostic system to reason about a problem that is outside its area of expertise.
- The model can simulate a physical system, for the purpose of monitoring or for verifying a hypothesis.

Model-based reasoning only works well in situations where there is a complete and accurate model. It is inappropriate for physical systems that are too complex to model

properly, such as medical diagnosis or weather forecasting. An ideal would be to build a model for monitoring and diagnosis directly from CAD data generated during the design stage. The model would then be available as soon as a device entered service.

12.5 Case Study: A Blackboard System for Interpreting Ultrasonic Images

One of the aims of this book is to demonstrate the application of a variety of techniques, including knowledge-based systems, computational intelligence, and procedural computation. The more complicated problems have many facets, where each facet may be best suited to a different technique. This is true of the interpretation of ultrasonic images, which will be discussed as a case study in the remainder of this chapter. A *blackboard system* has been used to tackle this complex interpretation problem (Hopgood et al. 1993). As we saw in Chapter 10, blackboard systems allow a problem to be divided into sub-tasks, each of which can be tackled using the most suitable technique.

An image interpretation system attempts to understand the processed image and describe the world represented by it. This problem requires symbolic reasoning (using rules, objects, relationships, list-processing, or other techniques) as well as numerical processing of signals (Walker and Fox 1987). ARBS (Algorithmic and Rule-based Blackboard System—Hopgood et al. 1998) and its successor DARBS (Distributed ARBS—Tait et al. 2008; Nolle et al. 2002c) have been designed to incorporate both types of processes. Numerically intensive signal processing, which may involve a large amount of raw data, is performed by conventionally coded agents. Facts, causal relationships, and strategic knowledge are symbolically encoded in one or more knowledge bases. This explicit representation allows encoded domain knowledge to be modified or extended easily. Signal processing routines are used to transform the raw data into a description of the key features in the data, that is, a symbolic image. Knowledge-based techniques are used to interpret this symbolic image.

The architecture allows signal interpretation to proceed opportunistically, building up from initial subsets of the data through to higher-level observations and deductions. When a particular set of rules needs access to a chunk of raw data, it can have it. No attempt has been made to write rules that look at the whole of the raw data, as the preprocessing agents avoid the need for this. Although a user interface is provided to allow monitoring and intervention, DARBS is designed to run independently, producing a log of its actions and decisions.

12.5.1 Ultrasonic Imaging

Ultrasonic imaging is widely used for the detection and characterization of features, particularly defects, in manufactured components. The technique belongs to a family of nondestructive testing methods that are distinguished by the ability to examine components without causing any damage. Various arrangements can be used for

producing images using ultrasound, that is, high frequency sound at approximately 1–10 MHz. Typically, a transmitter and receiver of ultrasound are situated within a single probe that makes contact with the surface of the component. The probe emits a short pulse of ultrasound and then detects waves returning as a result of interactions with the features within the specimen. If the detected waves are assumed to have been produced by single reflections, then the time of arrival can be readily converted to the depth (z) of these features. By moving the probe in one dimension (y), an image can be plotted of y versus z , with intensity often represented by color or grayscale. This image is called a *b-scan*, and it approximates to a cross-section through a component. It is common to perform such scans with several probes pointing at different angles into the specimen, and to collect several b-scans by moving the probes in a raster (Figure 12.11).

A typical b-scan image is shown in Figure 12.12. A threshold has been applied, so that only received signals of intensity greater than -30 dB are displayed, and these appear as black dots on the image. Ten probes were used, pointing into the specimen at five different angles. Because ultrasonic beams are not collimated, a point defect is detected over a range of y values, giving rise to a characteristic arc of dots on the b-scan image. Arcs produced by a single probe generally lie parallel to each other and normal to the probe direction. The point of intersection of arcs produced by different probes is a good estimate of the location of the defect that caused them.

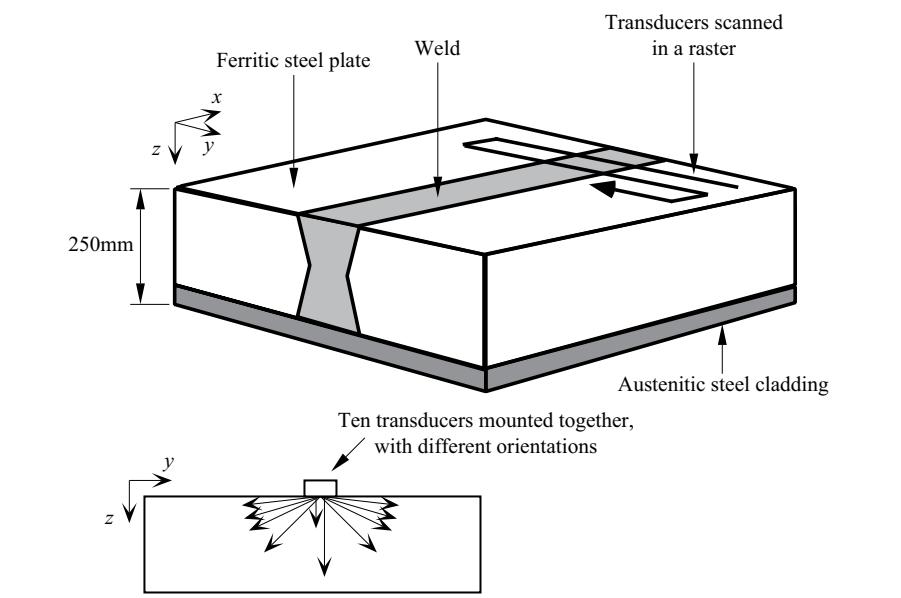


Figure 12.11 Gathering ultrasonic b-scan images.

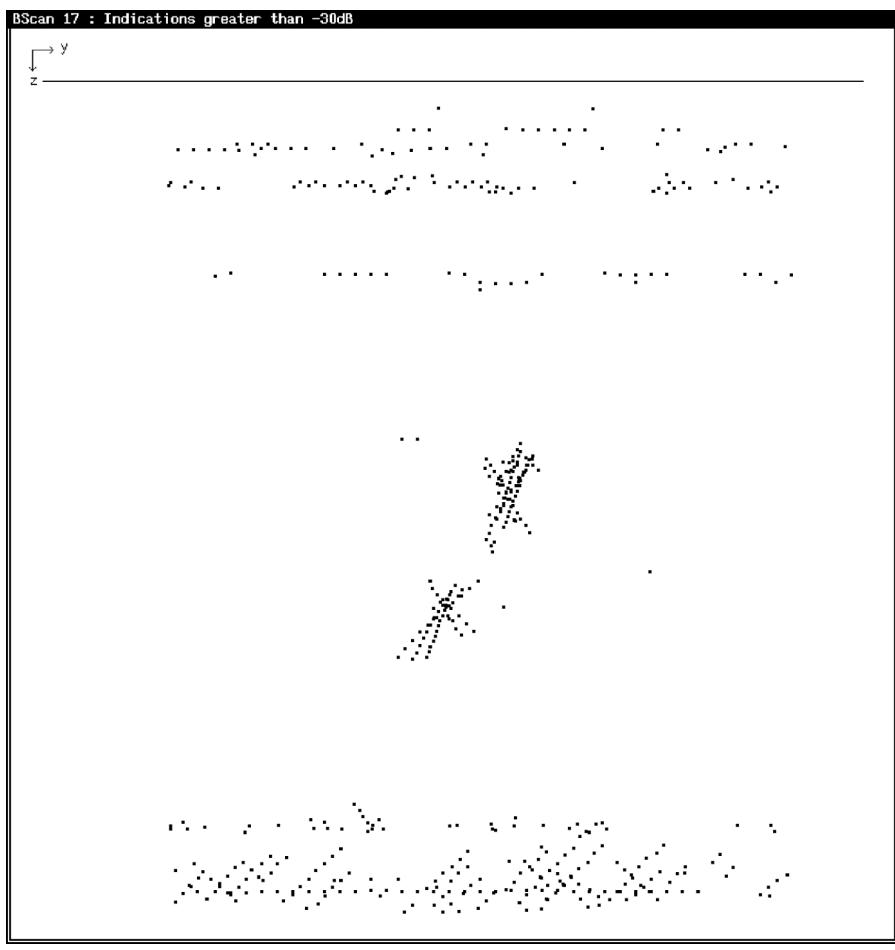


Figure 12.12 A typical b-scan image.

The problems of interpreting a b-scan are different from those of other forms of image interpretation. The key reason for this is that a b-scan is not a direct representation of the inside of a component, but rather it represents a set of wave interactions (reflection, refraction, diffraction, interference, and mode conversion) that can be used to *infer* some of the internal structure of the component.

12.5.2 Agents in DARBS

Each agent in DARBS is an autonomous parallel process. The agents can run concurrently on one computer or distributed across many computers connected by the Internet. Each agent is represented as a structure comprising seven primary elements,

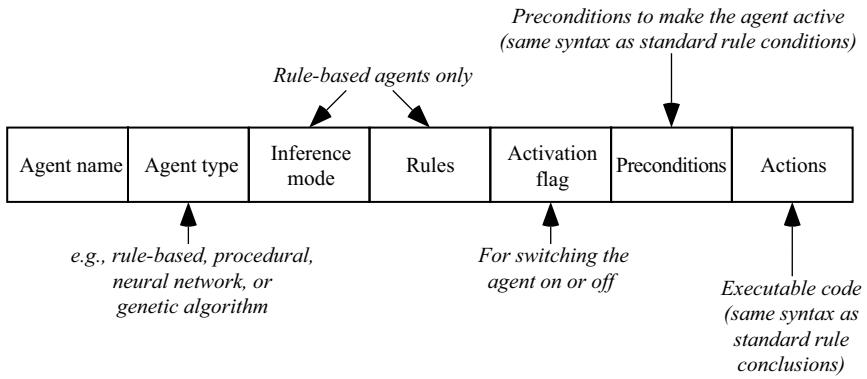


Figure 12.13 Structure of a DARBS agent.

shown in Figure 12.13. The open-source version of DARBS* represents all agents, rules, messages, and blackboard information in XML (Extensible Markup Language). XML is a standard format for textual information, widely used for Web documents but equally appropriate for structuring the textual information used in DARBS.

The first element of an agent specifies its name and the second specifies its type. For rule-based agents, the third and fourth elements specify the inference mode and rules, respectively. An activation flag in the fifth element allows individual agents to be switched on or off. The sixth element contains a set of preconditions that must be satisfied before the agent performs a task. Each agent constantly monitors the blackboard, waiting for its preconditions to be satisfied so that it can become active. As soon as the preconditions are satisfied, the agent performs its task opportunistically. Unlike earlier blackboard implementations, such as ARBS, there is no need for a scheduler in DARBS. The precondition may comprise subconditions joined with Boolean operators *and* and *or*. Finally, the seventh element states what actions are to be performed before the agent finishes its task. For procedural, neural network and other non-rule-based agents, the functions and procedures are listed in the action element.

Agents in DARBS can read data from the blackboard simultaneously. However, to avoid deadlock, only one agent is allowed to write data to the same partition of the blackboard at one time. Whenever the content of the blackboard changes, the blackboard server broadcasts a message to all agents. The agents themselves decide how to react to this change. For example, it may be appropriate for an agent to restart its current task. With this approach, agents are completely opportunistic, meaning that they activate themselves whenever they have some contributions to make to the blackboard.

When an agent is active, it applies its knowledge to the current state of the blackboard, adding to it or modifying it. If an agent is rule-based, then it is essentially a

* Information on accessing DARBS is available at adrianhopgood.com.

rule-based system in its own right. A rule's conditions are examined and, if they are satisfied by a statement on the blackboard, the rule fires and the actions dictated by its conclusion are carried out. Single or multiple instantiation of variables can be selected. When the rules are exhausted, the agent is deactivated and any actions in the `actions` element are performed. These actions usually involve reports to the user or the addition of control information to the blackboard. If the agent is procedural or contains a neural network or genetic algorithm, then the required code is simply included in the `actions` element and the elements relating to rules are ignored.

The blackboard architecture is able to bring together the most appropriate tools for handling specific tasks. Procedural tasks are not entirely confined to procedural agents, since rules within a rule-based agent can access procedural code from either their condition or conclusion parts. In the case of ultrasonic image interpretation, procedural agents written in C++ are used for fast, numerically intensive data-processing. An example is the procedure that preprocesses the image, using the Hough Transform to detect lines of indications (or dots). Rule-based agents are used to represent specialist knowledge, such as the recognition of image artifacts caused by probe reverberations. Neural network agents can be used for judgmental tasks, such as classification, that involve the weighing of evidence from various sources. Judgmental ability is often difficult to express in rules, and neural networks were incorporated into DARBS as a response to this difficulty.

12.5.3 Rules in DARBS

Rules are used in DARBS in two contexts:

- to express domain knowledge within a rule-based agent; and
- to express the applicability of an agent through its preconditions.

Just as agents are activated in response to information on the blackboard, so too are individual rules within a rule-based agent. The main functions of DARBS rules are to look up information on the blackboard, to draw inferences from that information, and to post new information on the blackboard. Rules can access procedural code for performing such tasks as numerical calculations or database lookup.

In early versions of DARBS, rules were implemented as lists (see Section 11.2.1), delineated by square brackets. These have subsequently been replaced by XML structures. The rules are interpreted by sending them to a piece of software, known as a parser, that breaks down a rule into its constituent parts and interprets them. The DARBS rule parser first extracts the condition statement and evaluates it, using recursion if there are embedded subconditions. If a rule is selected for firing and its overall condition is found to be true, all of the conclusion statements are interpreted and carried out.

Atomic conditions, that is, conditions that contain no subconditions, can be evaluated in any of the following ways:

- Test for the presence of information on the blackboard and look up the information if it is present.
- Call algorithms or external procedures that return Boolean or numerical results.
- Numerically compare variables, constants, or algorithm results.

The conclusions, or subconclusions, can comprise any of the following:

- Add or remove information to or from the blackboard.
- Call an algorithm or external procedure, and optionally add the results to the blackboard.
- Report actions to the operator.

The strategy for applying rules is a key decision in the design of a system. In many types of rule-based system, this decision is irrevocable, committing the rule-writer to either a forward-chaining or backward-chaining system. However, the blackboard architecture allows much greater flexibility, as each rule-based agent can use whichever inference mechanism is most appropriate. The rule structure has been designed so flexibly that fuzzy rules can be incorporated without changing either the rule syntax or inference engines (Hopgood et al. 1998).

DARBS offers a choice of inference engines, including options for multiple and single instantiation of variables, described in Section 2.7.1. Under multiple instantiation, all possible matches to the variables are found and acted upon with a single firing of a rule. In contrast, only the first match to the variables is found when single instantiation is used (Hopgood 1994). Figure 12.14 shows the difference between the two inference engines in the case of a rule that analyses rectangular areas of arc intersection in the image.

The hybrid inference mechanism described in Section 2.10 requires the construction of a network representing the dependencies between the rules. A separate dependence network can be built for each rule-based agent, prior to running the system. The networks are saved and only need to be regenerated if the rules are altered. The code to generate the networks is simplified by the fact that the only interaction between rules is via the blackboard. For rule *A* to enable rule *B* to fire, rule *A* must either add something to the blackboard that rule *B* needs to find or it must remove something that rule *B* requires to be absent. When a rule-based agent is activated, the rules within the agent are selected for examination in the order dictated by the dependence network.

Rule-based agents in DARBS can generate hypotheses that can then be tested and thereby supported, confirmed, weakened, or refuted. DARBS uses this hypothesize-and-test approach (see Section 12.2.3) to handle the uncertainty that is inherent in problems of abduction and the uncertainty that arises from sparse or noisy data. The hypothesize-and-test method reduces the solution search space by focusing attention on those agents relevant to the current hypothesis.

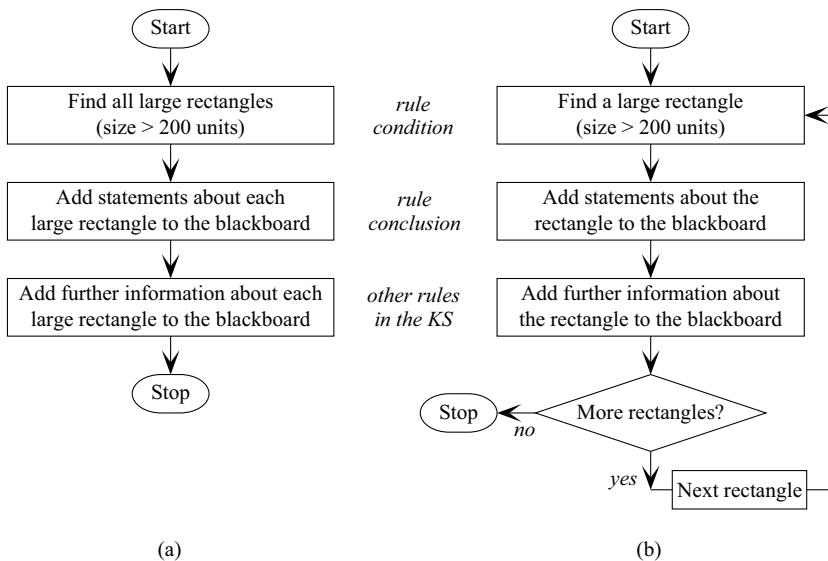


Figure 12.14 Firing a rule using: (a) multiple instantiation of variables, and (b) single instantiation of variables.

12.5.4 The Stages of Image Interpretation

The problem of ultrasonic image interpretation can be divided into three distinct stages: arc detection, gathering information about the regions of intersecting arcs, and classifying defects on the basis of the gathered information. These stages are now described in more detail.

12.5.4.1 Arc Detection Using the Hough Transform

The first step toward defect characterization is to place on the blackboard the important features of the image. This step is achieved by a procedural agent that examines the image (Figure 12.15a) and fits arcs to the data points (Figure 12.15b). In order to produce these arcs, a Hough transform (Duda and Hart 1972) is used to determine the groupings of points. The transform has been modified so that isolated points some distance from the others are not included. The actual positions of the arcs are determined by least-squares fitting.

This preprocessing phase is desirable in order to reduce the volume of data and to convert it into a form suitable for knowledge-based interpretation. Thus, knowledge-based processing begins on data concerning approximately 30 linear arcs rather than on data concerning 400–500 data points. No information is lost permanently. If, in the course of its operations, DARBS judges that more data concerning a particular line would help the interpretation, it retrieves the information about the individual points that along that line, and it represents those point data on the blackboard. It

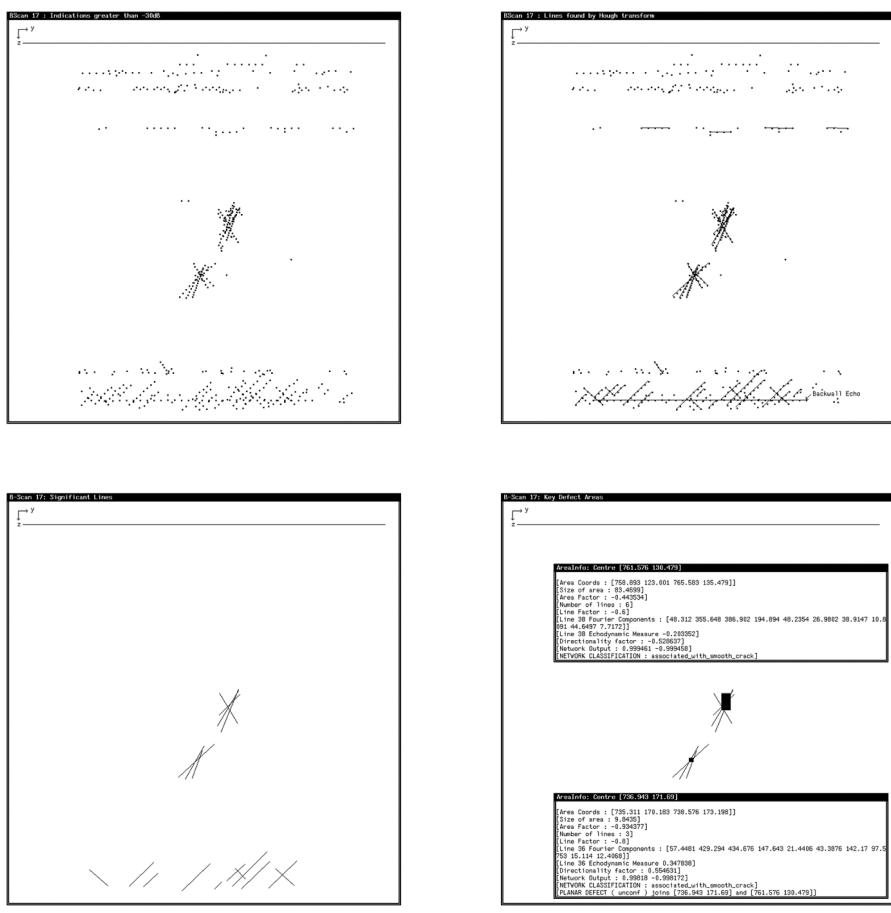


Figure 12.15 The stages of interpretation of a b-scan image: (a) before interpretation; (b) with lines found by a Hough transform; (c) significant lines; (d) conclusion: a crack runs between the two marked areas.

is natural, moreover, to work with lines rather than points in the initial stages of interpretation. Arcs of indications are produced by all defect types, and much of the knowledge used to identify flaws is readily couched in terms of the properties of lines and the relations between them.

12.5.4.2 Gathering the Evidence

Once a description of the lines has been recorded on the blackboard, a rule-based agent picks out those lines that are considered significant according to criteria such as intensity, number of points, and length. Rules are also used to recognize the back-wall echo and lines that are due to probe reverberation. Both are considered

“insignificant” for the time being. Key areas in the image are generated by finding points of intersection between significant lines and then grouping them together. Figure 12.15d shows the key areas found by applying this method to the lines shown in Figure 12.15c. For large smooth cracks, each of the crack tips is associated with a distinct area. Other defects are entirely contained in their respective areas.

Rule-based agents are used to gather evidence about each of the key areas. The evidence includes:

- Size of the area
- Number of arcs passing through it
- Shape of the echodynamic (defined in the following text)
- Intensity of indications
- Sensitivity of the intensity to the angle of the probe

The *echodynamic* associated with a defect is the profile of the signal intensity along one of the “significant” lines. This profile is of considerable importance in defect classification as different profiles are associated with different types of defects. In particular, the echodynamics are expected to be similar to those in Figure 12.16 for a smooth crack face, a spheroidal defect (e.g., a single pore or an inclusion) or crack tip, and a series of gas pores (Halmshaw 1987). In DARBS, pattern classification of the echodynamic is performed using a fast Fourier transform and a set of rules to analyze the features of the transformed signal. A neural network has also been used for the same task (see Section 12.5.5).

The sensitivity of the defect to the angle of the probe is another critical indicator of the nature of the defect in question. Roughly spherical flaws, such as individual gas pores, have much the same appearance when viewed from different angles. Smooth cracks on the other hand have a much more markedly directional character, so a small difference in the direction of the probe may result in a considerable reduction (or increase) in the intensity of indications. The directional sensitivity of an area of

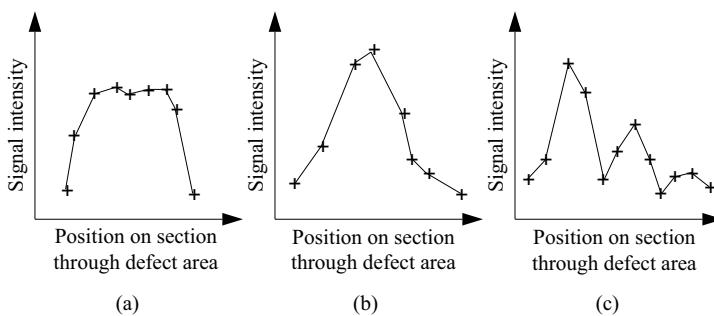


Figure 12.16 Echodynamics across (a) a crack face; (b) a crack tip, pore, or inclusion; (c) an area of porosity.

intersection is a measure of this directionality and is represented in DARBS as a number between -1 and 1.

12.5.4.3 Defect Classification

Quantitative evidence about key areas in an image is derived by rule-based agents, as described in the previous subsection. Each piece of evidence provides clues as to the nature of the defect associated with the key area. For instance, the indications from smooth cracks tend to be sensitive to the angle of the probe, and the echodynamic tends to be plateau-shaped. In contrast, indications from a small round defect, like an inclusion, tend to be insensitive to probe direction and have a cusp-shaped echodynamic. There are several factors like these that need to be taken into account when producing a classification, and each must be weighted appropriately.

Two techniques for classifying defects based upon the evidence have been tried using DARBS: a rule-based hypothesize-and-test approach and a neural network. In the former approach, hypotheses concerning defects are added to the blackboard. These hypotheses relate to smooth or rough cracks, porosity, or inclusions. They are tested by deriving from them *expectations* (or *predictions*) relating to other features of the image. On the basis of the correspondence between the expectations and the image, DARBS arrives at a conclusion about the nature of a defect, or, where this is not possible with any degree of certainty, it alerts the user to a particular problem case.

Writing rules to verify the hypothesized defect classifications is a difficult task and, in practice, the rules need continual refinement and adjustment in the light of experience. The use of neural networks to combine the evidence and produce a classification provides a means of circumventing this difficulty since the neural networks need only a representative training set of examples, instead of the formulation of explicit rules.

12.5.5 The Use of Neural Networks

Neural networks have been used in DARBS for two quite distinct tasks, described in the following subsections.

12.5.5.1 Defect Classification Using a Neural Network

Neural networks can perform defect classification, provided there are sufficient training examples and the evidence can be presented in numerical form. In this study (Hopgood et al. 1993), there were insufficient data to train a neural network to perform a four-way classification of defect types, as done under the hypothesize-and-test method. Instead, a backpropagation network was trained to classify defects as either critical (i.e., a smooth crack) or noncritical on the basis of four local factors: the size of the area, the number of arcs passing through it, the shape of the

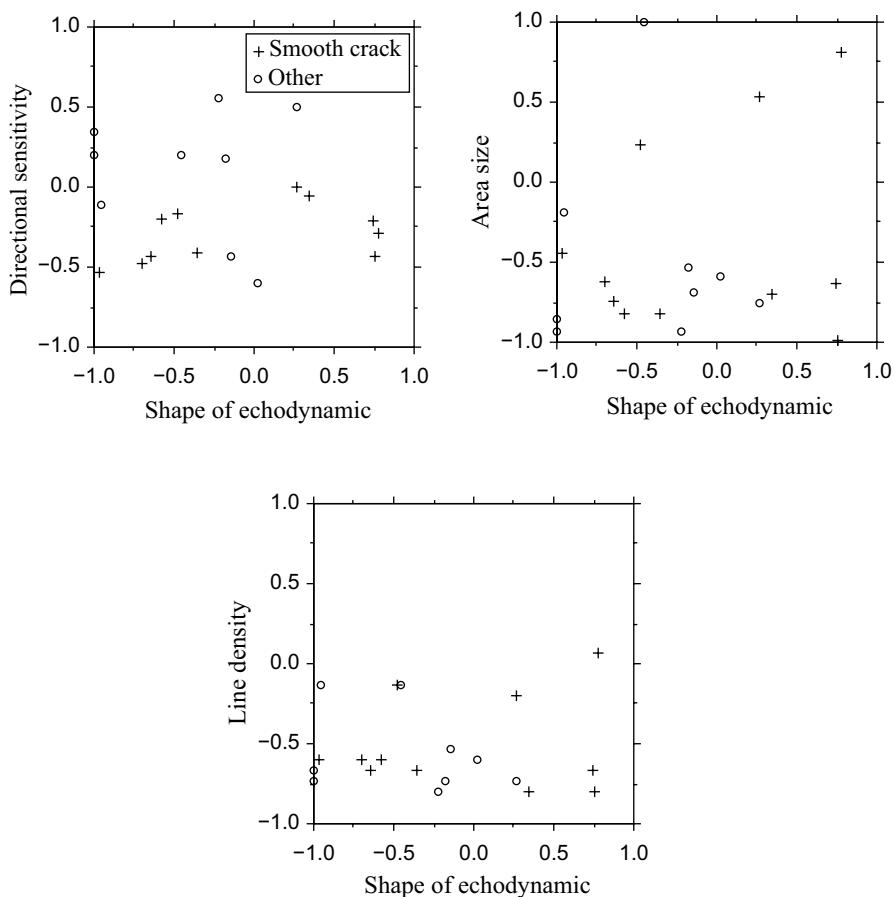


Figure 12.17 Evidence for the classification of 20 defects.

echodynamic, and the sensitivity of the intensity to the angle of the probe. Each of these local factors was expressed as a number between –1 and 1.

Figure 12.17 shows plots of evidence for the 20 defects that were used to train and test a neural network. The data are, in fact, points in four-dimensional space, where each dimension represents one of the four factors considered. Notice that the clusters of critical and noncritical samples might not be linearly separable. In that case, the traditional numerical techniques for finding linear discriminators (Duda and Hart 1973) are not powerful enough to produce good classification. However, a multilayer perceptron (MLP; Section 8.4) is able to discriminate between the two cases, since it is able to find the three-dimensional surface required to separate them. Using the leave-one-out technique (Section 8.4.6), an MLP with two hidden layers correctly classified 16 out of 20 images from defective components (Hopgood et al. 1993).

12.5.5.2 Echodynamic Classification Using a Neural Network

One of the inputs to the classification network requires a value between -1 and 1 to represent the shape of the echodynamic. This value can be obtained by using a rule-based agent that examines the Fourier components of the echodynamic and uses heuristics to provide a numerical value for the shape. An alternative approach is to use another neural network to generate this number.

An echodynamic is a signal intensity profile across a defect area and can be classified as a *cusp*, *plateau*, or *wiggle*. Ideally, a neural network would make a three-way classification, given an input vector derived from the amplitude components of the first n Fourier coefficients, where $2n$ is the echodynamic sample rate. However, cusps and plateaus are difficult to distinguish since they have similar Fourier components, so a two-way classification is more practical, with cusps and plateaus grouped together. A multilayer perceptron has been used for this purpose.

12.5.5.3 Combining the Two Applications of Neural Networks

The use of two separate neural networks in distinct agents for the classification of echodynamics and of the potential defect areas might seem unnecessary. Because the output of the former feeds, via the blackboard, into the input layer of the latter, this arrangement is equivalent to a hierarchical MLP (Section 8.4.4). In principle, the two networks could have been combined into one large neural network, thereby removing the need for a preclassified set of echodynamics for training. However, such an approach would lead to a loss of modularity and explanation facilities. Furthermore, it may be easier to train several small neural networks separately on subtasks of the whole classification problem than to attempt the whole problem at once with a single large network. These considerations are important when there are many subtasks amenable to connectionist treatment.

12.5.6 Rules for Verifying Neural Networks

Defect classification, whether performed by the hypothesize-and-test method or by neural networks, has so far been discussed purely in terms of evidence gathered from the region of the image that is under scrutiny. However, there are other features in the image that can be brought to bear on the problem. Knowledge of these features can be expressed easily in rule form and can be used to *verify* the classification. The concept of the use of rules to verify the outputs from neural networks was introduced in Section 10.5. In this case, an easily identifiable feature of a b-scan image is the line of indications due to the back-wall echo. A defect in the sample, particularly a smooth crack, will tend to cast a “shadow” on the back wall directly beneath it (Figure 12.18). The presence of a shadow in the expected position can be used to verify the location and classification of a defect. In this application, the absence of this additional evidence is not considered a strong enough reason to

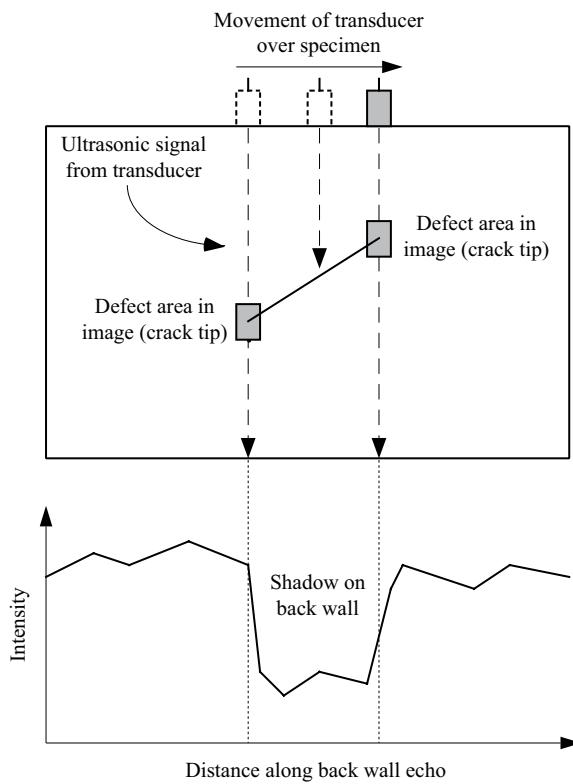


Figure 12.18 A shadow on the back wall can confirm the presence of a crack.

reject a defect classification. Instead, the classification in such cases is marked for the attention of a human operator, as there are grounds for doubt over its accuracy.

12.6 Summary

This chapter has introduced some of the techniques that can be used to tackle the problems of automated interpretation and diagnosis. Diagnosis is considered to be a specialized case of the more general problem of interpretation. It has been shown that a key part of diagnosis and interpretation is abduction, the process of determining a cause, given a set of observations. There is uncertainty associated with abduction, since causes other than the selected one might give rise to the same set of observations. Two possible approaches to dealing with this uncertainty are to explicitly represent the uncertainty using the techniques described in Chapter 3, or to *hypothesize-and-test*. As the name implies, the latter technique involves generating a hypothesis, or best guess, and either confirming or refuting the hypothesis depending on whether it is found to be consistent with the observations.

Several different forms of knowledge can contribute to a solution. We have paid specific attention to rules, case histories, and physical models. We have also shown that neural networks and conventional programming can play important roles when included as part of a blackboard system. Rules can be used to represent both shallow (heuristic) and deep knowledge. They can also be used for the generation and verification of hypotheses. Case-based reasoning, introduced in Chapter 5, involves comparison of a given scenario with previous examples and their solutions. Model-based reasoning relies on the existence of a model of the physical system, where that model can be used for monitoring, generation of hypotheses, and verification of hypotheses by simulation.

Blackboard systems have been introduced as part of a case study into the interpretation of ultrasonic images. These systems allow various forms of knowledge representation to come together in one system. They are, therefore, well suited to problems that can be broken down into subtasks. The most suitable form of knowledge representation for different subtasks is not necessarily the same, but each subtask can be individually represented by a separate intelligent agent.

A neural network agent was shown to be effective for combining evidence generated by other agents and for categorizing the shape of an echodynamic. This approach can be contrasted with the use of a neural network alone for interpreting images. The blackboard architecture avoids the need to abandon rules in favor of neural networks or vice versa, since the advantages of each can be incorporated into a single system. Rules can represent knowledge explicitly, whereas neural networks can be used where explicit knowledge is hard to obtain. Although neural networks can be rather impenetrable to the user and are unable to explain their reasoning, these deficiencies can be reduced by using them for small-scale localized tasks with reports generated in between.

Further Reading

- Ding, S. X. 2013. *Model-based Fault Diagnosis Techniques: Design Schemes, Algorithms, and Tools*. 2nd ed. Springer, Berlin, Germany.
- Korbicz, J., J. M. Koscielny, Z. Kowalcuk, and W. Cholewa, eds. 2004. *Fault Diagnosis: Models, Artificial Intelligence, Applications*. Springer, Berlin, Germany.
- Pascual, D. G. 2015. *Artificial Intelligence Tools: Decision Support Systems in Condition Monitoring and Diagnosis*, CRC Press, Boca Raton, FL.
- Price, C. J. 2000. *Computer-Based Diagnostic Systems*. Springer-Verlag, London, UK.

Chapter 13

Systems for Design and Selection

13.1 The Design Process

Before discussing how intelligent systems can be applied to design, it is important to understand what we mean by the word *design*. Traditionally, a distinction is drawn between engineering design and industrial design:

Engineering design is the use of scientific principles, technical information, and imagination in the definition of a mechanical structure, machine, or system to perform specified functions with the maximum economy and efficiency.

(Fielden 1963)

Industrial design seeks to rectify the omissions of engineering; it is a conscious attempt to bring form and visual order to engineering hardware where technology does not of itself provide these features.

(Moody 1980)

We will take a broad view of design, in which no distinction is drawn between the technical needs of engineering design and the aesthetic approach of industrial design. Our working definition of design will be as follows (Sriram et al. 1989):

[Design is] the process of specifying a description of an artifact that satisfies constraints arising from a number of sources by using diverse sources of knowledge.

Some of the constraints must be predetermined, and these constitute the *product design specification* (PDS). Other constraints may evolve as a result of decisions made during the design process (Pons and Raine 2005). The PDS is an expression of the *requirements* of a product, rather than a specification of the product itself. The latter, which emerges during the design process, is the design. The design can be interpreted for manufacture or construction, and it allows predictions about the performance of the product to be drawn.

Different authors have chosen to analyze the design process in different ways. An approximate consensus is that the broadest view of the design process comprises the following phases (Figure 13.1):

- Market
- Specification
- Design (narrow view)
- Manufacture
- Selling

The narrow view of design leads from a PDS to the manufacturing stage. It can be subdivided as follows:

- Conceptual design
- Optimization/evaluation
- Detailed design

The purpose of each phase in the broad design process is as follows.

1. *Market*: This phase is concerned with determining the need for a product. A problem is identified, resources allocated, and end-users targeted.

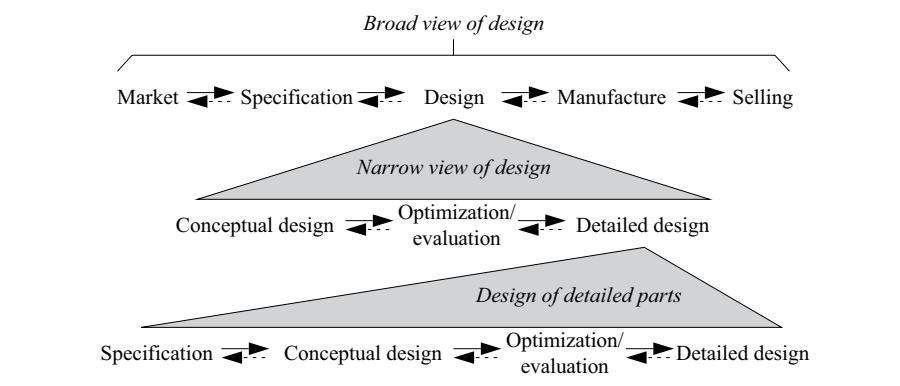


Figure 13.1 The principal phases of design.

2. *Specification:* A PDS is drawn up that describes the requirements and performance specifications of the product. The PDS for a motorcar might specify a product that can transport up to four people in comfort, traveling on roads at speeds up to the legal limit.
3. *Conceptual design:* Preliminary design decisions are made at this stage, with the aim of satisfying a few key constraints. Several alternatives would normally be considered. Decisions taken at the conceptual design stage determine the general form of the product, and so have enormous implications for the remainder of the design process. The conceptual design for a motorcar has altered little since the Ford Model T was unveiled in 1908. It describes a steel body with doors and windows, a wheel at each corner, two seats at the front (one of which has access to the controls), two seats at the back, and so on.
4. *Optimization/evaluation:* The conceptual design is refined, for instance by placing values on numerical attributes such as length and thickness. The performance of the conceptual design is tested for its response to external effects and its consistency with the PDS. The optimization and evaluation stage for a motorcar might include an assessment of the relationship between the shape of the body and its drag coefficient. As a further example, the energy consumption of a conceptual building design can be optimized using genetic algorithms (Caldas and Norford 2002). If the conceptual design cannot be made to meet the requirements, a new one is needed.
5. *Detailed design:* The design of the product and its components are refined so that all constraints are satisfied. Decisions taken at this stage might include the layout of a car's transmission system, the position of the entertainment system, the covering for the seats, and the total design of a door latch. The latter example illustrates that the complete design process for a component may be embedded in the detailed design phase of the whole assembly (Figure 13.1).
6. *Manufacture:* A product should not be designed without consideration of how it is to be manufactured, as it is all too easy to design a product that is uneconomical or impossible to produce. For a product that is to be mass-produced, the manufacturing plant needs to be designed just as rigorously as the product itself. Different constraints apply to a one-off product, as this can be individually crafted but mass-production techniques such as injection molding are not feasible.
7. *Selling:* The chief constraint for most products is that they should be sold at a profit. The broad view of design, therefore, takes into account not only how a product can be made, but also how it is to be sold.

Although the design process has been portrayed as a chronological series of events, in fact, there is considerable interaction between the phases—both forward and backward—as constraints become modified by the design decisions that are made. For instance, a decision to manufacture one component from polyethylene rather

than steel has ramifications for the design of other components and implications for the manufacturing process. It may also alter the PDS, as the polymer component may offer a product that is cheaper but less structurally rigid. Similarly, sales of a product can affect the market, thus linking the last design phase with the first.

In our description of conceptual and detailed design, we have made reference to the choice of materials from which to manufacture the product. Materials selection is one of the key aspects of the design process, and one where considerable effort has been placed in the application of intelligent systems. The process of materials selection is discussed in detail in Section 13.8. Selection is also the key to other aspects of the design process, as attempts are made to select the most appropriate solution to the problem.

The description of the design process that has been proposed is largely independent of the nature of the product. The product may be a single component (such as a turbine blade) or a complex assembly (such as a jet engine). It may be a one-off product or one that will be produced in large numbers. Many designs do not involve manufacture at all in the conventional sense. An example that is introduced in Section 13.4 is the design of a communications network. That example is a high-level design that is not concerned with the layout of wires or optical fibers. Rather, it concerns the overall configuration of the network. The product is a *service* rather than a physical thing. Although selection is again one of the key tasks, materials selection is not applicable in that example.

In summary, we can categorize products according to whether they are:

- Service-based or physical products
- Single component products or assemblies of many components
- One-off products or mass-produced products

Products in each category will have different requirements, leading to a different PDS. However, these differences do not necessarily alter the design process.

Three case studies are introduced in this chapter. The specification of a communications network is used to illustrate the importance and potential complexity of the PDS. The processes of conceptual design, optimization/evaluation, and detailed design are illustrated with reference to the floor of a passenger aircraft. This case study will introduce some aspects of the materials selection problem, and these are further illustrated by the third case study, which concerns the design of a kettle.

13.2 Design as a Search Problem

Design can be viewed as a search problem, as it involves searching for an optimum or adequate design solution (Moskewicz et al. 2001). Alternative solutions may be

known in advance (these are *derivation* problems), or they may be generated automatically (these are *formulation* problems). Designs may be tested as they are found in order to check whether they are feasible and meet the design requirements. This approach is the *generate-and-test* method. In application areas such as diagnosis (see Chapter 12), it may be sufficient to terminate the search as soon as a solution is found. In design, there are likely to be many solutions, and we would like to find “the best.” The search may, therefore, continue in order to find many feasible designs from which a selection can be made.

Search becomes impractical when large numbers of unreasonable designs are included. Consider, for example, the design of a house. In order to generate solutions automatically, we might write a computer program that generates every conceivable combination of shapes and sizes of rooms, walls, roofs, and foundations. Of this massive number of alternatives, only a small proportion would be feasible designs. In order to make the search problem manageable, some means of eliminating the unfeasible designs is needed. Better still would be a means of eliminating whole families of ill-conceived designs before the individual variants have been considered. The design-generator could be modified by heuristics so that it produced only designs with the roof above the walls and with the walls above the foundations. This would have the effect of pruning the search space (Figure 13.2). The search space can also be reduced by decomposing the design problem into subproblems of designing the rooms, roof, and foundations separately, each with its own smaller search tree.

The search problem is similar to the proposition that a monkey playing random notes on a grand piano will eventually play a Mozart piano concerto. The fault in this proposition is that the search space of compositions is so immense that the monkey would not stumble across the concerto within a practical time-frame. Only a composer with knowledge of suitable musical arrangements could

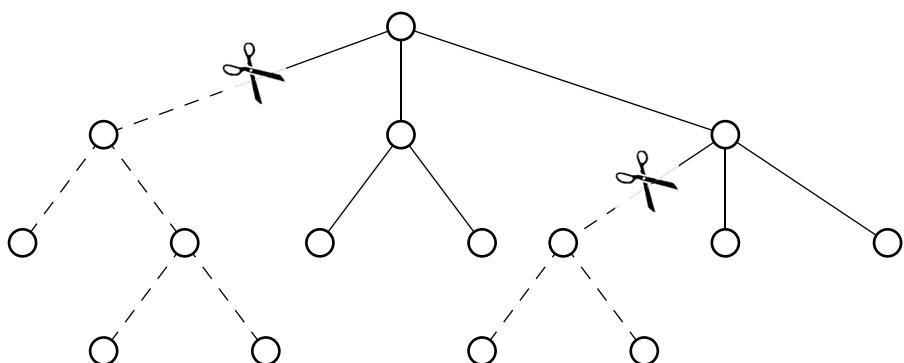


Figure 13.2 Pruning the search tree by eliminating classes of design that are unfeasible.

hope to generate the concerto, as he or she is able to prune the search space of compositions.

Even if we succeed in pruning the search space so that only feasible designs are considered, we will still be left with the problem of selecting between alternatives. The selection problem is discussed in Section 13.8 with particular reference to materials selection for design. The same techniques can be applied to selection between design alternatives.

Although heuristic rules can limit the search space, they do not offer unique solutions. This shortfall is because abductive rather than deductive rules are required (Chapter 1), as with diagnosis (Chapter 12). Consider this simple deductive rule in Flex format:

```
rule r13_1
  if Room includes bath
  and Room includes toilet
  then Room becomes a bathroom.
```

If a room fits the description provided by the condition part of the rule, we could use the rule to classify that room as a bathroom. The abductive interpretation of this rule is as follows:

```
rule r13_2
  if Room is a bathroom
  then bath of Room becomes true
  and toilet of Room becomes true.
```

The abductive rule states that, for a room to be a bathroom, it must have a bath and a toilet. The abductive rule poses two problems. First, we have made the closed-world assumption (see Chapters 1 and 2), and so the rule will never produce bathroom designs that have a shower and toilet but no bath. Second, the rule leads only to a partial design. It tells us that the bathroom will have a toilet and a bath, but it fails to tell us where these items should be placed or whether we need to add other items such as a basin.

13.3 Computer-Aided Design

The expression *computer-aided design*, or CAD, is used to describe computing tools that can assist in the design process. Most early CAD systems were intended primarily to assist in drawing a design, rather than directly supporting the decision-making process. CAD systems of this type carry out computer-aided *drafting* rather than computer-aided *design*. Typically, these drafting systems allow the designer to draw using a computer. All dimensions are automatically calculated, and the design can be easily reshaped, resized, or otherwise modified. Such systems have had an enormous impact on the design process since they remove much of the tedium and facilitate alterations.

Furthermore, CAD has altered the designers' working environment, as the traditional large flat drafting boards have been replaced by computer workstations.

Early CAD systems of this type do not make decisions and have little built-in intelligence. They do, however, make use of object-oriented programming techniques. Each line, box, circle, etc., that is created can be represented as an object instance. Rather than describing such systems in more detail, this chapter will concentrate on the use of intelligent systems that can help designers make design decisions. Some systems aim to integrate the computer-aided drafting elements with knowledge-based design (Kavakli 2001).

13.4 The Product Design Specification (PDS): A Telecommunications Case Study

13.4.1 Background

The PDS is a statement of the requirements of the product. In this section, we will consider a case study concerning the problems of creating a PDS that can be accessed by an intelligent design system. In this case study, the "product" is not a material product but a service, namely, the provision of a communications network. The model used to represent the PDS is called the *common traffic model* (Hopgood and Hopson 1991) because it is common to a variety of forms of communication traffic, including voice, packetized data, and synchronous data.

The common traffic model allows different views of a communications network to be represented simultaneously. The simplest view is a set of requirements defined in terms of links between sites and the applications to be used on these links, such as access to a website or database. The more specialized views contain implementation details, including the associated costs. The model allows nontechnical users to specify a set of communications requirements, from which an intelligent system can design and cost a network, thereby creating a specialized view from a nonspecialized one. The model consists of object class definitions, and a PDS is represented as a set of instances of those classes.

13.4.2 Alternative Views of a Network

Suppose that a small retailer has a central headquarters, a warehouse, and a retail store. The retailer may require various communications applications, including a customer order via the Web, a customer order by phone, and a reorder for replacement stock from its suppliers. The retailer views the network in terms of the sites and the telecommunications applications that are carried between them. This viewpoint is the simplest one, and it defines the PDS. From a more technical viewpoint, the network can be broken down into voice and data components. For the voice section, each site has a fixed number of lines connecting it to the network via a private

switching system, while the data section connects the head office to the other sites. The most detailed view of the network (the service-provider's viewpoint) includes a definition of the equipment and services used to implement the network. The detailed description is based on one of several possible implementations, while the less specialized views are valid regardless of the implementation.

