
Simulation-Based Optimization

OPERATIONS RESEARCH/COMPUTER SCIENCE INTERFACES SERIES

Series Editors

Professor Ramesh Sharda
Oklahoma State University

Prof. Dr. Stefan Voß
Universität Hamburg

Other published titles in the series:

- Greenberg, Harvey J. / *A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for ANALYZE*
- Greenberg, Harvey J. / *Modeling by Object-Driven Linear Elemental Relations: A Users Guidefor MODLER*
- Brown, Donald/Scherer, William T. / *Intelligent Scheduling Systems*
- Nash, Stephen G./Sofer, Ariela / *The Impact of Emerging Technologies on Computer Science & Operations Research*
- Barth, Peter / *Logic-Based 0-1 Constraint Programming*
- Jones, Christopher V. / *Visualization and Optimization*
- Barr, Richard S./ Helgason, Richard V./ Kennington, Jeffery L. / *Interfaces in Computer Science & Operations Research: Advances in Metaheuristics, Optimization, & Stochastic Modeling Technologies*
- Ellacott, Stephen W./ Mason, John C./ Anderson, Iain J. / *Mathematics of Neural Networks: Models, Algorithms & Applications*
- Woodruff, David L. / *Advances in Computational & Stochastic Optimization, Logic Programming, and Heuristic Search*
- Klein, Robert / *Scheduling of Resource-Constrained Projects*
- Bierwirth, Christian / *Adaptive Search and the Management of Logistics Systems*
- Laguna, Manuel / González-Velarde, José Luis / *Computing Tools for Modeling, Optimization and Simulation*
- Stilman, Boris / *Linguistic Geometry: From Search to Construction*
- Sakawa, Masatoshi / *Genetic Algorithms and Fuzzy Multiobjective Optimization*
- Ribeiro, Celso C./ Hansen, Pierre / *Essays and Surveys in Metaheuristics*
- Holsapple, Clyde/ Jacob, Varghese / Rao, H. R. / *BUSINESS MODELLING: Multidisciplinary Approaches — Economics, Operational and Information Systems Perspectives*
- Sleeker, Catherine M./ Wentling, Tim L./ Cude, Roger L. / *HUMAN RESOURCE DEVELOPMENT AND INFORMATION TECHNOLOGY: Making Global Connections*
- Voß, Stefan, Woodruff, David / *Optimization Software Class Libraries*
- Upadhyaya et al/ *MOBILE COMPUTING: Implementing Pervasive Information and Communications Technologies*
- Reeves, Colin & Rowe, Jonathan/ *GENETIC ALGORITHMS—Principles and Perspectives: A Guide to GA Theory*
- Bhargava, Hemant K. & Ye, Nong / *COMPUTATIONAL MODELING AND PROBLEM SOLVING IN THE NETWORKED WORLD: Interfaces in Computer Science & Operations Research*
- Woodruff, David L./ *NETWORK INTERDICTION AND STOCHASTIC INTEGER PROGRAMMING*
- Anandalingam, G. & Raghavan, S./ *TELECOMMUNICATIONS NETWORK DESIGN AND MANAGEMENT*
- Laguna, M. & Martí, R./ *SCATTER SEARCH: Methodology and Implementations in C*

SIMULATION-BASED OPTIMIZATION: Parametric Optimization Techniques and Reinforcement Learning

ABHIJIT GOSAVI
Department of Industrial Engineering
The State University of New York, Buffalo



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available from the Library of Congress.

Gosavi, A. / SIMULATION-BASED OPTIMIZATION: Parametric Optimization Techniques
and Reinforcement Learning

ISBN 978-1-4419-5354-4 ISBN 978-1-4757-3766-0 (eBook)
DOI 10.1007/978-1-4757-3766-0

Copyright © 2003 by Springer Science+Business Media New York

Originally published by Kluwer Academic Publishers in 2003

All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording, or otherwise, without the written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Permission for books published in Europe: permissions@wkap.nl

Permissions for books published in the United States of America: permissions@wkap.com

Printed on acid-free paper.