There are several possible views of the network, all of which are valid and can be represented by the common traffic model. It is the translation from the customer's view (defined in terms of the applications being used) to the service-provider's view (defined in terms of the equipment and services supplied) that determines the cost and efficiency of the communications network. This translation is the design task.

13.4.3 Implementation

The requirements of the network are represented as a set of object instances. For example, if the customer of the telecommunications company has an office in New York, that office is represented as an object. It has a name and position, and it is an instance of the object class `Customer_site`.

The common traffic model was originally designed using Coad and Yourdon's (1990) object-oriented analysis (OOA), but it is redrawn in Figure 13.3 using the Unified Modeling Language (UML) introduced in Chapter 4. The model is implemented as a set of object classes that act as templates for the object instances that are created when the system is used to represent a PDS. Various interclass relationships are employed. For example, a `Dispersion_link` is represented as a specialization of a `Link`. Similarly, an aggregation relationship is used to show that a `Network` comprises several instances of `Link`. Associations are used to represent physical connections, such as the connection between a `Link` and the instances of `Site` at its two ends.

The fact that instance connections are defined at the class level can be confusing. The common traffic model is defined entirely in terms of object classes, these being the templates for the instances that represent the user's communication needs. Although the common traffic model is only defined in terms of classes, it specifies the relationships that exist between instances when they are created.

13.4.4 The Classes

The classes that make up the common traffic model and the relationships between them are shown in Figure 13.3. A detailed understanding of Figure 13.3 is not necessary for this case study. Instead, it is hoped that the figure conveys the general idea of using OOA for generating both the PDS and a detailed network description. The main classes of the common traffic model are briefly described in the following subsections.

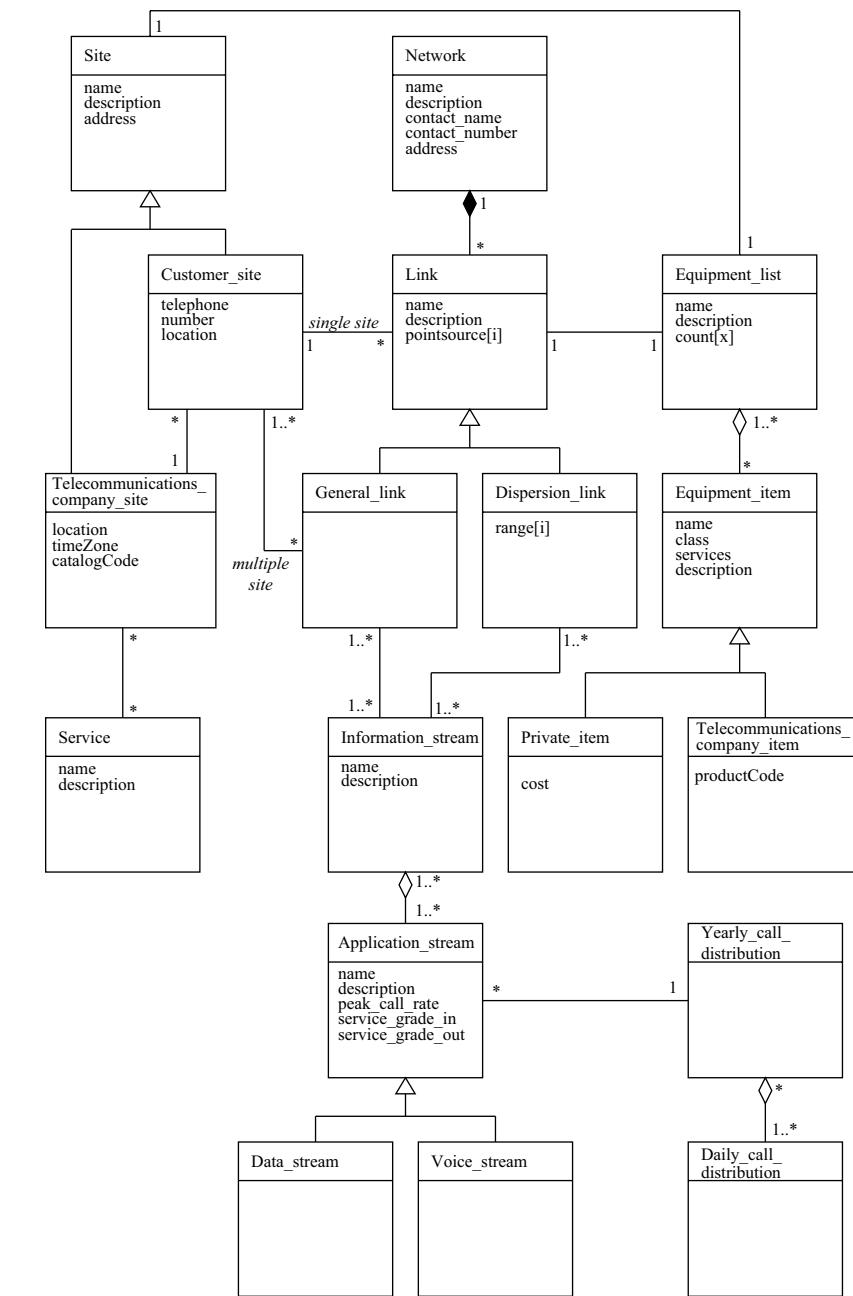


Figure 13.3 The main object classes and attributes in the common traffic model.
(Derived from Hopgood, A. A., and A. J. Hopson. 1991.)

13.4.4.1 Network

The `Network` object contains the general information relating to the network but is independent of the network requirements. It includes information such as contact people and their addresses. The specification of the network is constructed from a set of `Link` objects, described in the next subsection.

13.4.4.2 Link

A `Link` identifies the path between customer sites along which an information stream (described in the next subsection) occurs. Instance connections are used to associate links with appropriate customer sites, information streams, and equipment. Conceptually three link types are defined:

- *Multipoint links*, where information is exchanged between a single nominated site and a number of other sites. The links are instances of the class `General_link`, where an attribute (`pointsource`) indicates whether calls are initiated by the single site or by the multiple sites.
- *Point-to-point links*, which are treated as multipoint links, but where only one of the multiple sites is specified.
- *Dispersion links*, which carry application traffic that does not have a fixed destination site. This type of link applies to customers who want access to a public switched network.

13.4.4.3 Information Stream

The information streams specify the traffic on a link in terms of a set of application streams. Two subclasses of `Application_stream` are defined, `Data_stream` and `Voice_stream`. The first specifies digital applications, while the second specifies analog applications. Each application stream has a peak call rate and associated yearly and daily traffic profiles. Application streams can be broken down further into individual calls.

13.4.4.4 Site

Two classifications of sites are defined, namely, the customer's sites and the telecommunications company's sites. The latter specify sites that are part of the supplier's network, such as telephone exchanges. For most telecommunications services, the design and costing of a network is dependent on its spatial layout. For this reason, the common traffic model has access to a geographical database.

13.4.4.5 Equipment

An equipment list specifies a set of items that are present at a site or on a link. Two subclasses of equipment items are defined: those that are owned by the telecommunications company and those that are privately owned.

13.4.5 Summary of PDS Case Study

The common traffic model illustrates a formalized approach to creating a PDS, showing that the PDS and its implementation need to be carefully thought out before an intelligent design system can be employed. The common traffic model has proved an effective tool for representing a set of communication requirements in a way that satisfies more than one viewpoint. Nontechnical users can specify the PDS in terms of the types of use that they have in mind for the network. The common traffic model can also be used to represent the detailed network design, which may be one of many that are technically possible.

13.5 Conceptual Design

It has already been noted in Section 13.1 that conceptual design is the stage where broad decisions about the overall form of a product are made. A distinction can be drawn between cases where the designer is free to innovate and more routine cases where the designer is working within tightly bound constraints. An example of the former case would be the design of a can opener. Many designs have appeared in the past and the designer may call upon his or her experience of these. However, he or she is not bound by those earlier design decisions. In contrast, a designer might be tasked with arranging the layout of an electronic circuit on a VLSI (very large-scale integration) chip. While this task is undoubtedly complex, the conceptual design has already been carried out, and the designer's task is one that can be treated as a problem of mathematical optimization. We will call this *routine design*.

Brown and Chandrasekaran (1985) subdivide the innovative design category between *inventions* (such as the first helicopter) and more modest *innovations* (such as the first twin-rotor helicopter). Both are characterized by the lack of any prescribed strategy for design, and they rely on a spark of inspiration. The invention category makes use of new knowledge, whereas the innovation category involves the reworking of existing knowledge or existing designs. The three categories of design can be summarized as follows:

- Invention
- Innovative use of existing knowledge or designs
- Routine design

Researchers have different opinions of how designers work, and it is not surprising that markedly different software architectures have been produced. For instance, Sriram et al. (1989) claim to have based their CYCLOPS system on the following set of observations about innovative design:

1. Designers use multiple objectives and constraints to guide their decisions, but are not necessarily bound by them;
2. As new design criteria emerge, they are fed back into the PDS;
3. Designers try to find an optimum solution rather than settling on a satisfactory one;
4. Extensive use is made of past examples.

Demaid and Zucker (1988) do not dispute observations 1, 2, and 4. However, in contrast to observation 3, they emphasize the importance of choosing *adequate* materials for a product rather than trying to find an *optimum* choice.

The CYCLOPS (Sriram et al. 1989) and FORLOG (Dietterich and Ullman 1987) systems assume that innovative design can be obtained by generating a variety of alternatives and choosing between them. CYCLOPS makes use of previous design histories and attempts to adapt them to new domains. The success of this approach depends upon the ability to find diverse novel alternatives. In order to increase the number of past designs that might be considered, the design constraints are relaxed. Relaxation of constraints is discussed in Section 13.8.5 as part of an overall discussion of techniques for selecting between alternatives. CYCLOPS also has provision for modification of the constraints in the light of past experience.

As well as selecting a preexisting design for use in a novel way, CYCLOPS allows adaptation of the design to the new circumstances. This adaptation is achieved through having a stored explanation of the precedent designs. The example cited by Sriram et al. (1989) relates to houses in Thailand. Thai villagers put their houses on stilts to avoid flooding, and this forms a precedent design. The underlying explanation for the design, which is stored with it, is that stilts raise the structure. The association with flooding may not be stored at all, as it is not fundamental to the role of the stilts. CYCLOPS might then use this precedent to raise one end of a house that is being designed for construction on a slope.

Most work in intelligent systems for design relies on the application of a predetermined strategy. Dyer et al. (1986) see this approach as a limitation on innovation and have incorporated the idea of *brainstorming* into EDISON, a system for designing simple mechanical devices. Some of the key features of EDISON are:

- Brainstorming, by mutation, generalization, and analogy
- Problem-solving heuristics
- Class hierarchies of mechanical parts
- Heuristics describing relationships between mechanical parts

EDISON makes use of metarules (see Section 2.8.3) to steer the design process between the various strategies that are provided. Brainstorming and problem-solving often work in tandem, as brainstorming tends to generate new problems. Brainstorming involves retrieving a previous design from memory and applying *mutation*, *generalization*, and *analogical reasoning* until a new functioning device is “invented.” *Mutation* is achieved through a set of heuristics describing general modifications that can be applied to a variety of products. For example, slicing a door creates two slabs, each covering half a door frame. This operation results in a problem: the second slab is not connected to the frame. Two typical problem-solving heuristics might be:

- Hinged joints allow rotation about pin
- Hinged joints prohibit motion in any other planes

These heuristic rules provide information about the properties of hinges. Application of similar *problem-solving* rules might result in the free slab being connected either to the hinged slab or to the opposite side of the frame. In one case we have invented the swinging saloon door; in the other case the accordion door (Figure 13.4).

Generalization is the process of forming a generic description from a specific item. For instance, a door might be considered a subclass of the general class of entrances (Figure 13.5). Analogies can then be drawn (*analogical reasoning*) with another class of entrance, namely, a cat flap, leading to the invention of a door that hangs from hinges mounted at the top. Generalization achieves the same goal as the deep explanations used in the adaptation mode of CYCLOPS, described earlier.

Murthy and Addanki (1987) have built a system called PROMPT in which innovative structural designs are generated by reasoning from first principles, that is, using the fundamental laws of physics. Fundamental laws can lead to

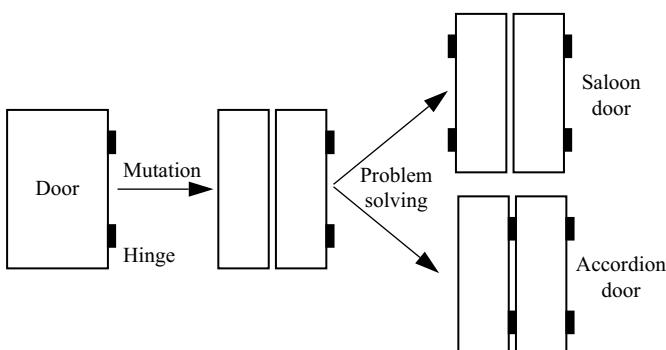


Figure 13.4 Inventing new types of doors by mutation and problem-solving. (Derived from Dyer, M. G. et al. 1986.)

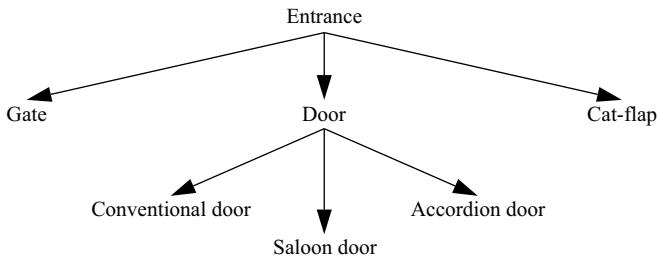


Figure 13.5 Hierarchical classification of types of entrances.

unconventional designs that heuristics based on conventional wisdom might have failed to generate. Other authors (Lirov 1990; Howe et al. 1986) have proposed a *systematic* approach to innovation that generates only feasible solutions, rather than large numbers of solutions from which the feasible ones must be extracted. In this approach, the goals are first determined and then the steps needed to satisfy those goals are found. These steps have their own subgoals, and so the process proceeds recursively.

13.6 Constraint Propagation and Truth Maintenance

The terms *constraint propagation* and *truth maintenance* are commonly used in the field of artificial intelligence to convey two separate but related ideas. They have particular relevance to design, as will be illustrated by means of some simple examples. Constraints are limitations or requirements that must be met when producing a solution to a problem (such as finding a viable design). Imagine that we are designing a product, and that we have already made some conceptual design decisions. Propagation of constraints refers to the problem of ensuring that new constraints arising from the decisions made so far are taken into account in any subsequent decisions. For instance, a decision to manufacture a car from steel rather than fiberglass introduces a constraint on the design of the suspension, namely, that it must be capable of supporting the mass of the steel body.

Suppose that we wish to investigate two candidate solutions to a problem, such as a steel-bodied car and a fiberglass car. Truth maintenance refers to the problem of ensuring that more detailed investigations, carried out subsequently, are associated with the correct premise. For example, steps must be taken to ensure that a lightweight suspension design is associated only with the lightweight (fiberglass) car design to which it is suited.

In order to illustrate these ideas in more detail, we have adapted the example provided by Dietterich and Ullman (1987). The problem is to place two batteries into a battery holder. There are four possible ways in which the batteries can be inserted, as shown in Figure 13.6. This situation is described by the following

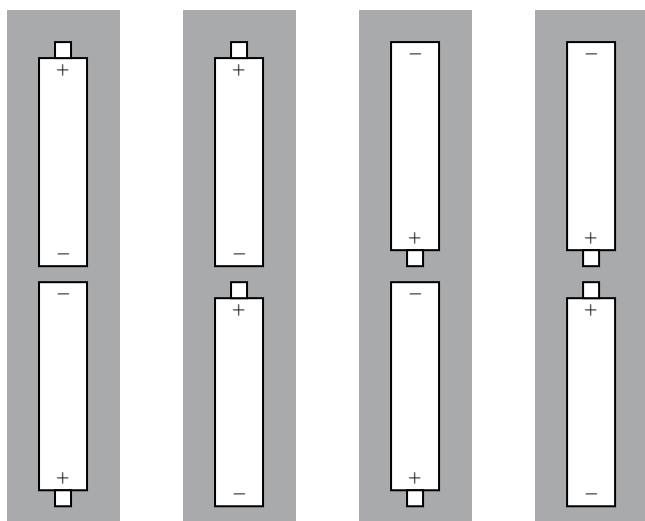


Figure 13.6 Four possible ways of inserting batteries into a holder.

Prolog clauses (Section 11.4 includes an overview of the syntax and workings of Prolog):

```
terminal(X) :- X=positive; X=negative.  
% battery terminal may be positive or negative
```

```
layout(T,B) :- terminal(T), terminal(B).  
% layout defined by top and bottom terminals
```

We can now query our Prolog system so that it will return all valid arrangements of the batteries:

```
?- layout(Top, Bottom).  
Top = Bottom = positive;  
Top = positive, Bottom = negative;  
Top = negative, Bottom = positive;  
Top = Bottom = negative;  
no
```

Now let us introduce the constraint that the batteries must be arranged in series. This constraint is achieved by adding a clause to specify that terminals at the top and bottom of the battery holder must be of opposite sign:

```
terminal(X) :- X=positive; X=negative.  
  
layout(T,B) :- terminal(T), terminal(B),  
not(T=B). % top terminal not equal to bottom terminal
```

We can now query our Prolog system again:

```
?- layout(Top,Bottom).
Top = positive, Bottom = negative;
Top = negative, Bottom = positive;
no
```

We will now introduce another constraint, namely, that a positive terminal must appear at the top of the battery holder:

```
terminal(X) :- X=positive; X=negative.

layout(T,B) :- terminal(T), terminal(B),
not(T=B),
T=positive. % positive terminal at top of holder
```

There is now only one arrangement of the batteries that meets the constraints:

```
?- layout(Top,Bottom).
Top = positive, Bottom = negative;
no
```

This example illustrates constraint propagation because it shows how a constraint affecting one part of the design (i.e., the orientation of the battery at the top of the holder) is propagated to determine some other part of the design (i.e., the orientation of the other battery). In this particular example, constraint propagation has been handled by the standard facilities of the Prolog language. Many researchers, including Dietterich and Ullman (1987), have found the need to devise their own means of constraint propagation in large design systems.

Truth maintenance becomes an important problem if we wish to consider more than one solution to a problem at a time, or to make use of nonmonotonic logic (see Section 12.2.3). For instance, we might wish to develop several alternative designs, or to assume that a particular design is feasible until it is shown to be otherwise. In order to illustrate the concept of truth maintenance, we will stay with our example of arranging batteries in a holder. However, we will veer away from a Prolog representation of the problem, as standard Prolog can consider only one solution to a problem at a time.

Let us return to the case where we had specified that the two batteries must be in series, but we had not specified an orientation for either. There were, therefore, two possible arrangements:

[Top = positive, Bottom = negative]

and

[Top = negative, Bottom = positive]

It is not sufficient to simply store the following four assertions together in memory:

```
Top = positive.  
Bottom = negative.  
Top = negative.  
Bottom = positive.
```

For these four statements to exist concurrently, it would be concluded that the two terminals of a battery are identical (i.e., negative = positive). This conclusion is clearly not the intended meaning. A frequently used solution to this difficulty is to label each fact, rule, or assertion, such that those bearing the same label are recognized as interdependent and, therefore, “belonging together.” These labeled linkages are the basis of DeKleer’s (1986a,b,c) assumption-based truth maintenance system (ATMS). If we choose to label our two solutions as `design1` and `design2`, then our four assertions might be stored as:

```
% Not Prolog code  
Top = positive {design1}  
Bottom = negative {design1}  
Top = negative {design2}  
Bottom = positive {design2}
```

Let us now make explicit the rule that the two terminals of a battery are different:

```
not (positive = negative) {global}
```

The English translation for these labels would be “if you believe the `global` assumptions, then you must believe `not (positive = negative)`.” Similarly, for `design1`, “if you believe `design1`, then you must also believe `Top = negative` and `Bottom = positive`.” Any deductions made by the inference engine should be appropriately labeled. For instance, the following deduction is compatible with the sets of beliefs defined by `design1` and `design2`:

```
negative = positive. {design1, design2}
```

However, this deduction is incompatible with our global rule, and so a warning should be produced of the form:

```
INCOMPATIBLE.{design1, design2, global}
```

This warning tells us that we cannot believe `design1`, `design2`, and `global` simultaneously. It is, however, alright to believe (`design1` and `global`) or (`design2` and `global`). This behavior is what we want, as there are two separate designs, and the inference engine has simply discovered that the two designs cannot be combined together.

13.7 Case Study: The Design of a Lightweight Beam

13.7.1 Conceptual Design

To illustrate some of the ideas behind the application of intelligent systems to conceptual design, we will consider the design of a lightweight beam. The beam is intended to support a passenger seat in a commercial aircraft. The whole aircraft will have been designed, and we are concerned with the design of one component of the whole assembly. The total design process for the beam is part of the detailed design process for the aircraft. The intended loading of the beam tends to cause it to bend, as shown in Figure 13.7. The objectives are for the beam to be:

- stiff enough that the deflection D is kept small;
- strong enough to support the load without fracture; and
- as light as possible, so as to maximize the ratio of cargo weight to fuel consumption.

Together, these three objectives form the basis of the PDS. The PDS can be made more specific by placing limits on the acceptable deflection (D) under the maximum design load (F). A limit could also be placed on the mass of the beam. However, a suitable mass limit is difficult to judge, as it presupposes the form of the beam (that is, its conceptual design) and the materials used. For this reason,

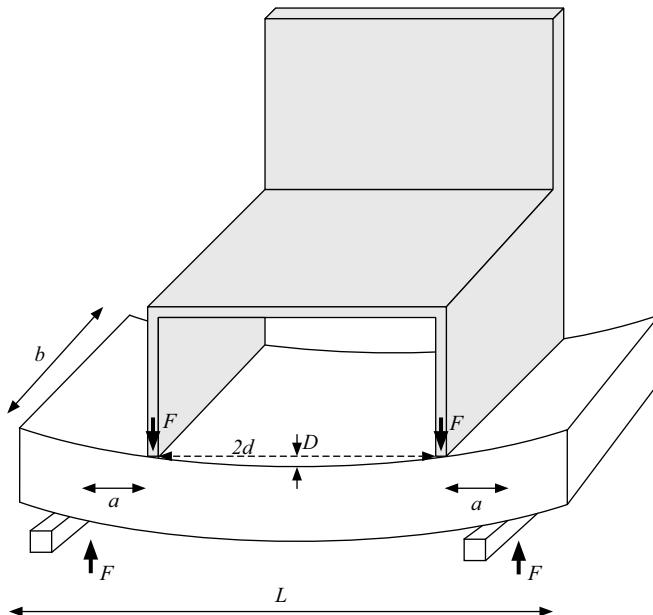


Figure 13.7 Four-point loading of a beam supporting a chair.

we will simply state that the beam is required to be as light as possible within the constraints of fulfilling the other two requirements. In practice, a number of additional constraints will apply, such as materials costs, manufacturing costs, and flammability.

Kim and Suh (1989) propose that the design process in general can be based upon two axioms, which can be implemented as metarules (see Section 2.8.3):

- Axiom 1: Maintain the independence of the functional requirements
- Axiom 2: Minimize the information content

Our statement of the PDS fulfills these two axioms, because we have identified three concise and independent requirements.

Many knowledge-based systems for conceptual design attempt to make use of past designs (e.g., CYCLOPS, mentioned in Section 13.5), as indeed do human designers. Some past designs that are relevant to designing the beam are shown in Figure 13.8. These are:

- I-beams used in the construction of buildings;
- box girder bridges; and
- sandwich structures used in aircraft wings.

All three structures have been designed to resist bending when loaded. For this knowledge to be useful, it must be accompanied by an explanation of the underlying principles of these designs, as well as their function. The principle underlying all three designs is that strength and stiffness are mainly provided by the top and bottom surfaces, while the remaining material keeps the two surfaces apart. The heaviest parts of the beam are, therefore, concentrated at the surfaces, where they are most effective. This explanation could be expressed as a rule, or perhaps by hierarchical classification of the structural objects that share this property (Figure 13.9). A set of conceptual design rules might seize upon the `Beam` class as being appropriate for the current application because beams maximize both the *stiffness/mass* ratio and the *strength/mass* ratio in bending.

At this stage in the design procedure, three markedly different conceptual designs have been found that fulfill the requirements as determined so far. A key difference between the alternatives is *shape*. So, a knowledge-based system for conceptual design might seek information about the shape requirements of the beam. If the beam needs both to support the seats and to act as a floor that passengers can walk on, it should be flat and able to fulfill the design requirements over a large area. Adding this criterion leaves only one suitable conceptual design, namely the sandwich beam. If the human who is interacting with the system is happy with this decision, the new application can be added to the `applications` attribute of the `Sandwich_beam` class so that this experience will be available in future designs (Figure 13.9).

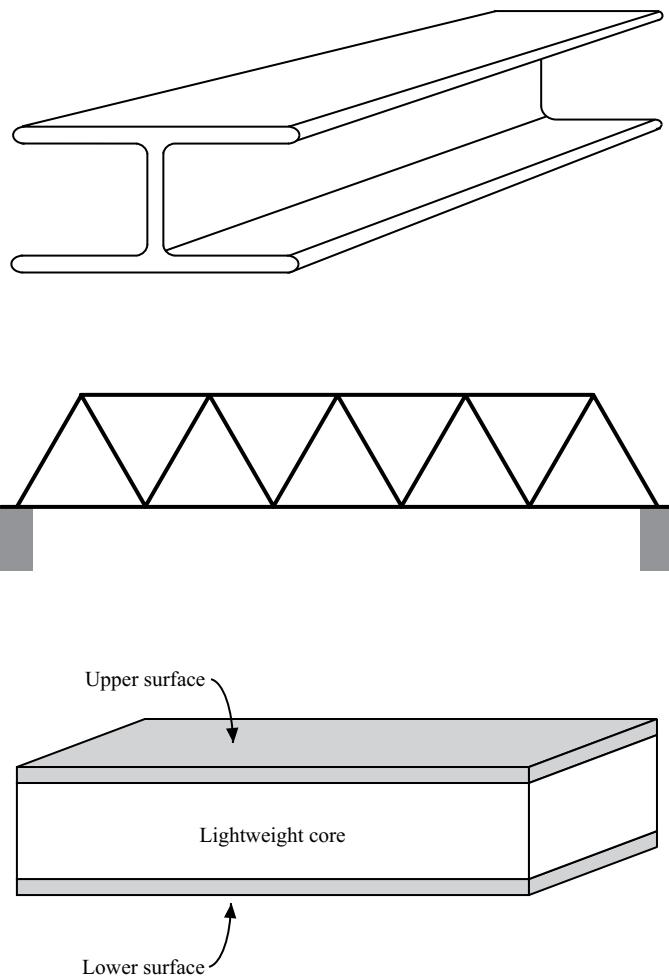


Figure 13.8 Some alternative conceptual designs for load-bearing beams.

13.7.2 Optimization and Evaluation

The optimization and evaluation stage of the design process involves performing calculations to optimize performance and to check whether specifications are being met. As this phase is primarily concerned with numerical problems, the tasks are mainly handled using computational intelligence or procedural programs. However, the numerical processes can be made more effective and efficient by the application of rules to steer the analysis. For instance, a design system may have access to a library of optimization procedures, the most appropriate for a specific task being chosen by a rule-based selection module.

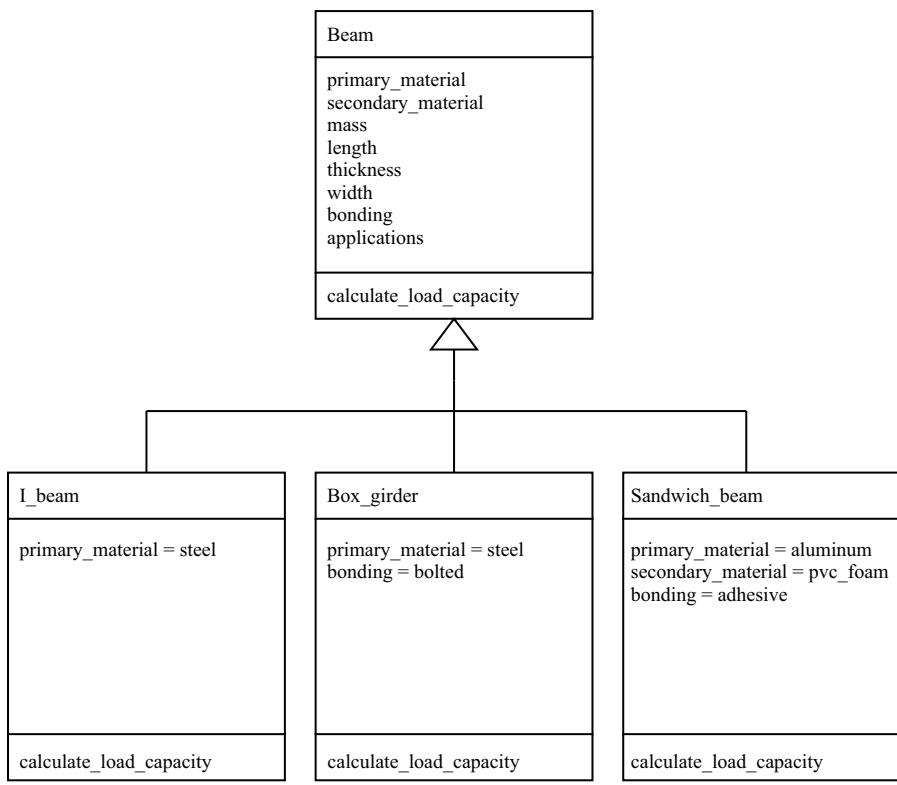


Figure 13.9 Hierarchical classification of beams.

Three important forms of numerical analysis are as follows:

- mathematical optimization (including computational intelligence techniques);
- finite-element analysis; and
- specialized modeling.

Several techniques for mathematical optimization were described in Chapters 6 and 7, including hill-climbing, simulated annealing, and genetic algorithms. *Finite-element analysis* is a general technique for modeling complex shapes. In order to analyze the performance of a three-dimensional physical product, a technique has to be devised for representing the product numerically within the computer. For regular geometric shapes, such as a cube or sphere, this task poses no great problem. But the shape of real products, such as a saucepan handle or a gas turbine blade, can be considerably more complex. Since the shape of an object is defined by its surfaces, or *boundaries*, the analysis of performance (for example, the flow of air over a turbine blade) falls into the class of *boundary-value problems*. Finite-element analysis provides a powerful technique for obtaining approximate solutions to such problems.

The technique is based on the concept of breaking up an arbitrarily complex surface or volume into a network of simple interlocking shapes. The performance of the whole product is then taken to be the sum of each constituent part's performance. There are many published texts that give a full treatment of finite-element analysis (e.g., Komzsik 2009; Kim and Sankar 2009; Cook et al. 2001).

Mathematical optimization or finite-element analysis might be used in their own right or as subtasks within a customized model. If equations can be derived that describe the performance of some aspects of the product under design, then it is sensible to make use of them. The rest of this section will, therefore, concentrate on the modeling of a physical system, with particular reference to the design of a sandwich beam.

In the case of the sandwich beam, expressions can be derived that relate the minimum mass of a beam that meets the stiffness and strength requirements to its dimensions and material properties. Mass, stiffness, and strength are examples of *performance variables*, as they quantify the performance of the final product. The thicknesses of the layers of the sandwich beam are *decision variables*, as the designer must choose values for them in order to achieve the required performance. Considering first the stiffness requirement, it can be shown (Reid and Greenberg 1980; Greenberg and Reid 1981) that the mass of a beam that just meets the stiffness requirement is given by:

$$M \approx bL \left(\frac{2\rho_s f F a d^2}{D E_s b t_c^2} + \rho_c t_c \right) \quad (13.1)$$

where:

M = mass of beam;

b, L, a, d = dimensions defined in Figure 13.7;

F = applied load;

f = safety factor ($f = 1.0$ for no margin of safety);

t_s, ρ_s, E_s = thickness, density, and Young's modulus of surface material; and

t_c, ρ_c, E_c = thickness, density, and Young's modulus of core material.

Equation 13.1 is written in terms of the core thickness t_c . For each value of core thickness, there is a corresponding surface thickness t_s that is required in order to fulfill the stiffness requirement:

$$t_s \approx \frac{f F a d^2}{D E_s b t_c^2} \quad (13.2)$$

Thus, given a choice of materials, the plan-view dimensions (b, L, a , and d), and the maximum deflection D under load F , there is a unique pair of values of t_c and t_s that correspond to the minimum-mass beam that meets the requirement. If this requirement were the only one, the analysis would be complete. However, as well as being sufficiently stiff, the beam must be sufficiently strong, so that it does not

break under the design load. A new pair of equations can be derived that describe the strength requirement:

$$M \approx bL \left(\frac{2\rho_s fFa}{\sigma_f bt_c} + \rho_c t_c \right) \quad (13.3)$$

$$t_c \approx \frac{fFa}{\sigma_f bt_c} \quad (13.4)$$

where σ_f is the failure stress of the surface material.

Assuming a choice of core and surface materials and given the plan-view dimensions and loading conditions, Equations 13.1 and 13.3 can be plotted to show the mass as a function of core thickness, as shown in Figure 13.10. The position of the two curves in relation to each other depends upon the materials chosen. It should be noted that the minimum mass to fulfill the stiffness requirement may be insufficient to fulfill the strength requirement, or vice versa.

There are still two other complications to consider before the analysis of the beam is complete. First, the core material must not fail in shear. In order to achieve this goal, the following condition must be satisfied:

$$t_c \geq \frac{3fF}{2b\tau_c} \quad (13.5)$$

where τ_c is the critical shear stress for failure of the core material.

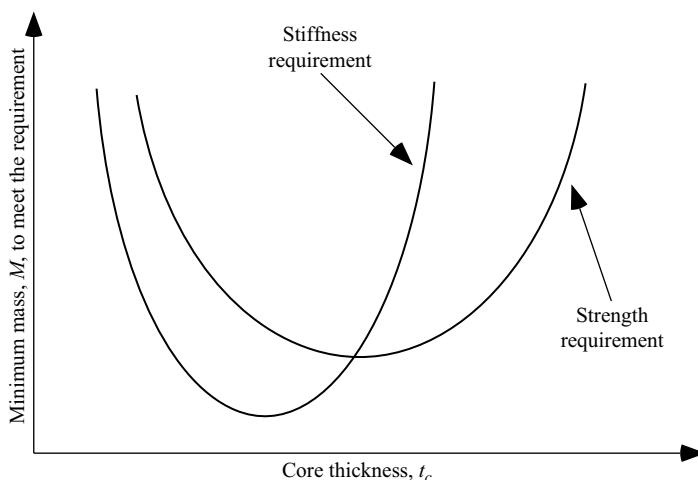


Figure 13.10 Mass of a sandwich beam that just meets some stiffness and strength requirements.

Second, the upper surface, which is in compression, must not buckle. This condition is described by the following equation:

$$t_s \geq \frac{2fFa}{bt_c(E_s E_c G_c)^{1/3}} \quad (13.6)$$

where G_c is the shear modulus of the core material.

Armed with these numerical models, reasonable choices of layer thicknesses can be made. Without such models, a sensible choice would be fortuitous.

13.7.3 Detailed Design

The detailed design phase allows the general view provided by the conceptual design phase to be refined. The optimization and evaluation phase provides the information needed to make these detailed design decisions. The decisions taken at this stage are unlikely to be innovative, as the design is constrained by decisions made during the conceptual design phase. In the case of the sandwich beam, choices need to be made regarding the following:

- Core material
- Upper surface material
- Lower surface material
- Core thickness
- Upper surface thickness
- Lower surface thickness
- Method of joining the surfaces to the core

There is clearly a strong interaction among these decisions. There is also an interaction with the optimization and evaluation process, as Equations 13.1–13.6 need to be reevaluated for each combination of materials considered. The decisions also need to take account of any assumptions or approximations that might be implicit in the analysis. For instance, Equations 13.1–13.4 were derived under the assumption that the top and bottom surfaces were made from identical materials and each had the same thickness.

13.8 Design as a Selection Exercise

13.8.1 Overview

The crux of both conceptual and detailed design is the problem of selection. Some of the techniques available for making selection decisions are described in the following sections. In the case of a sandwich beam, the selection of the materials and

glue involves making a choice from a very large but finite number of alternatives. Thickness, on the other hand, is a continuous variable, and it is tempting to think that the “right” choice is yielded directly by the analysis phase. However, this is rarely the case. The requirements on, say, core thickness will be different depending on whether we are considering stiffness, surface strength, or core shear strength. The actual chosen thickness has to be a compromise. Furthermore, although thickness is a continuous variable, the designer may be constrained by the particular set of thicknesses that a supplier is willing to provide.

This section will focus on the use of scoring techniques for materials selection, although neural network approaches can offer a viable alternative (Cherian et al. 2000). The scoring techniques are based on awarding candidate materials a score for their performances with respect to the requirements, and then selecting the highest-scoring materials. We will start by showing a naive attempt at combining materials properties to reach an overall decision, before considering a more successful algorithm called AIM (Hopgood 1989). AIM will be illustrated by considering the selection of a polymer for the manufacture of a kettle.

For the purposes of this discussion, selection will be restricted to polymer materials. The full range of materials available to designers covers metals, composites, ceramics, and polymers. As each of these categories is vast, restricting the selection to polymers still leaves us with a very complex design decision.

13.8.2 Merit Indices

The analysis of the sandwich beam yielded expressions for the mass of a beam that just meets the requirements. These expressions contained geometrical measurements and physical properties of the materials. Examination of Equation 13.1 shows that the lightest beam that meets the stiffness requirement will have a low-density core (ρ_c), while the surface material will have a low density (ρ_s) and a high Young’s modulus (E_s). However, this observation would not be sufficient to enable a choice between two materials where the first had a high value of E_s and ρ_s , and the second had a low value for each. Merit indices can help such decisions by enabling materials to be ranked according to *combinations* of properties. For instance, a merit index for the surface material of a sandwich beam would be E_s/ρ_s . This is because Equation 13.1 reveals that the important combination of properties for the surface material is the ratio ρ_s/E_s . As the latter ratio is to be minimized, while merit indices are normally taken to be a quantity that is to be maximized, the merit index is the reciprocal of this ratio. By considering Equations 13.1–13.6, we can derive the merit indices shown in Table 13.1.

Merit indices can be calculated for each candidate material. Tables can then be drawn up for each merit index, showing the rank order of the materials. Thus, merit indices go some way toward the problem of materials selection based on a combination of properties. However, if more than one merit index needs to be considered

Table 13.1 Merit Indices for a Sandwich Beam

<i>Minimum Weight for Given:</i>	<i>Merit Index for Surface Material</i>	<i>Merit Index for Core Material</i>
Stiffness	$\frac{E_s}{\rho_s}$	$\frac{1}{\rho_c}$
Strength	$\frac{\sigma_f}{\rho_s}$	$\frac{\tau_c}{\rho_c}$
Buckling resistance	$\frac{E_s^{1/3}}{\rho_s}$	$\frac{(E_c G_c)^{1/3}}{\rho_c}$

(as with the sandwich beam), the problem is not completely solved. Materials that perform well with respect to one merit index may not perform so well with another. The designer then faces the problem of finding the materials that offer the best compromise. The scoring techniques described in Sections 13.8.6 and 13.8.7 address this problem. Merit indices for the minimum-mass design of a range of mechanical structures are shown in Figure 13.11.

13.8.3 The Polymer Selection Example

With the huge number of polymers available, a human designer is unlikely to have sufficient knowledge to make the most appropriate choice of polymer for a specific application. The published data are often unreliable, and they may be produced by manufacturers with a vested interest in promoting their own products. Even when adequate data are available, the problem of applying them to the product design is likely to remain intractable unless the designer is an expert in polymer technology or has online assistance. The selection system described here is intended to help the designer by making the best use of the available polymer data. The quality of the recommendations made will be limited by the accuracy and completeness of these data. The use of a computerized material-selection system has the spin-off advantage of encouraging designers to consider and analyze their requirements of a material.

13.8.4 Two-Stage Selection

The selection system in this example is based on the idea of ranking a shortlist of polymers by comparing their relative performance against a set of materials properties. The length of the shortlist can be reduced by the prior application of numerical specifications, such as a minimum acceptable impact strength. The selection process then comprises two stages, *screening* and *ranking* (Jahan et al. 2010), as shown in Figure 13.12. First, any polymers that fail to meet the user's numerical specifications

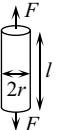
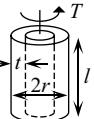
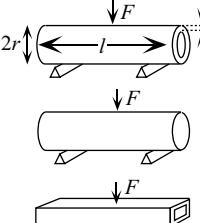
Mode of loading	Minimize mass for given:		
	Stiffness	Ductile strength	
Tie F, l specified r free		$\frac{E}{\rho}$	$\frac{\sigma_y}{\rho}$
Torsion bar T, l specified r free		$\frac{G}{\rho}$	$\frac{\sigma_y}{\rho}$
Torsion tube T, l, r specified t free		$\frac{G}{\rho}$	$\frac{\sigma_y}{\rho}$
Bending of rods and tubes F, l specified r or t free		$\frac{E^{1/2}}{\rho}$	$\frac{\sigma_y^{2/3}}{\rho}$

Figure 13.11 Merit indices for minimum-mass design (Ashby 1989). E = Young's modulus; G = shear modulus; ρ = density; σ_y = yield stress. (Reprinted from Ashby, M. F. 1989. On the engineering properties of materials. *Acta Metallurgica* 37:1273–1293. Copyright (1989). With permission from Elsevier Science.)

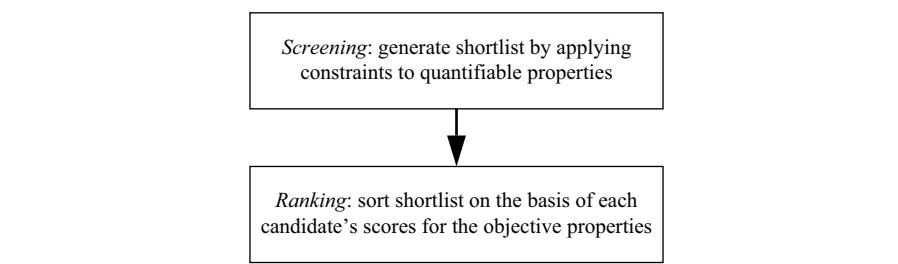


Figure 13.12 Two-stage selection.

are eliminated. These specifications are *constraints* on the materials, and they can be used to limit the number of candidate polymers. Constraints of this sort are sometimes described as *primary constraints*, indicating that they are nonnegotiable. A facility to alter the specifications helps the user of a selection system to assess the sensitivity of the system to changes in the constraints.

Second, the selection process requires the system to weigh the user's *objectives* to arrive at some balanced compromise solutions. The objectives are properties that are to be maximized or minimized so far as possible while satisfying constraints and other objectives. For instance, it may be desirable to maximize impact strength while minimizing cost. Cost is treated as a polymer property in the same way as the other physical properties. Each objective has a user-supplied importance rating associated with it. In the unlikely event that one polymer offers outstanding performance for every material objective, this polymer will appear at the top of the list of recommendations made by the selection system. More typically, the properties being optimized represent conflicting requirements for each polymer. For example, a polymer offering excellent impact resistance may not be easily injection molded. For such problems, there is no single correct answer, but several answers offering different levels of suitability. Objectives may also be known as *preferences* or *secondary constraints*.

13.8.5 Constraint Relaxation

Several authors have stressed the dangers of applying numerical constraints too rigidly and so risking the elimination of candidates that would have been quite suitable (Demaïd and Zucker 1988; Navichandra and Marks 1987; Sriram et al. 1989). This problem can be overcome by relaxing the constraints by some amount (Hopgood 1989; Navichandra and Marks 1987). In Hopgood's system, the amount of constraint relaxation is described as a tolerance, which is specified by the user. Relaxation overcomes the artificial precision that is built into a specification. It could be that it is difficult to provide an accurately specified constraint, the property itself may be ill-defined, or the property definition may only approximate to what we are really after. The application and relaxation of constraints can be illustrated by representing each candidate as a point on a graph where one property is plotted against another. A boundary is drawn between those materials that meet the constraints and those that do not, and relaxation of the constraints corresponds to sliding this boundary (Figure 13.13).

If our specification represents a minimum value that must be attained for a single property, for example, impact resistance must be at least 1 kJ/m, the boundary is moved toward the origin (Figure 13.13a). If one or both specifications are for a maximum value, then the boundary is moved in the opposite direction (Figures 13.13b, c, and d). Figure 13.13e illustrates the case where target specifications are provided, and constraint relaxation corresponds to increasing the tolerance on those specifications. Often the specifications cannot be considered independently, but instead

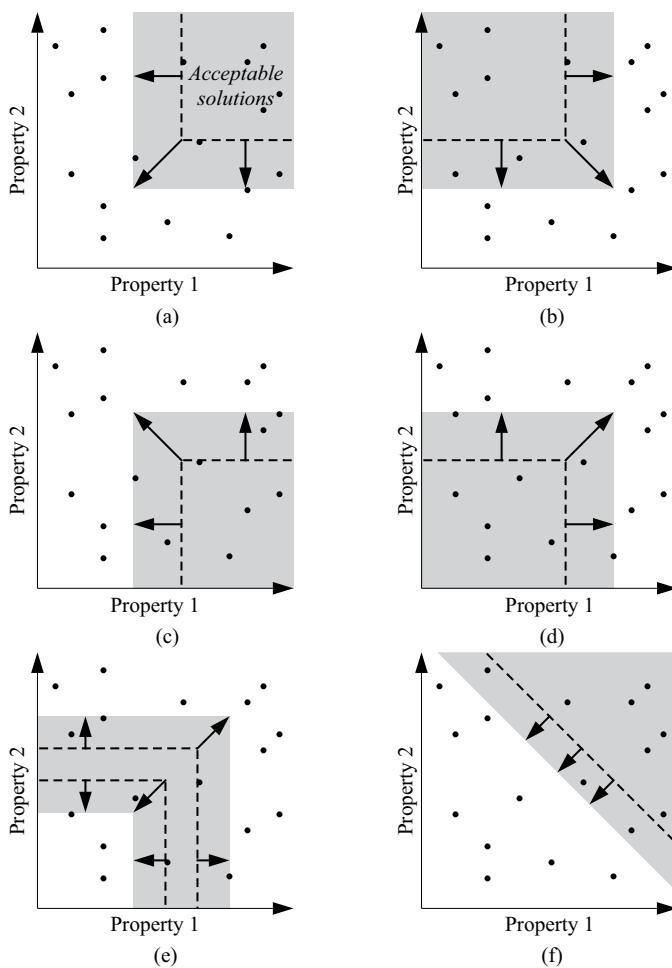


Figure 13.13 Relaxation of constraints: (a) both constraints are minimum specifications; (b) property 1 has a maximum specification, and property 2 has a minimum specification; (c) property 2 has a maximum specification, and property 1 has a minimum specification; (d) both constraints are maximum specifications; (e) constraints are target values with associated tolerances; (f) constraint is a trade-off between interdependent properties.

some combination of properties defines the constraint boundary (Figure 13.13f). In this case, there is a trade-off between the properties.

An alternative approach is to treat the category of satisfactory materials, that is, those that meet the constraints, as a fuzzy set (see Chapter 3). Under such a scheme, those materials that possessed properties comfortably within the specification would be given a membership value of 1, while those that failed completely to reach the specification would be given a membership value of 0. Materials close to

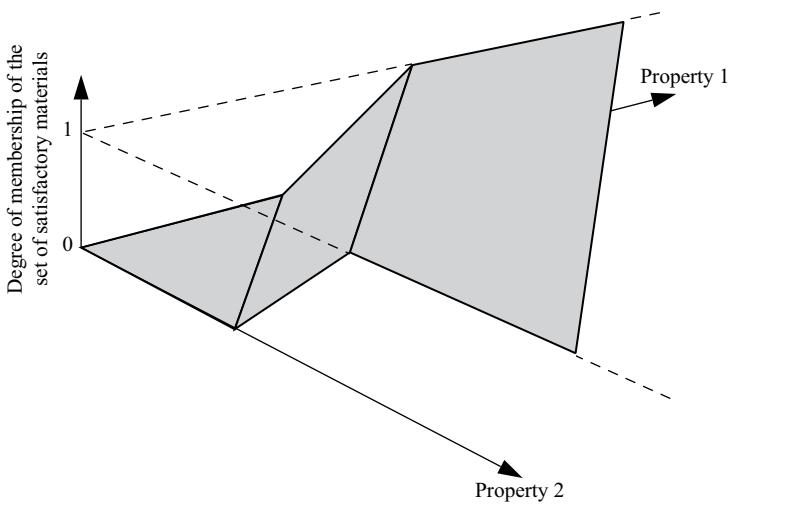


Figure 13.14 A fuzzy constraint.

the constraint boundary would be assigned a degree of membership between 0 and 1 (Figure 13.14). The membership values for each material might then be taken into account in the next stage of the selection process, based on scoring each material.