*To my parents:
Ashok and Sanjivani
Gosavi*

Contents

List of Figures	xvii
List of Tables	xxi
Acknowledgments	xxiii
Preface	xxv
1. BACKGROUND	1
1.1 Why this book was written	1
1.2 Simulation-based optimization and modern times	3
1.3 How this book is organized	7
2. NOTATION	9
2.1 Chapter Overview	9
2.2 Some Basic Conventions	9
2.3 Vector notation	9
2.3.1 Max norm	10
2.3.2 Euclidean norm	10
2.4 Notation for matrices	10
2.5 Notation for n -tuples	11
2.6 Notation for sets	11
2.7 Notation for Sequences	11
2.8 Notation for Transformations	11
2.9 Max, min, and arg max	12
2.10 Acronyms and Abbreviations	12
2.11 Concluding Remarks	12
3. PROBABILITY THEORY: A REFRESHER	15
3.1 Overview of this chapter	15
3.1.1 Random variables	15
3.2 Laws of Probability	16

3.2.1	Addition Law	17
3.2.2	Multiplication Law	18
3.3	Probability Distributions	21
3.3.1	Discrete random variables	21
3.3.2	Continuous random variables	22
3.4	Expected value of a random variable	23
3.5	Standard deviation of a random variable	25
3.6	Limit Theorems	27
3.7	Review Questions	28
4.	BASIC CONCEPTS UNDERLYING SIMULATION	29
4.1	Chapter Overview	29
4.2	Introduction	29
4.3	Models	30
4.4	Simulation Modeling of Random Systems	32
4.4.1	Random Number Generation	33
4.4.1.1	Uniformly Distributed Random Numbers	33
4.4.1.2	Other Distributions	36
4.4.2	Re-creation of events using random numbers	37
4.4.3	Independence of samples collected	42
4.4.4	Terminating and non-terminating systems	43
4.5	Concluding Remarks	44
4.6	Historical Remarks	44
4.7	Review Questions	45
5.	SIMULATION OPTIMIZATION: AN OVERVIEW	47
5.1	Chapter Overview	47
5.2	Stochastic parametric optimization	47
5.2.1	The role of simulation in parametric optimization	50
5.3	Stochastic control optimization	51
5.3.1	The role of simulation in control optimization	53
5.4	Historical Remarks	54
5.5	Review Questions	54
6.	RESPONSE SURFACES AND NEURAL NETS	57
6.1	Chapter Overview	57
6.2	RSM: An Overview	58
6.3	RSM: Details	59
6.3.1	Sampling	60
6.3.2	Function Fitting	60
6.3.2.1	Fitting a straight line	60
6.3.2.2	Fitting a plane	63
6.3.2.3	Fitting hyper-planes	64

6.3.2.4 Piecewise regression	65
6.3.2.5 Fitting non-linear forms	66
6.3.3 How good is the metamodel?	67
6.3.4 Optimization with a metamodel	68
6.4 Neuro-Response Surface Methods	69
6.4.1 Linear Neural Networks	69
6.4.1.1 Steps in the Widrow-Hoff Algorithm	72
6.4.1.2 Incremental Widrow-Hoff	72
6.4.1.3 Pictorial Representation of a Neuron	73
6.4.2 Non-linear Neural Networks	73
6.4.2.1 The Basic Structure of a Non-Linear Neural Network	75
6.4.2.2 The Backprop Algorithm	78
6.4.2.3 Deriving the backprop algorithm	79
6.4.2.4 Backprop with a Bias Node	82
6.4.2.5 Deriving the algorithm for the bias weight	82
6.4.2.6 Steps in Backprop	84
6.4.2.7 Incremental Backprop	86
6.4.2.8 Example D	88
6.4.2.9 Validation of the neural network	89
6.4.2.10 Optimization with a neuro-RSM model	90
6.5 Concluding Remarks	90
6.6 Bibliographic Remarks	90
6.7 Review Questions	91
7. PARAMETRIC OPTIMIZATION	93
7.1 Chapter Overview	93
7.2 Continuous Optimization	94
7.2.1 Gradient Descent	94
7.2.1.1 Simulation and Gradient Descent	98
7.2.1.2 Simultaneous Perturbation	101
7.2.2 Non-derivative methods	104
7.3 Discrete Optimization	106
7.3.1 Ranking and Selection	107
7.3.1.1 Steps in the Rinott method	108
7.3.1.2 Steps in the Kim-Nelson method	109
7.3.2 Meta-heuristics	110
7.3.2.1 Simulated Annealing	111
7.3.2.2 The Genetic Algorithm	117
7.3.2.3 Tabu Search	119
7.3.2.4 A Learning Automata Search Technique	123
7.3.2.5 Other Meta-Heuristics	128
7.3.2.6 Ranking and selection & meta-heuristics	128
7.4 Hybrid solution spaces	128
7.5 Concluding Remarks	129

7.6	Bibliographic Remarks	129
7.7	Review Questions	131
8.	DYNAMIC PROGRAMMING	133
8.1	Chapter Overview	133
8.2	Stochastic processes	133
8.3	Markov processes, Markov chains and semi-Markov processes	136
8.3.1	Markov chains	139
8.3.1.1	n -step transition probabilities	140
8.3.2	Regular Markov chains	142
8.3.2.1	Limiting probabilities	143
8.3.3	Ergodicity	145
8.3.4	Semi-Markov processes	146
8.4	Markov decision problems	148
8.4.1	Elements of the Markov decision framework	151
8.5	How to solve an MDP using exhaustive enumeration	157
8.5.1	Example A	158
8.5.2	Drawbacks of exhaustive enumeration	161
8.6	Dynamic programming for average reward	161
8.6.1	Average reward Bellman equation for a policy	162
8.6.2	Policy iteration for average reward MDPs	163
8.6.2.1	Steps	163
8.6.3	Value iteration and its variants: average reward MDPs	165
8.6.4	Value iteration for average reward MDPs	165
8.6.4.1	Steps	166
8.6.5	Relative value iteration	168
8.6.5.1	Steps	168
8.6.6	A general expression for the average reward of an MDP	169
8.7	Dynamic programming and discounted reward	170
8.7.1	Discounted reward	171
8.7.2	Discounted reward MDP	171
8.7.3	Bellman equation for a policy: discounted reward	173
8.7.4	Policy iteration for discounted reward MDPs	173
8.7.4.1	Steps	174
8.7.5	Value iteration for discounted reward MDPs	175
8.7.5.1	Steps	176
8.7.6	Getting value iteration to converge faster	177
8.7.6.1	Gauss Siedel value iteration	178
8.7.6.2	Relative value iteration for discounted reward	179
8.7.6.3	Span seminorm termination	180
8.8	The Bellman equation: An intuitive perspective	181
8.9	Semi-Markov decision problems	182
8.9.1	The natural process and the decision-making process	184
8.9.2	Average reward SMDPs	186

8.9.2.1	Exhaustive enumeration for average reward SMDPs	186
8.9.2.2	Example B	187
8.9.2.3	Policy iteration for average reward SMDPs	189
8.9.2.4	Value iteration for average reward SMDPs	191
8.9.2.5	Counterexample for regular value iteration	192
8.9.2.6	Uniformization for SMDPs	193
8.9.2.7	Value iteration based on the Bellman equation	194
8.9.2.8	Extension to random time SMDPs	194
8.9.3	Discounted reward SMDPs	194
8.9.3.1	Policy iteration for discounted SMDPs	195
8.9.3.2	Value iteration for discounted reward SMDPs	195
8.9.3.3	Extension to random time SMDPs	196
8.9.3.4	Uniformization	196
8.10	Modified policy iteration	197
8.10.1	Steps for discounted reward MDPs	198
8.10.2	Steps for average reward MDPs	199
8.11	Miscellaneous topics related to MDPs and SMDPs	200
8.11.1	A parametric-optimization approach to solving MDPs	200
8.11.2	The MDP as a special case of a stochastic game	201
8.11.3	Finite Horizon MDPs	203
8.11.4	The approximating sequence method	206
8.12	Conclusions	207
8.13	Bibliographic Remarks	207
8.14	Review Questions	208
9.	REINFORCEMENT LEARNING	211
9.1	Chapter Overview	211
9.2	The Need for Reinforcement Learning	212
9.3	Generating the TPM through straightforward counting	214
9.4	Reinforcement Learning: Fundamentals	215
9.4.1	Q -factors	218
9.4.1.1	A Q -factor version of value iteration	219
9.4.2	The Robbins-Monro algorithm	220
9.4.3	The Robbins-Monro algorithm and Q -factors	221
9.4.4	Simulators, asynchronous implementations, and step sizes	222
9.5	Discounted reward Reinforcement Learning	224
9.5.1	Discounted reward RL based on value iteration	224
9.5.1.1	Steps in Q -Learning	225
9.5.1.2	Reinforcement Learning: A “Learning” Perspective	227
9.5.1.3	On-line and Off-line	229
9.5.1.4	Exploration	230
9.5.1.5	A worked-out example for Q -Learning	231
9.5.2	Discounted reward RL based on policy iteration	234