Ashby (1989) has plotted maps similar to those in Figure 13.13 using logarithmic scales. These “Ashby maps” are a particularly effective means of representing a constraint on a merit index. Figure 13.15 shows the loci of points for which:

$$\frac{E}{\rho} = \text{constant}$$

$$\frac{E^{1/2}}{\rho} = \text{constant}$$

$$\frac{E^{1/3}}{\rho} = \text{constant}$$

E/ρ is a suitable merit index for the surface material of a stiff lightweight sandwich beam, $E^{1/2}/\rho$ is a suitable merit index for the material of a stiff lightweight tube, and $E^{1/3}/\rho$ is a suitable merit index for the material of a stiff lightweight plate. In Figure 13.15, the materials that meet the merit index specification most comfortably are those that are toward the top-left of the map.

When two desirable properties (such as strength and cheapness) are plotted against each other, the boundary of the population of acceptable materials may follow an arc, as shown in Figure 13.16, representing the trade-off between the properties. This boundary is known as the *Pareto boundary*. If more than two properties are considered,

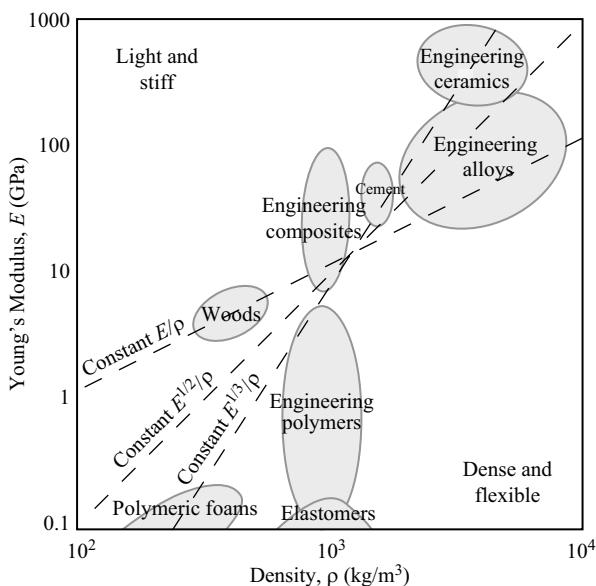


Figure 13.15 Ashby map for Young's modulus versus density. (Reprinted from Ashby, M. F. 1989. On the engineering properties of materials. *Acta Metallurgica* 37:1273–1293. Copyright (1989). With permission from Elsevier Science.)

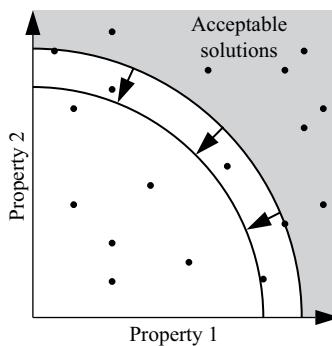


Figure 13.16 Constraint relaxation by sliding the Pareto boundary.

the boundary defines a surface in multidimensional space known as the *Pareto surface*. Materials that lie on the Pareto surface are said to be *Pareto optimal*, as an improvement in one property is always accompanied by deterioration in another if the acceptability criterion is maintained. Selection could be restricted to Pareto optimal candidates, but constraint relaxation allows materials *close* to the boundary to be considered as well (Figure 13.16). These same arguments apply to selection between design alternatives (Mouelhi et al. 2009; Sriram et al. 1989), as well as to selection between materials.

13.8.6 A Naive Approach to Scoring

We shall now move on to the problem of sorting the shortlist into an order of preference. Let us assume the existence of a data file containing, for each polymer, a set of performance values (ranging from 0 to 9) for each of a number of different properties. The user can supply a weighting for each property of interest to represent its importance. A naive approach to determining a polymer's score is to multiply the two figures together for each property, and then to take the sum of the values obtained to be the overall score for that polymer. The polymers with the highest scores are recommended to the user. This scoring system is summarized as follows:

$$\text{Total score for polymer } i = \sum_j [\text{performance}(i, j) \times \text{weight}(j)] \quad (13.7)$$

where:

$\text{performance}(i, j)$ = performance value of polymer i for property j ;
 $\text{weight}(j)$ = user-supplied weighting for property j .

An implication of the use of a summation of scores is that—even though a particular polymer may represent a totally inappropriate choice because of, for example, its poor impact resistance—it may still be highly placed in the ordered list of recommendations. An alternative to finding the arithmetic sum of all of the scores is to find their product:

$$\text{Product of scores for polymer } i = \prod_j [\text{performance}(i, j) \times \text{weight}(j)] \quad (13.8)$$

When combining by multiplication, a poor score for a given property is less readily compensated by the polymer's performance for other properties. A polymer that scores particularly badly on a given criterion tends to be filtered out from the final list of recommendations. Thus, using the multiplication approach, good “all-round performers” are preferred to polymers offering performance that varies between extremes.

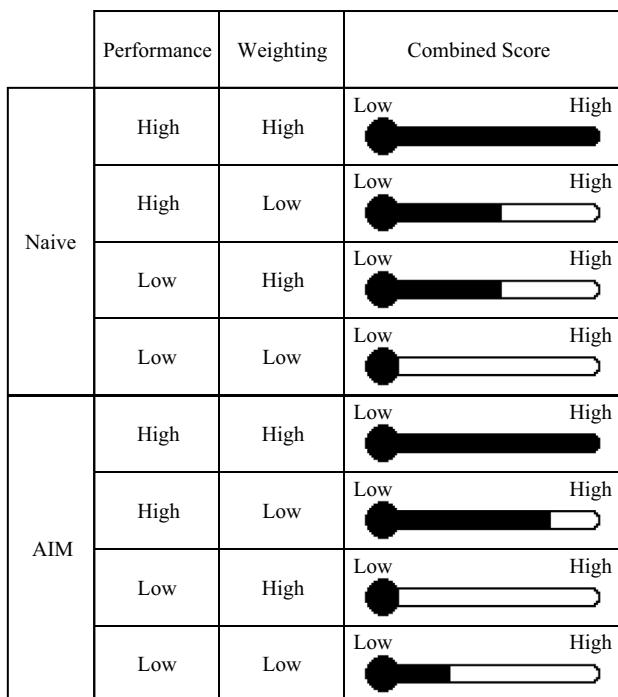
This distinction between the two approaches is illustrated by the simple example in Table 13.2. In this example, polymer B offers a uniform mediocre rating across the three properties, while the rating of polymer A varies from poor (score 1) to good (score 3). Under an additive scheme, the polymers are ranked equal, while under the multiplication scheme polymer B is favored.

A little reflection will show that both of these approaches offer an inadequate means of combining performance values with weightings. Where a property is considered important (so it has a high weighting) and a polymer performs well with respect to that property (so its performance value is high), the contribution to the

Table 13.2 Combining Scores by Addition and by Multiplication

	Score 1	Score 2	Score 3	Combination by Addition	Combination by Multiplication
Polymer A	1	2	3	6	6
Polymer B	2	2	2	6	8

polymer score is large. However, where a property is considered less important (low weighting) and a polymer performs poorly with respect to that property (performance value is low), this combination produces the smallest contribution to the polymer score. In fact, since the property in question has a low importance rating, the selection of the polymer should be still favored. The AIM algorithm (Section 13.8.7) was developed specifically to deal with this anomaly. The least appropriate polymer is actually one that has low values of performance for properties with high importance weightings. Figure 13.17 compares the naive algorithms with AIM.

**Figure 13.17 Comparison of naive and AIM scoring schemes.**

13.8.7 A Better Approach to Scoring

The shortcomings of a naive approach to scoring have been noted in the previous subsection and used as a justification for the development of an improved algorithm, AIM (Hopgood 1989). Using AIM, the score for each polymer is given by:

$$\text{Total score for polymer } i = \prod_j \left[\frac{(\text{performance}(i, j) - \text{offset})}{\times \text{weight}(j) + \text{scale_shift_term}} \right] \quad (13.9)$$

where *offset* is the midpoint of the performance range and *scale_shift_term* is the smallest number that will ensure that the combined weight and performance rating is positive. In an implemented system (Hopgood 1989), the following parameters were selected:

- polymer performance rating range: 0.0–9.0
- weighting range: 0.0–10.0
- *offset* = 4.5
- *scale_shift_term* = 46.0

The AIM equation for a single property, with these parameters inserted, is shown in Figure 13.18. Performance values lower than the offset value can be thought of as degrees of undesirability. On the weightings scale, zero means “I don’t care.”

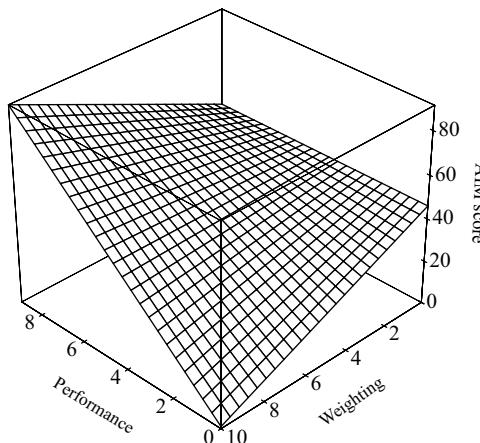


Figure 13.18 Combination of performance values with weightings for a single property, using AIM.

13.8.8 Case Study: The Design of a Kettle

Figure 13.19 shows a possible set of inputs and outputs from a polymer selection system that uses AIM. After receiving a list of recommended polymers, the user may alter one or more previous inputs in order to test the effect on the system's recommendations. These "what if?" experiments are also useful for designers whose materials specifications were only vaguely formed when starting a consultation. In these circumstances, the system serves not only to make recommendations for the choice of polymer, but also to assist the designer in deciding upon the materials requirements. The interface contains gaps in places where an entry would be inappropriate. For instance, the user can indicate that "glossiness" is to be maximized, as well as supplying a weighting. However, the user cannot supply a specification of the minimum acceptable glossiness, as only comparative data are available.

In the example shown in Figure 13.19, the designer is trying to determine a suitable polymer for the manufacture of a kettle. The designer has decided that the kettle must be capable of withstanding boiling water for intermittent periods. In

Input:

Property	Constraint	Tolerance	Weighting
Impact resistance			5
Resistance to aqueous environments			8
Maximum operating temperature	$\geq 100^\circ\text{C}$	30°C	
Glossiness			3
Cheapness			6
Ease of injection molding			9

Output:

Recommended polymers	Normalized score
Polypropylene copolymer	4.05
ABS (acrylonitrile-butadiene-styrene copolymer)	3.59
Polypropylene homopolymer	3.29
Fire-retardant polypropylene	2.53
30% glass-fibre coupled polypropylene	2.40
TPX (poly-4-methyl pent-1-ene)	2.06
(114 others meet the constraints)	

Figure 13.19 Use of the AIM polymer selection system during the design of a kettle.

addition, a high level of importance has been placed upon the need for the polymer to be injection moldable. The designer has also chosen material cheapness as a desired property, independent of manufacturing costs. Additionally, the glossiness and impact resistance of the polymer are to be maximized, within the constraints of attempting to optimize as many of the chosen properties as possible.

As we have already noted, care must be taken when entering any numerical specifications. In this example, it has been specified that a maximum operating temperature of at least 100°C is required. A tolerance of 30°C has been placed on this value to compensate for the fact that the polymer will only be intermittently subjected to this temperature. A polymer whose maximum operating temperature is 69°C would be eliminated from consideration, but one with a maximum operating temperature of 70°C would remain a candidate. In the current example, the temperature requirement is clearly defined, although the tolerance is more subjective. The provision of a tolerance is equivalent to constraint relaxation.

The recommendations shown in the example are reasonable. The preferred polymer (polypropylene) is sometimes used in kettle manufacture. The second choice (ABS, or acrylonitrile-butadiene-styrene copolymer) is used for the handles of some brands of kettles. The most commonly used polymer in kettle manufacture is an acetal copolymer, which was missing from the selection system's database. This example shows the importance of having access to adequate data.

13.8.9 Reducing the Search Space by Classification

The selection system described in the previous subsection relies on the ability to calculate a score for every polymer in the system database. In this example, only 150 polymers are considered, each of which has complete data for a limited set of properties. However, even with the search constrained to polymer materials, there are in reality thousands of candidate polymers and grades of polymer. Countless more grades could be specified by slight variations in composition or processing. Clearly, a system that relies on a complete and consistent set of data for each material cannot cope with the full range of available materials. Even if the data were available, calculating scores for every single one is unnecessary, and bears no resemblance to the approach adopted by a human expert, who would use knowledge about *families* of materials.

In general, chemically similar materials tend to have similar properties, as shown by the Ashby map in Figure 13.15. It would therefore be desirable to restructure the database so that polymers are hierarchically classified, with polymers of a given type grouped together. Thus, given only a vague specification, many categories could be eliminated from consideration early on. Within the category of materials called *polymer*, several subcategories exist, such as *acetal*. The selection task is simplified enormously by using knowledge of the range of values for a given property that apply to a particular subcategory. The initial searches would then scan only polymer groups, based upon ranges of properties for polymers within that group.

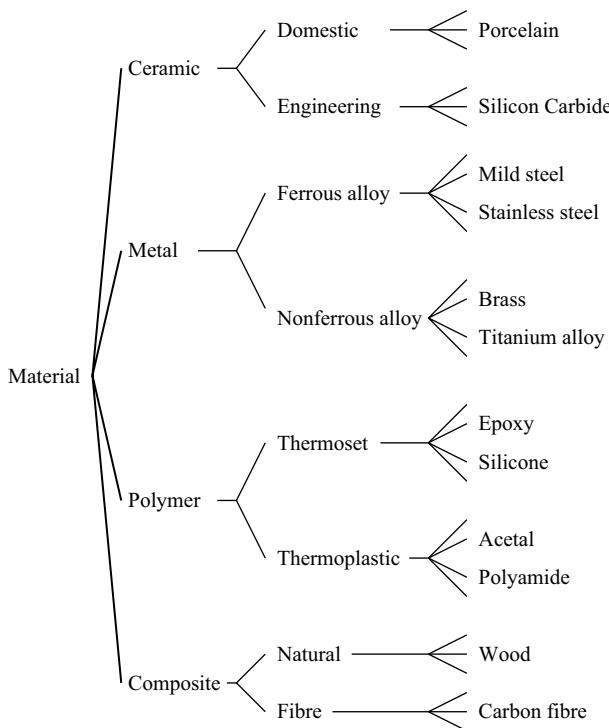


Figure 13.20 One of many possible ways to classify materials.

Only when the search has settled on one or two such families is it necessary to consider individual grades of polymer within those groups. As such a classification of materials is hierarchical, it can be represented using object or frame classes joined by specialization relationships (Chapter 4). One of many possible classification schemes is shown in Figure 13.20.

Demaid and Zucker (1992) make use of their own specialized object-oriented system to allow a full and detailed description of real materials and also of the hypothetical “ideal” material for the job. They specifically aim to overcome the restrictions inherent in systems that rely on a single number to describe a complex property. The knowledge-based system incorporating AIM makes an attempt at overcoming this limitation by using rules to modify the data in certain circumstances (Hopgood 1989). However, the real problem is that a single value describing a material property, such as stiffness, can only be valid at one temperature, after a fixed duration, under a fixed load, and in a particular environment. So, in order to choose a polymer that is sufficiently stiff to be suitable for a kettle body, we need more information than just its stiffness at room temperature. We also need to know its stiffness at 100°C and after, for example, two years of daily use. To illustrate how acute the problems can be when dealing with polymers, Figure 13.21 shows how

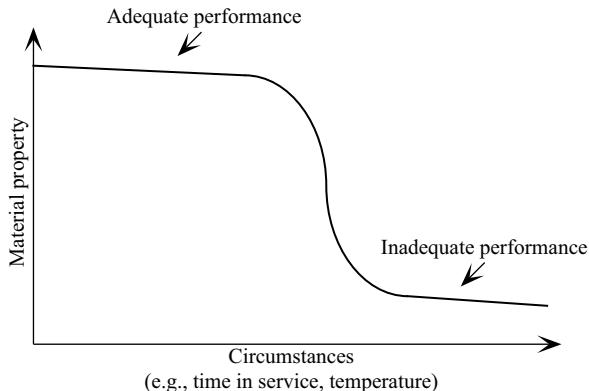


Figure 13.21 The “cliff-edge” effect.

a property such as stiffness might vary with temperature or duration of exposure. The designer (or the intelligent selection system) needs to be aware of possible “cliff-edge” effects like the one shown in Figure 13.21. For example, some polymers may have an adequate stiffness for many purposes at room temperature, but not necessarily after prolonged exposure to elevated temperatures.

13.9 Failure Mode and Effects Analysis (FMEA)

An important aspect of design is the consideration of what happens when things go wrong. If any component of a product should fail, the designer will want to consider the impact of that failure on the following:

- *Safety.* For example, would an explosion occur? Would a machine go out of control?
- *Indication of failure.* Will the user of the product notice that something is amiss? For example, will a warning light illuminate or an alarm sound?
- *Graceful or graceless degradation.* Will the product continue to function after a component has failed, albeit less efficiently? This capability is known as graceful degradation and has some advantages over designs in which the failure of a component is catastrophic. On the other hand, graceful degradation may require that the product contain more than the bare minimum of components, thereby increasing costs.
- *Secondary damage.* Will the failure of one component lead to damage of other components? Are those other components more or less vital to the function of the product? Is the secondary damage more expensive to fix than the original damage?

The assessment of all possible effects from all possible failures is termed *failure mode and effects analysis* (FMEA). FMEA is not concerned with the *cause* of failures (that is a diagnosis problem—see Chapter 11) but the *effects* of failures. FMEA comprises the following key stages:

- Identifying the possible failure modes
- Generating the changes to the product caused by the failure
- Identifying the consequences of those changes
- Evaluating the significance of the consequences

The scoring technique discussed in Section 13.8.7 could feasibly be adapted for the fourth stage, evaluating the *significance* of failure mode effects. The FLAME system (Price and Hunt 1991; Price 1998) uses product models in order to automate the first three stages of FMEA. Two modeling approaches have been used: *functional* and *structural* modeling. Functional modeling involves the breakdown of a system into subsystems, where each subsystem fulfills a specific function. The subsystems may be further decomposed, leading to a hierarchical breakdown based on functionality. The encapsulated nature of each subsystem favors the use of object-oriented programming or a multiagent system (see Chapter 4). In the case of the windshield washer system of a car (Figure 13.22), each subsystem is modeled by its response to one of three standard electrical inputs: positive voltage relative to earth, open circuit, or short-circuited to earth. The output from a subsystem then forms the input to another.

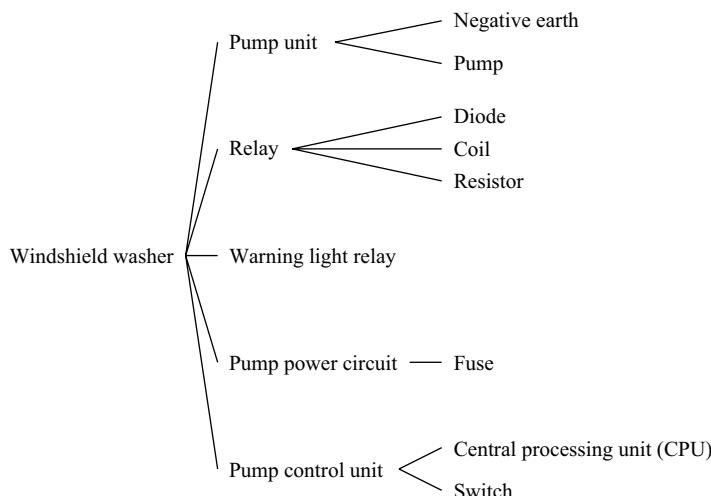


Figure 13.22 Functional decomposition of a windshield washer system. (Derived from Price, C. J., and J. E. Hunt. 1991.)

Price and Hunt argue that functional modeling is only adequate when the subsystems respond correctly to each of the modeled inputs. Under such circumstances, each subsystem can be relied upon to generate one of a few standard responses, which becomes the input to another subsystem. However, if the behavior of a subsystem is altered by a failure mode, a response may be generated that is not described in the functional model. If this response forms the input to another subsystem, the functional model can no longer cope. To model the functional response to *all* such inputs is impractical, as it would require a complete FMEA in advance. The FLAME system overcomes this problem by augmenting the functional model with a structural model that simulates the overall system. In this way, it can analyze the inputs that are generated at each subsystem.

13.10 Summary

This chapter has addressed some of the issues in developing intelligent systems to support design decision-making. Design can be viewed as a search problem in which alternatives must be found or generated and a selection made from among them. It is a particularly difficult task because it requires both creativity and a vast range of knowledge. Electrical and electronic engineering have been most amenable to the application of decision-support tools, as designs in these domains are often routine rather than innovative and can often be treated as optimization problems.

Selection between alternatives forms an integral part of the design problem. One important selection decision is the choice of materials, a problem that has been explored in some detail in this chapter. Similar techniques might be applied to other aspects of selection within design. Even within the apparently limited domain of materials selection, the range of relevant knowledge is so wide and the interactions so complex that automation is a major challenge.

We have seen, by reference to the design of a telecommunication network, that the design process can be applied to services as well as to manufactured products. This particular case study has also illustrated that producing a design specification can in itself be a complex task, and one that has to be formalized before computerized support tools can be considered.

The concepts of constraint propagation and truth maintenance have been illustrated by considering the problem of arranging batteries in a battery holder. Conceptual design, optimization and evaluation, and detailed design have been illustrated by considering the design of an aircraft floor. This design exercise included both geometric design and materials selection. The final case study, concerning the design of a kettle, was used to illustrate some additional ideas for materials selection.

CAD packages have been mentioned briefly. These are useful tools, but they are often limited to drafting rather than decision-making. The human designer remains at the center of the design process and a range of decision-support tools is being

developed that will assist rather than replace the human designer. To this end, it is likely that we will see a greater degree of integration of CAD tools with intelligent systems for decision support (Kavakli 2001).

Further Reading

- Gero, J. S., ed. 2011. *Design Computing and Cognition '10*. Springer, Berlin, Germany.
- Goel, A. K., and A. G. de Silva Garza. 2010. Special issue on artificial intelligence in design. *Journal of Computing and Information Science in Engineering* 10 (3).
- Noor, A. K. 2017. AI and the Future of Machine Design, *Mechanical Engineering* 139(10): 38–43.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 14

Systems for Planning

14.1 Introduction

The concept of planning is one that is familiar to all of us, as we constantly make and adjust plans that affect our lives. We may make long-range plans, such as selecting a particular career path, or short-range plans, such as what to eat for lunch. A reasonable definition of planning is the process of producing a *plan*, where:

A plan is a description of a set of actions or operations, in a prescribed order, that are intended to reach a desired goal.

Planning, therefore, concerns the analysis of actions that are to be performed in the future. It is a similar problem to designing (Chapter 13), except that it includes the notion of time. Design is concerned with the detailed description of an artifact or service, without consideration of when any specific part should be implemented. In contrast, the timing of a series of actions, or at least the order in which they are to be carried out, is an essential aspect of planning. Planning is sometimes described as “reasoning about actions,” which suggests that a key aspect to planning is the consideration of questions of the form “what would happen if ...?”

Charniak and McDermott (1985) have drawn an analogy between planning and programming, as the process of drawing up a plan can be thought of as programming oneself. However, they have also highlighted the differences, which are, in particular:

- Programs are intended to be repeated many times, whereas plans are frequently intended to be used once only.
- The environment for programs is predictable, whereas plans may need to be adjusted in the light of unanticipated changes in circumstances.

The need for computer systems that can plan, or assist in planning, is widespread. Potential applications include management decisions, factory configuration, organization of manufacturing processes, business planning and forecasting, and strategic military decision-making. Some complex software systems may feature *internal planning*, that is, planning of their own actions. For example, robots may need to plan their movements, while other systems may use internal planning to ensure that an adequate solution to a problem is obtained within an acceptable timescale.

According to our definition, planning involves choosing a set of operations and specifying either the timing of each or just their order. In many problems, such as planning a manufacturing process, the operations are known in advance. However, the tasks of allocating resources to operations and specifying the timing of operations can still be complicated and difficult. This specific aspect of planning is termed *scheduling* and is described in Sections 14.7 and 14.8.

Some of the principles and problems of planning systems are discussed in Section 14.2 and an early planning system (Stanford Research Institute Problem Solver, STRIPS) is described in Section 14.3. This system forms a basis, from which more sophisticated features are described in the remainder of this chapter.

14.2 Classical Planning Systems

In order to make automated planning a tractable problem, certain assumptions have to be made. All of the systems discussed in this chapter, except for the reactive planners in Section 14.9, assume that the world can be represented by taking a “snapshot” at a particular time to create a *world state*. Reasoning on the basis of this assumption is termed *state-based reasoning*. Planning systems that make this assumption are termed *classical planning systems*, as the assumption has formed the basis of much of the past research. The aim of a classical planner is to move from an initial world state to a different world state, which is the *goal state*. Determining the means of achieving this is *means–ends analysis* (see Section 14.3.1).

The inputs to a classical system are well defined:

- A description of the initial world state
- A set of actions or operators that might be performed on the world state
- A description of the goal state

The output from a classical planner comprises a description of the sequence of operators that, when applied to the current world state, will lead to the desired world state as described in the goal. Each operator in the sequence creates a new projected world state, upon which the next operator in the sequence can act, until the goal has been achieved.

The assumption that a plan can be based upon a snapshot of the world is only valid if the world does not change during the planning process. Classical planners may be inadequate for reasoning about a continuous process or dealing with unexpected catastrophes. If a system can react quickly enough to changes in the world to replan and to instigate the new plan, all in real time, the system is described as *reactive* (Section 14.9). As reactive planners must be continuously alert to changes in the world state, they are not classical systems.

Bel et al. (1989) have categorized types of planning decision on the basis of the timescale to which they apply, as shown in Figure 14.1. Long-range planning decisions concern general strategy and investments; medium-range planning applies to decisions such as production planning, with a horizon of weeks or months; short-range planning or scheduling refers to the day-to-day management of jobs and resources. Control decisions have the shortest time horizon and are made in real time (see Chapter 15).

A significant problem in the construction of a planning system is deciding which aspects of the world state are altered by an operation, and by how much they are altered, and which are not affected at all. Furthermore, some operations may have a different effect depending on the context, that is, depending on some other aspect of the world state. Collectively, these challenges are known as the *frame problem*, introduced in Chapter 1. As an example, consider the operation of walking from home to the shops. Two obvious changes in the world state are, first, that I am no longer at home and, second, that I have arrived at the shops. However, there are a number of other changes, such as the amount of rubber left on the soles of my shoes. There are also many things that will *not* have changed as a result of my walk, such as the price of oil. Some changes in the world state will depend on context. For example, if the initial world state includes rainy weather then I will arrive at the shops wet, but otherwise I will arrive dry.

Describing every feasible variant of an action with every aspect of the world state is impractical, as the number of such interactions increases dramatically with the

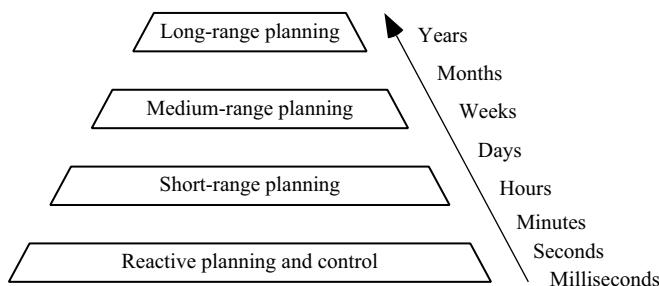


Figure 14.1 Classification by timescale of planning tasks. (Derived from Bel, G. et al. 1989.)

complexity of the model. One approach to dealing with the frame problem is the so-called *STRIPS assumption*, which is used in STRIPS (see the following section) and many other classical planners. This assumption is that all aspects of the world will remain unchanged by an operation, apart from those that are changed explicitly by the modeled operation. The STRIPS assumption is, therefore, similar to the closed-world assumption (see Chapters 1 and 2).

The planning problem becomes even more complex if we introduce multiple agents like those we met in Chapter 4 (Caridi and Cavalieri 2004). Imagine that we have programmed a robot as an autonomous agent, capable of planning and executing a set of operations in order to achieve some goal. Suppose now that our robot is joined by several other robots that also plan and execute various operations. It is quite probable that the plans of the robots will interfere. For example, they may bump into each other or require the same tool at the same time. Stuart (1985) has considered multiple robots, where each one reasons independently about the beliefs and goals of others, so as to benefit from them. Other researchers (e.g., Durfee et al. 1985), have considered collaborative planning by multiple agents (or robots), where plan generation is shared so that each agent is allocated a specific task. Shen (2002) argues that agent-based approaches to distributed manufacturing offer the best combination of responsiveness, flexibility, scalability, and robustness. However, a follow-up review (Shen et al. 2006) notes the lack of take-up of these approaches in the manufacturing industry.

14.3 Stanford Research Institute Problem Solver (STRIPS)

14.3.1 General Description

STRIPS (Fikes and Nilsson 1971; Fikes et al. 1972) is one of the oldest AI planning systems, but it remains the basis for much research into intelligent planning (e.g., Aineto et al. 2018; Galuszka and Swierniak 2010). In its original form, STRIPS was developed in order to allow a robot to plan a route between rooms, moving boxes along the way. However, the principles are applicable to a number of planning domains, and we will consider the operational planning of a company that supplies components for aircraft engines.

Like all classical planning systems, STRIPS is based upon the creation of a world model. The world model comprises a number of objects as well as operators that can be applied to those objects so as to change the world model. So, for instance, given an object *alloy block*, the operator *turn* (i.e., on a lathe) can be applied so as to create a new object called *turbine disk*. The problem to be tackled is to determine a sequence of operators that will take the world model from an initial state to a goal state.

Operators cannot be applied under all circumstances but, instead, each has a set of preconditions that must hold before it can be applied. Given knowledge of the preconditions and effects of each operator, STRIPS is able to apply a technique called *means–ends analysis* to solve planning problems. This technique, mentioned in Chapter 4, Section 4.3.3, involves looking for differences between the current state of the world and the goal state, and finding an operator that will reduce the difference. Many classical planning systems use means–ends analysis because it reduces the number of operators that need to be considered and, hence, the amount of search required. The selected operator has its own preconditions, the satisfaction of which becomes a new goal. STRIPS repeats the process recursively until the desired state of the world has been achieved.

14.3.2 An Example Problem

Consider the problem of responding to a customer's order for turbine disks. If we already have some suitable disks, then we can simply deliver them to the customer. If we do not have any disks manufactured, we can choose either to manufacture disks from alloy blocks (assumed to be the only raw material) or to subcontract the work. Assuming that we decide to manufacture the disks ourselves, we must ensure that we have an adequate supply of raw materials, that staff members are available to carry out the work, and that our machinery is in good working order. In the STRIPS model we can identify a number of objects and operators, together with the preconditions and effects of the operators (Table 14.1).

Table 14.1 shows specific instantiations of objects associated with operators (e.g., purchase raw materials), but other instantiations are often possible (e.g., purchase anything). The table shows that each operator has preconditions that must be satisfied before the operator can be executed. Satisfying a precondition is a subproblem of the overall task, and so a developing plan usually has a hierarchical structure.

We will now consider how STRIPS might solve the problem of dealing with an order for a turbine disk. The desired (goal) state of the world is simply *customer has turbine disk*, with the other parameters about the goal state of the world left unspecified.

Let us assume for the moment that the initial world state is as follows:

- The customer has placed an order for a turbine disk.
- The customer does not have the turbine disk.
- Staff members are available.
- We have no raw materials.
- Materials are available from a supplier.
- We can afford the raw materials.
- The machinery is working.

Table 14.1 Operators for Supplying a Product to Customer. Objects Are Italicized

Operator and object	Precondition	Effect
deliver <i>product</i>	<i>product</i> is ready and <i>order</i> has been placed by <i>customer</i>	<i>customer</i> has <i>product</i>
subcontract (manufacture of) <i>product</i>	<i>subcontractor</i> available and <i>subcontractor cost</i> is less than <i>product price</i>	<i>product</i> is ready
manufacture <i>product</i>	<i>staff</i> members are available, we have the <i>raw materials</i> , and the <i>machinery</i> is working	<i>product</i> is ready, our <i>raw materials</i> are reduced, and the <i>machinery</i> is closer to its next maintenance period
purchase raw materials	we can afford the <i>raw materials</i> and the <i>raw materials</i> are available	we have the <i>raw materials</i> and <i>money</i> is subtracted from our <i>account</i>
borrow money	good relationship with our <i>bank</i>	<i>money</i> is added to our <i>account</i>
sell assets	<i>assets</i> exist and there is sufficient time to sell them	<i>money</i> is added to our <i>account</i>
repair <i>machinery</i>	<i>parts</i> are available	<i>machinery</i> is working and next maintenance period scheduled

Means–ends analysis can be applied. The starting state is compared with the goal state and one difference is discovered, namely *customer has turbine disk*. STRIPS would now treat *customer has turbine disk* as its goal and would look for an operator that has this state in its list of effects. The operator that achieves the desired result is *deliver*, which is dependent on the conditions *turbine disk ready* and *order placed*. The second condition is satisfied already. The first condition becomes a sub-goal, which can be solved in either of two ways: by subcontracting the work or by manufacturing the disk. We will assume for the moment that the disk is to be manufactured in-house. Three preconditions exist for the *manufacture* operator. Two of these are already satisfied (staff members are available, and the machinery is working), but the precondition that we have raw materials is not satisfied and becomes a new sub-problem. This sub-problem can be solved by the operator *purchase*, whose two preconditions (that we can afford the materials and that materials

are available from a supplier) are already satisfied. Thus, STRIPS has succeeded in finding a plan, namely:

purchase raw materials → manufacture product → deliver product

The full search tree showing all operators and their preconditions is shown in Figure 14.2, while Figure 14.3 indicates the subset of the tree that was used in producing this plan. Note that performing an operation changes the state of the world model. For instance, the operator *purchase*, applied to raw materials, raises our stock of materials to a sufficient quantity to fulfill the order. In so doing, it fulfills the preconditions of another operator, namely, *manufacture*. This operator is then also able to change the world state, as it results in the product's being ready for delivery.

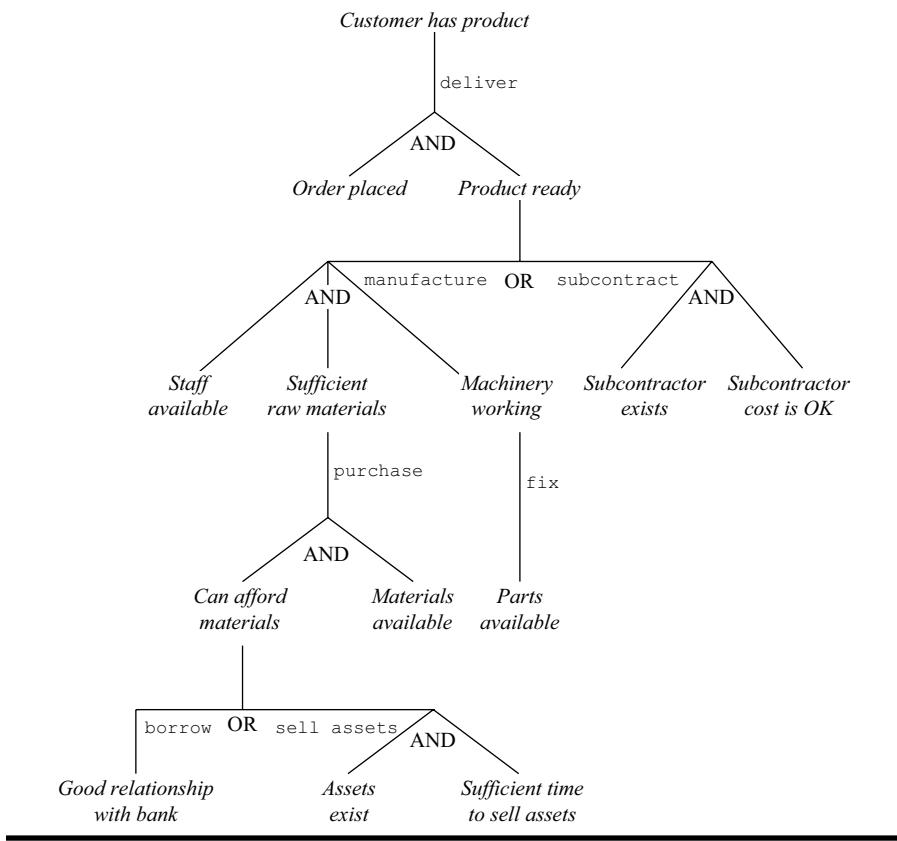


Figure 14.2 A search tree of effects and operators.

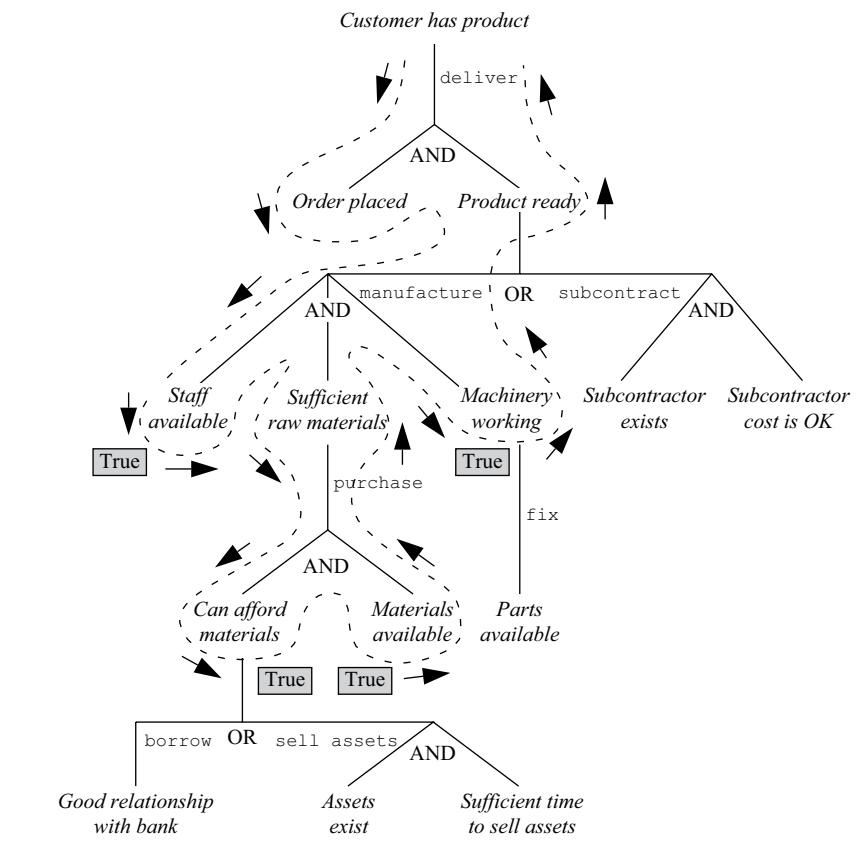


Figure 14.3 STRIPS searching for a plan to fulfill the goal *customer has product*, given the following initial world state: the customer has placed an order for a turbine disk; the customer does not have the turbine disk; staff members are available; we have no raw materials; materials are available from a supplier; we can afford the raw materials; the machinery is working.

14.3.3 A Simple Planning System in Prolog

STRIPS involves search, in a depth-first fashion, through a tree of states linked by operators. STRIPS initially tries to establish a goal. If the goal has preconditions, these become new goals and so on until either the original goal is established or all possibilities have been exhausted. In other words, STRIPS performs backward chaining in order to establish a goal. If it should find that a necessary condition cannot be satisfied for the branch that it is investigating, STRIPS will backtrack to the last decision point in the tree (i.e., the most recently traversed *or* node). STRIPS is heavily reliant on backward chaining and backtracking, features that are built into the Prolog language (Chapter 11). For this reason, it should be fairly straightforward to program a STRIPS-like system in Prolog. The actual STRIPS program described

by Fikes and Nilsson (1971) and Fikes et al. (1972) was implemented in Lisp. It was more sophisticated than the Prolog program presented here, and it used a different representation of objects and operators. More recent implementations of STRIPS-like systems include the GRT system in C++ (Refanidis and Vlahavas 2001) and numerous Python implementations such as the one by Garrett et al. (2017).

There are a number of different ways in which we might build a system for our example of supply of a product to a customer. For the purposes of illustration, we will adopt the following scheme:

- The sequence of operators used to achieve a goal is stored as a list, representing a plan. For example, assuming that we have no money in our account and no raw materials, a plan for obtaining a turbine blade that is ready for delivery would be the list:

```
[borrow_money, purchase_materials, manufacture]
```

- Each effect is represented as a clause whose last argument is the plan for achieving the effect. For instance, the effect *our customer has turbine disk* is represented by the following clause:
`has(our_customer, turbine_disk, Plan).`
- If a particular effect does not require any actions, then the argument corresponding to its plan is an empty list. We may wish to set up certain effects as part of our initial world state, which do not require a plan of action. An initial world state in which a subcontractor exists would be represented as:
`exists(subcontractor, []).`
- The preconditions for achieving an effect are represented as Prolog rules. For instance, one way in which a product can become ready is by subcontracting its manufacture. Assuming this decision, there are two preconditions, namely, that a subcontractor exists and that the cost of subcontracting is acceptable. The Prolog rule is as follows:

```
ready(Product, Newplan) :-  
    exists(subcontractor, Plan1),  
    subcontractor_price_OK(Plan2),  
    merge([Plan1, Plan2, subcontract], Newplan).
```

The final condition of this rule is the `merge` relation, which is used for merging sub-plans into a single sequence of actions. This relation is not a standard Prolog facility, so we will have to create it for ourselves. The first argument to `merge` is a list that may contain sublists, while the second argument is a list containing no sublists. The purpose of `merge` is to “flatten” a hierarchical list (the first argument) and to assign the result to the second argument. We can define `merge` by four separate rules, corresponding to different structures that the first argument might have. We can ensure that the four rules are considered mutually exclusive by using the cut facility (see Chapter 11).

The complete Prolog program is shown in Box 14.1. The program includes one particular initial world state, but of course this state can be altered. The world state shown in Box 14.1 is as follows:

BOX 14.1 A SIMPLE PLANNER IN PROLOG

```

has(Customer, Product, Newplan) :- % deliver product
    ready(Product, Plan1),
    order_placed(Product, Customer, Plan2),
    merge([Plan1, Plan2, deliver], Newplan).

ready(Product, Newplan) :-
    % subcontract the manufacturing
    exists(subcontractor, Plan1),
    subcontractor_price_OK(Plan2),
    merge([Plan1, Plan2, subcontract], Newplan).

subcontractor_price_OK([]) :-
    % no action is required if the subcontractor's
    % price is OK, i.e. less than the disk price
    price(subcontractor, Price1, _),
    price(product, Price2, _),
    Price1 < Price2.

ready(Product, Newplan) :-
    % manufacture the product ourselves
    exists(staff, Plan1),
    sufficient_materials(Plan2),
    machine_working(Plan3),
    merge([Plan1, Plan2, Plan3, manufacture], Newplan).

sufficient_materials([]) :-
    % no action required if we have sufficient stocks
    % already
    current_balance(materials, Amount, _),
    Amount >= 1.

sufficient_materials(Newplan) :- % purchase materials
    can_afford_materials(Plan),
    merge([Plan, purchase_materials], Newplan).

can_afford_materials([]) :-
    % no action is required if our bank balance is adequate
    current_balance(account, Amount, _),
    price(materials, Price, _),
    Amount >= Price.

```

(Continued)

Box 14.1 (Continued)

```

can_afford_materials(Newplan) :-           % borrow money
  bank_relationship(good, Plan),
  merge([Plan, borrow_money], Newplan).

can_afford_materials(Newplan) :-           % sell assets
  exists(assets, Plan1),
  exists(time_to_sell, Plan2),
  merge([Plan1, Plan2, sell_assets], Newplan).

machine_working(Newplan) :-                 % fix machinery
  exists(spare_parts, Plan2),
  merge([Plan1, Plan2, fix_machine], Newplan).

merge([], []) :- !.

merge([[] | Hierarchical], Flat) :-        !
, merge(Hierarchical, Flat).

merge([X | Hierarchical], [X | Flat]) :-    atom(X),
, merge(Hierarchical, Flat).

merge([X | Hierarchical], [A | Flat]) :-    X = [A | Rest],
, merge([Rest | Hierarchical], Flat).

% set up the initial world state
price(product, 1000, []).
price(materials, 200, []).
price(subcontractor, 500, []).
current_balance(account, 0, []).
current_balance(materials, 0, []).
bank_relationship(good, []).
order_placed(turbine, acme, []).
exists(subcontractor, []).
exists(spare_parts, []).
exists(staff, []).

% Assumed false under the closed-world assumption:
%   machine_working([]),
%   exists(assets, []),
%   exists(time_to_sell, []).

```

```

price(product,1000,[]).
% Turbine disk price is $1000
price(materials,200,[]).
% Raw material price is $200 per disk
price(subcontractor,500,[]).
% Subcontractor price is $500 per disk
current_balance(account,0,[]).
% No money in our account
current_balance(materials,0,[]).
% No raw materials
bank_relationship(good,[]).
% Good relationship with the bank
order_placed(turbine,acme,[]).
% Order has been placed by ACME, Inc.
exists(subcontractor,[]).
% A subcontractor is available
exists(spare_parts,[]).
% Spare parts are available
exists(staff,[]).
% Staff members are available

```

Because it is not specified that we have assets or time to sell them, or that the machinery is working, these assertions are all considered false under the closed-world assumption.

We can now ask our Prolog system for suitable plans to provide a customer (ACME, Inc.) with a product (a turbine disk), as follows:

```
?- has(acme, turbine, Plan).
```

Prolog offers the following plans in response to our query:

```

Plan = [subcontract, deliver];
Plan = [borrow_money, purchase_materials, fix_machine,
manufacture, deliver];
no

```

We can also ask for plans to achieve any other effect that is represented in the model. For instance, we could ask for a plan to give us sufficient raw materials, as follows:

```

?- sufficient_materials(Plan).
Plan = [borrow_money, purchase_materials];
no

```

Having discussed a simple planning system, the remainder of this chapter will concentrate on more sophisticated features that can be incorporated.

14.4 Considering the Side Effects of Actions

14.4.1 Maintaining a World Model

Means–ends analysis (see Section 14.3.1) relies upon the maintenance of a world model, as it involves choosing operators that reduce the difference between a given state and a goal state. Our simple Prolog implementation of a planning system does not explicitly update its world model, and this limitation leads to a deficiency in comparison with practical STRIPS implementations. When STRIPS has selected an operator, it applies that operator to the current world model, so that the model changes to a projected state. This change is important because an operator may have many effects, only one of which may be the goal that is being pursued. The new world model therefore reflects both the intended effects and the side effects of applying an operator, provided that they are both explicit in the representation of the operator. Under the STRIPS assumption (see Section 14.2), all other attributes of the world state are assumed to be unchanged by the application of an operator.

In the example considered in Section 14.3.3, the Prolog system produced a sequence of operators for achieving a goal, namely, to supply a product to a customer. What the system fails to tell us is whether there are any implications of the plan, other than achievement of the goal. For instance, we might like to be given some details of our projected cash flow, of our new stocks of materials, or of the updated maintenance schedule for our machinery. Because these data are not necessary for achieving the goal—although they are affected by the planned operators—they are ignored by a purely backward-chaining mechanism. (See Chapter 2 for a discussion of forward and backward chaining.) Table 14.1 indicates that purchasing raw materials has the effect of reducing our bank balance, and manufacturing reduces the time that can elapse before the machinery is due for servicing. Neither effect was considered in our Prolog system because these effects were not necessary for achieving the goal.

14.4.2 Deductive Rules

SIPE is a planning system that can deduce effects additional to those explicitly included in the operator representation (Wilkins 1983, 1984, 1989; Wilkins and desJardins 2001). This capability is powerful, as operators may be context-sensitive, meaning that the same operator may have different effects in different situations. Without this capability, context-sensitivity can only be modeled by having different operators to represent the same action taking place in different contexts.

SIPE makes use of two types of deductive rules, *causal* rules and *state* rules. Causal rules detail the auxiliary changes in the world state that are associated with the application of an operator. For example, the operator *purchase* is intended to change the world state from *we have no raw materials* to *we have raw materials*. This

change has at least one side effect, which is that our bank account balance is diminished. This side effect can be modeled as a causal rule.

State rules are concerned with maintaining the consistency of a world model, rather than explicitly bringing about changes in the model. Thus, if the assertion *machinery is working* is true in the current world state, then a state rule could be used to ensure that the assertion *machinery is broken* is made false.

Causal rules react to changes between states, whereas state rules enforce constraints within a state. Example causal and state rules are shown in Box 14.2, using syntax similar to that in SIPE. Note that parameters are passed to the rules in place of the named arguments. The rules are, therefore, more general than they would be without the arguments. A rule is considered for firing if its *trigger* matches the world state *after* an operator has been applied. In the case of the causal rule *update_bank_balance*, the trigger is the world state *we have sufficient supplies*, which is brought about by the operator *purchase*. Because causal rules react to a change in the world state, they also contain a *precondition*, describing the world state *before* the operator was applied, for example, not (*sufficient raw materials*). A causal rule will only fire if its trigger is matched after an operator has been applied and its precondition had been matched immediately before the operator was applied. State rules are not directly concerned with the application of an operator, and so do not have a precondition. There is, however, provision for naming further conditions (additional to the trigger) that must be satisfied.

In SIPE, when an operator is added to the current plan, causal rules are examined first in order to introduce any changes to the world model, and then state rules are applied to maintain consistency with the constraints on the model. Other than the order of applicability, there is no enforced difference between causal and state rules. According to the syntax, both can have preconditions and a trigger, although there appears to be no justification for applying a precondition to a state rule.

BOX 14.2 CAUSAL AND STATE RULES IN SIPE

```

causal-rule: update_bank_balance
arguments: cost, old_balance, new_balance
trigger: sufficient supplies of something
precondition: not (sufficient supplies of something)
effects: new_bank_balance = old_bank_balance - cost

state-rule: deduce_fixed
arguments: machine1
trigger: machine1 is working
other conditions: <none>
effects: not (machine1 is broken)

```

14.5 Hierarchical Planning

14.5.1 Description

Virtually all plans are hierarchical by nature, as exemplified by Figure 14.2, although they are not represented as such by all planning systems. STRIPS (a nonhierarchical planner) may produce the following plan for satisfying a customer's order:

borrow money → purchase materials → fix machinery → manufacture → deliver

Some of the actions in this plan are major steps (e.g., manufacture), whereas others are comparatively minor details (e.g., purchase materials). A *hierarchical planner* would first plan the major steps, for example:

be ready to deliver → deliver

The details of a step such as *be ready to deliver* might then be elaborated:

be ready to manufacture → manufacture → deliver

The step *be ready to manufacture* might be broken down into the following actions:

fix machinery → obtain raw materials

The action *obtain raw materials* can then be elaborated further:

borrow money → purchase materials

This hierarchical plan is depicted in Figure 14.4. An action that needs no further refinement is a *primitive action*. In some applications *purchase materials* may be considered a primitive action, but in other applications it may be necessary to elaborate this further (e.g., pick up the phone, dial a number, speak to a supplier, and so on).

Hierarchical planners explicitly represent the hierarchical nature of the plan and are, therefore, distinct from nonhierarchical planners like STRIPS. At the top of the hierarchy is a simplification or abstraction of the plan, while the lower levels contain the detailed requirements (Figure 14.4). A subplan is built for achieving each action in the main plan. While STRIPS does recognize that the achievement of some goals is dependent on subgoals, no distinction is drawn between goals that are major steps and those that are merely incremental. Furthermore, as the STRIPS hierarchy is not explicitly represented, it cannot be modified either by the user or by the system itself.

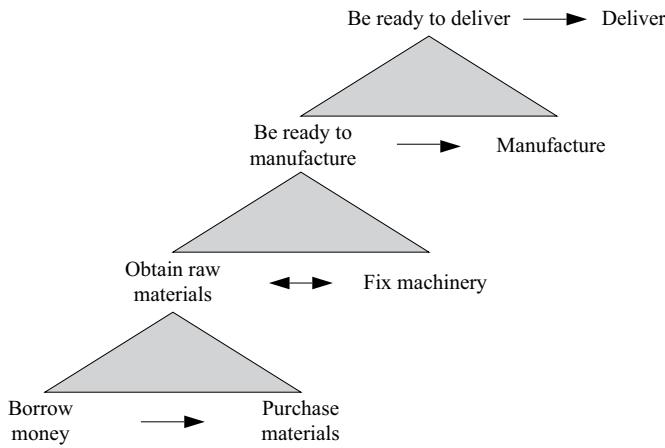


Figure 14.4 A hierarchical plan.

The method of hierarchical planning can be summarized as follows:

- Sketch a plan that is complete but too vague to be useful.
- Refine the plan into more detailed subplans until a complete sequence of problem-solving operators has been specified.

Since the plan is complete (though perhaps not useful in its own right) at each level of abstraction, the term *length-first* search is sometimes used to describe this technique for selecting appropriate operators that constitute a plan. Yang (1997) identifies three classes of abstraction: precondition-elimination, effect, and task. To these, Galindo et al. (2004) add a fourth, based on abstraction only of the world description, which forms the basis of their Hierarchical task Planning through World Abstraction (HPWA).

14.5.2 Benefits of Hierarchical Planning

Although means–ends analysis is an effective way of restricting the number of operators that apply to a problem, there may still be several operators to choose from, with no particular reason for preferring one over another. In other words, there may be several alternative branches of the search tree. Furthermore, there is no way of knowing whether the selected branch might lead to a dead end, which happens if one of its preconditions fails.

Consider the example of satisfying a customer's order for a product. Suppose that STRIPS has chosen to apply the `manufacture` operator, in other words, the left-hand branch of the tree shown in Figure 14.2 has been selected. STRIPS would now verify that staff members are available, plan to purchase raw materials (which in turn requires money to be borrowed), and then consider the state of the manufacturing

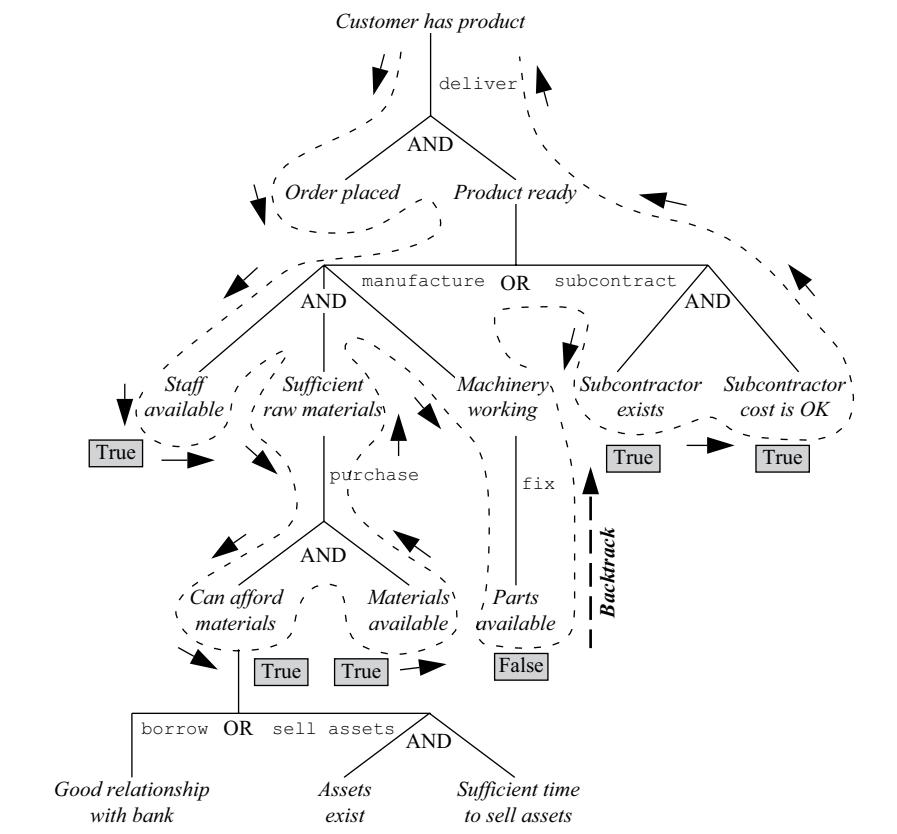


Figure 14.5 Inefficient search using STRIPS: the system backtracks on finding that there are no parts available for fixing the broken machinery.

equipment. Suppose that, at this point, it found that the machinery was broken and that spare parts were not available. The plan would have failed, and the planner would have to backtrack to the point where it chose to manufacture rather than to subcontract. All the intermediate processing would have been in vain, since STRIPS cannot plan to manufacture the product if the machinery is inoperable. The search path followed is shown in Figure 14.5.

Part of the expense of backtracking in this example arises from planning several operations that are minor details compared with the more important issue of whether the equipment for manufacturing is available. This question is relatively important, and we would expect to have established an answer earlier in the plan, before considering the details of how to obtain the money to buy the raw materials. The more natural approach to planning is to plan out the important steps first and then to fill in the details, that is, to plan hierarchically. Hierarchical planning is one way of postponing commitment to a particular action until more information about the appropriateness

of the action is available. This philosophy, sometimes called the *principle of least commitment*, occurs in different guises and is discussed further in Section 14.6.

Hierarchical planning requires the use of levels of abstraction in the planning process and in the description of the domain, where an abstraction level is distinguished by the granularity (or level of detail) of its description. It is unfortunate that the term “hierarchical planning” is sometimes used with different meanings. For instance, the term is sometimes used to describe levels of metaplanning, that is, planning the process of creating a plan.

14.5.3 Hierarchical Planning with ABSTRIPS

ABSTRIPS (abstraction-based STRIPS; Sacerdoti 1974) is an extension of STRIPS that incorporates hierarchical planning. In ABSTRIPS, preconditions and operators are unchanged from STRIPS, except that some preconditions are considered more important than others. Before attempting to generate a plan, ABSTRIPS assigns an importance rating, or *criticality*, to each precondition. The highest criticality is ascribed to those preconditions that cannot be altered by the planner, and lower criticalities are given to preconditions that can be satisfied by generating a subplan. Planning proceeds initially by considering only the operators that have the highest criticality, thereby generating a skeleton plan. This skeleton plan is said to be in the highest *abstraction space*. Details of the skeleton plan are filled by progressively considering lower criticality levels. In this way, subplans are generated to satisfy the preconditions in the higher-level plans until all the preconditions in a plan have been achieved. The plan at any given level (except for the highest and the lowest) is a skeleton plan for the level below and a refinement of the skeleton plan provided by the layer above.

ABSTRIPS adopts a semiautomated approach to the assigning of criticalities to preconditions. The user supplies a set of values, which are subsequently modified by ABSTRIPS using some simple heuristics. We will illustrate the process with reference to our example of supplying a product to a customer. The preconditions in our model include having something, something being affordable, or something existing. The existence or otherwise of something is beyond our powers to alter and, thus, intuitively warrants the highest criticality value. On the other hand, there are a variety of different ways in which having something can be achieved, and so these preconditions might be given the lowest criticality. A sensible set of user-supplied criticality values might be as shown in Table 14.2.

ABSTRIPS then applies heuristics for modifying the criticality values, given a particular world state. The preconditions are examined in order of decreasing user-supplied criticality and modified as follows:

- a. Any preconditions that remain true or false, irrespective of the application of an operator, are given the maximum criticality. Let us call these *fundamental preconditions*.

Table 14.2 User-Supplied Criticality Values Ascribed to Preconditions

<i>Precondition</i>	<i>User-Supplied Criticality</i>
We have an item	1
Something is affordable	2
Something exists or is available	3
Other considerations	2

- b. If a precondition can be readily established, assuming that all previously considered preconditions are satisfied (apart from unsatisfied fundamental preconditions), then the criticality is left unaltered.
- c. If a precondition cannot be readily established as described in (b), it is given a criticality value between that for category (a) and the highest in category (b).

The criticality values supplied by the user are dependent only on the nature of the preconditions themselves, whereas the modified values depend upon the starting world state and vary according to the circumstances. Consider, for instance, the following world state:

- customer does not have turbine disk;
- customer has placed an order for a turbine disk;
- staff members are available;
- we have no raw materials;
- we have no money in the bank;
- we have a good relationship with our bank;
- we do not have any assets, nor time to sell assets;
- the machinery is broken;
- spare parts are not available;
- a subcontractor exists;
- the subcontractor cost is reasonable.

The following preconditions are given a maximum criticality (say 5) because they are fundamental, and cannot be altered by any operators:

- order placed by customer;
- subcontractor exists;
- subcontractor cost is OK;
- staff available;
- raw materials available from supplier;
- machinery parts available;

- good relationship with bank;
- assets exist;
- sufficient time to sell assets.

The precondition *machinery working* falls into category (c) as it depends directly on *spare parts available*, a fundamental precondition that is false in the current world model. The remaining preconditions belong in category (b), and, therefore, their criticalities are unchanged. Although *can afford materials* is not immediately satisfied, it is readily achieved by the operator borrow, assuming that *good relationship with bank* is true. Therefore, *can afford materials* falls into category (b) rather than (c). Similar arguments apply to *product ready* and *sufficient raw materials*. Given the world model described, the criticalities shown in Table 14.3 might be assigned.

Once the criticalities have been assigned, the process of generating a plan can proceed as depicted by the flowchart in Figure 14.6. Planning at each abstraction level is treated as elaborating a skeleton plan generated at the level immediately

Table 14.3 Initial and Modified Criticality Values Ascribed to Preconditions

Precondition	Initial Criticality	Modified Criticality
Staff available	3	5
Subcontractor exists	3	5
Raw materials available	3	5
Machinery parts available	3	5
Assets exist	3	5
Order placed by customer	2	5
Machinery working	2	4
Subcontractor cost OK	2	5
Good relationship with bank	2	5
Sufficient time to sell assets	2	5
Can afford materials	2	2
Product ready	1	1
Sufficient raw materials	1	1

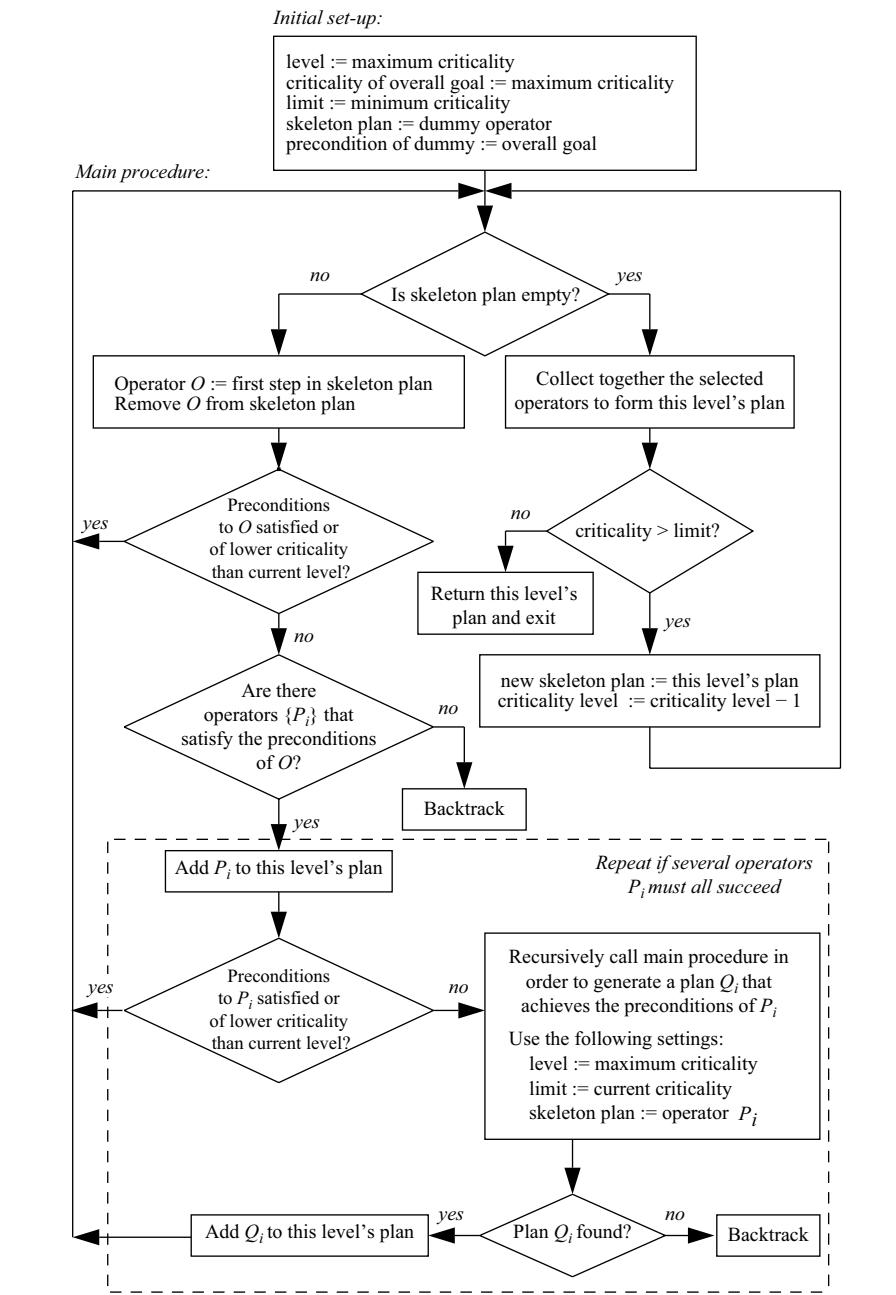


Figure 14.6 Planning with a hierarchical system based on ABSTRIPS.

higher. The main procedure is called recursively whenever a subplan is needed to satisfy the preconditions of an operator in the skeleton plan. Figure 14.6 is based on the ABSTRIPS procedure described by Sacerdoti (1974), except that we have introduced a variable lower limit on the criticality in order to prevent a subplan from being considered at a lower criticality level than the precondition it aims to satisfy.

When we begin planning, a dummy operator is used to represent the skeleton plan. The precondition of dummy is the goal that we are trying to achieve. Consider planning to achieve the goal *customer has product*, beginning at abstraction level 5 (Figure 14.7). The precondition to dummy is *customer has product*. This precondition is satisfied by the operator *deliver*, which has two preconditions. One of them (*order placed*) is satisfied, and the other (*product ready*) has a criticality less than 5. Therefore, *deliver* becomes the skeleton plan for a lower abstraction level.

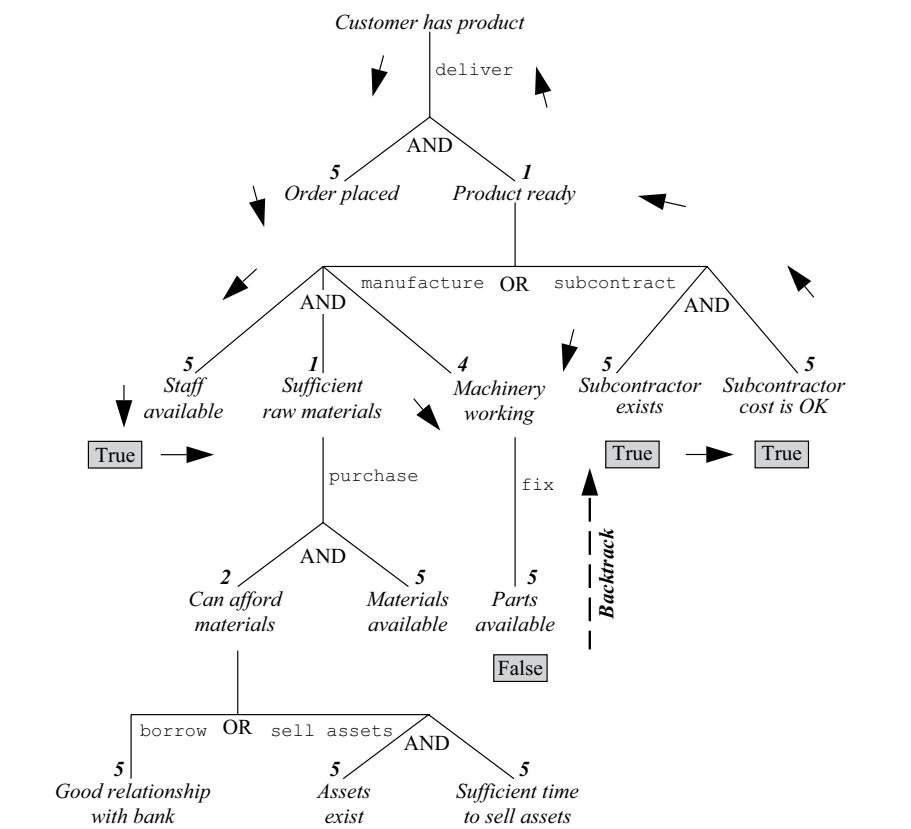


Figure 14.7 More efficient search using a hierarchical planner based on ABSTRIPS. The figures shown alongside the preconditions are the criticality levels.

The skeleton plan cannot be elaborated in levels 4, 3, or 2, as the criticality of *product ready* is only 1. At level 1, operators that achieve *product ready* are sought and two are found, namely, *manufacture* and *subcontract*. Both of these operators have preconditions of the highest criticality, so there is no reason to give one of them priority over the other. Supposing that *manufacture* is selected, the main procedure is then called recursively, with the preconditions to *manufacture* as the new goal. The precondition of the highest criticality is *staff available*, and this precondition is found to be satisfied. At the next level *machinery working* is examined, and the main procedure is called recursively to find a plan to satisfy this precondition. However, no such plan can be found as *parts available* is false. The plan to *manufacture* is abandoned at this stage and the planner backtracks to its alternative plan, *subcontract*. The preconditions to *subcontract* are satisfied and it becomes the new plan.

A hierarchical planner can solve problems with less searching and backtracking than its nonhierarchical equivalent. The preceding example (shown in Figure 14.7) is more efficient than the STRIPS version (Figure 14.5), as the hierarchical planner did not consider the details of borrowing money and buying raw materials before abandoning the plan to *manufacture*. Because a complete plan is formulated at each level of abstraction before the next level is considered, the hierarchical planner can recognize dead ends early, as it did with the problem of fixing the machinery. If more complex plans are considered, involving many more operators, the saving becomes much greater still.

The planner described here, based on ABSTRIPS, is just one of many approaches to hierarchical planning. Others adopt different means of determining the hierarchical layers, since the assignment of criticalities in ABSTRIPS is rather *ad hoc*. Some of the other systems are less rigid in their use of a skeleton plan. For example, the Nonlin system (Tate 1977) treats the abstraction levels as a guide to a skeleton solution, but it is able to replan or consider alternatives at any level if a solution cannot be found or if a higher-level choice is faulty.

14.6 Postponement of Commitment

14.6.1 Partial Ordering of Plans

We have already seen that an incentive for hierarchical planning is the notion that we are better off deferring detailed decisions until after more general decisions have been made. This is a part of the principle of *postponement of commitment*, or the principle of *least commitment*. In the same context, if the order of steps in a plan makes no difference to the plan, the planner should leave open the option of doing them in any order. A plan is said to be *partially ordered* if it contains actions that are unordered with respect to each other, that is, actions for which the planner has not yet determined an order and that may possibly be in parallel.

If we refer back to the Prolog planner described in Section 14.3.3, we see that, given a particular world state and the goal of supplying a product to a customer, the following plan was generated:

```
[borrow_money, purchase_materials, fix_machine, manufacture, deliver]
```

This plan contains a definite order of events. As can be inferred from the search tree in Figure 14.2, some events must occur before others. For example, the product cannot be delivered until after it has been manufactured. However, the operators `fix_machine` and `purchase_materials` have been placed in an arbitrary order. These operators are intended to satisfy two subgoals: *machinery working* and *sufficient raw materials*, respectively. As both subgoals need to be achieved, they are conjunctive goals. A planning system is said to be *linear* if it assumes that it does not matter in which order conjunctive goals are satisfied. This so-called *linear assumption*, which is not necessarily valid, can be expressed as follows:

According to the linear assumption, subgoals are independent and thus they can be sequentially achieved in an arbitrary order.

Nonlinear planners are those that do not rely upon this assumption. The generation of partially ordered plans is the most common form of nonlinear planning, but it is not the only form (Hendler et al. 1990). The generation of a partially ordered plan avoids commitment to a particular order of actions until information for selecting one order in preference to another has been gathered. Thus, a nonlinear planner might generate the following partially ordered plan.

$$\left[\left[\begin{array}{c} \text{borrow_money}, \text{purchase_materials} \\ \text{fix_machine} \end{array} \right], \text{manufacture}, \text{deliver} \right]$$

If it is subsequently discovered that fixing the machine requires us to borrow money, this requirement can be accommodated readily because we have not committed ourselves to fixing the machine before seeking a loan. Thus, a single loan can be organized for the purchase of raw materials and for fixing the machine.

The option to generate a partially ordered plan occurs every time a planner encounters a conjunctive node (i.e., *and*) on the search tree. Linear planners are adequate when the branches are decoupled, so that it does not matter which action is performed first. Where the ordering is important, a nonlinear planner can avoid an exponential search of all possible plan orderings. To emphasize how enormous this saving can be, just ten actions have more than three million (i.e., $10!$) possible orderings.

Some nonlinear planners adopt a different approach to limiting the number of orderings that need be considered, e.g., HACKER (Sussman 1975) and INTERPLAN (Tate 1975). These systems start out by making the linearity assumption. When confronted with a conjunctive node in the search tree, they select an arbitrary order for the actions corresponding to the separate branches. If the selected order is subsequently found to create problems, the plan is fixed by reordering. Depending on the problem being addressed, this approach may be inefficient as it can involve a large amount of backtracking.

HYBIS is a nonlinear planner that uses prior experience of previously solved planning problems to reduce the amount of backtracking (Fernandez et al. 2005). Goals are decomposed into subgoals at a lower level of the hierarchy, for each of which a partial order planner is evoked. The prior experience is represented as a trace of a search tree, whose nodes are tagged as previously successful, failed, abandoned, or unexplored. The previously successful nodes are prioritized for possible re-use, leading to efficiency gains.

We have already seen that the actions of one branch of the search tree can interact with the actions of another. As a further example, a system might plan to purchase sufficient raw materials for manufacturing a single batch, but some of these materials might be used up in the alignment of machinery following its repair. Detecting and correcting these interactions is a problem that has been addressed by most of the more sophisticated planners. The problem is particularly difficult in the case of planners like SIPE that allow actions to take place concurrently. SIPE tackles the problem by allocating a share of limited resources to each action and placing restrictions on concurrent actions that use the same resources. Modeling the process in this way has the advantage that resource conflicts are easier to detect than interactions between the effects of two actions.

14.6.2 The Use of Planning Variables

The use of planning variables is another technique for postponing decisions until they have to be made. Planners with this capability could, for instance, plan to purchase *something*, where *something* is a variable that does not yet have a value assigned. Thus, the planner can accumulate information before making a decision about what to purchase. The instantiation of *something* may be determined later, thus avoiding the need to produce and check a plan for every possible instantiation.

The use of planning variables becomes more powerful still if we can progressively limit the possible instantiations by applying constraints to the values that a variable can take. Rather than assuming that *something* is either unknown or has a specific value (e.g., *gearbox_part#7934*), we could start by applying the constraint that it is a gearbox component. We might then progressively tighten the constraints and, thereby, reduce the number of possible instantiations.

14.7 Job-Shop Scheduling

14.7.1 The Problem

As noted in Section 14.1, scheduling is a planning problem where time and resources must be allocated to operators that are known in advance. The term *scheduling* is sometimes applied to the internal scheduling of operations within a knowledge-based system. However, in this section we will be concerned with scheduling only in an engineering context.

Job-shop scheduling is a problem of great commercial importance. A job shop is either a factory or a manufacturing unit within a factory. Typically, the job shop consists of a number of machine tools connected by an automated palletized transportation system, as shown in Figure 14.8. The completion of a job may require the production of many different parts, grouped in lots. Flexible manufacturing is possible, because different machines can work on different part types simultaneously, allowing the job shop to adapt rapidly to changes in production mix and volume.

The planning task is to determine a schedule for the manufacturing of the parts that make up a job. As noted in Section 14.1, the operations are already known in this type of problem, but they still need to be organized in the most efficient way. The output that is required from a scheduling system is typically a Gantt chart like that shown in Figure 14.9. Therefore, the decisions required are:

- the allocation of machines (or other resources) to each operation;
- the start and finish times of each operation (although it may be sufficient to specify only the order of operations, rather than their projected timings).

The output of the job shop should display *graceful degradation*, that is, a reduced output should be maintained in the event of accidental or preprogrammed machine stops, rather than the whole job shop grinding to a halt. The schedule must ensure

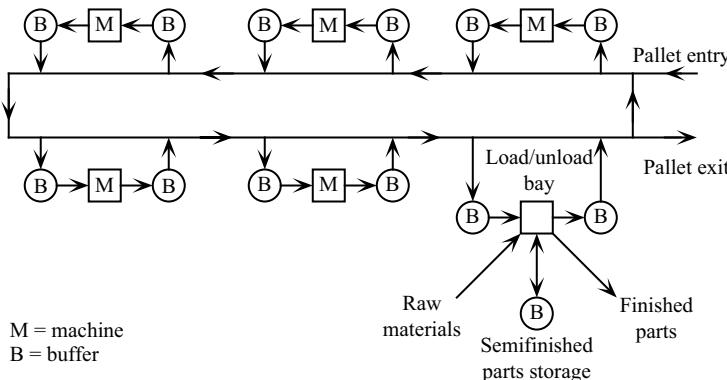


Figure 14.8 A possible job shop layout. (Derived from Bruno, G. et al. 1986.)

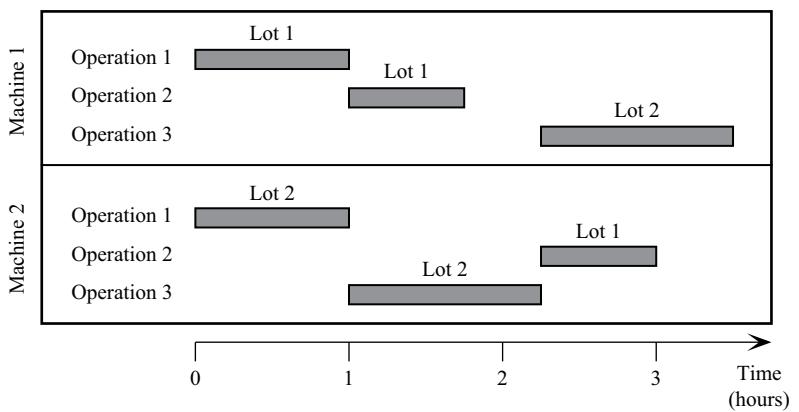


Figure 14.9 A Gantt chart, giving a visual representation of a schedule.

that all jobs are completed before their due dates, while taking account of related considerations such as minimizing machine idle times, queues at machines, work in progress, and allowing a safety margin in case of unexpected machine breakdown. Some of these considerations are *constraints* that *must* be satisfied, and others are *preferences* that we would *like* to satisfy to some degree. Several researchers, including Bruno et al. (1986) and Dattero et al. (1989), have pointed out that a satisfactory schedule is required, and that this may not necessarily be an optimum. A similar viewpoint is frequently adopted in the area of engineering design (Chapter 13).

14.7.2 Some Approaches to Scheduling

The approaches to automated scheduling that have been applied in the past can be categorized as either analytical, iterative, or heuristic (Liu 1988). None of these approaches has been particularly successful in its own right, but some of the more successful scheduling systems borrow techniques from all three approaches. The analytical approach requires the problem to be structured into a formal mathematical model. This approach normally requires several assumptions to be made, which can compromise the validity of the model in real-life situations. The iterative approach requires all possible schedules to be tested and the best one to be selected. The computational load of such an approach renders it impractical where there are large numbers of machines and lots. The heuristic approach relies on the use of rules to guide the scheduling. While this approach can save considerable computation, the rules are often specific to just one situation, and their expressive power may be inadequate.

Bruno et al. (1986) have adopted a semiempirical approach to the scheduling problem. A discrete event simulation (similar, in principle, to the ultrasonic simulation described in Chapter 4) serves as a test bed for the effects of action sequences. If a particular sequence causes a constraint to be violated, the simulation can backtrack

by one or more events and then test a new sequence. Objects are used to represent the key players in the simulation, such as lots, events, and goals. Rules are used to guide the selection of event sequences using priorities that are allocated to each lot by the simple expression:

$$\text{priority} = \frac{\text{remaining machining time}}{\text{due date} - \text{release time}}$$

Liu (1988) has extended the ideas of hierarchical planning (Section 14.5) to include job-shop scheduling. He has pointed out that one of the greatest problems in scheduling is the interaction between events, so that fixing one problem (e.g., bringing forward a particular machining operation) can generate new problems (e.g., another lot might require the same operation, but its schedule cannot be moved forward). Liu, therefore, sees the problem as one of maintaining the integrity of a global plan, and dealing with the effects of local decisions on that global plan. He solves the problem by introducing planning levels. He starts by generating a rough utilization plan that acts as a guideline for a more detailed schedule. This rough plan is typically based upon the one resource thought to be the most critical. The rough plan is not expanded into a more detailed plan, but rather a detailed plan is formed from scratch, with the rough plan acting as a guide.

As scheduling is an optimization problem, it is no surprise that many researchers have applied computational intelligence approaches. In the domain of scheduling some pumping operations within the water distribution industry, McCormick and Powell (2004) have used simulated annealing to produce near-optimal discrete pump schedules while Gogos et al. (2005) have used genetic algorithms to optimize their mathematical model. Other researchers in the same domain have taken a hybrid approach to scheduling. For example, AbdelMeguid and Ulanicki (2010) have used a genetic algorithm to calculate feedback rules. In fact, their approach did not outperform a traditionally produced time schedule, but it was more robust to unexpected changes in water flows and demand.

Rather than attempt to describe all approaches to the scheduling problem, one particular approach will now be described in some detail. This approach involves constraint-based analysis (CBA) coupled with the application of preferences.

14.8 Constraint-Based Analysis (CBA)

14.8.1 Constraints and Preferences

A review of CBA for planning and scheduling is provided by Bartak et al. (2010). There may be many factors to take into account when generating a schedule. As noted in Section 14.7.1, some of these factors are *constraints* that *must* be satisfied, and others are *preferences* that we would *like* to satisfy. Whether or not a constraint

is satisfied is generally clear-cut, for example, a product is either ready on time or it is not. The satisfaction of preferences is sometimes clear-cut, but often it is not. For instance, we might prefer to use a particular machine. This preference is clear-cut because either it will be met, or it will not. On the other hand, a preference such as “minimize machine idle times” can be met to varying degrees.

14.8.2 Formalizing the Constraints

Four types of scheduling constraints that apply to a flexible manufacturing system can be identified:

- *Production constraints*

The specified quantity of goods must be ready before the due date, and quality standards must be maintained throughout. Each lot has an earliest start time and a latest finish time.

- *Technological coherence constraints*

Work on a given lot cannot commence until it has entered the transportation system. Some operations must precede others within a given job, and sometimes a predetermined sequence of operations exists. Some stages of manufacturing may require specific machines.

- *Resource constraints*

Each operation must have access to sufficient resources. The only resource that we will consider in this study is time at a machine, where the number of available machines is limited. Each machine can work on only one lot at a given time, and programmed maintenance periods for machines must be taken into account.

- *Capacity constraints*

In order to avoid unacceptable congestion in the transportation system, machine use and queue lengths must not exceed predefined limits.

Our discussion of CBA will be based on the model of Bel et al. (1989). Their knowledge-based system, OPAL, solves the static (i.e., “snapshot”) job-shop scheduling problem. It is, therefore, a classical planner (see Section 14.2). OPAL contains five modules (Figure 14.10):

- an object-oriented database for representing entities in the system such as lots, operations, and resources (including machines);
- a CBA module that calculates the effects of time constraints on the sequence of operations (the module generates a set of precedence relations between operations, thereby partially or fully defining those sequences that are viable);
- a decision-support (DS) module that contains rules for choosing a schedule, based upon practical or heuristic experience, from among those that the CBA module has found to be viable;

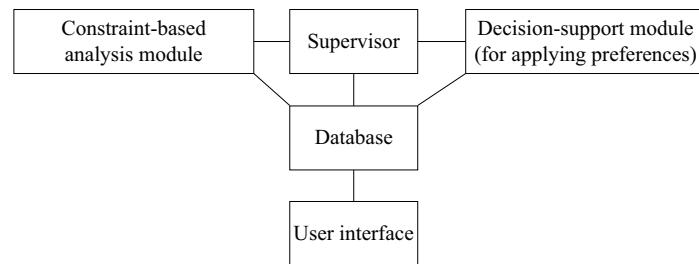


Figure 14.10 Principal modules in the OPAL scheduling system.

- a supervisor that controls communication between the CBA and DS modules, and builds up the schedule for presentation to the user;
- a user interface module.

Job-shop scheduling can be viewed in terms of juggling operations and resources. Operations are the tasks that need to be performed in order to complete a job, and several jobs may need to be scheduled together. Operations are characterized by their start time and duration. Each operation normally uses resources, such as a length of time at a given machine. There are two types of decisions, the timing or (sequencing) of operations and the allocation of resources. For the moment we will concentrate on the CBA module, which is based upon the following set of assumptions:

- There is a set of jobs J comprised of a set of operations O .
- There is a limited set of resources R .
- Each operation has the following properties:
cannot be interrupted;
uses a subset r of the available resources;
uses a quantity q_i of each resource r_i in the set r ;
has a fixed duration d_i .

A schedule is characterized by a set of operations, their start times, and their durations. We will assume that the operations that make up a job and their durations are predefined. Therefore, a schedule can be specified by just a set of start times for the operations. For a given job, there is an earliest start time (est_i) and latest finish time (lft_i) for each operation O_i . Each operation has a time window that it *could* occupy and a duration within that window that it *will* occupy. The problem is then one of positioning the operation within the window as shown in Figure 14.11.

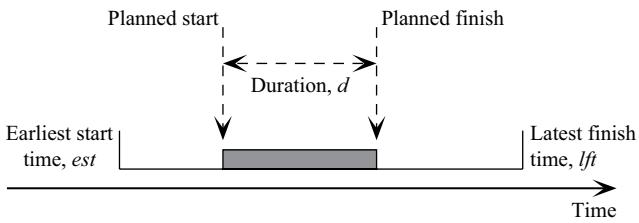


Figure 14.11 Scheduling an operation within its available time window.

14.8.3 Identifying the Critical Sets of Operations

The first step in the application of CBA is to determine if and where conflicts for resources arise. These conflicts are identified through the use of *critical sets*, a concept that is best described by example. Suppose that a small factory employs five workers who are suitably skilled for carrying out a set of four operations O_i . Each operation requires some number q_i of the workers, as shown in Table 14.4.

A critical set of operations I_c is one that requires more resources than are available, but where this conflict would be resolved if *any* of the operations were removed from the set. In our example, the workers are the resource and the critical sets are $\{O_1, O_3\}$, $\{O_2, O_3\}$, $\{O_4, O_3\}$, $\{O_1, O_2, O_4\}$. Note that $\{O_1, O_2, O_3\}$ is not a critical set since it would still require more resources than the five available workers if we removed O_1 or O_2 . The critical sets define the conflicts for resources, because the operations that make up the critical sets cannot be carried out simultaneously. Therefore, the first sequencing rule is as follows:

One operation of each critical set must precede at least one other operation in the same critical set.

Applying this rule to the preceding example produces the following conditions:

1.	either	$(O_1 \text{ precedes } O_3) \text{ or } (O_3 \text{ precedes } O_1);$
2.	either	$(O_2 \text{ precedes } O_3) \text{ or } (O_3 \text{ precedes } O_2);$
3.	either	$(O_4 \text{ precedes } O_3) \text{ or } (O_3 \text{ precedes } O_4);$
4.	either	$(O_1 \text{ precedes } O_2) \text{ or } (O_2 \text{ precedes } O_1) \text{ or }$ $(O_1 \text{ precedes } O_4) \text{ or } (O_4 \text{ precedes } O_1) \text{ or }$ $(O_2 \text{ precedes } O_4) \text{ or } (O_4 \text{ precedes } O_2).$

Table 14.4 Example Problem Involving Four Operations: O_1 , O_2 , O_3 , and O_4

	O_1	O_2	O_3	O_4
Number of workers required, q_i	2	3	5	2

These conditions have been deduced purely on the basis of the available resources, without consideration of time constraints. If we now introduce the known time constraints (i.e., the earliest start times and latest finish times for each operation), the schedule of operations can be refined further and, in some cases, defined completely. The schedule of operations is especially constrained in the case where each conflict set is a *pair* of operations. This is the *disjunctive* case, which we will consider first before moving on to consider the more general case.

14.8.4 Sequencing in the Disjunctive Case

As each conflict set is a pair of operations in the disjunctive case, no operations can be carried out simultaneously. Each operation has a defined duration (d_i), an earliest start time (est_i), and a latest finish time (lft_i). The scheduling task is one of determining the actual start time for each operation.

Consider the task of scheduling the three operations A , B , and C shown in Figure 14.12. If we try to schedule operation A first, we find that there is insufficient time for the remaining operations to be carried out before the final lft , irrespective of how the other operations are ordered (Figure 14.12b). However, there is a feasible schedule if operation C precedes A . This observation is an example of the following general rule, informally stated:

```
rule r14_1 /* informally stated; not Flex format */
  if (final lft - estA) < Σdi
    then at least one operation must precede A.
```

Similarly, there is no feasible schedule that has A as the last operation since there is insufficient time to perform all operations between the earliest est and the lft for A (Figure 14.12c). The general rule that describes this situation is:

```
rule r14_2 /* informally stated; not Flex format */
  if (lftA - earliest est) < Σdi
    then at least one operation must follow A.
```

14.8.5 Sequencing in the Nondisjunctive Case

In the nondisjunctive case, at least one critical set contains more than two operations. The precedence rules described in Section 14.8.3 can be applied to those

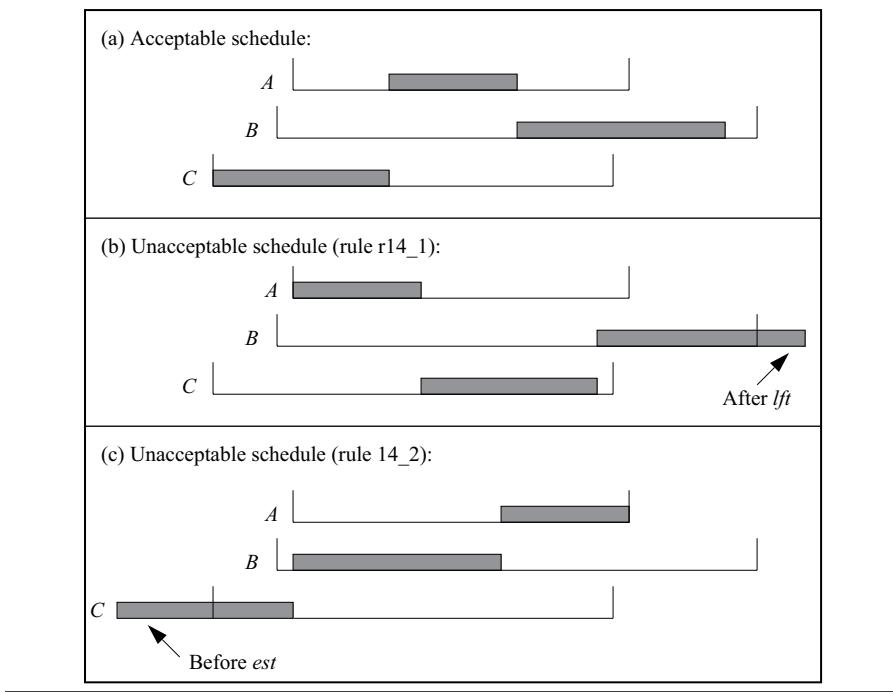


Figure 14.12 Sequencing in the disjunctive case: (a) an acceptable schedule; (b) A cannot be the first operation (rule r14_1); (c) A cannot be the last operation (rule 14_2).

critical sets that contain only two operations. Let us consider the precedence constraints that apply to the operations O_i of a critical set having more than two elements. From our original definition of a critical set, it is not possible for all of the operations in the set to be carried out simultaneously. At any one time, at least one operation in the set must either have finished or be waiting to start. This constraint provides the basis for some precedence relations. Let us denote the critical set by the symbol S , where S includes the operation A . Another set of operations that contains all elements of S apart from operation A will be denoted by the letter W . The two precedence rules that apply are as follows:

```

rule r14_3 /* informally stated; not Flex format */
  if ( $lft_i - est_j < (d_i + d_j)$  for every pair of operations  $\{O_i, O_j\}$  in set W
  /* condition that  $O_i$  cannot be performed after  $O_j$  has
  finished */
  and  $lft_i - est_A < d_A + d_i$  for every operation  $O_i$  in set W
  /* condition that  $O_i$  cannot be performed after A has
  finished) */
  then at least one operation in set W must have finished
  before A starts.

```

```

rule r14_4 /* informally stated; not Flex format */
if ( $lft_i - est_j < (d_i + d_j)$  for every pair of operations  $\{O_i, O_j\}$  in set W
/* condition that  $O_i$  cannot be performed after  $O_j$  has
finished */
and ( $lft_A - est_i < (d_A + d_i)$  for every operation  $O_i$  in set W
/* condition that A cannot be performed after  $O_i$  has
finished */
then A must finish before at least one operation in set W
starts.

```

The application of rule r14_3 is shown in Figure 14.13a. Operations B and C have to overlap (the first condition) and it is not possible for A to finish before one of

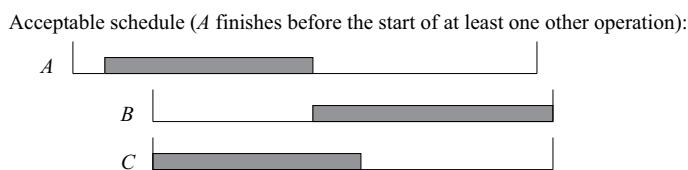
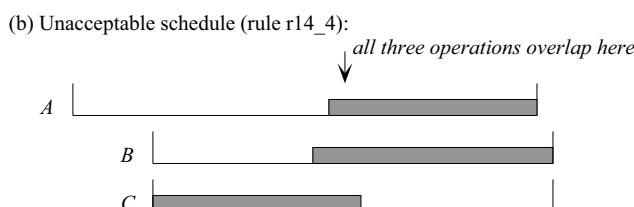
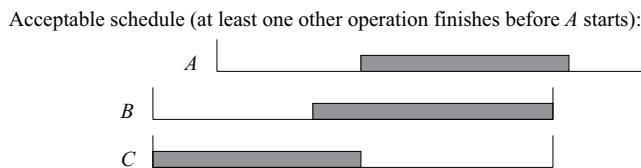
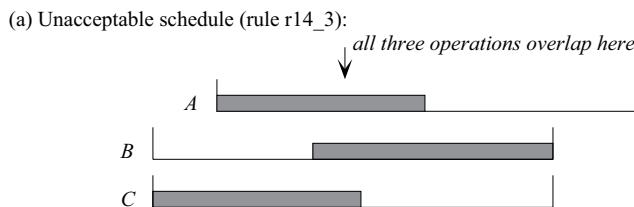


Figure 14.13 Sequencing in the nondisjunctive case.

either B or C has started (the second condition). Therefore, operation A *must* be preceded by at least one of the other operations. Note that, if this constraint is not possible either, then there is no feasible schedule. Overlap of all the operations is unavoidable in such a case but, since we are dealing with a critical set, there is insufficient resource to support this overlap.

Rule r14_4 is similar and covers the situation depicted in Figure 14.13b. Here operations B and C again have to overlap, and it is not possible to delay the start of A until after one of the other operations has finished. Under these circumstances, operation A *must* precede at least one of the other operations.

14.8.6 Updating Earliest Start Times and Latest Finish Times

If rule r14_1 or r14_3 has been fired, so we know that at least one operation must precede A , it may be possible to update the earliest start time of A to reflect this restriction, as shown in Figures 14.14a and 14.14b. The rule that describes this update is:

```
rule r14_5 /* informally stated; not Flex format */
  if some operations must precede A /* by rule r14_1 or r14_3
  */
  and [the earliest (est+d) of those operations] > estA
  then the new estA is the earliest (est+d) of those
  operations.
```

Similarly, if rule r14_2 or r14_4 has been fired, so we know that at least one operation must follow operation A , it may be possible to modify the *lft* for A , as shown in Figures 14.14a and 14.14c. The rule that describes this update is:

```
rule r14_6 /* informally stated; not Flex format */
  if some operations must follow A /* by rule r14_2 or r14_4
  */
  and [the latest (lft-d) of those operations] < lftA
  then the new lftA is the latest (lft-d) of those operations.
```

The new earliest start times and latest finish times can have an effect on subsequent operations. Consider the case of a factory that is assembling cars. Axle assembly must precede wheel fitting, regardless of any resource considerations. This sort of precedence relation, which is imposed by the nature of the task itself, is referred to as a *technological coherence constraint*. Suppose that, as a result of a rule like r14_5, the *est* for axle assembly is delayed to a new time est_a . If the axle assembly takes time d_a per car, then the new *est* for wheel-fitting (est_w) will be $est_a + d_a$, ignoring any time taken to move the vehicle between assembly stations.