9.5.2.1	<i>Q</i> -factor version of regular policy iteration	235
9.5.2.2	Steps in the <i>Q</i> -factor version of regular policy iteration	235
9.5.2.3	Steps in <i>Q</i> - <i>P</i> -Learning	237
9.6	Average reward Reinforcement Learning	238
9.6.1	Discounted RL for average reward MDPs	238
9.6.2	Average reward RL based on value iteration	238
9.6.2.1	Steps in Relative <i>Q</i> -Learning	239
9.6.2.2	Calculating the average reward of a policy in a simulator	240
9.6.3	Other algorithms for average reward MDPs	241
9.6.3.1	Steps in <i>R</i> -Learning	241
9.6.3.2	Steps in SMART for MDPs	242
9.6.4	An RL algorithm based on policy iteration	244
9.6.4.1	Steps in <i>Q</i> - <i>P</i> -Learning for average reward	244
9.7	Semi-Markov decision problems and RL	245
9.7.1	Discounted Reward	245
9.7.1.1	Steps in <i>Q</i> -Learning for discounted reward DTMDPs	245
9.7.1.2	Steps in <i>Q</i> - <i>P</i> -Learning for discounted reward DTMDPs	246
9.7.2	Average reward	247
9.7.2.1	Steps in SMART for SMDPs	248
9.7.2.2	Steps in <i>Q</i> - <i>P</i> -Learning for SMDPs	250
9.8	RL Algorithms and their DP counterparts	252
9.9	Actor-Critic Algorithms	252
9.10	Model-building algorithms	253
9.10.1	<i>H</i> -Learning for discounted reward	254
9.10.2	<i>H</i> -Learning for average reward	255
9.10.3	Model-building <i>Q</i> -Learning	257
9.10.4	Model-building relative <i>Q</i> -Learning	258
9.11	Finite Horizon Problems	259
9.12	Function approximation	260
9.12.1	Function approximation with state aggregation	260
9.12.2	Function approximation with function fitting	262
9.12.2.1	Difficulties	262
9.12.2.2	Steps in <i>Q</i> -Learning coupled with neural networks	264
9.12.3	Function approximation with interpolation methods	265
9.12.4	Linear and non-linear functions	269
9.12.5	A robust strategy	269
9.12.6	Function approximation: Model-building algorithms	270
9.13	Conclusions	270
9.14	Bibliographic Remarks	271
9.14.1	Early works	271
9.14.2	Neuro-Dynamic Programming	271
9.14.3	RL algorithms based on <i>Q</i> -factors	271
9.14.4	Actor-critic Algorithms	272
9.14.5	Model-building algorithms	272