However, a car plant is unlikely to be moving just one car through the assembly process, rather a whole batch of cars will need to be scheduled. These circumstances

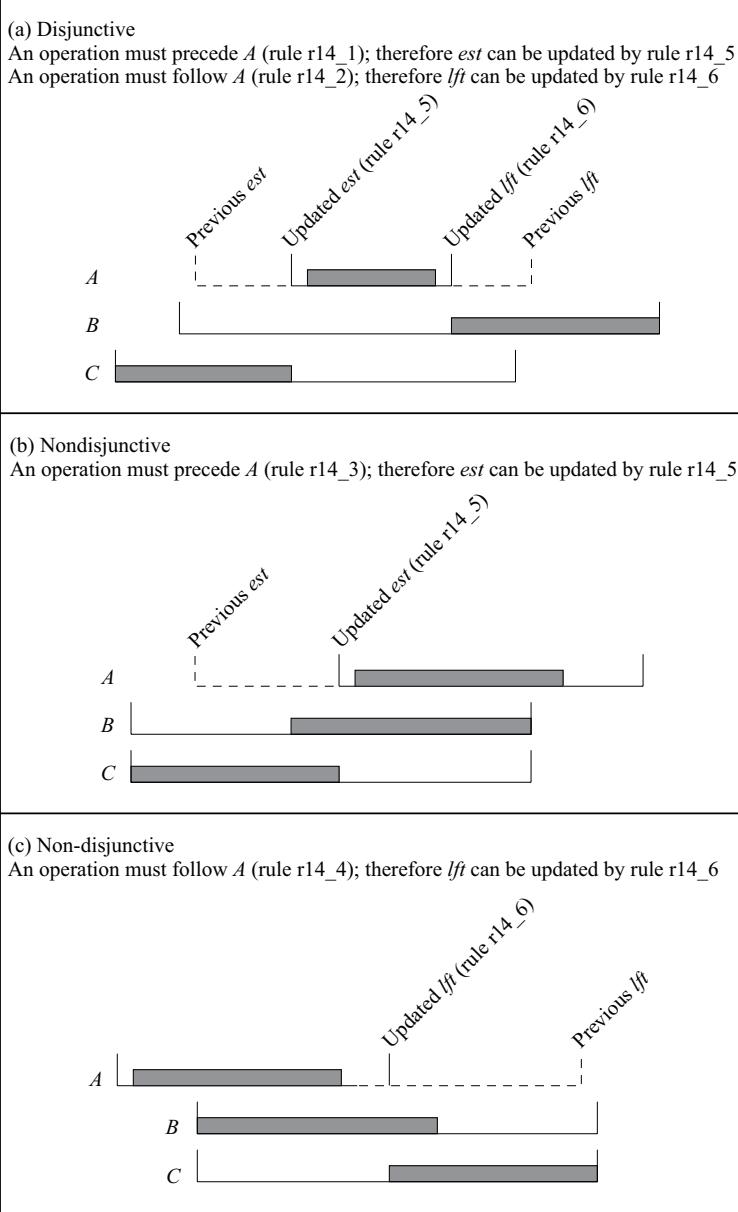


Figure 14.14 Updating earliest start times and latest finish times.

offer greater flexibility as the two assembly operations can overlap, provided that the order of assembly is maintained for individual cars. Two rules can be derived, depending on whether axle assembly or wheel fitting is the quicker operation. If wheel fitting is quicker, then a sufficient condition is for wheels to be fitted to the

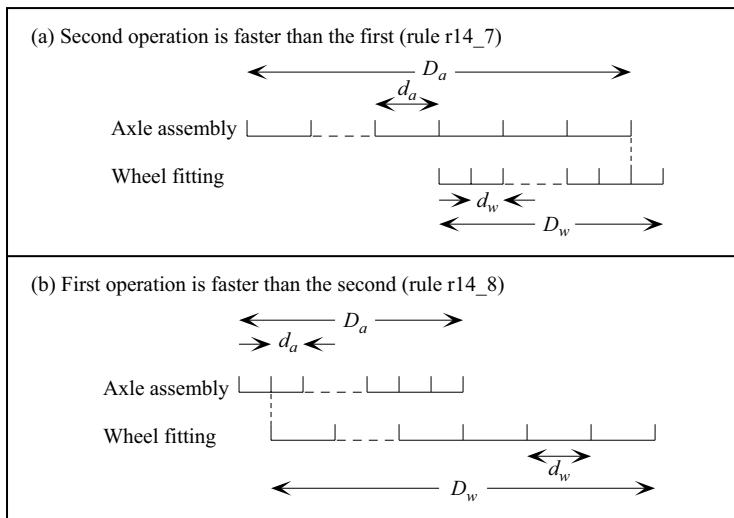


Figure 14.15 Overlapping operations in batch processing: (a) rule r14_7 applies if wheel fitting is faster than axle assembly; (b) rule r14_8 applies if axle assembly is faster than wheel fitting.

last car in the batch immediately after its axles have been assembled. This situation is shown in Figure 14.15a, and is described by the following rule:

```
rule r14_7 /* informally stated; not Flex format */
  if  $d_a > d_w$ 
    then  $est_w$  becomes  $est_a + D_a - (n-1)d_w$ 
    and  $lft_a$  becomes  $lft_w - d_w$ .
```

where n is the number of cars in the batch, D_a is nd_a , and D_w is nd_w . If axle assembly is the quicker operation, then wheel fitting can commence immediately after the first car has had its axle assembled (Figure 14.15b). The following rule applies:

```
rule r14_8 /* informally stated; not Flex format */
  if  $d_a < d_w$ 
    then  $est_w$  becomes  $est_a + d_a$ 
    and  $lft_a$  becomes  $lft_w - D_w + (n-1)d_a$ .
```

14.8.7 Applying Preferences

CBA can produce one of three possible outcomes:

- The constraints cannot be satisfied, given the allocated resources.
- A unique schedule is produced that satisfies the constraints.
- More than one schedule is found that satisfies the constraints.

In the case of the first outcome, the problem can be solved only if more resources are made available or the time constraints are slackened. In the second case, the scheduling problem is solved. In the third case, pairs of operations exist that can be sequenced in either order without violating the constraints. The problem then becomes one of applying *preferences* so as to find the most suitable order. Preferences are features of the solution that are considered desirable but, unlike constraints, they are not compulsory. Preferences can be applied by using fuzzy rules (Bel et al. 1989). Such a fuzzy rule-based system constitutes the decision-support module in Figure 14.10. (See Sections 3.4–3.6 for a discussion of fuzzy logic.)

The pairs of operations that need to be ordered are potentially large in number and broad in scope. It is, therefore, impractical to produce a rule base that covers specific pairs of operations. Instead, rules for applying preferences usually make use of local variables (see Section 2.6), so that they can be applied to different pairs of operations. The rules may take the form:

```
rule r14_9 /* informally stated; not Flex format */
  if (x of Operation1) > (x of Operation2)
  then Operation1 precedes Operation2.
```

Here, any pair of operations can be substituted for Operation1 and Operation2, but the attribute x is specified in a given rule. A typical expression for x might be the duration of its available time window (i.e., *lft – est*). The rules are chosen so as to cover some general guiding principles or goals, such as:

- Maximize overall slack time.
- Perform operations with the least slack time first.
- Give preference to schedules that minimize resource utilization.
- Avoid tool changes.

In OPAL, each rule is assigned a degree of relevance *R* with respect to each goal. Given a goal or set of goals, some rules will tend to favor one ordering of a pair of operations, while others will favor the reverse. A consensus is arrived at by a scoring (or “voting”) procedure. The complete set of rules is applied to each pair of operations that need to be ordered, and the scoring procedure is as follows:

- The degree to which the condition part of each rule is satisfied is represented using fuzzy sets. Three fuzzy sets are defined (*true*, *maybe*, and *false*), and the degree of membership of each is designated μ_t , μ_m , and μ_f . Each membership is a number between 0 and 1, such that for every rule:

$$\mu_t + \mu_m + \mu_f = 1$$

- Every rule awards a score to each of the three possible outcomes (a precedes b , b precedes a , or no preference). These scores (V_{ab} , V_{ba} , and $V_{no_preference}$) are determined as follows:

$$V_{ab} = \min(\mu_t, R)$$

$$V_{ba} = \min(\mu_f, R)$$

$$V_{no_preference} = \min(\mu_m, R)$$

- The scores for each of the three possible outcomes are totaled across the whole rule set. The totals can also be normalized by dividing the sum of the scores by the sum of the R values for all rules. Thus:

$$\text{Total score for } ab = \Sigma(V_{ab}) / \Sigma(R)$$

$$\text{Total score for } ba = \Sigma(V_{ba}) / \Sigma(R)$$

$$\text{Total score for no preference} = \Sigma(V_{no_preference}) / \Sigma(R)$$

The scoring procedure can have one of three possible outcomes:

- One ordering is favored over the other. This outcome is manifested by $\Sigma(V_{ab}) > \Sigma(V_{ba})$ or vice versa.
- Both orderings are roughly equally favored, but the scores for each are low compared with the score for the impartial outcome. This outcome indicates that there is no strong reason for preferring one order over the other.
- Both orderings are roughly equally favored, but the scores for each are high compared with the score for the impartial outcome. Under these circumstances, there are strong reasons for preferring one order, but there are also strong reasons for preferring the reverse order. In other words, there are strong conflicts between the rules.

14.8.8 Using Constraints and Preferences

OPAL makes use of a supervisor module that controls the CBA and DS modules (Figure 14.10). In OPAL and other scheduling systems, CBA is initially applied in order to eliminate all those schedules that cannot meet the time and resource constraints. A preferred ordering of operations is then selected from the schedules that are left. As we have seen, the DS module makes this selection by weighting the rules of optimization, applying these rules, and selecting the order of operations that obtains the highest overall score.

If the preferences applied by the DS module are not sufficient to produce a unique schedule, the supervising module calls upon the CBA and DS modules repeatedly until a unique solution is found. The supervising module stops the process when an acceptable schedule has been found, or if it cannot find a workable schedule.

There is a clear analogy between the two stages of scheduling (CBA and the application of preferences) and the stages of materials selection (see Section 13.8). In the case of materials selection, constraints can be applied by ruling out all materials that fail to meet a numerical specification. The remaining materials can then be put into an order of preference, based upon some means of comparing their performance scores against various criteria, where the scores are weighted according to a measure of the perceived importance of the properties.

14.9 Replanning and Reactive Planning

The discussion so far has concentrated on predictive planning, that is, building a plan that is to be executed at some time in the future. Suppose now that, while a plan is being executed, something unexpected happens, such as a machine breakdown. In other words, the actual world state deviates from the expected world state. A powerful capability under these circumstances is to be able to replan, that is, to modify the current plan to reflect the current circumstances. Systems that are capable of planning or replanning in real time, in a rapidly changing environment, are described as *reactive* planners. Such systems monitor the state of the world during execution of a plan and are capable of revising the plan in response to their observations. Since a reactive planner can alter the actions of machinery in response to real-time observations and measurements, the distinction between reactive planners and control systems (Chapter 15) is blurred.

To illustrate the distinction between predictive planning, replanning, and reactive planning, let us consider an intelligent robot that is carrying out some gardening. It may have planned in advance to mow the lawn and then to prune the roses (predictive planning). If it finds that someone has borrowed the mower, it might decide to mow after pruning, by which time the mower may have become available (replanning). If a missile is thrown at the robot while it is pruning, it may choose to dive for cover (reactive planning).

A system called SONIA was devised to be capable of both predictive and reactive planning of factory activity (Collinot et al. 1988). This system offers more than just a scheduling capability, as it has some facilities for selecting which operations will be scheduled. However, it assumes that a core of essential operations is pre-selected. SONIA brings together many of the techniques that we have seen previously. The operations are ordered by the application of constraints and preferences (Section 14.8) in order to form a predictive plan. During execution of the plan,

the observed world state may deviate from the planned world state. Some possible causes of the deviation might be:

- machine failure;
- personnel absence;
- arrival of urgent new orders;
- ill-conception of the original plan.

Under these circumstances, SONIA can modify its plan or, in an extreme case, generate a new plan from scratch. The particular type of modification chosen is largely dependent on the time available for SONIA to reason about the problem. Some possible plan modifications might be:

- cancel, postpone, or curtail some operations in order to bring some other operations back on schedule;
- reschedule to use any slack time in the original plan;
- reallocate resources between operations;
- reschedule a job, comprising a series of operations, to finish later than previously planned;
- delay the whole schedule.

A final point to note about SONIA is that it has been implemented as a blackboard system (see Chapter 10). The versatility of the blackboard architecture is demonstrated by the variety of applications in which it is used. An application in data interpretation was described in Chapter 12, whereas here it is used for predictive planning. SONIA uses the blackboard to represent both the planned and observed world states. Separate agents perform the various stages of predictive planning, monitoring, and reactive planning. In fact, SONIA uses two blackboards: one for representing information about the shop floor and a separate one for its own internal control information.

14.10 Summary

This chapter began by defining a classical planner as one that can derive a set of operations to take the world from an initial state to a goal state. The initial world state is a “snapshot” that is assumed not to alter except as a result of the execution of the plan. While being useful in a wide range of situations, classical planners are of limited use when dealing with continuous processes or a rapidly changing environment. In contrast, reactive planners can respond rapidly to unexpected events.

Scheduling is a special case of planning, where the operators are known in advance and the task is to allocate resources to each and to determine when they

should be applied. Scheduling is particularly important in the planning of manufacturing processes.

A simple classical planner similar to STRIPS has been described. More sophisticated features that can be added to extend the capabilities of a planning system were then described and are summarized as follows:

■ *World modeling*

Unlike STRIPS, the Prolog program shown in Section 14.3.3 does not explicitly update its world model as operators are selected. Proper maintenance of a world model ensures that any side effects of a plan are recorded along with intended effects.

■ *Deductive rules*

Deductive rules permit the deduction of effects that are additional to those explicitly included in the operator representation. Because they do not form part of the operator representation, they can allow different effects to be registered depending on the current context.

■ *Hierarchical planning*

STRIPS may commit itself to a particular problem-solving path too early, with the result that it must backtrack if it cannot complete the plan that it is pursuing. Hierarchical planners can plan at different levels of abstraction. An abstract (or “skeleton”) plan is formed first, such as to build a house by digging foundations, then building walls, and finally putting on a roof. The detailed planning might include the precise shape, size, and placement of the timber. The abstract plan restricts the range of possibilities of the detailed planning.

■ *Nonlinearity*

Linear planners such as STRIPS make the assumption that it does not matter in which order the subgoals for a particular goal are satisfied. Partial ordering of operations is a form of nonlinear planning in which the ordering of operations is postponed until either more information becomes available or a decision is forced. In some cases, it may be possible for operations to be scheduled to run in parallel.

■ *Planning variables*

The use of variables also allows postponement of decisions. Plans can be generated using variables that have not yet been assigned a value. For instance, we might plan to go somewhere without specifying where. The instantiation of somewhere may become determined later, and we have saved the effort of considering all of the possibilities in the meantime.

■ *Constraints*

The use of planning variables is more powerful still if we can limit the possible instantiations by applying constraints on the values that a variable can take. CBA can be used to reduce the number of possible plans, or even to find a unique plan that meets the constraints.

■ *Preferences*

If CBA yields more than one viable plan or schedule, preferences can be applied in order to select between the alternatives.

■ *Replanning*

The ability to modify a plan in the light of unexpected occurrences was discussed.

Hierarchical plans, partially ordered plans, and the use of planning variables are all means of postponing commitment to a particular plan. This approach is known as the *principle of least commitment*.

Systems that have only some of the aforementioned features are adequate in many situations. A few specific applications have been considered in this chapter, but much of the research effort in planning has been concerned with building general-purpose planning systems that are domain-independent. The purpose of such systems is to allow knowledge relevant to any particular domain to be represented, rather like an expert system shell (see Chapters 1 and 11).

Further Reading

- Geffner, H., and B. Bonet. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, San Rafael, CA.
- Ghallab, M., D. Nau, and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann, San Mateo, CA.
- Ghallab, M., D. Nau, and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press, New York, NY.
- van Wezel, W., R. J. Jorna, and A. M. Meystel. 2006. *Planning in Intelligent Systems: Aspects, Motivations, and Methods*. John Wiley & Sons, Hoboken, NJ.
- Vlahavas, I., and D. Vrakas. 2004. *Intelligent Techniques for Planning*. IGI Publishing, Hershey, PA.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 15

Systems for Control

15.1 Introduction

The application of intelligent systems to control has far-reaching implications for manufacturing, robotics, and other areas of engineering. The control problem is closely allied with some of the other applications that have been discussed so far. For instance, a controller of manufacturing equipment will have as its aim the implementation of a manufacturing plan (see Chapter 14). It will need to interpret sensor data, recognize faults (see Chapter 12), and respond to them. Similarly, it will need to replan the manufacturing process in the event of breakdown or some other unexpected event, that is, to plan reactively (Section 14.9). Indeed, some researchers treat automated control as a loop of plan generation, monitoring, diagnosis, and replanning (Micalizio 2009; Bennett 1987).

The systems described in Chapters 12 to 14 gather data describing their environment and make decisions and judgments about those data. Controllers are distinct in that they can go a step further by altering their environment. They may do this actively by sending commands to the hardware or passively by recommending to a human operator that certain actions be taken. The passive implementation assumes that the process decisions can be implemented relatively slowly.

Control problems appear in various guises, and different techniques may be appropriate in different cases. For example, a temperature controller for an industrial oven may modify the current flowing in the heating coils in response to the measured temperature, where the temperature may be registered as a potential difference across a thermocouple. This is *low-level* control, in which a rapid response is required but little intelligence is involved. In contrast, *high-level* or *supervisory* control takes a wider view of the process being controlled. For example, in the control of the manufacturing process for a steel component, an oven temperature may be just

one of many parameters that need to be adjusted. High-level control requires more intelligence, but there is often more time available in which to make the decisions.

The examples of control discussed in the preceding paragraph implicitly assumed the existence of a model of the system being controlled. In building a temperature controller, it is known that an increased current will raise the temperature, that this temperature is registered by the thermocouple, and that there will be a time lag between the two. The controller is designed to exploit this model. There may be some circumstances where no such model exists, or the model is too complex to represent. The process under control can then be thought of as a black box, whose input is determined by the controller, and whose output we wish to regulate. As it has no other information available to it, the controller must learn how to control the black box through trial and error. In other words, it must construct a model of the system through experience. We will discuss two approaches, the BOXES algorithm (Section 15.7) and neural networks (Section 15.8).

As well as drawing a distinction between low-level and high-level control, we can distinguish between *adaptive* and *servo* control. The aim of an adaptive controller is to maintain a steady state. In a completely stable environment, an adaptive controller would need to do nothing. In the real world, an adaptive controller must adapt to changes in the environment that may be brought about by the controlled process itself or by external disturbances. A temperature controller for an industrial oven is an adaptive controller, the task of which is to maintain a constant temperature. It must meet this challenge in spite of disturbances such as the oven door opening, large thermal masses being inserted or removed, fluctuations in the power supply, and changes in the temperature of the surroundings. Typically, it will achieve its task by using negative feedback (see Section 15.2).

A servo controller is designed to drive the output of the plant from a starting value to a desired value. Choosing a control action to achieve the desired output requires that a prediction be made about the future behavior of the controlled plant. This requirement, again, relies on a model of the controlled plant. Often, a high-level controller is required to decide upon a series of servo control actions. This situation is known as *sequence* control. For instance, alloyed components are normally taken through a heat-treatment cycle. They are initially held at a temperature close to the melting temperature, then they are rapidly quenched to room temperature, and finally they are “aged” at an intermediate temperature.

15.2 Low-Level Control

15.2.1 Open-Loop Control

The open-loop control strategy is straightforward: given a control requirement, the controller simply sends a control action to the plant (Figure 15.1a). The

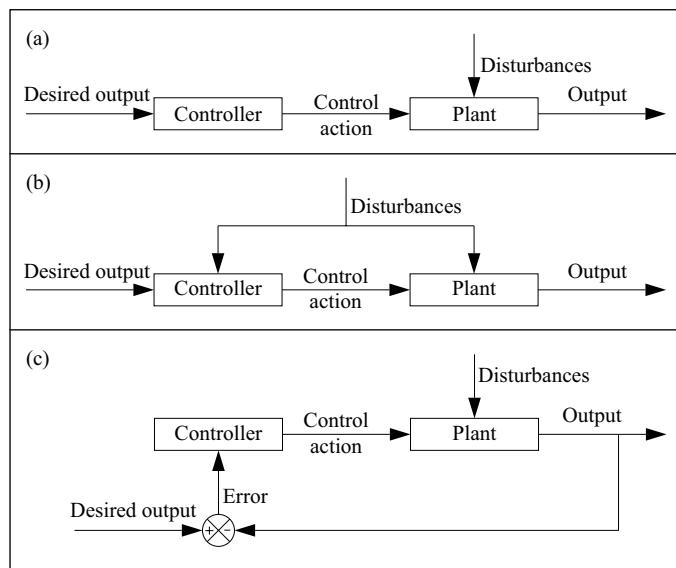


Figure 15.1 Three alternative strategies for control: (a) open loop; (b) feedforward; (c) feedback (closed loop).

controller must have a model of the relationship between the control action and the behavior of the plant. The accuracy of control is solely dependent on the accuracy of the model, and no checks are made to ensure that the plant is behaving as intended. An open-loop temperature controller, for example, would send a set current to the heating coils of an industrial oven in order to achieve an intended temperature. In order to choose an appropriate current, it would have an implicit model of the thermal capacity of the oven and of the rate of heat loss. This strategy cannot correct for any disturbance to the plant, which is the oven in this example.

15.2.2 Feedforward Control

The feedforward strategy takes account of disturbances to the plant by measuring the disturbances and altering the control action accordingly (Figure 15.1b). In the example of the temperature controller, fluctuations in the electrical power supply might be monitored. The nominal current sent to the coils may then be altered to compensate for fluctuations in the actual current. Note that the disturbance is monitored and not the controlled variable itself, that is, the power supply is monitored for fluctuations, not the oven temperature. Any disturbances that are not measured are not taken into account. As with open-loop control, a model of the plant is needed in order to calculate the impacts of the control action and the measured disturbance.

15.2.3 Feedback Control

Feedback control is the most common control strategy in both high-level and low-level control applications. The measured output from the plant is compared with the required value, and the difference or *error* is used by the controller to regulate its control action. As shown in Figure 15.1c, the control action affects the plant and the plant output affects the controller, thereby forming a closed loop. Hence the strategy is also known as *closed-loop control*. This strategy takes account of all disturbances, regardless of how they are caused and without having to measure them directly. There is often a lag in the response of the controller, as corrections can only be made after a deviation in the plant output has been detected. For example, the temperature of the industrial oven considered previously would need to drift slightly before a corrective adjustment is made to the electrical power.

15.2.4 First- and Second-Order Models

It has already been emphasized that a controller can be built only if we have a model, albeit a simple one, of the plant being controlled. For low-level control, it is often assumed that the plant can be adequately modeled on first- or second-order linear differential equations. Let us denote the input to the plant (which may also be the output of the controller) by the letter x , and the output of the plant by the letter y . In an industrial oven, x would represent the current flowing and y would represent the oven temperature. The first-order differential equation would be:

$$\tau \frac{dy}{dt} + y = k_1 x \quad (15.1)$$

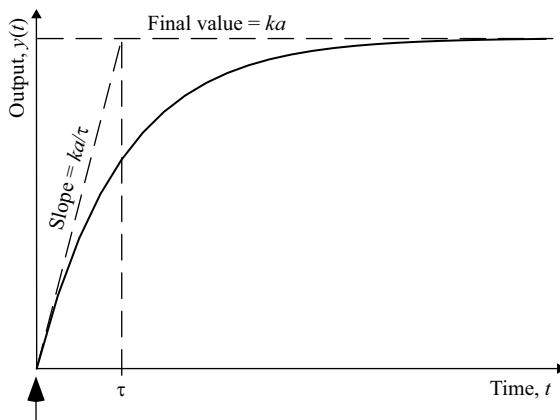
and the second-order differential equation:

$$\frac{d^2 y}{dt^2} + 2\zeta\omega_n \frac{dy}{dt} + \omega_n^2 y = k_2 x \quad (15.2)$$

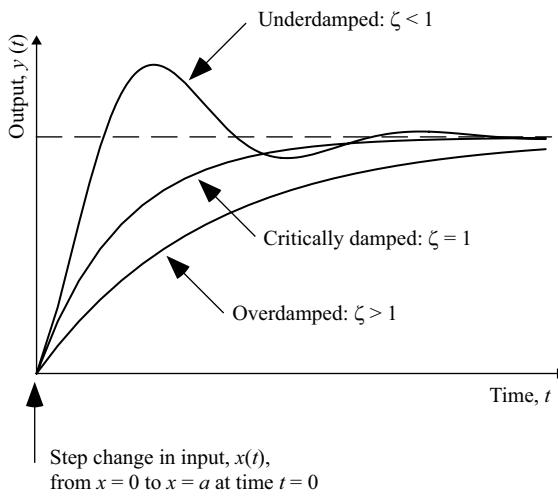
where τ , k_1 , k_2 , ζ , and ω_n are constants for a given plant, and t represents time. These equations can be used to tell us how the output y will respond to a change in the input x .

Figures 15.2a and 15.2b show the response to a step change in x for a first- and second-order model, respectively. In the first-order model, the time for the controlled system to reach a new steady state is determined by τ , which is the *time constant* for the controlled system.

In the second-order model, the behavior of the controlled system is dependent on two characteristic constants. The *damping ratio* ζ determines the rate at which y will approach its intended value, and the *undamped natural angular frequency* ω_n determines the frequency of oscillation about the final value in the underdamped case (Figure 15.2b).



(a)



(b)

Figure 15.2 (a) First-order response to a step change; (b) second-order response to a step change.

15.2.5 Algorithmic Control: The PID Controller

Control systems may be either analog or digital. In analog systems, the output from the controller varies continuously in response to continuous changes in the controlled variable. This book will concentrate on digital control, in which the data are sampled and discrete changes in the controller output are calculated accordingly.

There are strong arguments to support the view that low-level digital control is best handled by algorithms, which can be implemented either in electronic hardware or by procedural coding. Such arguments are based on the observation that low-level control usually requires a rapid response, but little or no intelligence. This view would tend to preclude the use of intelligent systems. In this section, we will look at a commonly used control algorithm, and in Section 15.6 we will examine the possibilities for improvement by using fuzzy logic.

It was noted in Section 15.2.3 that feedback controllers determine their control action on the basis of the error e , which is the difference between the measured output y from the plant at a given moment and the desired value, known as the reference r . The control action is often simply the assignment of a value to a variable, such as the electrical current for an industrial oven. This *action variable* is usually given the symbol u . The action variable is sometimes known as the *control* variable, although this terminology can lead to confusion with the *controlled* variable y .

In a simple controller, u may be set in proportion to e . A more sophisticated approach is adopted in the PID (proportional + integral + derivative) controller. The value for u that is generated by a PID controller is the sum of three terms:

- P —a term proportional to the error e ;
- I —a term proportional to the integral of e with respect to time;
- D —a term proportional to the derivative of e with respect to time.

Thus, the value assigned to the action variable u by a PID controller would ideally be given by:

$$u = K_p \left(e + \frac{1}{\tau_i} \int e dt + \tau_d \frac{de}{dt} \right) \quad (15.3)$$

where K_p , τ_i , and τ_d are adjustable parameters of the controller that can be set to suit the characteristics of the plant being controlled. K_p is the *proportional gain*, τ_i is the *integral time*, and τ_d is the *derivative time*. If the values for e are sampled at time intervals Δt , then the integral and derivative terms must be approximated:

$$u_k = K_p \left(e_k + \frac{\Delta t}{\tau_i} \sum_{n=0}^k e_n + \tau_d \frac{e_k - e_{k-1}}{\Delta t} \right) \quad (15.4)$$

where k is the sample number, such that time $t = k\Delta t$, and u_k and e_k are the action variable and the error, respectively, at sample k . The role of the P term is intuitively obvious: the greater the error, the greater the control action that is needed. Through the I term, the controller output depends on the accumulated historical values of the error. This term is used to counteract the effects of long-term disturbances on the controlled plant. The magnitude of the D term depends on the rate of change

of the error. This term allows the controller to react quickly to sharp fluctuations in the error. The D term is low for slowly varying errors, and zero when the error is constant. In practice, tuning the parameters of a PID controller can be difficult. A commonly used technique is the Ziegler–Nichols method (Franklin et al. 2020). Genetic algorithms offer an alternative approach in which the parameter-tuning is treated as an optimization task (Passow et al. 2008; AltInten et al. 2008; Zhang et al. 2009). Passow et al.’s system was demonstrated on the control of a model helicopter, where the fitness was the inverse sum of the squared errors between the helicopter’s setpoint heading and its actual heading. Their system used physical measurements from a tethered helicopter, rather than a software model, and it showed that the GA tuning method could outperform a human engineer.

15.2.6 Bang-Bang Control

Bang-bang controllers rely on switching the action variable between its upper and lower limits, with intermediate values disallowed. As previously noted, servo control involves forcing a system from one state to a new state. The fastest way of doing this, known as time-optimal control, is by switching the action variable from one extreme to another at precalculated times, a method known as *bang-bang* control. Two extreme values are used, although Sripada et al. (1987) also allow a final steady-state value for the action variable (Figure 15.3).

Consider the control of an industrial electric oven. As soon as the new (increased) temperature requirement is known, the electrical current is increased to the maximum value sustainable until the error in the temperature is less than a critical value e^* . The current is then dropped to its minimum value (i.e., zero) for time Δt , before being switched to its final steady-state value. There are, therefore, two parameters that determine the performance of the controller, e^* and Δt . Figure 15.3 shows the effects of errors in these two parameters.

15.3 Requirements of High-Level (Supervisory) Control

Our discussion so far has concentrated on fairly straightforward control tasks, such as maintaining the temperature of an industrial oven or driving the temperature to a new set point. Both are examples of low-level control, where a rapid response is needed but scarcely any intelligence. In contrast, high-level control may be a complex problem concerning decisions about the actions to take at any given time. This problem may involve aspects of both adaptive and servo control. An important example of high-level control is the control of a manufacturing process. Control decisions at this level concern diverse factors such as choice of reactants and raw materials; conveyor belt speeds; rates of flow of solids, liquids, and gases; temperature and pressure cycling; and batch transport.

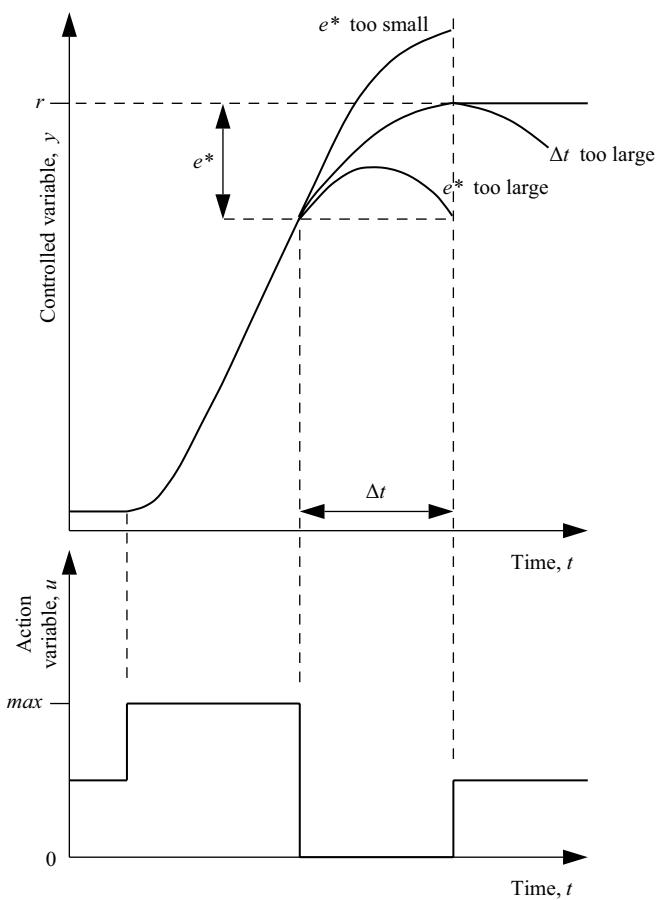


Figure 15.3 Bang-bang servo control. (Derived from Sripada, N.R. et al., 1987.)

Leitch et al. (1991) have identified six key requirements for a real-time supervisory controller:

- ability to make decisions and act upon them within a time constraint;
- handling asynchronous events—the system must be able to break out of its current set of operations to deal with unexpected occurrences;
- temporal reasoning, that is, the ability to reason about time and sequences;
- reasoning with uncertainty;
- continuous operation;
- multiple sources of knowledge.

We have come across the third requirement in the context of planning (Chapter 14), and the last three requirements as part of monitoring and diagnosis (Chapter 12). Only the first two requirements are new.

15.4 Blackboard Maintenance

One of the requirements listed in the previous section was for multiple sources of knowledge. They are required because high-level control may have many sources of input information and many subtasks to perform. It is no surprise, therefore, that the *blackboard model* (see Chapter 10) is chosen for many control applications (Oh and Quek 2001).

Leitch et al. (1991) point out that, because continuous operation is required, a mechanism is needed for ensuring that the blackboard does not contain obsolete information. They meet this need by tagging blackboard information with the time of its posting. As time elapses, some information may remain relevant, some may gradually lose accuracy, and some may suddenly become obsolete. There are several ways of representing the lifetime of blackboard information:

1. A default lifetime for all blackboard information may be assumed. Any information that is older than this default may be removed. A drawback of this approach is that deductions made from information that has become obsolete may remain on the blackboard.
2. At the time of posting to the blackboard, individual items of information may be tagged with an expected lifetime. Suppose that item A is posted at time t_A with expected lifetime l_A . If item B is deduced from A at time t_B , where $t_B < t_A + l_A$, then the lifetime of B , l_B , would be $t_A + l_A - t_B$.
3. Links between blackboard items are recorded, showing the interdependencies between items. Figure 15.4 illustrates the application of this approach to the control of a boiler, using rules borrowed from Chapter 2. There is no need to record the expected lifetimes of any items, as all pieces of blackboard information are ultimately dependent on sensor data, which are liable to change. When changes in sensor values occur, updates are rippled through the dependent items on the blackboard. In the example shown in Figure 15.4, as soon as the flow rate fell, *flow rate high* would be removed from the blackboard along with the inferences *steam escaping* and *steam outlet blockage* and the resulting control action. All other information on the blackboard would remain valid.

The third technique is a powerful way of maintaining the blackboard integrity, since it avoids unnecessary reevaluation of information. This benefit is counterbalanced by the additional complexity of the blackboard and the computational load of maintaining it. These problems can be minimized by careful partitioning of the blackboard into levels of abstraction (e.g., from low-level statements about sensor values to high-level analysis of trends and policy) and subject groupings (e.g., low-level data about a conveyor belt are kept separate from low-level information about a boiler).

The control actions on the blackboard in Figure 15.4 are shown with a tag indicating whether or not the action has been carried out. The tag is necessary because there is a time lag between an action being taken and a resultant change in the

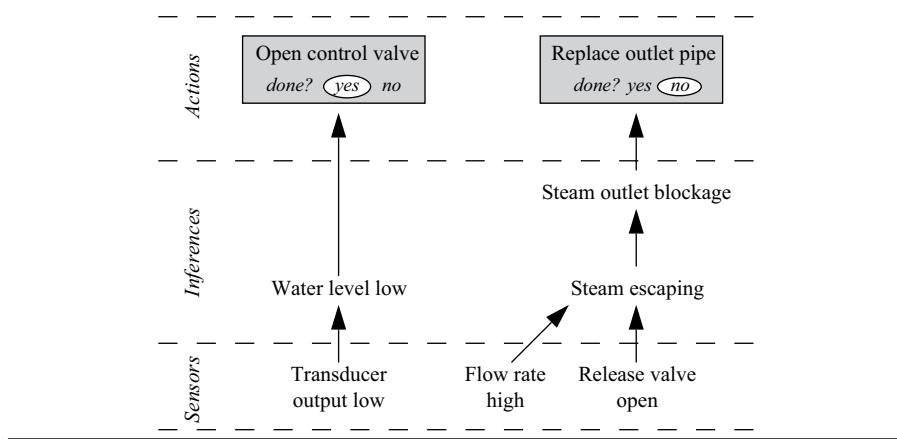


Figure 15.4 Storing dependencies between items of information on the blackboard.

sensor data. When the action is first added to the blackboard, the tag is set to “not yet done.” This tag is changed as soon as the action is carried out. The action is not removed at this stage, as the agent that generated it would simply generate it again. Instead, the control action remains on the blackboard until the supporting evidence is removed, when the sensor reading changes.

15.5 Time-Constrained Reasoning

An automatic control system must be capable of operating in real time. This requirement does not necessarily mean fast, but merely *fast enough*, as described in the following definition of real-time performance (Laffey et al. 1988):

[Real-time performance requires] the system responds to incoming data at a rate as fast or faster than it is arriving.

This definition conveys the idea that real-time performance is the ability to keep up with the physical world to which the system is interfaced. The RESCU process control system, applied to a chemical plant, receives data in blocks at 5-minute intervals (Leitch et al. 1991). Therefore, it has 5 minutes in which to respond to the data in order to achieve real-time performance. This example illustrates a system that is fast enough, but not particularly fast. The rate of arrival of data is one factor in real-time control, but there may be other reasons why decisions have to be made within a specified time frame. For instance, the minimum time of response to an overheated

reactor is not dependent on the rate of updating the temperature data, but on the time over which overheating can be tolerated.

In order to ensure that a satisfactory (though not necessarily optimum) solution is achieved within the available timescale, an intelligent system needs to be able to schedule its activities appropriately. This need is not straightforward to meet, as the time taken to solve a problem cannot be judged accurately in advance. A conservative approach would be to select only those reasoning activities that are judged able to be completed comfortably within the available time. However, this approach may lead to an unsatisfactory solution while failing to use all the time available for reasoning.

Some of the techniques that have been applied to ensure a satisfactory response within the time constraints are described in the following subsections.

15.5.1 Prioritization of Processes

The processes in the RESCU process control system are partitioned on two levels. First, the processes of the overall system are divided into *operator, communications, log, monitor*, and *knowledge-based system*. Second, the knowledge-based system is implemented as a blackboard system (Chapter 10) whose knowledge is divided among its agents.

Each of the five processes at the system level is assigned a priority number between 1 and 6. The communications and monitoring processes are given the highest priorities, so these processes can cause the temporary suspension of the other (lower priority) activities. The application of priorities to processes is similar to the application of priorities to rules in other systems (see Section 2.8.2).

As RESCU uses an early form of blackboard system, its agents are not truly opportunistic. Prioritization of agents is implemented using a dedicated control agent. The control agent checks the preconditions of other agents and assigns them a priority rating. Agents that are not activated, despite being applicable, have their priority iteratively increased, thereby ensuring that all applicable agents are activated eventually.

In contrast, Oh and Quek (2001) adhere to the principle that agents in a blackboard system should be opportunistic rather than explicitly scheduled, but the opportunism is constrained. The required processes are dynamically prioritized, and the number of available agents is restricted in order to meet the time constraints.

15.5.2 Approximation

The use of priorities, described in the previous subsection, ensures that the *most important* tasks are completed within the time constraints, and less important tasks are completed only if time permits. An alternative approach is to ensure that an *approximate* solution is obtained within the time constraints, and the solution is

embellished only if time permits (Lesser et al. 1988). Three aspects of a solution that might be sacrificed to some degree are as follows:

- Completeness
- Precision
- Certainty

Loss of completeness means that some aspects of the solution are not explored. Loss of precision means that some parameters are determined less precisely than they might have been. (The maximum precision is determined by the precision of the input data.) Loss of certainty means that some evidence in support of the conclusion has not been evaluated, or alternatives have been ignored. Thus, there is a trade-off between the quality of a solution and the time taken to derive it.

One approach to approximation would be to make a rough attempt at solving the problem initially and to use any time remaining to refine the solution incrementally. Provided that sufficient time was available to at least achieve the rough solution, a solution of some sort would always be guaranteed. In contrast, the approach adopted by Lesser et al. is to plan the steps of solution generation so that there is just the right degree of approximation to meet the deadline (Lesser et al. 1988).

Lesser et al. distinguish between *well-defined* and *ill-defined* approximations. According to their definition, well-defined approximations have the following properties:

- a predictable effect on the quality of the solution and the time taken to obtain it;
- graceful degradation, that is, the quality of the solution decreases smoothly as the amount of approximation is increased;
- loss of precision is not accompanied by loss of accuracy. If, for example, a reactor temperature is determined to lie within a certain temperature range, then this range should straddle the value that would be determined by more precise means.

Lesser et al. recommend that ill-defined approximations be used only as a last resort, when it is known that a well-defined approximation is not capable of achieving a solution within the available time. They consider six strategies for approximation, all of which they consider to be well defined. These strategies are classified into three groups: *approximate search*, *data approximations*, and *knowledge approximations*. These classifications are described in the following three subsections.

15.5.2.1 Approximate Search

Two approaches to pruning the search tree, that is, reducing the number of alternatives to be considered, are elimination of corroborating evidence and elimination of competing interpretations.

1. Eliminating corroborating evidence

Once a hypothesis has been generated, and perhaps partially verified, time can be saved by dispensing with further corroborating evidence. This streamlining

will have the effect of reducing the certainty of the solution. For corroborating data to be recognized as such, it must be analyzed to a limited extent before being discarded.

2. *Eliminating competing interpretations*

Elimination of corroborating evidence is a means of limiting the input data, whereas elimination of competing interpretations limits the output data. Solutions that have substantially lower certainties than their alternatives can be eliminated. If it is recognized that some solutions will have a low certainty regardless of the amount of processing that is carried out on them, then these solutions can be eliminated before they are fully evaluated. The net result is a reduced level of certainty of the final solution.

15.5.2.2 *Data Approximations*

Time can be saved by cutting down the amount of data considered. *Incomplete event processing* and *cluster processing* are considered here, although the elimination of corroborating evidence (see the previous subsection) might also be considered in this category.

1. *Incomplete event processing*

This approximation technique is, in effect, a combination of prioritization (see Section 15.5.1) and elimination of corroborating evidence. Suppose that a chemical reactor is being controlled, and that data are needed regarding the temperature and pressure. If the temperature has the higher priority, then any data that support the estimation of the pressure can be ignored in order to save time. The result is a less complete solution.

2. *Cluster processing*

Time can be saved by grouping together data items that are related and examining the overall properties of the group rather than the individual data items. For instance, the temperature sensors mounted on the walls of a reactor chamber might be clustered. Then, rather than using all of the readings, only the mean and standard deviation might be considered. This approach may lead to a loss of precision, but the certainty may be increased owing to the dilution of erroneous readings.

15.5.2.3 *Knowledge Approximations*

Changes can be made to the knowledge base in order to speed up processing. Two possible approaches are as follows:

1. *Knowledge adaptation to suit data approximations*

This approach is not really a technique in its own right, but simply a recognition that data approximations (see the previous subsection) require a modified knowledge base.

2. Eliminating intermediate steps

As noted in Section 12.3, shallow knowledge represents a means of bypassing the steps of the underlying deep knowledge. Therefore, it has potential for time saving. However, it may lead to a loss of certainty, as extra corroborating evidence for the intermediate steps might have been available. Shallow knowledge is also less adaptable to new situations.

15.5.3 Single and Multiple Instantiation

Single instantiation and multiple instantiation of variables in a rule-based system are described in Section 2.7.1. Under multiple instantiation, a single rule, fired only once, finds all sets of instantiations that satisfy the condition and then performs the conclusion on each. Under single instantiation, a separate rule firing is required for each set of instantiations. The choice between these inference engines can affect the order in which the conclusions are drawn, as shown in the examples in Section 2.7.1 and Section 12.5.3.

In time-constrained control applications, the choice between multiple instantiation and repeated single instantiation can be critical. Consider the case of an automatic controller for a telecommunications network (Hopgood 1994). Typically, the controller will receive statistics describing the network traffic at regular intervals Δt . Upon receiving the statistics, the controller must interpret the data, choose appropriate control actions, and carry out those actions before the next set of statistics arrives. Typically, there is a cycle of finding overloaded routes, planning alternative routes through the network, and executing the plan. The most efficient way of carrying out these tasks is by multiple instantiation, as the total duration of the find-plan-execute cycle for all routes is smaller (Figure 15.5). However, the controller must finish within time Δt . The more overloaded links exist, the less likely it is that

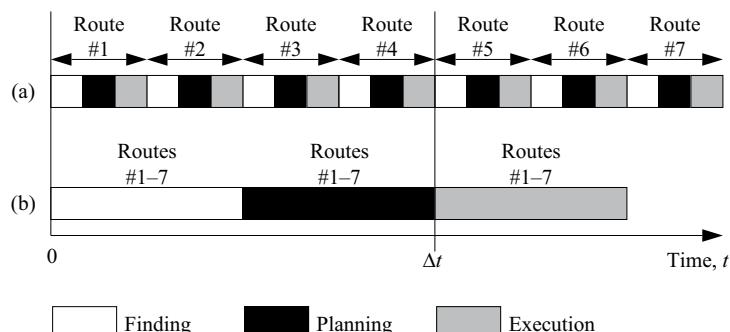


Figure 15.5 Controlling a telecommunications network within a time constraint (Δt): (a) repeated single instantiation of variables; (b) multiple instantiation of variables.

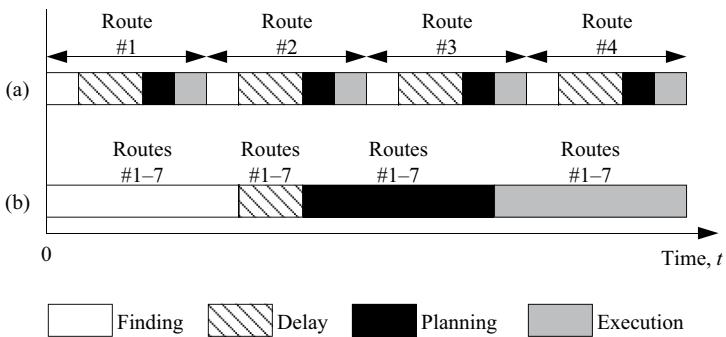


Figure 15.6 Controlling a telecommunications network where there is a delay in receiving information: (a) repeated single instantiation of variables; (b) multiple instantiation of variables.

the controller will finish. In fact, it is feasible that the controller might have found all of the overloaded links and determined an alternative routing for each one, but that it has failed to perform *any* control actions (Figure 15.5b).

This problem is avoided by repeated use of single instantiation, shown in Figure 15.5a, which is guaranteed to have performed *some* control actions within the available time. There is a crucial difference between the two approaches in a control application. Multiple instantiation involves the drawing up of a large plan, followed by execution of the plan. Repeated single instantiation, on the other hand, involves the interleaving of planning and execution.

Repeated single instantiation does not always represent the better choice, as the comparison depends on the specific application. Multiple instantiation is generally faster as it requires the rules to be selected and interpreted once only. Multiple instantiation can have other advantages as well. Consider the telecommunications example again, but now imagine that, in order to choose a suitable rerouting, the controller must request some detailed information from the remote network nodes. There will be a delay in receiving this information, thereby delaying the planning stage. If all the requests for information are sent at the same time (i.e., multiple instantiation, Figure 15.6b) then the delays overlap. Under single instantiation the total delay is much greater, since a separate delay is encountered for every route under consideration (Figure 15.6a).

15.6 Fuzzy Control

The principles of fuzzy controllers were introduced in Section 3.5. LINKman (Taunton and Haspel 1988) is a fuzzy control system that has been applied to cement kiln control and other manufacturing processes. The amount of overlap of the membership functions is deliberately restricted, thereby reducing the number of

rule firings. This restriction is claimed to simplify the defuzzification and tuning of the membership functions, and to make the control actions more transparent. All variables are normalized to lie within the range +1 to -1, representing the maximum and minimum extremes, where 0 represents the normal steady-state value or set point. It is claimed that, by working with normalized variables, the knowledge base can be more easily adapted to different plants.

DARBS and its predecessor ARBS are iterations of a blackboard system introduced in Chapters 10 and 12. ARBS has been applied to controlling plasma deposition, an important process in electronic component manufacture (Hopgood et al. 1998). This work has demonstrated the use of fuzzy rules that adjust more than one action variable simultaneously (i.e., *multivariable control*), in order to control a separate state variable that is measurable but not directly adjustable.