9.14.6 Function Approximation	273
9.14.7 Some other references	273
9.14.8 Further reading	273
9.15 Review Questions	273
10. MARKOV CHAIN AUTOMATA THEORY	277
10.1 Chapter Overview	277
10.2 The MCAT framework	278
10.2.1 The working mechanism of MCAT	278
10.2.2 Step-by-step details of an MCAT algorithm	280
10.2.3 An illustrative 3-state example	282
10.2.4 What if there are more than two actions?	284
10.3 Concluding Remarks	285
10.4 Bibliographic Remarks	285
10.5 Review Questions	285
11. CONVERGENCE: BACKGROUND MATERIAL	287
11.1 Chapter Overview	287
11.2 Vectors and Vector Spaces	288
11.3 Norms	290
11.3.1 Properties of Norms	291
11.4 Normed Vector Spaces	291
11.5 Functions and Mappings	291
11.5.1 Domain and Range of a function	291
11.5.2 The notation for transformations	293
11.6 Mathematical Induction	294
11.7 Sequences	297
11.7.1 Convergent Sequences	298
11.7.2 Increasing and decreasing sequences	300
11.7.3 Boundedness	300
11.8 Sequences in \mathcal{R}^n	306
11.9 Cauchy sequences in \mathcal{R}^n	307
11.10 Contraction mappings in \mathcal{R}^n	308
11.11 Bibliographic Remarks	315
11.12 Review Questions	315
12. CONVERGENCE: PARAMETRIC OPTIMIZATION	317
12.1 Chapter Overview	317
12.2 Some Definitions and a result	317
12.2.1 Continuous Functions	318
12.2.2 Partial derivatives	319
12.2.3 A continuously differentiable function	319
12.2.4 Stationary points, local optima, and global optima	319

12.2.5 Taylor's theorem	320
12.3 Convergence of gradient-descent approaches	323
12.4 Perturbation Estimates	327
12.4.1 Finite Difference Estimates	327
12.4.2 Notation	328
12.4.3 Simultaneous Perturbation Estimates	328
12.5 Convergence of Simulated Annealing	333
12.6 Concluding Remarks	341
12.7 Bibliographic Remarks	341
12.8 Review Questions	341
13. CONVERGENCE: CONTROL OPTIMIZATION	343
13.1 Chapter Overview	343
13.2 Dynamic programming transformations	344
13.3 Some definitions	345
13.4 Monotonicity of $T, T_{\hat{\mu}}, L$, and $L_{\hat{\mu}}$	346
13.5 Some results for average & discounted MDPs	347
13.6 Discounted reward and classical dynamic programming	349
13.6.1 Bellman Equation for Discounted Reward	349
13.6.2 Policy Iteration	356
13.6.3 Value iteration for discounted reward MDPs	359
13.7 Average reward and classical dynamic programming	364
13.7.1 Bellman equation for average reward	365
13.7.2 Policy iteration for average reward MDPs	368
13.7.3 Value Iteration for average reward MDPs	372
13.8 Convergence of DP schemes for SMDPs	379
13.9 Convergence of Reinforcement Learning Schemes	379
13.10 Background Material for RL Convergence	380
13.10.1 Non-Expansive Mappings	380
13.10.2 Lipschitz Continuity	380
13.10.3 Convergence of a sequence with probability 1	381
13.11 Key Results for RL convergence	381
13.11.1 Synchronous Convergence	382
13.11.2 Asynchronous Convergence	383
13.12 Convergence of RL based on value iteration	392
13.12.1 Convergence of Q -Learning	392
13.12.2 Convergence of Relative Q -Learning	397
13.12.3 Finite Convergence of Q -Learning	397
13.13 Convergence of Q - P -Learning for MDPs	400
13.13.1 Discounted reward	400
13.13.2 Average Reward	401
13.14 SMDPs	402

13.14.1 Value iteration for average reward	402
13.14.2 Policy iteration for average reward	402
13.15 Convergence of Actor-Critic Algorithms	404
13.16 Function approximation and convergence analysis	405
13.17 Bibliographic Remarks	406
13.17.1 DP theory	406
13.17.2 RL theory	406
13.18 Review Questions	407
14. CASE STUDIES	409
14.1 Chapter Overview	409
14.2 A Classical Inventory Control Problem	410
14.3 Airline Yield Management	412
14.4 Preventive Maintenance	416
14.5 Transfer Line Buffer Optimization	420
14.6 Inventory Control in a Supply Chain	423
14.7 AGV Routing	424
14.8 Quality Control	426
14.9 Elevator Scheduling	427
14.10 Simulation optimization: A comparative perspective	429
14.11 Concluding Remarks	430
14.12 Review Questions	430
15. CODES	433
15.1 Introduction	433
15.2 C programming	434
15.3 Code Organization	436
15.4 Random Number Generators	437
15.5 Simultaneous Perturbation	439
15.6 Dynamic Programming Codes	441
15.6.1 Policy Iteration for average reward MDPs	442
15.6.2 Relative Iteration for average reward MDPs	447
15.6.3 Policy Iteration for discounted reward MDPs	450
15.6.4 Value Iteration for discounted reward MDPs	453
15.6.5 Policy Iteration for average reward SMDPs	460
15.7 Codes for Neural Networks	464
15.7.1 Neuron	465
15.7.2 Backprop Algorithm — Batch Mode	470
15.8 Reinforcement Learning Codes	478
15.8.1 Codes for <i>Q</i> -Learning	478
15.8.2 Codes for Relative <i>Q</i> -Learning	486
15.8.3 Codes for Relaxed-SMART	495