Sripada et al. (1987) have used fuzzy logic in two different ways to enhance bang-bang control, introduced in Section 15.2.6. First, they have used fuzzy logic to build a *self-tuning* servo controller. Their bang-bang servo controller has three switch values (extreme high, extreme low, and steady state), coded as a set of three rules. A separate set of rules adjusts the parameters e^* , Δt , and K_p in the light of experience. This parameter adjustment for self-tuning can be thought of as controlling the controller, or meta-control. The extent to which the parameters are adjusted is in proportion to the degree of membership of the fuzzy sets *too_large* and *too_small*. Thus, a typical fuzzy rule might be:

```
fuzzy_rule r15_1f
  if output is overshoot
    then critical_error becomes too_large
      and critical_error_change becomes reduce.
```

The proposition that the output overshoots is fuzzy, so the fuzzy variable *output* has a degree of membership of the fuzzy set *overshoot* that lies between 0 and 1, reflecting the amount of overshoot. When the fuzzy rule fires, this membership is propagated to the fuzzy proposition that the critical error is too large. The degree of membership of the fuzzy set *reduce* for the fuzzy variable *critical_error_change* is also assigned and, from this membership, the adjustment Δe is calculated by defuzzification, described in Section 3.4.3. A self-tuning bang-bang controller using this approach has been shown to outperform a well-tuned PI controller, that is, a PID controller with the *D* term set to zero (Sripada et al. 1987).

As well as looking at servo control, the same authors have applied knowledge-based techniques to low-level adaptive control. They assert that, while a PID controller is adequate for reducing the drift in a typical plant output, it cannot cope with slight refinements to this basic requirement. In particular, they consider the application of constraints on the plant output, such that the output y must not be allowed to drift beyond $y_0 \pm y_c$, where y_0 is the set-point for y and y_c defines

the constraints. Bang-bang control is required to move y rapidly toward y_0 if it is observed to be approaching $y_0 \pm y_c$. A further requirement is that control of the plant output be as smooth as possible close to the set point, precluding bang-bang control under these conditions. The controller was, therefore, required to behave differently in different circumstances. This adaptability is possible with a system based on heuristic rules, but not for a PID controller, which has a fixed predetermined behavior.

To determine the type and extent of control action required, the error e and the rate of change of the plant output dy/dt were classified with the following fuzzy sets:

- $e = \text{zero};$
- $e = \text{small positive};$
- $e = \text{small negative};$
- $e = \text{large positive};$
- $e = \text{large negative};$
- $e = \text{close to constraint};$
- $dy/dt = \text{small (positive or negative)};$
- $dy/dt = \text{large positive};$
- $dy/dt = \text{large negative}.$

According to the degree of membership of each of these nine fuzzy sets, a rule base was used to determine the degree of membership for each of six fuzzy sets applied to control actions:

- zero change;
- small positive change;
- small negative change;
- large positive change;
- large negative change;
- drastic change (bang-bang).

Incorporating the last action as a fuzzy set enabled a smooth transition to bang-bang control as the plant output approached the constraints.

15.7 The BOXES Controller

15.7.1 *The Conventional BOXES Algorithm*

It has already been emphasized (Section 15.1) that a controller can only function if it has a model for the system being controlled. Neural networks (see Chapters 8 and 9, and Section 15.8) and the BOXES algorithm are techniques for generating

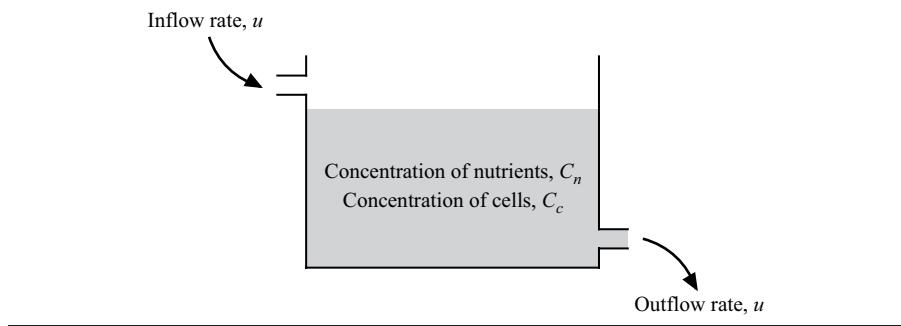


Figure 15.7 A bioreactor.

such a model without any prior knowledge of the mechanisms occurring within the controlled system. Such an approach may be useful if:

- the system is too complicated to model accurately;
- insufficient information is known about the system to enable a model to be built; or
- satisfactory control rules have not been found.

The BOXES algorithm may be applied to adaptive or servo control. Only the following information about the controlled system is required:

- its inputs, which are the possible control actions;
- its outputs, which define its state at any given time;
- the desired state (adaptive control) or the final state (servo control);
- constraints on the input and output variables.

Note that no information is needed about the relationships between the inputs and outputs.

As an example, consider a bioreactor (Ramaswamy et al. 2005; Ungar 1990; Woodcock et al. 1991a), which is a tank of water containing cells and nutrients (Figure 15.7). When the cells multiply, they consume nutrients. The rate at which the cells multiply is dependent only on the nutrient concentration C_n . The aim of the controller is to maintain the concentration of cells C_c at some desired value by altering the rate of flow u of nutrient-rich water through the tank. The state of the bioreactor at any time can be defined by the two variables C_n and C_c , and it can be represented as a point in state space (Figure 15.8). For any position in state space there will be an appropriate control action u . By defining intervals in C_n and C_c we can create a finite number of boxes in state-space, where each box represents a collection of states that are similar to each other. A control action

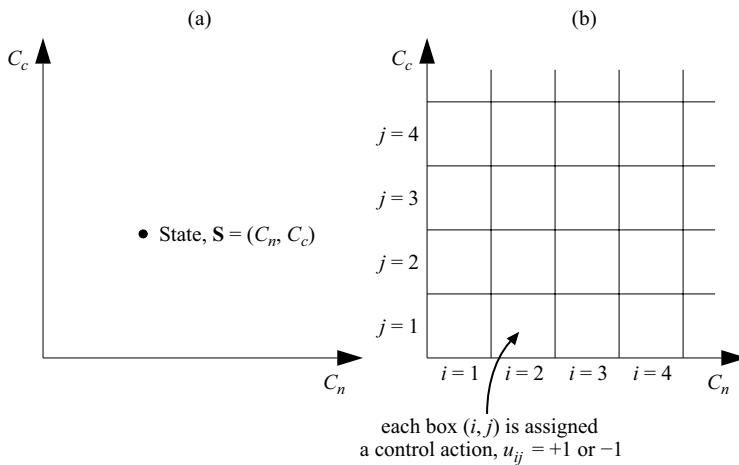


Figure 15.8 (a) State-space for a bioreactor; (b) state-space partitioned into boxes.

can then be associated with each box. A BOXES controller is completely defined by such a set of boxes and control actions, which together implicitly model the controlled system.

Control actions are performed at time $n\Delta t$, where n is an integer and Δt is the interval between control actions. At any such time, the system state will be in a particular box. That box is considered to be “visited” and its control action is performed. The system may be in the same box or a different one when the next control action is due.

The BOXES controller must be trained to associate appropriate control actions with each box. This requirement is most easily met using bang-bang control, where the control variables can take only their maximum or minimum value, denoted +1 and -1, respectively. In the case of the bioreactor, a valve controlling the flow would be fully open or fully shut. For each box, there is a recorded score for both the +1 and -1 action. When a box is visited, the selected control action is the one with the highest score. Learning is achieved by updating the scores.

In order to learn, the system must receive some measure of its performance, so that it can recognize beneficial or deleterious changes in its control strategy. It uses a form of *reinforcement learning*, which is a machine-learning technique based on positive or negative feedback, sometimes referred to as reward and punishment (Sutton et al. 2018). This feedback is achieved through use of a *critic* that evaluates the controller’s performance. In the case of the bioreactor, the controller’s time to failure might be monitored, where “failure” occurs if the cell concentration C_c drifts beyond prescribed limits. The longer the time to failure, the better the performance of the controller. Woodcock et al. (1991a) consider this approach to be midway between supervised and unsupervised learning (Chapters 5 and 8), as the controller

receives an indication of its performance but not a direct comparison between its output and the desired output.

For each box, a score is stored for both the +1 action and the -1 action. These scores are a measure of “degree of appropriateness” and are based on the average time between selecting the control action in that particular box and the next failure.

The learning strategy of Michie and Chambers (1968) for bang-bang control is shown in Figure 15.9. During a run, a single box may be visited N times. For each box, the times ($t_1, \dots, t_p, \dots, t_N$) at which it is visited are recorded. At the end of a run, that is, after a failure, the time t_f is noted and the +1 and -1 scores for each visited box are updated. Each score is based on the average time to failure after that

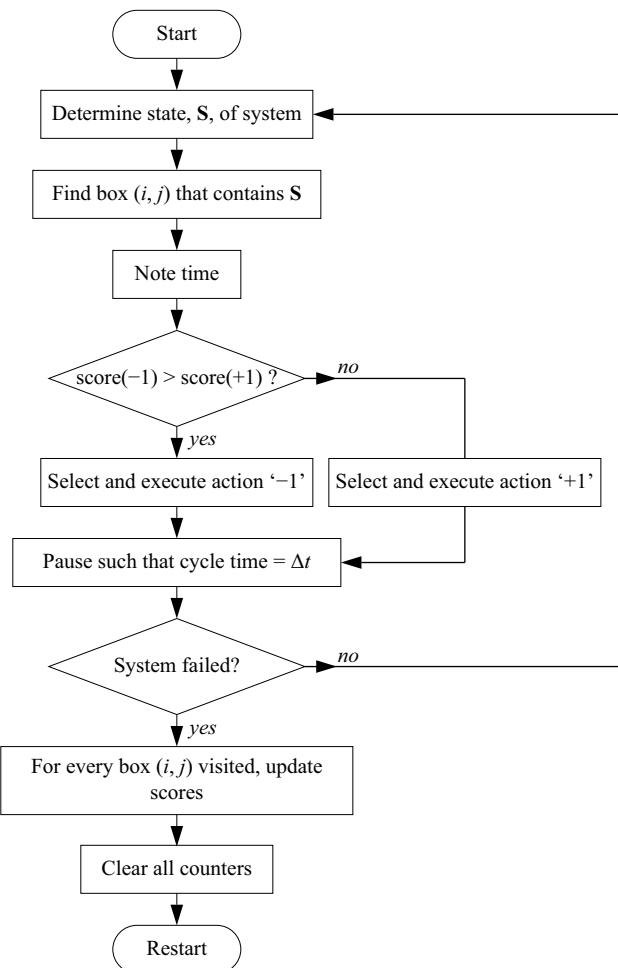


Figure 15.9 The BOXES learning algorithm. (Derived from Michie and Chambers, 1968.)

particular control action had been carried out, which is the lifetime l . The lifetimes are modified by a usage factor n , a decay factor α , a global lifetime l_g , a global usage factor n_g , and a constant β , thereby yielding a score. These modifications ensure that, for each box, both alternative actions have the chance to demonstrate their suitability during the learning process and that recent experience is weighted more heavily than old experience. The full updating procedure is as follows:

$$l_g = \alpha l_g + t_f \quad (15.5)$$

$$n_g = \alpha n_g + 1 \quad (15.6)$$

For each box where $score_{(+1)} > score_{(-1)}$:

$$l_{(+1)} = \alpha l_{(+1)} + \sum_{i=1}^N (t_f - t_i) \quad (15.7)$$

$$n_{(+1)} = \alpha n_{(+1)} + N \quad (15.8)$$

$$score_{(+1)} = \frac{l_{(+1)} + \beta \frac{l_g}{n_g}}{u_{(+1)} + \beta} \quad (15.9)$$

For each box where $score_{(-1)} > score_{(+1)}$:

$$l_{(-1)} = \alpha l_{(-1)} + \sum_{i=1}^N (t_f - t_i) \quad (15.10)$$

$$n_{(-1)} = \alpha n_{(-1)} + N \quad (15.11)$$

$$score_{(-1)} = \frac{l_{(-1)} + \beta \frac{l_g}{n_g}}{u_{(-1)} + \beta} \quad (15.12)$$

After a controller has been run to failure and the scores associated with the boxes have been updated, the controller becomes competent at balancing in only a limited part of the state-space. In order to become expert in all regions of state-space, the controller must be run to failure several times, starting from different regions in state-space.

The BOXES algorithm has been used for control of a bioreactor as described, and also for balancing a pole on a mobile cart (Figure 15.10). The latter is a similar problem to the bioreactor, but the state is described by four rather than two variables. The boxes are four-dimensional and difficult to represent graphically.

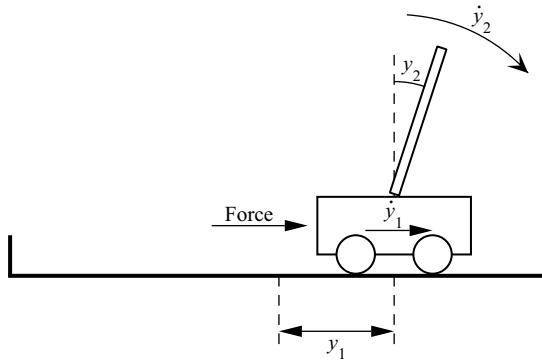


Figure 15.10 The cart-and-pole control problem.

In principle, the BOXES algorithm can be applied to state-space with any number of dimensions.

The cart-and-pole problem, shown in Figure 15.10, has been used extensively as a benchmark for intelligent controllers. A pole is attached by means of a hinge to a cart that can move along a finite length of track. The cart and the pole are restricted to movement within a single plane. The controller attempts to balance the pole while keeping the cart on the length of track by applying a force to the left or right. If the force has a fixed magnitude in either direction, this example is another application of bang-bang control. The four state variables are the cart's position y_1 and velocity \dot{y}_1 and the pole's angle y_2 and angular velocity \dot{y}_2 . Failure occurs when y_1 or y_2 breach the constraints placed upon them. The constraint on y_1 represents the limited length of the track.

Rather than use a BOXES system as an intelligent controller *per se*, Sammut and Michie (1991) have used it as a means of eliciting rules for a rule-based controller. After running the BOXES algorithm on a cart-and-pole system, they found clear relationships between the learned control actions and the state variables. They expressed these relationships as rules and then proceeded to use analogous rules to control a different black box simulation, namely, a simulated spacecraft. The spacecraft was subjected to a number of unknown external forces, but the rule-based controller was tolerant of these forces. Similarly, the BOXES controller of Woodcock et al. (1991a) was virtually unaffected by random variations superimposed on the control variables.

One of the attractions of the BOXES controller is that it is a fairly simple technique, and so an effective controller can be built quite quickly. Woodcock et al. (1991a) rapidly built their controller and a variety of black box simulations using the Smalltalk object-oriented language (see Chapter 4). Although both the controller and simulation were developed in the same programming environment, the workings of the simulators were hidden from the controller. Sammut and Michie

(1991) also report that they were able to build quickly their BOXES controller and the rule-based controller that it inspired.

15.7.2 Fuzzy BOXES

Woodcock et al. (1991a) have investigated the suggestion that the performance of a BOXES controller might be improved by using fuzzy logic to smooth the bang-bang control (Bernard 1988). Where different control actions are associated with neighboring boxes, it was proposed that states lying between the centers of the boxes should be associated with intermediate actions. The controller was trained as described in the previous subsection in order to determine appropriate bang-bang actions. After training, the box boundaries were fuzzified using triangular fuzzy sets. The maximum and minimum control actions (bang-bang) were normalized to +1 and -1, respectively, and intermediate actions were assigned a number between these extremes.

Consider again the bioreactor, which is characterized by two-dimensional state-space. If a particular state S falls within the box (i, j) , then the corresponding control action is u_{ij} . This relationship can be stated as an explicit rule:

```
if state S belongs in box(i, j)
  then the control action is uij.
```

If we consider C_n and C_c separately, this rule can be rewritten:

```
if Cn belongs in interval i AND Cc belongs in interval j
  then the control action is uij.
```

The same rule can be applied in the case of fuzzy BOXES, except that now it is interpreted as a fuzzy rule. We know from Equation 3.36 that:

$$\mu(C_n \text{ belongs in interval } i \text{ AND } C_c \text{ belongs in interval } j) = \min[\mu(C_n \text{ belongs in interval } i), \mu(C_c \text{ belongs in interval } j)]$$

Thus, if the membership functions for C_n belongs in interval i and C_c belongs in interval j are both triangular, then the membership function for state S belongs in box (i, j) , denoted by $\mu_{ij}(S)$, is a surface in state space in the shape of a pyramid (Figure 15.11). As the membership functions for neighboring pyramids overlap, a point in state space may be a member of more than one box. The control action u_{ij} for each box to which S belongs is scaled according to the degree of membership $\mu_{ij}(S)$. The normalized sum of these actions is then interpreted as the defuzzified action u_o :

$$u_o = \frac{\sum_i \sum_j \mu_{ij}(S) u_{ij}}{\sum_i \sum_j \mu_{ij}(S)} \quad (15.13)$$

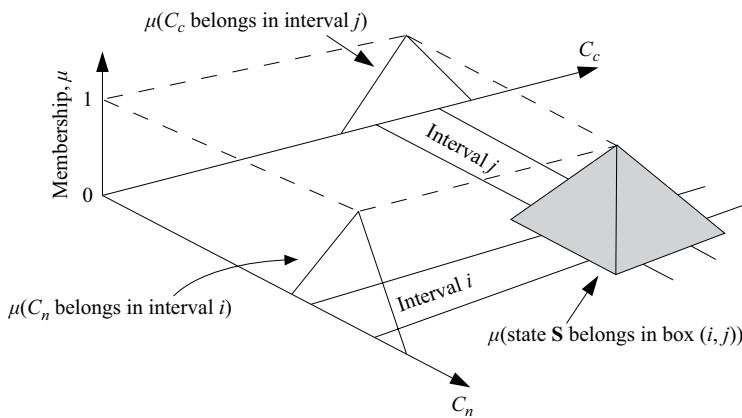


Figure 15.11 Fuzzy membership functions for boxes in the bioreactor state space. (Derived from Woodcock, N. et al., 1991a.)

This interpretation is equivalent to defuzzification using the centroid method (Section 3.4.3), if the membership functions for the control actions are assumed to be symmetrical about a vertical line through their balance points.

Woodcock et al. (1991a) have tested their fuzzy BOXES controller against the cart-and-pole and bioreactor simulations (described earlier), both of which are adaptive control problems. They have also tested it in a servo control application, namely, reversing a tractor and trailer up to a loading bay. In none of these examples was there a clear winner between the nonfuzzy and the fuzzy BOXES controllers. The comparison between them was dependent on the starting position in state-space. This dependency was most clearly illustrated in the case of the tractor and trailer. If the starting position was such that the tractor could reverse the trailer in a smooth sweep, the fuzzy controller was able to perform best because it was able to steer smoothly. The nonfuzzy controller, on the other hand, was limited to using only full steering lock in either direction. If the starting condition was such that full steering lock was required, then the nonfuzzy controller outperformed the fuzzy one.

15.8 Neural Network Controllers

Neural network controllers tackle a similar problem to BOXES controllers, which is the control of a system using a model that is automatically generated during a learning phase. Two distinct approaches have been adopted by Valmiki et al. (1991) and by Willis et al. (1991). Valmiki et al. have trained a neural network to associate directly particular sets of state variables with particular action variables, in an analogous fashion to the association of a box in state-space with a control action in a BOXES controller. Willis et al. adopted a less direct approach, using a neural

network to estimate the values of those state variables that are critical to control but cannot be measured directly. The estimated values are then fed to a PID controller as though they were real measurements. These two approaches are discussed separately below.

15.8.1 Direct Association of State Variables with Action Variables

Valmiki et al. (1991) have applied a neural network to a control problem that had previously been tackled using rules and objects, namely, the control of a glue dispenser (Chandraker et al. 1990). As part of the manufacture of mixed technology circuit boards, surface-mounted components are held in place by a droplet of glue. The glue is dispensed from a syringe by means of compressed air. The size of the droplet is the state variable that must be controlled, and the change in the air pressure is the action variable.

Valmiki et al. have built a 6–6–5 multilayer perceptron (Figure 15.12), where specific meanings are attached to values of 0 and 1 on the input and output nodes. Five of the six input nodes represent ranges for the error in the size of the droplet. The node corresponding to the measured error is sent a 1, while the other four nodes are sent a 0. The sixth input node is set to 0 or 1 depending on whether the error is positive or negative. Three of the five output nodes are used to flag particular actions, while the

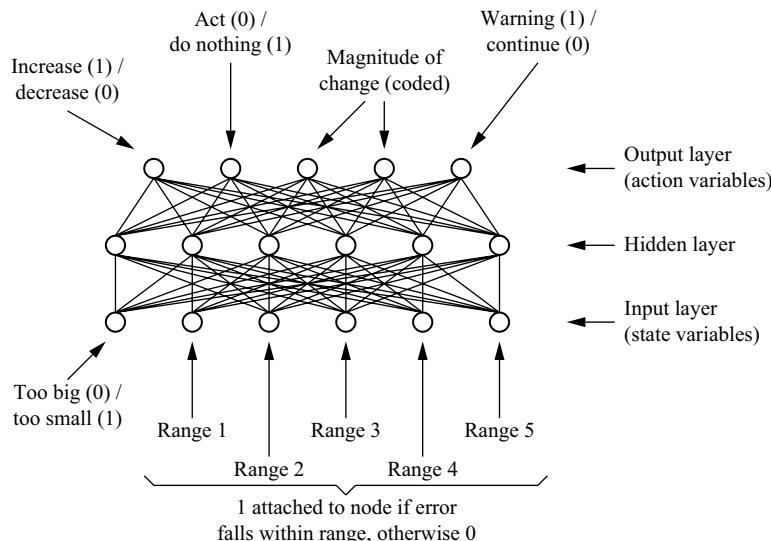


Figure 15.12 Using a neural network to map directly state variables to action variables. (Derived from the glue-dispensing application of Valmiki, A. H., et al., 1991.)

other two are a coded representation of the amount by which the air pressure should be changed, if a change is required. The three action flags on the output are:

- decrease (0) or increase (1) the pressure;
- do something (0) or do nothing (1), overriding the increase/decrease flag;
- no warning (0) or warning (1) if the error in the droplet size is large.

The required mapping of the input states to the outputs was known in advance, allowing the training data to be generated by hand. One advantage of a neural network approach is that an interpolated meaning can be attached to output values that lie between 0 and 1. This same effect could, alternatively, have been achieved using fuzzy rules, given the prior knowledge of the mapping of the input states to the outputs. Nevertheless, Valmiki et al.'s experiment is important in demonstrating the feasibility of using a neural network to learn to associate state variables with control actions. This capability is useful where rules or functions that link the two are unavailable, although this limitation did not apply to their experiment.

15.8.2 Estimation of Critical State Variables

Willis et al. (1991) have demonstrated the application of neural network controllers to industrial continuous and batch-fed fermenters, and to a commercial-scale high purity distillation column. Each application is characterized by a delay in obtaining the critical state variable (i.e., the *controlled* variable), as it requires chemical or pathological analysis. The neural network allows comparatively rapid estimation of the critical state variable from secondary state variables. The use of a model for such a purpose is discussed in Section 12.4.4. The only difference here is that the controlled plant is modeled using a neural network. The estimated value for the critical state variable can be sent to a PID controller (see Section 15.2.5) to determine the action variable (Figure 15.13). As the critical variable can be measured off-line in each case, there is no difficulty in generating training sets of data. Each of the three applications demonstrates a different aspect to this problem, described in the following text. The chemotaxis learning algorithm was used in each case (see Section 8.4.3).

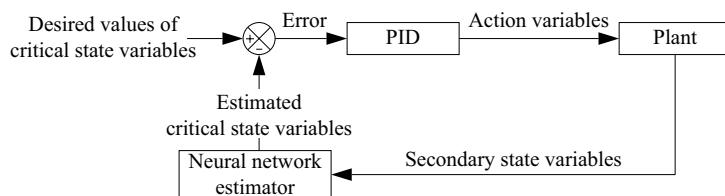


Figure 15.13 Using a neural network to estimate values for critical state variables.

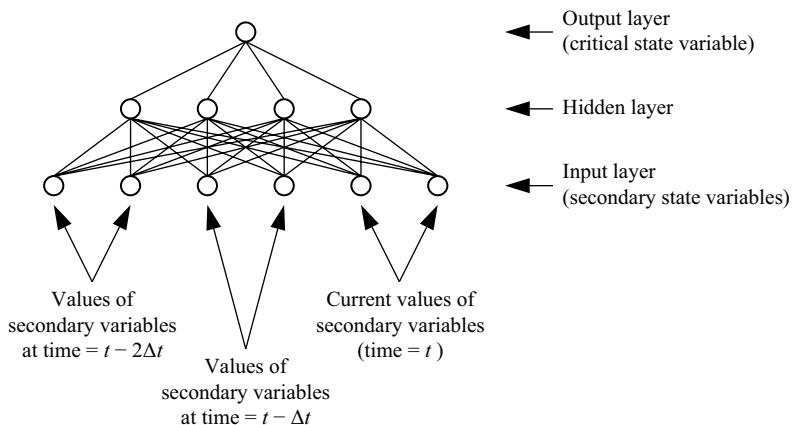


Figure 15.14 Using time histories of state variables in a neural network. (Derived from the continuous fermentation application of Willis, M. J. et al., 1991.)

The continuous fermentation process is dynamic, that is, the variables are constantly changing, and a *change* in the value of a variable may be just as significant as the absolute value. The role of a static neural network, on the other hand, is to perform a mapping of static input variables onto static output variables. One way around this weakness is to use the recent history of state variables as input nodes. In the continuous fermentation process, two secondary state variables were considered. Nevertheless, six input nodes were required, since the two previously measured values of the variables were used as well as the current values (Figure 15.14).

In contrast, the batch fermentation process should move smoothly and slowly through a series of phases, never reaching equilibrium. In this case, the time since the process began, rather than the time history of the secondary variable, was important. Thus, for this process there were only two input nodes, the current time and a secondary state variable.

In the methanol distillation process, an alternative approach was adopted to the problem of handling dynamic behavior. As it is known that the state variables must vary continuously, sudden sharp changes in any of the propagated values can be disallowed. This constraint is achieved through a simple low-pass digital filter. There are several alternative forms of digital filter (see, for example, Diniz et al. (2010)), but Willis et al. (1991) used the following:

$$y(t) = \Omega y(t-1) + (1-\Omega)x(t) \quad 0 \leq \Omega \leq 1 \quad (15.14)$$

where $x(t)$ and $y(t)$ are the input and output of the filter, respectively, at time t . The filter ensures that no value of $y(t)$ can be greatly different from its previous value,

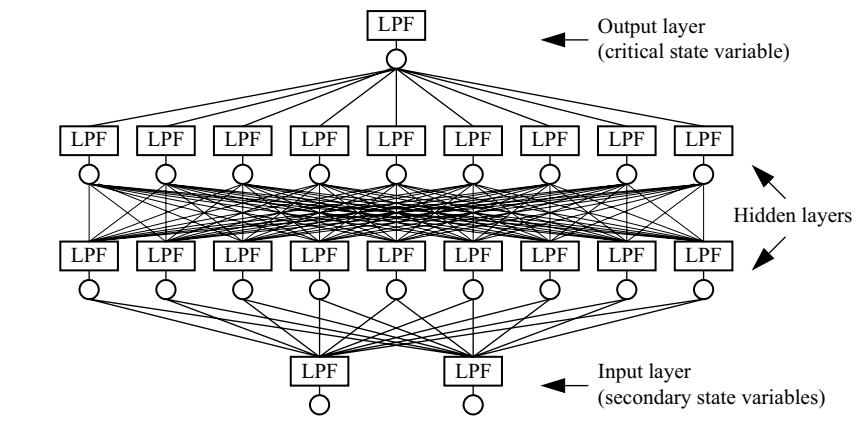


Figure 15.15 Dealing with changing variables by using low-pass filters (LPF). (Derived from the industrial distillation application of Willis, M. J. et al., 1991.)

and so high-frequency fluctuations are eliminated. Such a filter was attached to the output side of each neuron (Figure 15.15), so that the unfiltered output from the neuron was represented by $x(t)$, and the filtered output was $y(t)$. Suitable values for the parameters Ω were learned along with the network weightings.

Willis et al. were able to show improved accuracy of estimation and tighter control by incorporating the digital filter into their neural network. Further improvements were possible by comparing the estimated critical state variable with the actual values, as these became known. The error was then used to adjust the output of the estimator. There were two feedback loops, one for the PID controller and one for the estimator (Figure 15.16).

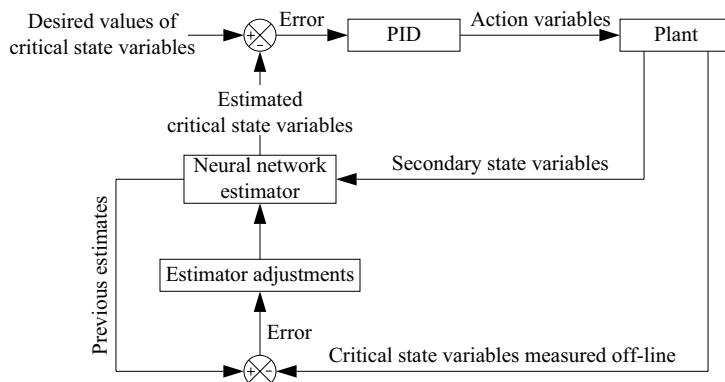


Figure 15.16 Feedback control of both the plant and the neural network estimator.

15.9 Statistical Process Control (SPC)

15.9.1 Applications

Statistical process control (SPC) is a technique for monitoring the quality of products as they are manufactured. Critical parameters are monitored, and adjustments are made to the manufacturing process *before* any products are manufactured that lie outside of their specifications. The appeal of SPC is that it minimizes the number of products that are rejected at the quality control stage, thereby improving productivity and efficiency. Since the emphasis of SPC lies in *monitoring* products, this section could equally belong in Chapter 12.

SPC involves inspecting a sample of the manufactured products, measuring the critical parameters, and inferring from these measurements any trends in the parameters for the whole population of products. The gathering and manipulation of the statistics is a procedural task, and some simple heuristics are used for spotting trends. The monitoring activities, therefore, lend themselves to automation through procedural and rule-based programming. Depending on the process, the control decisions might also be automated.

15.9.2 Collecting the Data

Various statistics can be gathered, but we will concentrate on the mean and standard deviation of the monitored parameters (although the *range* of sample values is often used instead of the standard deviation). Periodically, a sample of consecutively manufactured products is taken, and the critical parameter x is measured for each item in the sample. The sample size n is typically in the range 5–10. In the case of the manufacture of silicon wafers, thickness may be the critical parameter. The mean \bar{x} and standard deviation σ for the sample are calculated. After several such samples have been taken, it is possible to arrive at a mean of means $\bar{\bar{x}}$ and a mean of standard deviations $\bar{\sigma}$. The values $\bar{\bar{x}}$ and $\bar{\sigma}$ represent the normal, or set-point, values for \bar{x} and σ , respectively. Special set-up procedures exist for the manufacturing plant to ensure that $\bar{\bar{x}}$ corresponds to the set-point for the parameter x . Bounds called *control limits* are placed above and below these values (Figure 15.17). Inner and outer control limits, referred to as *warning limits* and *action limits*, respectively, may be set such that:

$$\text{warning limit for } \bar{x} = \bar{\bar{x}} \pm 2 \frac{\bar{\sigma}}{\sqrt{n}}$$

$$\text{action limit for } \bar{x} = \bar{\bar{x}} \pm 3 \frac{\bar{\sigma}}{\sqrt{n}}$$

Action limits are also set on the values of σ such that:

$$\text{upper action limit for } \sigma = C_U \bar{\sigma}$$

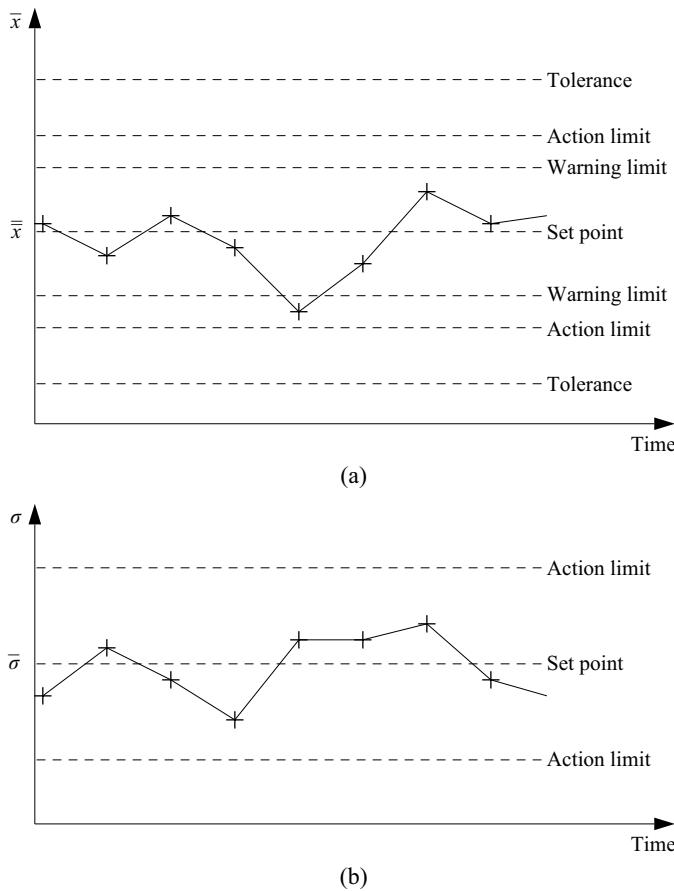


Figure 15.17 Control limits (action and warning) applied to: (a) sample means (\bar{x}); (b) sample standard deviations (σ).

$$\text{lower action limit for } \sigma = \frac{\bar{\sigma}}{C_L}$$

where suitable values for C_U and C_L can be obtained from standard tables for a given sample size n . Note that both C_U and C_L are greater than 1. The heuristics for interpreting the sample data with respect to the control limits are described in Section 15.9.3. Any values of \bar{x} that lie beyond the action limits indicate that a control action is needed. The *tolerance* that is placed on a parameter is the limit beyond which the product must be rejected. It follows that if the tolerance is tighter than the action limits, then the manufacturing plant is unsuited to the product and attempts to use it will result in a large number of rejected products irrespective of SPC.

15.9.3 Using the Data

As the data are gathered, a variety of heuristics can be applied. Some typical ones are reproduced here:

- if a single \bar{x} value lies beyond an action limit
then a special disturbance has occurred that
must be investigated and eliminated.
- if there are \bar{x} values beyond both action limits
then the process may be deteriorating.
- if two consecutive values of \bar{x} lie beyond a worrying limit
then the process mean may have moved.
- if eight consecutive values of \bar{x} lie on an upward or downward trend
then the process mean may be moving.
- if seven consecutive values of \bar{x} lie all above or all below $\bar{\bar{x}}$
then the process mean may have moved.
- if there are σ values beyond the upper action limit
then the process may be deteriorating.
- if eight consecutive values of σ lie on an upward trend
then the process may be deteriorating.
- if there are σ values beyond the lower action limit
then the process may have improved
and attempts should be made to incorporate the improvement
permanently.

The conclusions of these rules indicate a high probability that a control action is needed. They cannot be definite conclusions, as the evidence is statistical. Furthermore, it may be that the process itself has not changed at all, but instead some aspect of the measuring procedure has altered. Each of the aforementioned rules calls for investigation of the process to determine the cause of any changes, perhaps using case-based or model-based reasoning (Chapters 5 and 12).

15.10 Summary

Intelligent systems for control applications draw upon the techniques used for interpreting data (Chapter 12) and planning (Chapter 14). Frequently, the stages of planning are interleaved with the execution of the plans, so that the controller can react to changes in the controlled plant as they occur. This reactivity contrasts with

the classical planning systems described in Chapter 14, where the world is treated as a static “snapshot.” As control systems must interact with a dynamic environment, time constraints are placed upon them. There is often a trade-off between the quality of a control decision and the time taken to derive it. In most circumstances it is preferable to perform a suboptimal control action than to fail to take any action within the time limits.

The control problem can be thought of as one of mapping a set of state variables onto a set of action variables. The state variables describe the state of the controlled plant, while the action variables, set by the controller, are used to modify the state of the plant. Adaptive controllers attempt to maintain one or more critical state parameters at a constant value, minimizing the effects of any disturbance. In contrast, servo controllers attempt to drive the plant to a new state, which may be substantially different from its previous state. The problems of adaptive and servo control are similar, as both involve minimizing the difference, or error, between the current values of the state variables and the desired values.

An approximate distinction can be drawn between low-level “reflex” control and high-level supervisory control. Low-level control often requires little intelligence and can be most effectively coded procedurally, for instance, as the sum of proportional, integral, and derivative (PID) terms. Improvements over PID control can be made by using fuzzy rules, which also allow some subtleties to be included in the control requirements, such as bounds on the values of some variables. Fuzzy rules offer a mixture of some of the benefits of procedures and crisp rules. Like crisp rules, fuzzy rules allow a linguistic description of the interaction between state and action variables. On the other hand, like an algebraic procedure, fuzzy rules allow smooth changes in the state variables to bring about smooth changes in the action variables. The nature of these smooth changes is determined by the membership functions that are used for the fuzzy sets.

Any controller requires a model of the controlled plant. Even a PID controller holds an implicit model in the form of its parameters, which can be tuned to specific applications. When a model of the controlled plant is not available, it is possible to build one automatically using the BOXES algorithm or a neural network. Both techniques can be used to provide a mapping between the state variables and the action variables. They can also be used in a monitoring capacity, where critical state variables (which may be difficult to measure directly) are inferred from secondary measurements. The inferred values can then be used as feedback to a conventional controller. If a plant is modeled with sufficient accuracy, then predictive control becomes a possibility. A predictive controller has two goals, to tackle the immediate control needs and to minimize future deviations, based on the predicted behavior.

Further Reading

- Franklin, G. F., J. D. Powell, and A. Emami-Naeini. 2020b. *Feedback Control of Dynamic Systems*. 8th ed. Pearson Education, Upper Saddle River, NJ.
- Jantzen, J. 2013. *Foundations of Fuzzy Control: A Practical Approach*, 2nd ed. John Wiley & Sons, Chichester, UK.
- Nanayakkara, T., F. Sahin, and M. Jamshidi. 2009. *Intelligent Control Systems with an Introduction to System of Systems Engineering*. CRC Press, Boca Raton, FL.
- Suykens, J. A. K., J. P. L. Vandewalle, and B. L. R. De Moor. 2010. *Artificial Neural Networks for Modelling and Control of Non-Linear Systems*. Kluwer, Dordrecht, Germany.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 16

The Future of Intelligent Systems

16.1 Benefits

This book has discussed a wide range of intelligent systems techniques and their applications. A question remains over whether any implemented intelligent system, using these techniques, can truly display human-like intelligence. Nevertheless, the following practical benefits have stemmed from the development of intelligent systems techniques.

- *Reliability and Consistency*

An intelligent system makes decisions that are consistent with its input data and its knowledge base (for a knowledge-based system) or numerical parameters (for a computational intelligence technique). It is, therefore, more reliable than a person when repetitive mundane judgments have to be made.

- *Automation*

In many applications, such as visual inspection on a production line, judgmental decision-making has to be performed repeatedly. A well-designed intelligent system can deal with the majority of such cases, while highlighting any that lie beyond the scope of its capabilities. Therefore, only the most difficult cases, which are normally the most interesting, are deferred to a person.

- *Speed*

Intelligent systems are designed to make automatic decisions that would otherwise require human reasoning, judgment, expertise, or common sense. Any lack of true intelligence is compensated by the system's processing speed. An

intelligent system can make decisions informed by a wealth of data and information that a person would have insufficient time to assimilate.

■ *Improved Domain Understanding*

The process of constructing a knowledge-based system requires the decision-making criteria to be clearly identified and assessed. This process frequently leads to a better understanding of the problem being tackled. Similar benefits can be obtained by investigating the decision-making criteria used by computational intelligence techniques.

■ *Knowledge Archiving*

The knowledge base is a repository for the knowledge of one or more people. When these people move on to new jobs, some of their expert knowledge is saved in the knowledge base, which continues to evolve after their departure.

16.2 Trends in Implementation

Since intelligent systems are supposed to be flexible and adaptable, development is usually based upon continuous refinements of an initial prototype. This is the *prototype–test–refine* cycle, which applies to both knowledge-based systems and computational intelligence techniques. The key stages in the development of a system are as follows:

- decide the requirements;
- design and implement a prototype;
- continuously test and refine the prototype, with feedback from users.

In the past, many software engineers would have been skeptical of the prototype–test–refine cycle, particularly for large projects. Instead, they would have preferred the traditional linear “waterfall” process of meticulous specification, analysis, and design phases prior to implementation and testing. These attitudes have now changed, and rapid prototyping and iterative development have gained respectability across most areas of software engineering.

The development of a mock-up knowledge-based system using an expert system shell may be helpful to demonstrate a concept at the first prototyping stage, prior to reimplementing in a more sophisticated programming environment. However, working in a shell that lacks flexibility and representational capabilities can lead to convoluted programming in order to force the desired behavior. For these reasons, it is now commonplace to work from the outset in a flexible programming environment that provides all the tools that are likely to be needed, or which allows extra tools to be added as required.

16.3 Intelligent Systems and the Internet

Intelligent systems are becoming increasingly distributed in terms of both their applications and their implementation. Although large centralized corporate systems remain important, they can nevertheless draw upon a wealth of information, of variable quality, from across the Internet. At the same time, smaller embedded intelligent systems are also appearing in the home and workplace. Communication between them is likely to extend further their influence on our daily lives, especially as more and more devices that would not normally be regarded as computers are now communicating via the Internet, collectively known as the *Internet of Things* (Al-Fuqaha et al. 2015).

In addition to being distributed in their applications, intelligent systems are also becoming distributed in their implementation. Chapter 5 looked at the important technique of intelligent agents. The explosion in the use of the Internet has led to increased communication between agents that reside on separate computers. Mobile agents that can travel over the Internet in search of information are also viable, but they are likely to be constrained by security considerations.

Jennings (2000) argues that agent-based techniques are appropriate both for developing large complex systems and for mainstream software engineering. Chapter 10 discussed the blackboard architecture for dividing problems into sub-tasks that can be shared among specialized agents that may be widely distributed. Hsu and McGuinness (2003) have demonstrated the concept of distributed knowledge accessed by an expert system for wine selection. There is abundant information about wines on the Web, but techniques are needed for making available the meaning of Web information, that is, a *semantic web* is required (Shadbolt et al., 2006).

Paradoxically, there is also a sense in which intelligent systems are becoming more integrated, by allowing a single system to be act as a server for multiple users (Eriksson, 1996). Watson & Gardingen (1999) describe a sales support application that had become integrated by use of the World Wide Web, as a single definitive copy of the software accessible via the Web had replaced distributed copies. Similarly, Wen (2008) describes a centralized expert system for traffic-light control, accessed via wireless nodes.

As a further aspect of integration, computers may be required to assist in commercial decision-making, based upon a wide view of the organization. For example, production decisions need to take into account and influence design, marketing, personnel, sales, materials stocks, and product stocks. These separate, distributed functions are becoming integrated by the need for communication between them. (The use of computers to support an integrated approach to manufacturing is termed *computer-integrated manufacturing*, or *CIM*.) Nevertheless, smaller-scale systems are likely to remain at least as important. These include intelligent agents that perform as personal consultants to advise and inform us, and others that function silently

and anonymously while performing tasks such as data interpretation, monitoring, and control.

16.4 Computational Power

Several reasons can be proposed for the excitement around AI since the start of the third millennium, including:

- maturation and iterative improvement of ideas that had been proposed in prior decades;
- significant technical developments, notably deep-learning algorithms and data analytics (for big data);
- the rise of the Internet which, as noted in Section 16.3, has enabled any online system to access vast amounts of information and to distribute intelligent behaviors among networked devices;
- the availability of huge volumes of data to train machine-learning systems;
- widespread availability of large computational power.

The last point is particularly significant, as computer hardware has improved to the extent that many computationally demanding AI concepts no longer require supercomputers and have become practical on affordable computers and mobile devices. *Quantum computing* holds the potential to cause a step-change in this trend (Welser et al. 2018). Unlike classical computing that uses bits that are in one of two distinct states (0 and 1), quantum bits can be in many superimposed states at the same time, like the subatomic particles on which the technology is based. The inherent parallelism of the quantum computing model gives it the potential to run enormously faster than conventional computers. The model may have two types of impact on the field of AI. Firstly, it can open the door to much greater computational power, which has arguably been holding back the field of AI until now. Secondly and more speculatively, it may transpire that the nondeterministic nature of quantum computing makes it a more natural tool with which to model the non-deterministic world in which we live.

16.5 Ubiquitous Intelligent Systems

Early examples of intelligent systems were mostly consultative in nature, for example, diagnostic expert systems that reached a conclusion following dialogue with a human, or neural networks that produced classifications from data stored in a file. Many of the more modern intelligent systems are *situated*, that is, they interact with their environment through sensors that detect the environment and actuators that operate upon the environment in real time. Situated artificial intelligence is demonstrated by an application of DARBS (see Section 10.2 and Section 12.5) that

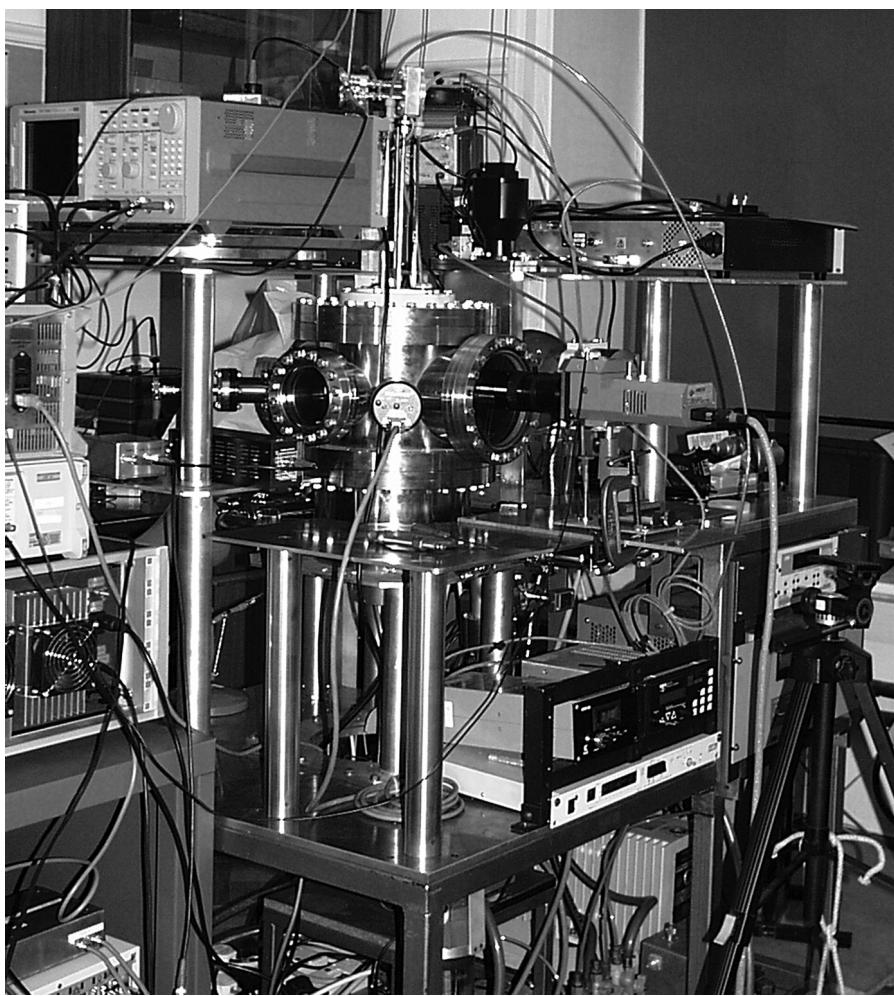


Figure 16.1 An intelligent system situated in a complex environment. Here, DARBS is used to monitor and control plasma deposition equipment. (Photograph by Lars Nolle.) (Reprinted from Hopgood, A.A. 2005. The state of artificial intelligence, Copyright (2005), pp. 1–75, with permission from Elsevier.)

involves monitoring and controlling plasma deposition equipment used in semiconductor device manufacture (Al-Kuzee et al., 2003). As Figure 16.1 graphically illustrates, the computer system in this application is indeed situated within its environment with a wealth of devices for interacting with the environment.

If intelligent systems are to become more widely situated into everyday environments, they need to become smaller, cheaper, and more reliable. The next key stage in the development of artificial intelligence is likely to be a move toward *embedded*

intelligence, that is, intelligent systems that are embedded in machines, devices, and appliances. The work of Choy et al. (2003) is significant in this respect, as they have demonstrated that the DARBS blackboard system can be ported to a compact platform of parallel low-cost processors.

We are likely to see a growing mixture of intelligent agents that serve as personal consultants to advise and inform us, and others that function silently and anonymously while performing tasks such as data interpretation, monitoring, and control. We are close to the possibility of various forms of intelligent behavior in everyday domestic, workplace, and public environments. Examples that have already appeared include washing machines that incorporate knowledge-based control systems, elevators that use fuzzy logic to decide at which floor to wait for the next passenger, robotic vacuum cleaners that incorporate the BDI model of intelligent agents, tablet devices that use neural networks to learn the characteristics of their owner's handwriting, and driverless cars that rely on sophisticated vision and control technologies. It is surely only a matter of time before artificial intelligence becomes truly ubiquitous.

16.6 Ethics

As the capabilities of intelligent systems have improved and they take on increasing autonomy, so have the ethical dilemmas raised. Dignum (2018) has proposed three levels of the relationship between ethics and AI:

- (1) *Ethics by Design: the technical/algorithmic integration of ethical reasoning capabilities as part of the behavior of an artificial autonomous system.* This category corresponds with the proposition by Waser (2009) that the decision-making of intelligent systems should be constrained by a coherent, integrated, and consistent moral/ethical structure. A classic example is an autonomous car that faces an inevitable collision and must choose between different fatal outcomes. Stahl (2004) argues for a Moral Turing Test. In the original Turing Test, or the *imitation game* (Turing, 1950), a machine was deemed to display intelligence if a human could not distinguish the computer from another human in a blind conversation. Likewise, the Moral Turing Test would deem an intelligent system to be moral if its decisions cannot be distinguished from those of a moral being.
- (2) *Ethics in Design: the regulatory and engineering methods that support the analysis and evaluation of the ethical implications of AI systems as these integrate or replace traditional social structures.* This category is more focused on the ethical implications of the deployment of intelligent systems than the system's own ethics. One such consideration is the potential for intelligent systems to displace human employment. It may be argued that the loss of employment is damaging for a person's self-worth and financial wellbeing. On the other hand,

intelligent systems may liberate humankind from drudgery, create more leisure time, and focus employment on the most interesting jobs.

- (3) *Ethics for Design: the codes of conduct, standards and certification processes that ensure the integrity of developers and users as they research, design, construct, employ and manage artificial intelligent systems.* This category relates to the motives of those who develop and deploy intelligent systems. Issues that could come under this category include the design of intelligent systems for warfare, and questions over who is held responsible for the decisions of an intelligent system. Indeed, if we imagine a future where AI displays true intelligence and emotions, we could speculate that it should share the same rights and responsibilities as humans.

16.7 Conclusions

Chapter 1 of this book proposed a simple definition of artificial intelligence, that is, the science of mimicking or exceeding human mental faculties in a computer. The subsequent sections have reviewed a wide range of techniques. Some approaches are highly structured, while others specify only low-level behavior, leaving the intelligence to emerge through complex interactions. Some approaches are based on the use of knowledge expressed in words and symbols, whereas others use only mathematical and numerical constructions.

Overall, the tools and techniques of artificial intelligence are ingenious, practical, and useful. If these were the criteria by which success were measured, artificial intelligence would surely be heralded as one of the most accomplished technological disciplines. However, human mental faculties are incredibly complex and have proved to be extremely difficult to mimic. In this respect, the discipline still has a long way to travel.

The techniques presented here have undoubtedly advanced humankind's progress toward the construction of an intelligent machine. Artificial intelligence research has made significant advances from both ends of the intelligence spectrum shown in Figure 1.1, and is closing the gap in the middle. Nevertheless, it is still difficult to build a system that can make sensible decisions about unfamiliar situations in everyday, nonspecialist, domains. This challenge requires progress in simulating behaviors that humans take for granted, such as perception, language, interaction, common sense, and adaptability.

Through the further development of AI models and the increasing raw power of computer hardware, we can look forward to the full spectrum of intelligent behaviors finally being bridged. Already, intelligent systems have shown their utility in a wide range of domains, albeit that each domain is typically quite narrow. Although we are still a long way from a generalized human-like artificial intelligence, it will surely arrive eventually, and society should begin to prepare.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

References

- Aamodt, A., and E. Plaza. 1994. Case-based reasoning – Foundational issues, methodological variations, and system approaches. *AI Communications* 7 (1):39–59.
- AbdelMeguid, H., and B. Ulanicki. 2010. *Feedback rules for operation of pumps in a water supply system considering electricity tariffs*. In *WDSA 2010:12th annual Water Distribution Systems Analysis conference*. Tucson, AZ.
- Abe, S. 2010. *Support Vector Machines for Pattern Classification*. 2nd ed., Advances in Pattern Recognition. Springer, London, UK.
- Aineto, D., S. Jiménez, and E. Onaindia. 2018. *Learning STRIPS action models with classical planning*. In *28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 399–407.
- Al-Fuqaha, A., M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. 2015. Internet of Things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* 17 (4):2347–2376.
- Al-Kuzee, J., T. Matsuura, A. Goodyear, L. Nolle, A. A. Hopgood, P. D. Picton, and N. St. J. Braithwaite. 2003. *Intelligent control of low-pressure plasma processing*. In *IEEE Industrial Electronics Society Conference (ICON03)*, 2, 1932–1937.
- Albashiri, K. A., F. Coenen, and P. Leng. 2009. EMADS: An extendible multi-agent data miner. *Knowledge Based Systems* 22 (7):523–528.
- Alpaydin, E. 2010. *Introduction to Machine Learning*. 2nd ed. MIT Press, Cambridge, MA.
- AltInten, A., F. Ketevanlioglu, S. Erdogan, H. Hapoglu, and M. Alpbaz. 2008. Self-tuning PID control of jacketed batch polystyrene reactor using genetic algorithm. *Chemical Engineering Journal* 138 (1–3):490–497.
- Altug, S., M. Y. Chow, and H. J. Trussell. 1999. Heuristic constraints enforcement for training of and rule extraction from a fuzzy/neural architecture – Part II: Implementation and application. *IEEE Transactions on Fuzzy Systems* 7 (2):151–159.
- Ampratwum, C. S., P. D. Picton, and A. A. Hopgood. 1997. *A rule-based system for optical emission spectral analysis*. In *Symposium on industrial applications of Prolog (INAP'97)*. Kobe, Japan. 99–102.
- Aoki, H., and K. Sasaki. 1990. Group supervisory control system assisted by artificial intelligence. *Elevator World*, February, 70–91.
- Arroyo-Figueroa, G., Y. Alvarez, and L. E. Sucar. 2000. SEDRET – An intelligent system for the diagnosis and prediction of events in power plants. *Expert Systems with Applications* 18:75–86.
- Ashby, M. F. 1989. On the engineering properties of materials. *Acta Metallurgica* 37:1273–1293.

- Baker, J. E. 1985. *Adaptive selection methods for genetic algorithms*. In *International Conference on Genetic Algorithms and their Application*. 101–111.
- Balakrishnan, A., and T. Semmelbauer. 1999. Circuit diagnosis support system for electronics assembly operations. *Decision Support Systems* 25:251–269.
- Baldwin, J. M. 1896. A new factor in evolution. *American Naturalist* 30 (354):441–451.
- Bandyopadhyay, S., S. Saha, U. Maulik, and K. Deb. 2008. A simulated annealing-based multiobjective optimization algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation* 12 (3):269–283.
- Barnett, J. A. 1981. *Computational methods for a mathematical theory of evidence*. In *7th International Joint Conference on Artificial Intelligence (IJCAI'81)*. Vancouver, Canada. 868–875.
- Barr, A., and E. A. Feigenbaum. 1986. *The Handbook of Artificial Intelligence*, vol. 1. Addison-Wesley, Reading, MA.
- Bartak, R., M. A. Salido, and F. Rossi. 2010. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing* 21 (1):5–15.
- Beaudoin, J. F., S. Delisle, M. Dugre, and J. St-Pierre. 2005. Reengineering the knowledge component of a data warehouse-based expert diagnosis system. In *Database and Expert Systems Applications*, Lecture Notes in Computer Science, vol. 3588, edited by K. V. Andersen, J. Debenham and R. Wagner. Springer-Verlag, Berlin, Germany. 910–919.
- Bel, G., E. Bensana, D. Dubois, J. Erschler, and P. Esquivrol. 1989. A knowledge-based approach to industrial job-shop scheduling. In *Knowledge-based Systems in Manufacturing*, edited by A. Kusiak. Taylor and Francis, Philadelphia, PA. 207–246.
- Bellifemine, F., A. Poggi, and G. Rimassi. 1999. *JADE: A FIPA-compliant agent framework*. In *Proc. Practical Applications of Intelligent Agents and Multi-Agents*. 97–108.
- Bennett, M. E. 1987. Real-time continuous AI. *IEE Proceedings-D* 134 (4):272–277.
- Bennett, S., S. McRobb, and R. Farmer. 2010. *Object-oriented Systems Analysis and Design using UML*. 4th ed. McGraw-Hill, London, UK.
- Bernard, J. A. 1988. Use of a rule-based system for process control. *IEEE Control Systems Magazine* 8 (5):3–13.
- Berstel, B. 2002. *Extending the RETE Algorithm for event management*. In *Proceedings of Ninth International Symposium on Temporal Representation and Reasoning*. 49–51.
- Bichindaritz, I., and C. Marling. 2006. Case-based reasoning in the health sciences: What's next? *Artificial Intelligence in Medicine* 36(2):127–135.
- Booch, G., J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide*. 2nd ed. Addison-Wesley, Reading, MA.
- Booch, G., R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. 2007. *Object-oriented Analysis and Design with Applications*. 3rd ed. Addison-Wesley, Reading, MA.
- Bratko, I. 2011. *Prolog Programming for Artificial Intelligence*. 4th ed. Addison-Wesley, Reading, MA.
- Bratman, M. E. 1987. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.
- Bratman, M. E., D. J. Israel, and M. E. Pollack. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* 4:349–355.
- Brooks, R. A. 1991. *Intelligence without reason*. In *12th International Joint Conference on Artificial Intelligence (IJCAI'91)*. Sydney, Australia. 569–595.
- Brown, D., and B. Chandrasekaran. 1985. Expert systems for a class of mechanical design activity. In *Knowledge Engineering in Computer-aided Design*, edited by J. Gero. Elsevier, Amsterdam, Netherlands. 259–282.

- Bruninghaus, S., and K. D. Ashley. 2003. Combining case-based and model-based reasoning for predicting the outcome of legal cases. *Lecture Notes in Artificial Intelligence* 2689:65–79.
- Bruno, G., A. Elia, and P. Laface. 1986. A rule-based system to schedule production. *IEEE Computer* 19 (7):32–40.
- Brzykcy, G., J. Martinek, A. Meissner, and P. Skrzypczynski. 2001. *Multi-agent blackboard architecture for a mobile robot*. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2369–2374.
- Buchanan, B. G., and R. O. Duda. 1983. Principles of rule-based expert systems. In *Advances in Computers*, edited by M. C. Yovits. Academic Press, New York, NY.
- Buchanan, B. G., G. L. Sutherland, and E. A. Feigenbaum. 1969. Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. *Machine Intelligence* 4: 209–254.
- Bykat, A. 1990. NICBES-2, a nickel-cadmium battery expert system. *Applied Artificial Intelligence* 4:133–141.
- Caldas, L. G., and L. K. Norford. 2002. A design optimization tool based on a genetic algorithm. *Automation in Construction* 11 (2):173–184.
- Campolucci, P., S. Fiori, A. Uncini, and F. Piazza. 1997. *A new IIR-MLP learning algorithm for on-line signal processing*. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing ICASSP-97*. 3293–3296.
- Caridi, M., and S. Cavalieri. 2004. Multi-agent systems in production planning and control: An overview. *Production Planning & Control* 15 (2):106–118.
- Carpenter, G. A., and S. Grossberg. 1987. ART2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics* 26:4919–4930.
- Cernanský, M., M. Makula, and L. Benusková. 2007. Organization of the state space of a simple recurrent network before and after training on recursive linguistic structures. *Neural Networks* 20 (2):236–244.
- Chakravarthy, V. 2007. Info cardio advanced medical image analysis using GIIM, IISM with CBR & RBR intelligence. *Proceedings of World Academy of Science, Engineering and Technology* 33:315–319.
- Chalup, S. K., and A. D. Blair. 2003. Incremental training of first order recurrent neural networks to predict a context-sensitive language. *Neural Networks* 16 (7):955–972.
- Chandraker, R., A. A. West, and D. J. Williams. 1990. Intelligent control of adhesive dispensing. *Int. J. Computer Integrated Manufacturing* 3 (1):24–34.
- Charniak, E., and D. McDermott. 1985. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, MA.
- Cheetham, W. 2005. Tenth anniversary of plastics color matching. *Artificial Intelligence Magazine* 26 (3):51–61.
- Cheetham, W., and K. Goebel. 2007. Appliance call center: A successful mixed-initiative case study. *AI Magazine* 28 (2):89–100.
- Cherian, R. P., L. N. Smith, and P. S. Midha. 2000. A neural network approach for selection of powder metallurgy materials and process parameters. *Artificial Intelligence in Engineering* 14:39–44.
- Chow, M. Y., S. Altug, and H. J. Trussell. 1999. Heuristic constraints enforcement for training of and knowledge extraction from a fuzzy/neural architecture – Part I: Foundation. *IEEE Transactions on Fuzzy Systems* 7 (2):143–150.
- Choy, K. W., A. A. Hopgood, L. Nolle, and B. C. O'Neill. 2003. *Design and implementation of an inter-process communication model for an embedded distributed processing network*.

- In *International Conference on Software Engineering Research and Practice (SERP'03)*. Las Vegas, NV. 239–245.
- Coad, P., and E. Yourdon. 1990. *OOA: Object-oriented analysis*. Prentice Hall, Englewood Cliffs, NJ.
- Collinot, A., C. Le Pape, and G. Pinoteau. 1988. SONIA: A knowledge-based scheduling system. *Artificial Intelligence in Engineering* 3:86–94.
- Cook, R. D., D. S. Malkus, M. E. Plesha, and R. J. Witt. 2001. *Concepts and Applications of Finite Element Analysis*. 4th ed. John Wiley & Sons, New York, NY.
- Cunningham, P. 1998. A case study on the use of model-based systems for electronic fault diagnosis. *Artificial Intelligence in Engineering* 12:283–295.
- Cvijovic, D., and J. Klinowski. 1995. Taboo search – An approach to the multiple minima problem. *Science* 267:664–666.
- Dague, P., O. Jehl, P. Deves, P. Luciani, and P. Taillibert. 1991. When oscillators stop oscillating. In *International Joint Conference on Artificial Intelligence (IJCAI'91)*. Sydney, Australia. 1109–1115.
- Dague, P., P. Deves, Z. Zein, and J. P. Adam. 1987. DEDALE: An expert system in VM/Prolog. In *Knowledge-based expert systems in industry*, edited by J. Kriz. Ellis Horwood, Chichester, UK. 57–68.
- Dantzig, G. B. 1998. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.
- Dash, E. 1990. Diagnosing furnace problems with an expert system. *SPIE-Applications of Artificial Intelligence VIII* 1293:966–971.
- Dattero, R., J. J. Kanet, and E. M. White. 1989. Enhancing manufacturing planning and control systems with artificial intelligence techniques. In *Knowledge-based Systems in Manufacturing*, edited by A. Kusiak. Taylor and Francis, Philadelphia, PA. 137–150.
- Davidson, E. M., S. D. J. McArthur, and J. R. McDonald. 2003. A toolset for applying model-based reasoning techniques to diagnostics for power systems protection. *IEEE Transactions on Power Systems* 18 (2):680–687.
- Davis, L., and S. Coombs. 1987. Optimizing network link sizes with genetic algorithms. In *Modelling and Simulation Methodology: Knowledge systems paradigms*, edited by M. S. Elzas, T. I. Oren and B. P. Zeigler. North Holland, Amsterdam, Netherlands.
- Davis, L., ed. 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY.
- de Castro, L. N., and J. I. Timmis. 2003. Artificial immune systems as a novel soft computing paradigm. *Soft Computing* 7:526–544.
- de Koning, K., B. Bredeweg, J. Breuker, and B Wielinga. 2000. Model-based reasoning about learner behaviour. *Artificial Intelligence* 117 (2):173–229.
- de Mantaras, R.L., D. McSherry, D. Bridge, D. Leake, B. Smyth, S. Craw, B. Faltings, M-L Maher, M. T. Cox, K. Forbus, M. Keane, A. Aamody, and I. Watson 2006. Retrieval, reuse, revision, and retention in case-based reasoning. *Knowledge Engineering Review* 20 (3):215–240.
- DeKleer, J. 1986a. An assumption-based TMS. *Artificial Intelligence* 28:127–162.
- DeKleer, J. 1986b. Extending the ATMS. *Artificial Intelligence* 28:163–196.
- DeKleer, J. 1986c. Problem-solving with the ATMS. *Artificial Intelligence* 28:197–224.
- Dekleer, J., and B. C. Williams. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32:97–130.
- Demaid, A., and J. Zucker. 1988. A conceptual model for materials selection. *Metals and Materials*, May, 291–297.

- Demaid, A., and J. Zucker. 1992. Prototype-oriented representation of engineering design knowledge. *Artificial Intelligence in Engineering* 7:47–61.
- Dietterich, T. G., and D. G. Ullman. 1987. FORLOG: A logic-based architecture for design. In *Expert Systems in Computer-aided Design*, edited by J. Gero. Elsevier, Amsterdam, Netherlands, 1–17.
- Dignum, V. 2018. Ethics in artificial intelligence: Introduction to the special issue. *Ethics and Information Technology* 20 (1).
- Diniz, P. S. R., E. A. B. da Silva, and S. L. Netto. 2010. *Digital Signal Processing: System Analysis and Design*. 2nd ed. Cambridge University Press, Cambridge, UK.
- Dorigo, M. 2001. Ant algorithms solve difficult optimization problems. *Lecture Notes in Artificial Intelligence* 2159:11–22.
- Dorigo, M., and T. Stützle. 2004. *Ant Colony Optimization*. MIT Press, Cambridge, MA.
- Duda, R. O., J. G. Gashnig, and P. E. Hart. 1979. Model design in the PROSPECTOR consultant system for mineral exploration. In *Expert Systems in the Micro-electronic Age*, edited by D. Michie. Edinburgh University Press, Edinburgh, UK.
- Duda, R. O., and P. E. Hart. 1972. Use of the Hough transform to detect lines and curves in pictures. *Communications of the ACM* 15 (1):11–15.
- Duda, R. O., and P. E. Hart. 1973. *Pattern Classification and Scene Analysis*. Wiley, New York, NY.
- Duda, R. O., P. E. Hart, and N. J. Nilsson. 1976. Subjective Bayesian methods for rule-based inference systems. In *National Computer Conference*. AFIPS. 1075–1082.
- Durfee, E. H., V. R. Lesser, and D. D. Corkhill. 1985. Increasing coherence in a distributed problem solving network. In *9th International Joint Conference on Artificial Intelligence (IJCAI'85)*. Los Angeles, CA. 1025–1030.
- Dyer, M. G., M. Flowers, and J. Hodges. 1986. EDISON: An engineering design invention system operating naively. *Artificial Intelligence in Engineering* 1:36–44.
- Elman, J. L. 1991. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning* 7 (2–3):195–225.
- El-Mihoub, T., A. A. Hopgood, and L. Nolle. 2020. Self-adaptive learning for hybrid genetic algorithms. *Evolutionary Intelligence*.
- El-Mihoub, T., A. A. Hopgood, L. Nolle, and A. Battersby. 2004. Performance of hybrid genetic algorithms incorporating local search. In *18th European Simulation Multiconference (ESM2004)*, edited by G. Horton. Magdeburg, Germany. 154–160.
- El-Mihoub, T., A. A. Hopgood, L. Nolle, and A. Battersby. 2006a. Hybrid genetic algorithms – A review. *Engineering Letters* 13:124–137.
- El-Mihoub, T., A. A. Hopgood, L. Nolle, and A. Battersby. 2006b. Self-adaptive Baldwinian search in hybrid genetic algorithms. In *Computational Intelligence: Theory and Applications – Advances in Soft Computing*, vol. 38, edited by B. Reusch. Springer, Berlin, Germany. 597–602.
- Eriksson, H. 1996. Expert systems as knowledge servers. *IEEE Intelligent Systems* 11 (3):14–19.
- Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. 1980. The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys* 12 (2):213–253.
- Feigenbaum, E. A.. 1988. *Blackboard systems*, edited by R. S. Englemore and A. J. Morgan. Addison-Wesley, Reading, MA.

- Fenton, W. G., T. M. McGinnity, and L. P. Maguire. 2001. Fault diagnosis of electronic systems using intelligent techniques: A review. *IEEE Transactions on Systems Man and Cybernetics Part C – Applications and Reviews* 31 (3):269–281.
- Ferguso, A., and D. Bridge. 2000. Options for query revision when interacting with case retrieval systems. *Expert Update* 3 (1):16–27.
- Ferguson, I. A. 1995. Integrated control and coordinated behaviour: A case for agent models. In *Intelligent Agents: Theories, Architectures and Languages*, edited by M. Wooldridge and N. R. Jennings. Springer-Verlag, Berlin, Germany. 203–218.
- Fernandez, S., R. Aler, and D. Borrajo. 2005. Machine learning in hybrid hierarchical and partial-order planners for manufacturing domains. *Applied Artificial Intelligence* 19 (8):783–809.
- Fielden, G. B. R. 1963. *Engineering design*. HMSO, London, UK.
- Fikes, R. E., and N. J. Nilsson. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.
- Fikes, R. E., P. E. Hart, and N. J. Nilsson. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.
- Finin, T., Y. Labrou, and J. Mayfield. 1997. KQML as an agent communication language. In *Software Agents*, edited by J. M. Bradshaw. MIT Press, Cambridge, MA. 291–316.
- Fink, P. K., and J. C. Lusth. 1987. Expert systems and diagnostic expertise in the mechanical and electrical domains. *IEEE Transactions on Systems, Man, and Cybernetics* 17 (3):340–349.
- Firth, N. 2007. Stepping up a gear: The latest version of Honda's Asimo boasts a host of features that takes the world's most advanced robot closer to full autonomy. *Engineer* 293:26–27.
- Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19 (1):17–37.
- Frank, S. L. 2006. Learn more by training less: Systematicity in sentence processing by recurrent networks. *Connection Science* 18 (3):287–302.
- Franklin, G. F., J. D. Powell, and A. Emami-Naeini. 2020. *Feedback Control of Dynamic Systems*. 8th ed. Pearson Education, Upper Saddle River, NJ.
- Fulton, S. L., and C. O. Pepe. 1990. An introduction to model-based reasoning. *AI Expert* 5 (1):48–55.
- Galindo, C., J. A. Fernandez-Madrigal, and J. Gonzalez. 2004. Improving efficiency in mobile robot task planning through world abstraction. *IEEE Transactions on Robotics and Automation* 20 (4):677–690.
- Galuszka, A., and A. Swierniak. 2010. Planning in multi-agent environment using STRIPS representation and non-cooperative equilibrium strategy. *Journal of Intelligent & Robotic Systems* 58 (3–4):239–251.
- Garrett, C. R., T. Lozano-Pérez, and L. P. Kaelbling. 2017. STRIPS planning in infinite domains. Preprint arXiv:1701.00287.
- Gers, F. A., J. Schmidhuber, and F. Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural Computation* 12 (10): 2451–2471.
- Ghosh-Dastidar, S., and H. Adeli. 2009. Spiking neural networks. *International Journal of Neural Systems* 19 (4):295–308.
- Glover, F., and M. Laguna. 1997. *Tabu Search*. Kluwer, Norwell, MA.
- Gogos, C., P. Alefragis, and E. Housos. 2005. Public enterprise water pump scheduling system. In *10th IEEE Conference on Emerging Technologies and Factory Automation*.

- Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading, MA.
- Goldberg, D. E., B. Korb, and K. Deb. 1989. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems* 3:493–530.
- Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NIPS'14)*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Red Hook, NY. 3:2672–2680.
- Gorman, B., and M. Humphrys. 2007. *Imitative learning of combat behaviours in first-person computer games*. In *Proceedings of 10th International Conference on Computer Games (CGAMES07)*. Louisville, KY. 85–90.
- Gouaillier, D., V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier. 2009. *Mechatronic design of NAO humanoid*. In *IEEE Int. Conf. on Robotics and Automation*. Kobe, Japan.
- Gray, F. 1953. Pulse code communication. U.S. Patent 2,632,058.
- Greenberg, J., and C. N. Reid. 1981. *A simple design task (with the aid of a microcomputer)*. In *2nd International Conference on Engineering Software*. Southampton, UK. 926–942.
- Greff, K., R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. 2017. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems* 28 (10): 2222–2232.
- Griewank, A. O. 1981. Generalized descent for global optimization. *Journal of Optimization Theory and Applications* 34:11–39.
- Güera, D. and E. J. Delp. 2018. Deepfake video detection using recurrent neural networks. In *15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, Auckland, New Zealand. 1–6.
- Hallam, N. J., A. A. Hopgood, and N. Woodcock. 1990. Defect classification in welds using a feedforward network within a blackboard system. In *International Neural Network Conference (INNC'90)*. Paris, France. 353–356.
- Halmshaw, R. 1987. *Non-destructive Testing*. Edward Arnold, London, UK.
- Harel, D. 1988. On visual formalisms. *Communications of the ACM* 31 (5):514–530.
- Hart, P. E., R. O. Duda, and M. T. Einaudi. 1978. PROSPECTOR: A computer-based consultation system for mineral exploration. *Math Geology* 10 (5):589–610.
- Hausen, R. and B. E. Robertson. 2020. Morpheus: A deep learning framework for the pixel-level analysis of astronomical image data. *The Astrophysical Journal Supplement Series* 248(1).
- Hecht-Nielson, R. 1990. *Neurocomputing*. Addison-Wesley, Reading, MA.
- Hendler, J., A. Tate, and M. Drummond. 1990. AI planning: Systems and techniques. *AI Magazine* 11 (2):61–77.
- Herrera F. and M. Lozano. 1996. Adaptation of genetic algorithm parameters based on fuzzy logic controllers. In *Genetic Algorithms and Soft Computing*, edited by F. Herrera and J. L. Verde-Gay, Physica-Verlag, Heidelberg, Germany. 95–125.
- Hinton, G. E. and R. R. Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science*, 313 (5786):504–507.
- Hirschman, I. I., and D. V. Widder. 2005. *The Convolution Transform*. Dover Publications, Mineola, NY.
- Hochreiter, S., and J. Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9 (8): 1735–1780.

- Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
- Holland, J. H., and J. S. Reitman. 1978. Cognitive systems based on adaptive algorithms. In *Pattern-directed Inference Systems*, edited by D. A. Waterman and F. Hayes-Roth. Academic Press, New York, NY. 313–329.
- Holmes, N. 2003. Artificial intelligence: Arrogance or ignorance? *IEEE Computer* 36 (11): 118–120.
- Hopfield, J. J. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proc. National Academy of Science. USA* 2554–2558.
- Hopgood, A. A. 1989. An inference mechanism for selection, and its application to polymers. *Artificial Intelligence in Engineering* 4:197–203.
- Hopgood, A. A. 1994. Rule-based control of a telecommunications network using the blackboard model. *Artificial Intelligence in Engineering* 9 (1): 29–38.
- Hopgood, A. A. 2003. Artificial intelligence: Hype or reality? *IEEE Computer* 36 (5): 24–28.
- Hopgood, A. A. 2005. The state of artificial intelligence. In *Advances in Computers*, edited by M. V. Zelkowitz. Elsevier, Oxford, UK. 65:1–75.
- Hopgood, A. A., and A. J. Hopson. 1991. *The common traffic model: A universal model for communications networks*. In *Institution of Radio and Electronic Engineers Conference (IREECON'91)*. Sydney, Australia. 61–64.
- Hopgood, A. A., and A. Mierzejewska. 2008. Transform ranking: A new method of fitness scaling in genetic algorithms. In *Research and Development in Intelligent Systems XXV*, edited by M. Brammer, F. Coenen and M. Petridis. Springer, Berlin, Germany. 349–354.
- Hopgood, A. A., H. J. Phillips, P. D. Picton, and N. St. J. Braithwaite. 1998. Fuzzy logic in a blackboard system for controlling plasma deposition processes. *Artificial Intelligence in Engineering* 12:253–260.
- Hopgood, A. A., N. Woodcock, N. J. Hallam, and P. D. Picton. 1993. Interpreting ultrasonic images using rules, algorithms and neural networks. *European Journal of Nondestructive Testing* 2:135–149.
- Hornick, K., M. Stinchcombe, and H. White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2 (5):359–366.
- Hossack, J., S. D. J. McArthur, E. Davidson, J. R. McDonald, and T. Cumming. 2002. A multi-agent intelligent interpretation system for power system disturbance diagnosis. In *Applications and Innovations in Intelligent Systems X*, edited by A. L. Macintosh, R. Ellis and F. Coenen. Springer-Verlag, London, UK. 91–104.
- Howe, A. E., P. R. Cohen, J. R. Dixon, and M. K. Simmons. 1986. DOMINIC: A domain-independent program for mechanical engineering design. *Artificial Intelligence in Engineering* 1:23–28.
- Hsu, E., and D. L. McGuinness. 2003. *KSL wine agent: A semantic web testbed application*. In *International Workshop on Description Logics (DL2003)*. Rome, Italy.
- Huang, J. K., M. T. Ho, and R. L. Ash. 1992. Expert systems for automated maintenance of a Mars oxygen production system. *Journal of Spacecraft and Rockets* 29:425–431.
- Huang, J. T., and Y. S. Liao. 2000. A wire-EDM maintenance and fault-diagnosis expert system integrated with an artificial neural network. *International Journal of Production Research* 38:1071–1082.
- Hunt, J. 1997. Case-based diagnosis and repair of software faults. *Expert Systems* 14:15–23.
- Jahan, A., M. Y. Ismail, S. M. Sapuan, and F. Mustapha. 2010. Material screening and choosing methods – A review. *Materials & Design* 31 (2):696–705.

- Jang, J.-S. R. 1993. ANFIS: Adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man, and Cybernetics* 23 (03):665–685.
- Jennings, A. J. 1989. *Artificial intelligence: A tool for productivity*. In *Institution of Engineers (Australia) National Conference*. Perth, Australia.
- Jennings, N. R. 2000. On agent-based software engineering. *Artificial Intelligence* 117:277–296.
- John, R., and S. Coupland. 2007. Type-2 fuzzy logic: A historical view. *IEEE Computational Intelligence Magazine* 2 (1):57–62.
- Johnson, J. H., and P. D. Picton. 1995. *Concepts in Artificial Intelligence*. Butterworth-Heinemann, Oxford, UK.
- Johnson, J. H., N. J. Hallam, and P. D. Picton. 1990. *Safety critical neurocomputing: Explanation and verification in knowledge augmented neural networks*. In *IEE Colloquium on Human-Computer Interaction*. London, UK.
- Jozefowicz, R., W. Zaremba, and I. Sutskever. 2015. An empirical exploration of recurrent network architectures. In *32nd International Conference on Machine Learning, ICML 2015*, 3:2332–2340.
- Kang, C. W., and M. W. Golay. 1999. A Bayesian belief network-based advisory system for operational availability focused diagnosis of complex nuclear power systems. *Expert Systems with Applications* 17:21–32.
- Karr, C. L. 1991a. Applying genetics to fuzzy logic. *AI Expert* 6 (3):38–43.
- Karr, C. L. 1991b. Genetic algorithms for fuzzy controllers. *AI Expert* 6 (2):26–33.
- Kavakli, M. 2001. NoDes: kNObledge-based modeling for detailed DESign process – From analysis to implementation. *Automation in Construction* 10:399–416.
- Khmeleva, E., A. A. Hopgood, L. Tipi, and M. Shahidan. 2018. Fuzzy-logic controlled genetic algorithm for the rail-freight crew-scheduling problem. *KI – Künstliche Intelligenz* 32 (1):61–75.
- Kim, K. H., and J. K. Park. 1993. Application of hierarchical neural networks to fault diagnosis of power systems. *International Journal of Electrical Power and Energy Systems* 15 (2):65–70.
- Kim, N. H., B. V. Sankar, and A. V. Kumar. 2018. *Introduction to Finite Element Analysis and Design*. 2nd ed. John Wiley & Sons, New York, NY.
- Kim, S. H., and N. P. Suh. 1989. Formalizing decision rules for engineering design. In *Knowledge-based Systems in Manufacturing*, edited by A. Kusiak. Taylor and Francis, Philadelphia, PA. 33–44.
- Kinny, D., and M. Georgeff. 1991. Commitment and effectiveness in situated agents. In *12th International Joint Conference on Artificial Intelligence (IJCAI'91)*. Sydney, Australia. 82–88.
- Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing: Quantitative studies. *Science* 220:671–680.
- Klarreich, E. 2002. Inspired by immunity. *Nature* 415:468–470.
- Knight, K. 1990. Connectionist ideas and algorithms. *Communications of the ACM* 33 (11):59–74.
- Kohonen, T. 1987. Adaptive, associative, and self-organizing functions in neural computing. *Applied Optics* 26 (23):4910–4918.
- Kohonen, T. 1988. *Self-organization and Associative Memory*. 2nd ed. Springer-Verlag, Berlin, Germany.
- Komzsik, L. 2009. *What Every Engineer Should Know About Computational Techniques of Finite Element Analysis*. 2nd ed. CRC Press, Boca Raton, FL.

- Koza, J. R., I. I. I. Bennett, H. Forrest, D. Andre, and M. A. Keane (eds.) 1999. *Genetic Programming III: Darwinian invention and problem solving*. Morgan Kaufmann, San Francisco, CA.
- Kreinovich, V., C. Quintana, and O. Fuentes. 1993. Genetic algorithms: What fitness scaling is optimal? *Cybernetics and Systems: An International Journal* 24:9–26.
- Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM* 60 (6):84–90.
- Kumar, M., N. Stoll, D. Kaber, K. Thurow, and R. Stoll. 2007. *Fuzzy filtering for an intelligent interpretation of medical data*. IEEE International Conference on Automation Science and Engineering. Scottsdale, AZ. 225–230.
- Laffey, T. J., P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read. 1988. Real-time knowledge-based systems. *AI Magazine* 9 (1):27–45.
- Laird, J. E., A. Newell, and P. S. Rosenbloom. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33:1–64.
- Laird, J. E., P. S. Rosenbloom, and A. Newell. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1:11–46.
- Lamarck, J.-B. 1809. *Philosophie Zoologique*. Paris, France.
- LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11):2278–2323.
- Lee, C. C. 1990a. Fuzzy logic in control systems: Fuzzy logic controller – Part I. *IEEE Transactions on Systems, Man and Cybernetics* 20 (2):404–418.
- Lee, C. C. 1990b. Fuzzy logic in control systems: Fuzzy logic controller – Part II. *IEEE Transactions on Systems, Man and Cybernetics* 20 (2):419–435.
- Lee, S., and H. Rowlands, H. 2005. Finding robust optima with a diploid genetic algorithm. *International Journal of Simulation: Systems, Science and Technology* 6 (9):73–79.
- Leitch, R., R. Kraft, and R. Luntz. 1991. RESCU: A real-time knowledge based system for process control. *IEE Proceedings-D* 138 (3):217–227.
- Lesser, V. R., J. Pavlin, and E. Durfee. 1988. Approximate processing in real-time problem solving. *AI Magazine* 9 (1):49–61.
- Lesser, V. R., R. D. Fennell, L. D. Erman, and D. R. Reddy. 1975. Organization of the Hearsay-II speech understanding system. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23:11–23.
- Li, G., A. A. Hopgood, and M. J. Weller. 2003. Shifting matrix management: A model for multi-agent cooperation. *Engineering Applications of Artificial Intelligence* 16:191–201.
- Lippmann, R. P. 1987. An introduction to computing with neural nets. *IEEE ASSP Magazine* 4 (2):4–22.
- Lirov, Y. 1990. Systematic invention for knowledge engineering. *AI Expert* 5 (7):28–33.
- Liu, B. 1988. Scheduling via reinforcement. *Artificial Intelligence in Engineering* 3:76–85.
- Lu, Y., T. Q. Chen, and B. Hamilton. 1998. A fuzzy diagnostic model and its application in automotive engineering diagnosis. *Applied Intelligence* 9:231–243.
- Lumer, E. D., and B. Faieta. 1994. Diversity and adaptation in populations of clustering ants. In *Proc. 3rd Int. Conf. on Simulation of Adaptive Behavior: From animal to animats*, edited by D. Cliff, P. Husbands, J. Meyer and S. Wilson. MIT Press/Bradford Books, Cambridge, MA.
- Maass, W. 1997. Networks of spiking neurons: The third generation of spiking neural network models. *Neural Networks* 10 (9):1659–1671.

- Maderlechner, G., E. Egeli, and F. Klein. 1987. Model guided interpretation based on structurally related image primitives. In *Knowledge-based expert systems in industry*, edited by J. Kriz. Ellis Horwood, Chichester, UK. 91–97.
- Mamdani, E. H. 1977. Application of fuzzy logic to approximate reasoning using linguistic synthesis. *IEEE Transactions on Computers* 26 (12):1182–1191.
- Manoj, C., and N. Nagarajan. 2003. The application of artificial neural networks to magnetotelluric time-series analysis. *Geophysical Journal International* 153 (2):409–423.
- Marefat, M., and J. Britanik. 1997. Case-based process planning using an object-oriented model representation. *Robotics and Computer Integrated Manufacturing* 13 (3):229–251.
- Mateis, C., M. Stumptner, and F. Wotawa. 2000. Locating bugs in Java programs – First results of the Java diagnosis experiments project. *Lecture Notes in Artificial Intelligence* 1821:174–183.
- Matsui, Y., M. Kanoh, S. Kato, and H. Itoh. 2008. Generating interactive facial expression of communication robots using simple recurrent network. In *PRICAI 2008: Trends in Artificial Intelligence*, edited by T. B. Ho and Z. H. Zhou. 1016–1021.
- Matsui, Y., M. Kanoh, S. Kato, T. Nakamura, and H. Itoh. 2009. *Evaluating a model for generating interactive facial expressions using simple recurrent network*. In *SMC 2009: IEEE International Conference on Systems, Man and Cybernetics*. 1639–1644.
- McCormick, G., and R. S. Powell. 2004. Derivation of near-optimal pump schedules for water distribution by simulated annealing. *Journal of the Operational Research Society* 55 (7):728–736.
- McCulloch, W. S., and W. Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics* 5 (4): 115–133.
- McQueen, T., A. A. Hopgood, T. J. Allen, and J. A. Tepper. 2005. Extracting finite structure from infinite language. *Knowledge-Based Systems* 18 (4–5):135–141.
- Melioli, G., C. Spenser, G. Reggiardo, G. Passalacqua, E. Compalati, A. Rogkakou, A. M. Riccio, E. Di Leo, E. Nettis, and G. W. Canonica. 2014. Allergenius, an expert system for the interpretation of allergen microarray results. *The World Allergy Organization Journal* 7 (1):15.
- Mendel, J. M. 2007. Type-2 fuzzy sets and systems: An overview. *IEEE Computational Intelligence Magazine* 2 (1):20–29.
- Mettrey, W. 1991. A comparative evaluation of expert system tools. *IEEE Computer* 24 (2):19–31.
- Micalizio, R. 2009. *A distributed control loop for autonomous recovery in a multi-agent plan*. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*. Pasadena, USA. 1760–1765.
- Michie, D., and R. A. Chambers. 1968. BOXES: An experiment in adaptive control. In *Machine Intelligence 2*, edited by E. Dale and D. Michie. Oliver and Boyd, Edinburgh, UK. 137–152.
- Milne, R. 1987. Strategies for diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics* 17 (3):333–339.
- Minton, S., J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40:63–118.
- Mintzberg, H. 1979. *The Structuring of Organizations*. Prentice-Hall, Englewood Cliffs, NJ.
- Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

- Mitchell, T. M., P. E. Utgoff, and R. Banerji. 1983. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine learning: An artificial intelligence approach*, vol. 1, edited by R. Michalski, J. G. Carbonell and T. M. Mitchell, 163–190.
- Moallem, P., and S. A. Monadjemi. 2010. An efficient MLP-learning algorithm using parallel tangent gradient and improved adaptive learning rates. *Connection Science* 22 (4):373–392.
- Montani, S., P. Magni, R. Bellazzi, C. Larizza, A. V. Roudsari, and E. R. Carson. 2003. Integrating model-based decision support in a multi-modal reasoning system for managing type 1 diabetic patients. *Artificial Intelligence in Medicine* 29 (1–2):131–151.
- Moody, S. 1980. The role of industrial design in technological innovation. *Design Studies* 1 (6):329–339.
- Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. T. Zhang, and S. Malik. 2001. *Chaff: Engineering an efficient SAT solver*. In *38th Design Automation Conference Proceedings*. ACM, New York, NY. 530–535.
- Motta, E., M. Eisenstadt, K. Pitman, and M. West. 1988. Support for knowledge acquisition in the Knowledge Engineer's Assistant (KEATS). *Expert Systems* 5 (1):6–28.
- Mouelhi, O., P. Couturier, and T. Redarce. 2009. *Multicriteria optimization and evaluation in complex products design*. In *2009 International Conference on Computers and Industrial Engineering*. IEEE, New York, NY. 590–595.
- Muraina, I. O., M. A. Rahman, and I. A. Adeleke. 2016. Statistical approaches and decision making towards bivariate and multivariate analyses with Visirule. *British Journal of Education, Society & Behavioural Science* 14 (2): 1–10.
- Murthy, S. S., and S. Addanki. 1987. PROMPT: An innovative design tool. In *Expert Systems in Computer-Aided Design*, edited by J. Gero. North-Holland, Amsterdam, Netherlands. 323–341.
- Mustonen, V., and M. Lässig. 2009. From fitness landscapes to seascapes: Non-equilibrium dynamics of selection and adaptation. *Trends in Genetics* 25 (3):111–119.
- Mustonen, V., and M. Lässig. 2010. Fitness flux and ubiquity of adaptive evolution. *Proceedings of the National Academy of Sciences* 107 (9):4248–4253.
- Mutawa, A. M., and M. A. Alzuwawi. 2019. Multilayered rule-based expert system for diagnosing uveitis. *Artificial Intelligence in Medicine* 99:101691.
- Navichandra, D., and D. H. Marks. 1987. Design exploration through constraint relaxation. In *Expert Systems in Computer-aided Design*, edited by J. Gero. Elsevier, Amsterdam, Netherlands. 481–509.
- Netten, B. D. 1998. Representation of failure context for diagnosis of technical applications. *Advances in Case-based Reasoning* 1488:239–250.
- Newell, A. 1982. The knowledge level. *Artificial Intelligence* 18 (1):87–127.
- Niemüller, T., A. Ferrein, and G. Lakemeyer. 2010. A Lua-based behavior engine for controlling the humanoid robot Nao. In *RoboCup 2009: Robot Soccer World Cup XIII*, edited by J. Baltes, M. Lagoudakis, T. Naruse, S. Ghidary. Lecture Notes in Computer Science, vol. 5949. Springer, Berlin, Germany. 240–251.
- Nii, H. P. 1986. Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine* 7:38–53.
- Nolle, L., D. A. Armstrong, A. A. Hopgood, and A. Ware. 2002a. Simulated annealing and genetic algorithms applied to finishing mill optimisation for hot rolling of wide steel strip. *Int. Journal of Knowledge-Based Intelligent Engineering Systems* 6:104–110.

- Nolle, L., A. Goodyear, A. A. Hopgood, P. D. Picton, and N. S. Braithwaite. 2002b. Automated control of an actively compensated Langmuir probe system using simulated annealing. *Knowledge-Based Systems* 15:349–354.
- Nolle, L., K. C. P. Wong, and A. A. Hopgood. 2002c. DARBS: A distributed blackboard system. In *Research and Development in Intelligent Systems XVIII*, edited by M. Bramer, F. Coenen and A. Preece. Springer-Verlag, London, UK. 161–170.
- Nolle, L., A. Goodyear, A. A. Hopgood, P. D. Picton, and N. S. Braithwaite. 2005. Improved simulated annealing with step width adaptation for Langmuir probe tuning. *Engineering Optimization* 37:463–477.
- Nolle, L., D. A. Armstrong, A. A. Hopgood, and J. A. Ware. 1999. Optimum work roll profile selection in the hot rolling of wide steel strip using computational intelligence. *Lecture Notes in Computer Science* 1625:435–452.
- O’Leary, D. E. 1998. Knowledge acquisition from multiple experts: An empirical study. *Management Science* 44 (8):1049–1058.
- Oh, K. W., and C. Quek. 2001. BBIPS: A blackboard-based integrated process supervision. *Engineering Applications of Artificial Intelligence* 14 (6):703–714.
- Pascoe, G. A. 1986. Elements of object-oriented programming. *Byte* 11 (8):139–144.
- Passow, B. N., M. A. Gongora, S. Coupland, and A. A. Hopgood. 2008. *Real-time evolution of an embedded controller for an autonomous helicopter*. In *Proc. 2008 IEEE Congress on Evolutionary Computation (CEC 2008)*. Hong Kong.
- Pedersen, G. K. M., and D. E. Goldberg. 2004. Dynamic uniform scaling for multiobjective genetic algorithms. *Lecture Notes in Computer Science* 3103:11–23.
- Penrose, R. 1999. *The Emperor’s New Mind*. Oxford University Press, Oxford, UK.
- Philipp, A., P. M. Della-Marta, J. Jacobbeit, D. R. Fereday, P. D. Jones, A. Moberg, and H. Wanner. 2007. Long-term variability of daily North Atlantic-European pressure patterns since 1850 classified by simulated annealing clustering. *Journal of Climate* 20 (16):4065–4095.
- Picton, P. D., J. H. Johnson, and N. J. Hallam. 1991. *Neural networks in safety-critical systems*. In *3rd International Congress on Condition Monitoring and Diagnostic Engineering Management*. Southampton, UK.
- Pons, D. J., and J. K. Raine. 2005. Design mechanisms and constraints. *Research in Engineering Design* 16 (1–2):73–85.
- Portinale, L., D. Magro, and P. Torasso. 2004. Multi-modal diagnosis combining case-based and model-based reasoning: A formal and experimental analysis. *Artificial Intelligence* 158 (2):109–153.
- Poslad, S. and P. Charlton. 2001. Standardizing agent interoperability: The FIPA approach. In *Multi-Agent Systems and Applications*, edited by M. Luck, V. Mařík, O. Štěpánková, and R. Trappl. Lecture Notes in Computer Science, vol. 2086. Springer, Berlin, Germany. 98–117.
- Prang, J., H. D. Huemmer, and W. Geisselhardt. 1996. A system for simulation, monitoring and diagnosis of controlled continuous processes with slow dynamics. *Knowledge-based Systems* 9:525–530.
- Price, C. J. 1998. Function-directed electrical design analysis. *Artificial Intelligence in Engineering* 12:445–456.
- Price, C. J., and J. E. Hunt. 1991. Automating FMEA through multiple models. In *Research and development in expert systems VIII*, edited by I. Graham and R. Milne. Cambridge University Press, Cambridge, UK, 25–39.

- Price, C. J., and J. Hunt. 1989. Simulating mechanical devices. In *Pop-11 Comes of Age: The advancement of an AI programming language*, edited by J. A. D. W. Anderson. Ellis Horwood, Chichester, UK, 217–237.
- Quinlan, J. R. 1983. Inferno: A cautious approach to uncertain inference. *The Computer Journal* 26 (2):255–269.
- Ramaswamy, S., T. J. Cutright, and H. K. Qammar. 2005. Control of a continuous bioreactor using model predictive control. *Process Biochemistry* 40 (8):2763–2770.
- Ramos, V., and F. Almeida. 2000. Artificial ant colonies in digital image habitats – A mass behaviour effect study on pattern recognition. In *Proc 2nd Int. Workshop on Ant Algorithms: From ant colonies to artificial ants*, edited by M. Dorigo, M. Middendorf, and T. Stüzle. Brussels, Belgium, 113–116.
- Refanidis, I., and I. Vlahavas. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.
- Reid, C. N., and J. Greenberg. 1980. An exercise in materials selection. *Metals and Materials*, July, 385–387.
- Rescher, N. 1976. *Plausible Reasoning*. Van Gorcum, Assen, Netherlands.
- Riesbeck, C. K., and R. C. Schank. 1989. *Inside Case-based Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Rumbaugh, J., I. Jacobson, and G. Booch. 2010. *The Unified Modeling Language Reference Manual*. 2nd ed. Addison-Wesley, Reading, MA.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986a. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the micro-structures of cognition*, edited by D. E. Rumelhart and J. L. McClelland. MIT Press, Cambridge, MA.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986b. Learning representations by back-propagating errors. *Nature* 323 (9):533–536.
- Russell, S., and P. Norvig. 2016. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education, London, UK.
- Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.
- Sadjadi, F. 2004. Comparison of fitness scaling functions in genetic algorithms with applications to optical processing. In *Proc. SPIE: Optical Information Systems II*, edited by B. Javidi and D. Psaltis. 356–364.
- Saini, R., N. J. Ahuja, and K. D. Bahukhandi. 2016. Fuzzy logic based advisory for handling landfill operational problems for early warning and emergency response planning. *International Journal of ChemTech Research* 9 (08):282–297.
- Sakagami, Y., R. Watanabe, C. Aoyama, S. Matsunaga, N. Higaki, and K. Fujimura. 2002. *The intelligent ASIMO: System overview and integration*. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Lausanne, Switzerland.
- Sammut, C., and D. Michie. 1991. Controlling a black-box simulation of a spacecraft. *AI Magazine* 12 (1):56–63.
- Scarl, E. A., J. R. Jamieson, and C. I. Delaune. 1987. Diagnosis and sensor validation through knowledge of structure and function. *IEEE Transactions on Systems, Man, and Cybernetics* 17 (3):360–368.
- Schafer, G. 1976. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, NJ.

- Schmedding, F., N. Sawas, and G. Lausen. 2007. Adapting the Rete-algorithm to evaluate F-logic rules. In *Lecture Notes in Computer Science*, 4824 LNCS:166–173. Springer Verlag, Orlando, FL.
- Schwefel, H.-P. 1981. *Numerical Optimization of Computer models*. John Wiley & Sons, New York, NY.
- Sejnowski, T. J., and C. R. Rosenberg. 1986. NETtalk: A parallel network that learns to read aloud. *Cognitive Science* 14:179–211.
- Shadbolt, N., W. Hall, and T. Berners-Lee. 2006. The semantic web revisited. *IEEE Intelligent Systems* 21 (3):96–101.
- Shen, W. M. 2002. Distributed manufacturing scheduling using intelligent agents. *IEEE Intelligent Systems* 17 (1):88–94.
- Shen, W. M., Q. Hao, H. J. Yoon, and D. H. Norrie. 2006. Applications of agent-based systems in intelligent manufacturing: An updated review. *Advanced Engineering Informatics* 20 (4):415–431.
- Shieh, J-S, C-F Chou, S-J Huang, and M-C Kao. 2004. Intracranial pressure model in intensive care unit using a simple recurrent neural network through time. *Neurocomputing* 57:239–256.
- Shortliffe, E. H. 1976. *Computer-Based Medical Consultations: MYCIN*. Elsevier, New York, NY.
- Shortliffe, E. H., and B. G. Buchanan. 1975. A model of inexact reasoning in medicine. *Mathematical Biosciences* 23:351–379.
- Skyttner, L. 2005. *General Systems Theory: Problems, Perspectives, Practice*. 2nd ed. World Scientific, Hackensack, NJ.
- Smith, R. G., and R. Davis. 1981. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics* 11 (1):61–70.
- Spenser, C. 2007. Drawing on your knowledge with VisiRule. *IEEE Potentials* 26 (1): 20–25.
- Sripada, N. R., D. G. Fisher, and A. J. Morris. 1987. AI application for process regulation and process control. *IEE Proceedings-D* 134 (4):251–259.
- Sriram, D., G. Stephanopoulos, R. Logcher, D. Gossard, N. Groleau, D. Serrano, and D. Navinchandra. 1989. Knowledge-based system applications in engineering design: Research at MIT. *AI Magazine* 10 (3):79–96.
- Stahl, B. C. 2004. Information, ethics, and computers: The problem of autonomous moral agents. *Minds and Machines* 14:67–83.
- Steele, G. L. 1990. *Common Lisp: The language*. 2nd ed. Digital Press, Bedford, MA.
- Steels, L. 1989. Diagnosis with a function-fault model. *Applied Artificial Intelligence* 3 (2–3):129–153.
- Stroustrup, B. 1988. What is object-oriented programming? *IEEE Software* 5 (3):10–20.
- Stuart, C. J. 1985. An implementation of a multi-agent plan synchronizer. In *9th International Joint Conference on Artificial Intelligence (IJCAI'85)*. Los Angeles, CA. 1031–1033.
- Suman, B., and P. Kumar. 2006. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the Operational Research Society* 57 (10):1143–1160.
- Suman, B., N. Hoda, and S. Jha. 2010. Orthogonal simulated annealing for multiobjective optimization. *Computers & Chemical Engineering* 34 (10):1618–1631.
- Sussman, G. J. 1975. *A Computer Model of Skill Acquisition*. Elsevier, New York, NY.
- Sutton, R. S., A. G. Barto, and F. Bach. 2018. *Reinforcement Learning: An introduction*. 2nd ed. MIT Press, Cambridge, MA.
- Syswerda, G. 1989. Uniform crossover in genetic algorithms. In *4th International Conference on Genetic Algorithms*, edited by J. D. Schaffer. Morgan Kaufmann, San Mateo, CA.

- Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. *Going deeper with convolutions*. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA. 1–9.
- Tait, R. J., G. Schaefer, and A. A. Hopgood. 2008. Intensity-based image registration using multiple distributed agents. *Knowledge-Based Systems* 21(3):256–264.
- Takagi, T., and M. Sugeno. 1985. *Fuzzy identification of systems and its applications to modeling and control*. *IEEE Trans. Systems, Man, and Cybernetics* 15:116–132.
- Tate, A. 1975. *Interacting goals and their use*. In *4th International Joint Conference on Artificial Intelligence (IJCAI'75)*. Tbilisi, Georgia. 215–218.
- Tate, A. 1977. *Generating project networks*. In *5th International Joint Conference on Artificial Intelligence (IJCAI)*. Cambridge, MA. 888–893.
- Taunton, J. C., and D. W. Haspel. 1988. The application of expert system techniques in on-line process control. In *Expert Systems in Engineering*, edited by D. T. Pham. IFS Publications, Bedford, UK / Springer-Verlag, Berlin, Germany.
- To, C. C., and J. Vohradsky. 2007. A parallel genetic algorithm for single class pattern classification and its application for gene expression profiling in Streptomyces coelicolor. *BMC Genomics* 8 (49).
- Torrey, L. and J. Shavlik. 2009. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, methods, and techniques*, edited by E. Soria, J. Martin, R. Magdalena, M. Martinez, and A. Serrano. 242–264. IGI Global, Hershey, PA.
- Tsukimoto, H. 2000. Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks* 11 (2):377–389.
- Turing, A. M. 1950. Computing machinery and intelligence. *Mind*, LIX (236):433–460.
- Ulrich, I., F. Mondada, and J. D. Nicoud. 1997. Autonomous vacuum cleaner. *Robotics and Autonomous Systems* 19:233–245.
- Ungar, L. H. 1990. A bioreactor benchmark for adaptive network-based process control. In *Neural Networks for Control*, edited by W. T. Miller, R. S. Sutton and P. J. Werbos. MIT Press, Cambridge, MA.
- Valmiki, A. H., A. A. West, and D. J. Williams. 1991. *The evolution of a neural network controller for adhesive dispensing*. In *IFAC workshop on computer software structures integrating AI/KBS systems in process control*. Bergen, Norway. 93–101.
- von Seggern, D. 2007. *CRC Standard Curves and Surfaces with Mathematics*. 2nd ed. CRC Press, Boca Raton, FL.
- Wagner, C., and H. Hagras. 2010. Toward general type-2 fuzzy logic systems based on zSlices. *IEEE Transactions on Fuzzy Systems* 18 (4):637–660.
- Wagner, F., R. Schmuki, T. Wagner, and P. Wolstenholme. 2006. *Modeling Software with Finite State Machines*. CRC Press, Boca Raton, FL.
- Walker, N., and J. Fox. 1987. Knowledge-based interpretation of images: A biomedical perspective. *Knowledge Engineering Review* 2 (4):249–264.
- Waser, M. R. 2009. A safe ethical system for intelligent machines. In *Biologically Inspired Cognitive Architectures II*. North America: AAAI Fall Symposium Series (FS-09-01).
- Watson, I., and D. Gardinen. 1999. *A distributed case-based reasoning application for engineering sales support*. In *16th International Joint Conference on Artificial Intelligence (IJCAI'99)*. Stockholm, Sweden. 1:600–605.
- Welser, J., J. W. Pitera, and C. Goldberg. 2018. Future computing hardware for AI. In *Technical Digest – IEEE International Electron Devices Meeting, IEDM2018*.

- Wen, W. 2008. A dynamic and automatic traffic light control expert system for solving the road congestion problem. *Expert Systems with Applications* 34:2370–2381.
- Widrow, B., and M. E. Hoff. 1960. Adaptive switching circuits. In *Wescon Conference Record Part 4*, 96–104.
- Wilkins, D. E. 1983. Representation in a domain-independent planner. In *8th International Joint Conference on Artificial Intelligence (IJCAI'83)*. Karlsruhe, Germany. 733–740.
- Wilkins, D. E. 1984. Domain-independent planning: Representation and plan generation. *Artificial Intelligence* 22:269–301.
- Wilkins, D. E. 1989. *Practical Planning: Extending the classical AI planning paradigm*. Morgan Kaufmann, San Mateo, CA.
- Wilkins, D. E., and M. desJardins. 2001. A call for knowledge-based planning. *AI Magazine* 22 (1):99–115.
- Willis, M. J., C. Di Massimo, G. A. Montague, M. T. Tham, and A. J. Morris. 1991. Artificial neural networks in process engineering. *IEE Proceedings-D* 138 (3):256–266.
- Wong, K., K. Leung, and M. Wong. 2010. Protein structure prediction on a lattice model via multimodal optimization techniques. In *GECCO-2010*. 155–162.
- Wong, T.-T. 2015. Performance evaluation of classification algorithms by k -fold and leave-one-out cross validation. *Pattern Recognition* 48:2839–2846.
- Woodcock, N., N. J. Hallam, and P. D. Picton. 1991a. Fuzzy BOXES as an alternative to neural networks for difficult control problems. In *Applications of Artificial Intelligence in Engineering VI*, edited by G. Rzevski and R. A. Adey. Computational Mechanics / Elsevier, Southampton, UK. 903–919.
- Woodcock, N., N. J. Hallam, P. D. Picton, and A. A. Hopgood. 1991b. *Interpretation of ultrasonic images of weld defects using a hybrid system*. In *International Conference on Neural Networks and their Applications*. Nimes, France.
- Wooldridge, M. J. 1997. Agent-based software engineering. *IEE Proc. Software Engineering* 144 (1):26–37.
- Wooldridge, M. J. 1999. Intelligent agents. In *Multiagent Systems: A modern approach to distributed artificial intelligence*, edited by G. Weiss. MIT Press, Cambridge, MA. 27–77.
- Wooldridge, M. J. 2009. *An Introduction to Multiagent Systems*. 2nd ed. Wiley, Chichester, UK.
- Wooldridge, M. J., and N. R. Jennings. 1994a. Formalising the cooperative problem solving process. In *13th International Workshop on Distributed Artificial Intelligence (IWDAI'94)*. Lake Quinault, WA. 403–417.
- Wooldridge, M. J., and N. R. Jennings. 1994b. Towards a theory of cooperative problem solving. In *6th European Conference on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'94)*. Odense, Denmark. 15–26.
- Wooldridge, M. J., and N. R. Jennings. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10 (2):115–152.
- Wotawa, F. 2000. Debugging VHDL designs using model-based reasoning. *Artificial Intelligence in Engineering* 14 (4):331–351.
- Wu, T. H., C. C. Chang, and S. H. Chung. 2008. A simulated annealing algorithm for manufacturing cell formation problems. *Expert Systems with Applications* 34 (3):1609–1617.
- Yamada, T., and R. Nakano. 1992. A genetic algorithm applicable to large-scale job-shop problems. *Parallel Problem Solving from Nature* 2:281–290.
- Yamashita, Y., H. Komori, E. Aoki, and K. Hashimoto. 2000. Computer aided monitoring of pump efficiency by using ART2 neural networks. *Kagaku Kogaku Ronbunshu* 26:457–461.

- Yang, Q. 1997. *Intelligent Planning: A decomposition and abstraction based approach*. Springer-Verlag, Berlin, Germany.
- Yao, X. 1999. Evolving artificial neural networks. *Proceedings of the IEEE* 87 (9):1423–1447.
- Yella, S., M. S. Dougherty, and N. K. Gupta. 2006. Artificial intelligence techniques for the automatic interpretation of data from non-destructive testing. *Insight* 48 (1):10–20.
- Yosinski, J., J. Clune, Y. Bengio, and H. Lipson. 2014. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems (NIPS'14)*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Red Hook, NY. 4:3320–3328.
- Zadeh, L. A. 1965. Fuzzy sets. *Information and Control* 8:338–353.
- Zadeh, L. A. 1975. Fuzzy logic and approximate reasoning. *Synthese* 30:407–428.
- Zadeh, L. A. 1983a. Commonsense knowledge representation based on fuzzy logic. *IEEE Computer* 16 (10):61–65.
- Zadeh, L. A. 1983b. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy Sets and Systems* 11:199–227.
- Zadeh, L. A. 1996. Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems* 4 (2):103–111.
- Zhang, J., J. Zhuang, H. Du, and S. Wang. 2009. Self-organizing genetic algorithm based tuning of PID controllers. *Information Sciences* 179 (7):1007–1018.
- Zhang, J., W. L. Lo, and H. Chung. 2006. Pseudocoevolutionary genetic algorithms for power electronic circuits optimization. *IEEE Trans Systems, Man, and Cybernetics, Part C* 36 (4):590–598.
- Zhang, X., X. Zhou, M. Lin, and J. Sun. 2018. *ShuffleNet: An extremely efficient convolutional neural network for mobile devices*. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT. 6848–6856.
- Zhang, Z., and M. Simaan. 1987. A rule-based interpretation system for segmentation of seismic images. *Pattern Recognition* 20 (1):45–53.

Index

Page numbers in *italics* refer to figures; page numbers in **bold** refer to tables.

A

abduction, 10–11, 299–300
abstract data types, 115, 116
abstraction-based STRIPS (ABSTRIPS), 392–397, **395, 396**
abstraction space, 392
initial and modified criticality values, 394, **394**
Nonlin system, 397
planning flow chart, 394, **395**
preconditions and operators, 392–394
search using hierarchical planner, 396, 396–397
semiautomated approach, 392
user-supplied criticality values, 392, **393**
abstraction links, 151–152, **152**
access function, 137–139
activation function, 201, **201**
active values, 134, **135**
adaptive-network based fuzzy inference system (ANFIS), 255
Adaptive Resonance Theory networks (ART1 and ART2), 220–221, **220–221**
ad-hoc approach, 66, 67, 72, 96, 397
adjudicator, 258
adversarial, 241
agent communication languages (ACLs), 110, 139
agents, 2, 10, 97–111, 127, 135–136, 139, 194, 248–250, 260, 307, 321–323, **322, 325–328, 330, 332, 378, 415, 428, 429, 455, 458**

definition, 98
emergent behavior architecture, 100, **100**
intelligent agent, 98–99
knowledge-level architecture, 100–101, **101**
layered architecture, 101–102, **102**
logic-based architecture, 100
multiagent systems, *see* multiagent systems (MASs)
swarm intelligence, 111
AIM algorithm, 357, 365–367, **365–367, 369**
Alexnet, 238
algorithmic and rule-based blackboard system (ARBS), 44–45, 248–249, 319, 322, 434
alleles, 173, 175
ant colony optimization, 194
artificial immune systems (AISs), 194
artificial intelligence, definition, 1–2
Ashby maps, 362, **363, 368**
aspiration criteria, 164
associative network, 9–10, **10**
assumption-based truth maintenance system (ATMS), 349
asynchronous communication, 109–110
autoencoder networks, 240–241, **241**
autonomy, 98–99, 136, 139, 458

B

back-error propagation algorithm, 208–209, **209, 210, 211, 216**
backtracking, 16, **16, 38–39, 39, 283–287, 285, 295–296, 382, 391, 397, 399**

- backward chaining, 11–12, 38–45, 249, 324, 382, 387
 format of, 43–44
 implementation, 39, 40, 41
 mechanism, 38–39, 39, 44, 45
 variations of, 41–43, 42
- Baldwinian inheritance, 247, 253, 255, 256, 256
- bang-bang servo controller, 425, 434, 435
- Bayesian updating, 8, 18, 19, 52–73, 93–95, 298
 advantages, 66–67
 application of Bayes' theorem, 53–55
 certainty theory, 69, 70, 72, 73
 combining evidence, 59–62, 61
 Dempster–Shafer theory of evidence, 94
 disadvantages, 67–68
 example of, 63, 63–66
- Inferno, 95
 likelihood ratios, 55–58, 57
 production rule, 62–63
 uncertain evidence, 58–59, 59
 uncertainty by probability, 52–53
- beliefs-desires-intentions (BDI) architecture, 101, 101, 106, 458
- big data, 2, 18, 19, 456
- binary chop/structural isolation strategy, 316
- biological neurons, 226–227, 227
- blackboard system/model/architecture, 19, 247–250, 248, 257, 319–332, 415, 427, 429, 434
- arc detection, 325–326, 326
- DARBS, 321–325, 322, 325
- defect classification, 328–330, 329
- echodynamic classification, 330
- echodynamics, 327, 327
- hypothesize-and-test method, 324, 328, 330, 331
- image interpretation system, 319
- maintenance, 427–428
- multifaceted tasks, 248, 248–250
- rule-based agents, 327
- ultrasonic imaging, 319–321, 320, 321
- black box, 228, 257, 420, 440
- blind search, 17
- Boltzmann fitness scaling, 182, 183, 185–187
- BOXES controller, 435–442, 450
 bioreactor, 436, 436–437, 437
 cart-and-pole control problem, 439, 440
 control actions, 437
- degree of appropriateness, 438
 full updating procedure, 439
 fuzzy BOXES, 441–442, 442
 learning strategy, 438, 438
 positive/negative feedback, 437
 reinforcement learning, 437
 Smalltalk object-oriented language, 440
- breadth-first search, 16–17, 17, 33, 38, 41, 42, 42
- Brooks' subsumption architecture, 100, 100, 111
- bucket-brigade algorithm, 257
- building block hypothesis, 191–192
- C**
- capability enhancement, 247, 253
 learning classifier systems (LCSs), 257, 257
 memetic algorithms, 255–256, 256
 neuro-fuzzy systems, 253–255, 254
- Cartesian coordinates, 222
- case-based reasoning (CBR), 2, 142–143, 150–156, 298, 307, 318, 332
 adapting case history, 153–156
 characteristics, 150
 expert system, 150–151
 retrieving cases, 153
 storing cases, 151–153, 152
- centroid method, 82, 82–83, 90, 442
- certainty factor (CF), 18, 19, 68–71, 70, 72–75, 75, 86, 95
- certainty theory, 68–75, 79, 93, 95
 certainty factors and probability, 74–75, 75
 conjunction, 71
 disjunction, 71
 example of, 72–74
 negation, 71–72
- chunking, 149–150
- C++ language, 112, 115–119, 122–126, 128–132, 136, 144, 259, 261, 262, 264, 266, 288, 295, 323, 383
- access control, 125, 125
- attributes, 116
- dynamic binding/late binding, 128–130
- instances, 115, 117–118
- member functions, 117
- metaclasses, 131, 131
- public and private derivation, 125–126, 126
- repeated inheritance, 122–123, 123

- single inheritance, 119
- specialized method, 124
- static type checking, 132–133
- class attributes, 116, 125
- class browser, 124–125
- classes, 9–10, 114–115
- “cliff-edge” effect, 370, **370**
- closed-world assumption, 27–28
- Common Lisp, 265, 274
- Common Lisp Object System (CLOS), 112, 265
- common traffic model, 334, 339, 340, 342, 343, 350, 351
 - communication applications, 339
 - dispersion links, 342
 - equipment, 343
 - implementation, 340
 - information stream, 342
 - multipoint links, 342
 - Network object, 342
 - object classes and attributes, 340, **341**
 - point-to-point links, 342
 - sites, 342
- competitive models, 104
- computational intelligence (CI), 2, 17–19, **18**, 160, 250, 261, 319, 352, 402, 454
- computational power, 69, 456
- computer-aided design (CAD), 338–339
- concurrency, 133
- conditional probability $P(H|E)$, 74–75, **75**;
 - see also* Bayesian updating
- conflict resolution, 31, 35–37, 150
 - “first come, first served” scheme, 35–36, **36**
 - metarules, 37
 - priority values, 36–37
- conjugate gradient descent, 163–164
- connectionism, 197
- constraint-based analysis (CBA), 402–414
 - capacity constraints, 403
 - constraints and preferences, 402–403, 413–414
- critical set of operations, 405–406, **406**
 - earliest start times and latest finish times
 - updating, 403–411, **410**, **411**
 - job-shop scheduling, 404, **405**
- OPAL, 403–404, **404**
- outcomes, 411–413
- production constraints, 403
- resource constraints, 403
- sequencing in disjunctive case, 406, **407**
- sequencing in nondisjunctive case, 406–409, **408**
- technological coherence constraints, 403
- constructor, 118
- content-addressable memory (CAM), 200
- context neurons, 215, **216**, 243
- contract nets, **105**, 105–106, 108–109, **109**
- control system, 17, 81, 419–451
 - adaptive and servo control, 420
 - blackboard maintenance, 427–428
 - BOXES controller, 435–442
 - fuzzy control, 86, 86–90, **88–90**, 433–435
 - high-level or supervisory control, 419, 425–426, **426**
 - low-level control, *see* low-level control
 - neural network controllers, 442–446
 - passive implementation, 419
 - sequence control, 420
 - statistical process control (SPC), 447–449
 - temperature controller, 419, 420
 - time-constrained reasoning, *see* time-constrained reasoning
- convergence factor (CF), 252
- convergence of techniques, 247–248
- convolutional neural networks (CNNs), 230–239
 - convolution layers, 233
 - feature maps, 235–237, **236**
 - image recognition, 230, **231**
 - input layer, 233–235, **235**
 - origin, 230
 - pooling and classification layers, 237–238
 - pretrained networks and transfer learning, 238–239
 - primitives, 232
 - structure, 233, **234**
- cooperative models, 104–109
 - contract nets, **105**, 105–106, 108–109, **109**
 - cooperative problem-solving, **106**, 106–109, **109**
 - shifting matrix management, 107–109, **107–109**
- cooperative problem-solving (CPS), 105–109, **106**, **109**
- credit-apportionment system, 257
- credit-assignment problem, 146
- crew-scheduling problem (CSP), 252, **253**
- crossover probability P_c , 173–175, **174**, 191
- cybersecurity development, 2
- CYCLOPS system, 344, 345

D

daemons, 134, **135**
 DARBS, 319–331, **322, 325**
 data abstraction, 112, 114–118, 128, 135,
 137, 139
 attributes, 116
 classes, 114–115
 instances, **114**, 115–118
 operations/functions, 116–117
 data members, 116, 125, 127, 134
 data mining, 104, 111, 143, 200
 declarative programming, 12–14
Deepfake techniques, 240
 deep knowledge, 303–305, **304**, 316, 332
 deep learning, 229
 deep neural networks, 2, 18, 229–245
 autoencoder networks, 240–241, **241**
 convolutional neural networks (CNNs),
 230–239
 GANs, 241–242, **242**
 generative *vs.* discriminative algorithms,
 239–240
 LSTM networks, 242–245, **244**
 multilayer perceptron's limitations,
 229–230
 deduction, 10–11, 299–300
 defuzzification, 80, 82–86, 89–90, 434, 442
 anomaly, **85**, 85–86
 centroid method, **82**, 82–83
 in control system, **89**, 89–90, **90**
 extreme membership function, **83**,
 83–84
 membership function, **81, 81**
 Sugeno defuzzification, **84**, 84–85
 delta rule, 209, 211, 226
 Dempster–Shafer theory of evidence, 93–95
 DENDRAL, 298, 301
 depth-first search, 16, **16**, 38, 39, 41, 42, **42**
 derivational adaptation, 154–156
 design process, 333–336, 345–349, 370–372
 analogical reasoning, 345
 brainstorming and problem-solving, 345
 CAD, 338–339
 conceptual design, 335
 constraint propagation, 346–349, **347**
 CYCLOPS system, 344
 definition, 333
 detailed design, 335
 EDISON, 344, 345
 engineering design, 333
 FMEA, 370–372, **371**

FORLOG, 344
 forward and backward phases, 335
 generalization, 345, **346**
 industrial design, 333
 inventions and innovations, 343, 344
 lightweight beam design, *see* lightweight
 beam design
 manufacture, 335
 market, 334
 materials selection, 336
 mutation, 345
 optimization/evaluation, 335
 principal phases, 334, **334**
 problem-solving rules, 345
 product design specification (PDS),
 334–336, 339–344, 350–351
 PROMPT, 345
 routine design, 343
 search problem, 336–338, **337**
 selection decisions, *see* selection
 decisions
 selling, 335
 specification, 335
 systematic approach, 346
 truth maintenance, 346–349, **347**
 deterministic methods, 160
 directed forward chaining, 45
 directory facilitator (DF) agent, 104, **104**
 disjunctive-concept problem, 149
 distributed ARBS (DARBS), 44, 249, 259,
 319–332, 434, 456–458
 atomic conditions, 324
 main functions, 323
 multiple instantiation of variables,
 324, **325**
 open-source version, 321
 precondition, 322
 rule-based agents, 323, 324
 single instantiation of variables, 324, **325**
 structure of, 321, **322**
 XML, 322
 distributed artificial intelligence (DAI), 102
 dynamic binding, 128–130, 137–139
 dynamic type checking, 132–133

E

echodynamic, 327–330, **327**
 EDISON, 344, 345
 elitism, 187
 engineering design, 110, 333
 epistasis, 162

- ethics, 458–459
 Euclidean distance, 224
 evaluation/objective value, 180
 evolutionary algorithm, 171; *see also*
 genetic algorithms (GAs)
 exemplar links, 152, 153
 exhaustive search, 15, 194
 expert system, 5, 14, 48, 150–151, 259
 exploding gradient problem, 243
 Extendable Multi-Agent Data-mining System
 (EMADS), 104
- F**
- fact base, 22, 30, 34, 38, 43
 failure links, 152, 153
 failure mode and effects analysis (FMEA),
 370–372, 371
 feature maps, 233, 235–238, 236
 feedback control, 422, 446
 feedforward control, 421
 feedforward network, 203, 219–220,
 223, 227
 finite-element analysis, 353, 354
 FIPA-ACL, 110, 111
 fitness function, 159, 163, 172, 173, 187, 195
 fitness landscape, 161–163, 161, 186, 187,
 195, 256
 fitness-proportionate selection, 177,
 177–179, 179
 fitness scaling, 180–187
 Boltzmann fitness scaling, 182, 183
 linear scaling, 180–181
 nonlinear rank scaling, 184, 184, 186
 probabilistic nonlinear rank scaling,
 184–186
 rank scaling/rank selection, 183,
 184, 186
 sigma scaling, 181–182, 186
 transform ranking, 185–186
 truncation selection, 185
 fixed-locus assumption, 189
 FLAME system, 371, 372
 Flex, 6, 22, 27, 36, 43, 63, 115, 138, 143,
 259–261, 338
 Flint, 58, 62, 64, 259
 footprint of uncertainty, 91, 92
 FORLOG, 344
 forward chaining, 11–12, 25, 30–37, 43–45,
 249, 324
 conflict resolution, 35–37, 36
- cycle of events, 30–31, 31
 mechanism, 44, 45
 multiple instantiation of local variables,
 31–33, 32–34
 Rete algorithm, 33–35, 35
 single instantiation of local variables,
 31–33, 32–34
- frame-based system, 2, 97, 137–139, 138
 frame problem, 11, 146, 377, 378
 fuzzification, 76, 81, 83, 85, 86, 254
 fuzzy BOXES, 441–442, 442
 fuzzy control system, 86–90, 433–435
 crisp sets, 86, 86–87
 defuzzification, 89, 89–90, 90
 fuzzy control rules, 87–89, 88
 fuzzy–genetic systems, 252, 253
 fuzzy logic, 75–93, 198, 298, 433–435
 control system, *see* fuzzy control system
 goal of, 75
 type-1, *see* type-1 fuzzy logic
 type-2, 90–91, 92, 93, 93
- G**
- Gantt chart, 400, 401
 garbage collection, 117, 264
 Gaussian function, 224, 225
 generalization techniques, 146–150
 chunking, 149–150
 class hierarchy, 149, 150
 conjunction, 148–149
 disjunction, 149
 replacing constants with local variables,
 147–148
 universalization, 147
 generalized delta rule, 208–209, 226
 generate-and-test method, 16, 337
 generation gap, 187
 generative adversarial networks (GANs),
 241–242, 242
 generative *vs.* discriminative algorithms,
 239–240
 genetic algorithms (GAs), 2, 18, 19, 142,
 160, 162, 171–196, 250–253, 257,
 335, 353, 425
 artificial immune systems, 194
 assumptions, 173
 Boltzmann scaling, 186–187
 building block hypothesis, 191–192
 characteristics, 171, 172
 chromosomes, 172–173

crossover, 173–175, **174**
 elitism, 187
 fitness-proportionate selection, **177**,
 177–179, **179**
 fitness scaling, 180–186, **183**, **184**
 genetic programming, 193–194
 Gray code, 188–189, **188–189**
 Griewank function, 186
 memetic algorithm, 255–256, **256**
 monitoring evolution, 192–193
 multiobjective optimization, 187–188
 mutation, 175
 niches, 193
 parameter selection, 192
 premature convergence, 176
 Schwefel function, 186
 species, 193
 stalled evolution, 176
 tournament selection, 186
 validity check, 175–176
 variable-length chromosome, 189–190
 genetic–fuzzy systems, **251**, 251–252
 genetic–neural systems, 250–251
 genetic programming (GP), 193–194
 genome, 172
 genotype, 172
 global optimum, 161–163, **161**, 169, 171,
 180, 195, 255
 GoogLeNet, 238
 gradient descent, 163, 164, 209, 211
 Gray code, 188–189, **188–189**

H

HACKER, 155, 399
 Hamming distance, 161, 163
 Hamming network, **219**, 219–220
Hearsay-II blackboard system, 248, 249
 heuristic search, 17
 hierarchical planning, 389–397, 402, 416
 ABSTRIPS, 392–397, **393**, **395**, **396**
 advantages, 390–392
 description, 379–380
 Hierarchical task Planning through World
 Abstraction (HPWA), 390
 high-level or supervisory control, 87, 419,
 425–426, **426**
 hill-climbing algorithm, 162–165, 168, 353
 Hopfield network, 216, **217**, 218, 219
 hypothesize-and-test method, 301, 302, 324,
 328, 330, 331

I

imitation game, 458
 index links, 151–152, **152**
 induction, 10–11, 143–150
 industrial design, 333
 inference engine, 4–5, **5**, 11–12
 inference network, 7, 7–8
 Inferno, 93, 95
 instance-of links, 152, **152**
 instances, 9–10, 115–116
 Integrated Diagnostic Model (IDM), 305
 intelligent behavior, 3, 3–4, **4**, 100, 102, 111,
 136, 458
 intelligent systems, 2–3, 15, 19, 150–156,
 419–459
 automation, 453
 benefits, 453–454
 case-based reasoning, *see* case-based
 reasoning (CBR)
 computational power, 456
 control system, *see* control system
 definition, 2–3
 design process, *see* design process
 ethics, 458–459
 future of, 453–459
 and internet, 455–456
 knowledge archiving, 454
 prototype–test–refine cycle, 454
 reliability and consistency, 453
 speed, 453–454
 ubiquitous intelligent systems,
 456–458, **457**
 intelligent systems tools, 6, 22, 27, 36, 43,
 63, 115, 138, 143, 259–294
 brakes diagnosis system, 260, **260**
 computational intelligence, 259
 data types, 263, **263**
 expert-system shell, 259, 260
 Flex, 6, 22, 27, 36, 43, 63, 115, 138, 143,
 259–261, 338
 Flint, 58, 62, 64, 259
 knowledge-based systems (KBSs), 259, 260
 Lisp, 262–276
 Lists, 261–262
 programming environments, 264
 programming languages, 260
 Prolog, 276–287
 Python, 287–294
 rule-based expert systems, 259
 VisiRule, 260, **260**
 inversion, 191–192

J

Java, 112, 136, 259, 261
 Java Agent Development (JADE)
 framework, 104
 job-shop scheduling, 400, 400–404, 401

K

KEATS, 309, 316
k-fold cross-validation method, 213
 Kirchoff's first law, 304
 knowledge acquisition, 15, 141
 knowledge base, 4, 5, 6–10
 inference network, 7, 7–8
 rules and facts, 6–7
 semantic network, 8–10, 10
 knowledge-based systems (KBss), 2, 4–17,
 67, 94, 100–101, 112, 137, 247,
 259, 319, 351, 369, 403, 429, 453,
 454, 458
 agents, 100–101, 101
 Bayesian updating, 60
 blackboard system, 19, 247
 computational intelligence, 17–19, 18
 declarative and procedural
 programming, 12–14
 deduction, abduction, and induction,
 10–11, 299–300
 expert system, 5, 14
 explicit separation of knowledge, 5
 inference engine, 4–5, 5, 11–12
 knowledge acquisition, 15
 knowledge base, 4, 5, 6–10
 search, 15–17, 16, 17
 teacher role, 142
 Knowledge Query and Manipulation
 Language (KQML), 110, 111
 knowledge sources (KSs), 248
 Kohonen self-organizing networks, 222,
 222–223
 Kolmogorov's Existence Theorem, 208, 208,
 250

L

Lamarckian inheritance, 247, 255, 256, 256
 Larsen's product operation rule, 81, 81, 82,
 85, 89, 89
 learning by induction, 143–150
 generalization techniques, 146–150, 150

overview, 143–145
 search problem, 145, 145–146
 specialization techniques, 146, 147,
 149, 150
 learning classifier systems (LCSs), 253,
 257, 257
 leave-one-out technique, 213, 329
 Lex system, 145
 lightweight beam design, 350–356
 conceptual designs, 351, 352
 hierarchical classification, 351, 353
 knowledge-based systems, 351
 optimization and evaluation,
 352–356, 355
 passenger seat in commercial
 aircraft, 350
 PDS, 350, 351
 stiffness/mass ratio, 351
 total design process, aircraft,
 350, 350
 likelihood ratios, 55–58, 66–69, 96
 affirms weight *A*, 57–58
 definition, 55
 denies weight *D*, 56–58
 function, 56, 57
 odds O(H), 55–57
 probability P(H), 55–56
 linear membership function, 77,
 77, 78
 linear scaling, 180–181
 LINKman, 433
 Lisp, 39, 110, 112, 259, 261–276
 CLOS, 265
 Common Lisp, 265
 conditional statement, 271
 declaration of variables, 263
 defvar function, 270
 dolist control function, 271
 dotted pair, 268, 272, 273
 first and rest functions, 267
 if function, 271
 linked list, 262
 nested list, 262
 quote function, 266
 reading or writing, 267
 remove function, 273
 rules, 265
 setf and dolist function, 274
 setf function, 270
 syntax and features, 264
 undeclared variables, 263
 worked example, 275–276

local optimum, 161, 161–164
 local receptive field (LRF), 235–237
 long short-term memory (LSTM)
 networks, 242
 low-level control, 419, 420–425, 450
 bang-bang controllers, 425
 feedback control, 422
 feedforward strategy, 421
 first-and second-order models, 422, 423
 open-loop control, 420–421, 421
 PID controller, 423–425

M

Machine learning, 2, 18–19
 Mamdani-style fuzzy inference, 82, 84
 MAXNET, 217, 218–220
 max-pooling, 237
 memetic algorithms, 255–256, 256
 memory cell, 243, 244
 merit indices, 357–358, 358, 359
 messy chromosomes, 189–191
 metaclasses, 131, 131
 metaheuristic algorithm, 164
 metarules, 37
 mirror rule, 83, 83–85, 89–90, 90
 mobile agent, 99, 455
 model-based reasoning, 305–319
 ADAPTER, 307
 advantages, 318–319
 explicit models, 314
 EXPRES, 315
 fault repair, 317
 fault simulation, 316–317
 feedback mechanism, 314
 finite-state machine (FSM), 310
 first principles, 306
 flow diagram, 313, 313
 function, 307–308, 308
 heuristic approach, 316
 implicit model, 314
 limitations of rules, 305–306
 model parameters, 314
 object-oriented programming, 307
 potential uses, 312
 problem trees, 317–318, 318
 RESCU system, 314
 shotgun approach, 315–316
 single point of failure assumption, 313
 state map, 310, 311, 311, 312
 structural isolation, 316

structure, 308–310, 309, 310
 VHDL, 307
 monomorphism, 129, 130
 moral Turing Test, 458
 multiagent systems (MASs), 102–111, 249
 benefits, 103
 building, 104, 104–105
 communication, 109–111
 contract nets, 105, 105–106,
 108–109, 109
 cooperative problem-solving, 106,
 106–109, 109
 definition, 102
 shifting matrix management, 107–109,
 107–109
 multilayer perceptron (MLP), 203–215, 218,
 237, 245, 329, 330, 443
 buffered perceptron, 212
 data scaling, 214–215
 hierarchical perceptron, 211, 212
 k-fold cross-validation method, 213
 leave-one-out technique, 213
 network topology, 203, 203–204
 overtraining, 212–213, 214, 215
 perceptrons as classifiers, 204–208, 205,
 206, 207
 training, 208–211, 209, 210
 multiobjective optimization, 168, 187–188
 mutation probability, 175
 MYCIN, 68, 298, 301

N

neural networks
 shallow neural networks 197–228
 deep neural networks 229–245
 neural network controllers, 443–446
 critical state variables, 444–446,
 444–446
 state variables and action variables, 443,
 443–444
 neural network outputs, clarification and
 verification, 257–258
 neural network-rule-based system, 298
 neuro-fuzzy systems, 253–255, 254
 Newell's Principle of Rationality, 101, 101
 nonlinear function, 201, 201, 202, 202
 nonlinear rank scaling, 184, 184, 186
 Nonlin system, 397
 null adaptation, 154
 numerical learning, 141–142, 159, 197, 227

O

objective function, 160, 162
 object-oriented system 112
 object-oriented programming (OOP), 97,
 112–136, 139, 259, 288, 307, 310, 371
 active values, 134, **135**
 class browser, 124–125
 concurrency, 133–134
 daemons, 134, **135**
 data abstraction, 114–118
 detection of defects, 113, **113**
 dynamic binding/late binding, 128–130
 encapsulation, 125–126, **125–126**
 message passing and function calls, 130
 metaclasses, 131, **131**
 multiple inheritance, 121–123, **122**, **123**
 persistence, 133
 programming environment, 112
 repeated inheritance, 122–123, **123**
 single inheritance, 119–121, **120**, **121**
 specialized method, 124
 type checking, 132–133
 Unified Modeling Language, 126–127,
 127, **128**
 OPAL, 403–404, **404**
 open-loop control, 420–421, **421**
 optimization algorithm, 159–169; *see also*
 genetic algorithms (GAs)
 overtraining, 212–213, **214**, **215**

P

Pareto boundary, 362–363, **363**
 Pareto optimal, 188, 363
 persistence, 133
 phenotype, 172
 PID controller, 423–425, 434, 435, 444,
 446, 450
 planning systems, 12, 17, 155, 375–417
 ABSTRIPS, 392–397, **395**, **396**
 CBA, 402–414
 classical planning systems, 376–378, **377**
 definition, 375
 design, 375
 HACKER, 399
 hierarchical planning, 389–397
 HYBIS, 399
 internal planning, 376
 INTERPLAN, 399
 job-shop scheduling, **400**, 400–402, **401**
 means–ends analysis, 376, 387

nonlinear planning, 398
 partial ordering of plans, 397–399
 postponement of commitment, 397–399
 replanning and reactive planning,
 414–415
 scheduling, 376, 401–402
 SIPE, 387–388
 state-based reasoning, 376
 STRIPS, 378–386
 use of planning variables, 399
 polymorphism, 129, 130, 135
 pooling layers, 233, **234**, 237–238
 possibility theory, *see* fuzzy logic
 premature convergence, 176, 177, 180, 182,
 183, 185, 187
 prior probability, 53, 54, 62, 66, 67, 94
 probabilistic nonlinear rank scaling,
 184–186
 probabilistic rule, 64–66
 problem trees, 317–318, **318**
 product design specification (PDS),
 334–336, 339–343, 344, 350–351
 Prolog, 14, 22, 30, 39–41, 43, 110, 111, 115,
 133, 276–287
 backtracking, 283–287, **285**
 building blocks, 277
 features of, 277
 list separator, 280
 logical problems, 276
 member relation, 281
 underscore character, 278
 worked example, 282
 PROMPT, 345
 PROSPECTOR, 52, 297–298, 301
 Python, 13, 39, 110, 113, 262, 268,
 287–296, 383

Q

quantum computing, 456

R

radial basis function (RBF) networks,
 223–225, 223–226
 ramp function, 201, **202**
 rank scaling/rank selection, 183, **184**, 186
 reasoning by analogy, 143, **152**, 154–155
 recurrent neural networks (RNNs), 215–220,
 226, 242–245, **244**
 Hamming network, **219**, 219–220

- Hopfield network, 216, **217**, 218
 long short-term memory (LSTM)
 networks, 242–245, **244**
MAXNET, **217**, 218–219
 simple recurrent network, 215–216, **216**
 refrigerator, fault diagnosis, 150–155, **152**,
 299–304
 abduction, 300
 blackboard system, *see* blackboard system
 case-based reasoning (CBR), 150–155
 deductive rules, 299
 deep knowledge, 303–304, **304**
 default reasoning, 302
 exhaustive testing, 300–301
 explicit modeling of uncertainty, 301
 frame-based representation, 299
 hypothesize-and-test, 301–302
IDM, 305
 model-based reasoning, *see* model-based
 reasoning
 schematic diagram, 299, **299**
 shallow knowledge, 302–303, **304**
 reinstatiation, 155–156
 Rete algorithm, 26, 33–36, **35**
 root-mean-squared (RMS) error, 208,
 212–213, **215**
 roulette wheel selection with replacement
 (RWSR), 177, **177**, 178, 180,
 184–186
 rule-based diagnostic systems, 298
 rule-based system, 21–49, 145, 249, 260, 432
 backward chaining, 38–44, **39**, **40**, **42**
 for boiler control, 22–24, **23**
 closed-world assumption, 27–28
 explanation facilities, 48–49
 facts, 21–22
 forward chaining, 30–35, **31**–**35**
 hybrid strategy, 44–47, **45**
 local variables, **23**, 28–30
 maintaining consistency, 25–27
 production rule, 21, 62–63
 rule examination and firing, 24–25
 rule dependence network, **45**, 45–47
 rule extraction, 258
 rule induction, *see* learning by induction
- S**
- scene links, **152**, 152–153
 scheduling, 252–253, 376, 400–414
 schema theorem, 191
 Schwefel function, 186, 256
- search control, 150
 search problem, 15, **145**, 145–146,
 336–338, **337**
 search space, 255, 337, 338, 368–370,
 369, **370**
 genetic algorithm, 172–173, 176, 185,
 194, 195
 knowledge-based system, 15
 single-candidate optimization algorithm,
 160–162, **161**
 secondary membership function, 91, 93, **93**
 selection in genetic algorithms, 176–187
 selection decisions, 268–269, 356–370
 AIM, 366, **366**
 Ashby map, 368, **369**
 chemically similar materials, 368
 “cliff-edge” effect, 370, **370**
 constraint relaxation, 360–363, **361**–**363**
 kettle design, **367**, 367–368
 merit indices, 357–358, **358**, **359**
 overview, 356–357
 polymer selection, 358
 scoring system, 364–365, **365**
 two-stage selection, 358–360, **359**
 selective value, 180
 self-adapting genetic algorithm, 256
 self-adaptive parameters, 192
 self-organizing maps (SOMs), 216, **222**,
 222–223
 semantic network, 8–10, **10**
 semantic web, 455
 semi-supervised learning, 200, 223
 shallow knowledge, 24, 302–303, **304**,
 305, 316
 shallow neural network, 18, 197–228, 231, 233
 classification, 199
 clustering, 200
 content-addressable memory, 200
 multilayer perceptron, *see* multilayer
 perceptron (MLP)
 nodes and interconnections, 198,
 201–202, **201**–**202**
 nonlinear estimation and prediction, 199
 pattern recognition, 198–199
 recurrent networks, *see* recurrent neural
 networks
 single-layered perceptron, **203**, 203–204,
 206–207, **207**
 spiking neural network, 226–227, **227**
 unsupervised learning, **220**–**225**,
 220–226
 vectors, 198

- shifting matrix management (SMM), 105, 107–109, **107–109**
- ShuffleNet, 238
- sigma scaling, 181–182, 186
- sigma truncation, 181–182
- sigmoid function, 201, 202, **202**, 207, 209
- simple recurrent network (SRN), 215–216, **216**, 243
- simulated annealing, 142, 164, 171, 353, 402
- shallow neural network, 211
- single-candidate optimization algorithm, 18, 159–169, **166**, **167**
- conjugate gradient descent, 163–164
 - gradient-proportional descent, 163
 - hill-climbing, 162–163
 - parameter space, 162
 - search space, 160–162, **161**
 - simulated annealing, 164–165, **166**, **167**, 167–168
 - steepest gradient descent, 163
 - tabu search, 164
- single-layered perceptron (SLP), 202–204
- network topology, **203**, 203–204
 - perceptrons as classifiers, 206–207, **207**
- SIPE, 387–388, 399
- situation-identification problem, 146
- Smalltalk, 112, 130–133, 259, 264, 440
- SOAR, 149–150
- social capability, 99
- societal norms, 104, **104**
- soft computing, 18
- SONIA, 414
- Spatio Temporal Self-Organizing Recurrent Map (STORM), 216
- specialization techniques, 146, 147, 149, **150**
- spectrum of intelligent behavior, 3
- spiking neural networks (SNNs), 226–227, **227**
- spiking response model, 227
- Stanford certainty theory, *see* certainty theory
- Stanford Research Institute Problem Solver (STRIPS), 376, 378–388
- general description, 378–379
 - means–ends analysis, 379, 380
 - in Prolog, 382–386
 - search tree, 381, **381**
- statistical process control (SPC), 447–450
- applications, 447
 - data collection, 447–448, **448**
- steady-state selection, 187
- steepest gradient descent method, 163, 164
- stochastic methods, 160
- stochastic universal selection (SUS), 178–180, **179**, 184–186
- stride length, 235
- STRIPS, 376, 378–388
- structural adaptation, 153–155
- critics, 155
 - null adaptation, 154
 - parameterization, 154
 - reasoning by analogy, **152**, 154–155
- structured operators, 176
- Sugeno defuzzification, 84, 84–85
- supervised learning, 142, 198, 199, 200, 222, 223
- support vector machines (SVMs), 228
- swarm intelligence, 111, 194
- symbolic learning, 2, 141–157
- case-based reasoning, 150–157
 - learning by induction, 143–150, **145**, **150**
 - overview, 141
- synchronous communication, 109

T

- tabu search, 164
- Tanese's implementation, 182
- time-constrained reasoning, 428–433
- approximate search, 430–431
 - data approximations, 431
 - defined and ill-defined approximations, 430
 - definition, 428
 - knowledge approximations, 431–432
 - knowledge-based system, 429
 - loss of certainty, 430
 - loss of completeness, 430
 - loss of precision, 430
 - RESCU process, 428, 429
 - single and multiple instantiation, **432**, 432–433, **433**
- Touring Machines, 101–102, **102**
- tournament selection, 186
- transfer learning, 238
- transform ranking, 185–186
- triangular membership functions, 251, **251**
- truncation method, 81, **81**
- truncation selection, 185
- Turing test, 458
- two-point crossover, 174

- type-1 fuzzy logic, 75–90
 crisp set, 76, 76–77
 defuzzification, 80–86, **81–85**
 fuzzy rules, 78–80, **80**
 fuzzy set, 76–78, 77
 type-2 fuzzy logic, 90, 93, **93**
 secondary membership function, 91,
93, 93
- U**
- ubiquitous intelligent systems,
 456–458, **457**
- ultrasonic imaging, 112–121, 127, 248, 249,
 319–321, **320, 321**
- uncertainty, 8, 18, 51–96, 300, 301, 313, 324
 Bayesian updating, *see* Bayesian
 updating
 certainty theory, *see* certainty theory
 Dempster–Shafer theory of evidence, 94
 fuzzy logic, *see* fuzzy logic
 Inferno, 95
 plausibility theory, 93
 sources, 51–52
- unification, 30
- Unified Modeling Language (UML),
 126–127, **127, 128**
- uniform crossover, 175
- unimprovement factor (UF), 252
- unsupervised learning, 142–143, 198, 200,
 202, 220–226, 239, 437
 Adaptive Resonance Theory, 220–221,
220–221
- Kohonen self-organizing networks, 222,
 222–223
- radial basis function (RBF) networks,
223–225, 223–226
- user agent, 98
- V**
- vanishing gradient problem, 243
- variance factor (VF), 252
- verification rules, 248
- VisiRule, 260, **260**
- Z**
- zero-order Sugeno fuzzy inference, **84,**
84–85
- Ziegler–Nichols method, 425