15.9 Codes for the Preventive Maintenance Case Study	506
15.9.1 Learning Codes	507
15.9.2 Fixed Policy Codes	521
15.10 MATLAB Codes	531
15.11 Concluding Remarks	535
15.12 Review Questions	535
16. CONCLUDING REMARKS	537
References	539
Index	551

List of Figures

4.1	A Schematic of a queue that forms in a bank or supermarket.	30
4.2	The event clock showing arrivals and departures.	39
6.1	Fitting a straight line.	61
6.2	Fitting a piecewise linear function.	65
6.3	Fitting a non-linear equation with one independent variable	66
6.4	Fitting a non-linear equation with two independent variables	67
6.5	A linear network — a neuron with three input nodes and one output node. The approximated function is a plane with two independent variables: $x(1)$ and $x(2)$. The node with input $x(0)$ assumes the role of the constant a in regression.	74
6.6	A non-linear neural network with an input layer, one hidden layer and one output node. The term $w(i, h)$ denotes a weight on the link from the i th input node to the h th hidden node. The term $x(h)$ denotes the weight on the link from the h th hidden node to the output node.	75
6.7	Deriving the value of the output for given values of inputs and weights.	77
6.8	A neural network with a bias node. The topmost node is the bias node. The weight on the direct link to the output node is b .	83
6.9	Actual implementation with a bias node	88
7.1	A surface with multiple minima	98
7.2	The Mechanism Underlying LAST	124
8.1	A single-server queue	134
8.2	The figure shows the queue in two different <i>states</i> . The state is defined by the number in the queue.	135
8.3	Schematic of a two-state Markov chain, where circles denote states.	136

8.4	Schematic of a two-state Markov chain, where circles denote states, arrows depict possible transitions, and the numbers on the arrows denote the probabilities of those transitions.	140
8.5	Schematic of a Markov chain with 3 states	141
8.6	Schematic of a Markov chain with 4 states	141
8.7	A Markov chain with one transient state	147
8.8	A Markov chain with two transient states	147
8.9	A Markov chain underlying a simple single server queue.	149
8.10	Schematic showing how the TPM of policy (2, 1) is constructed from the TPMs of action 1 and 2.	153
8.11	Calculation of expected immediate reward	156
8.12	A two state MDP	159
8.13	Total discounted reward calculation on an infinitely long trajectory	181
8.14	Immediate reward in one transition	182
8.15	A Schematic for a Finite Horizon MDP	204
9.1	Solving an MDP using DP <i>after</i> estimating the TPMs and TRMs in a simulator	216
9.2	A schematic highlighting the differences in the methodologies of RL and DP	217
9.3	The updating of the Q -factors in a simulator. Each arrow denotes a state transition in the simulator. After going to state j , the Q -factor for the previous state i and the action a selected in i , that is, $Q(i, a)$ is updated.	225
9.4	Trial and error mechanism of RL. The action selected by the RL agent (algorithm) is fed into the simulator. The simulator simulates the action, and the resultant feedback (immediate reward) obtained is fed back into the knowledge-base (Q -factors) of the agent. The agent uses the RL algorithm to update its knowledge-base, becomes smarter in the process, and then selects a better action.	228
9.5	Function approximation using interpolation: The circles denote the representative Q -factors. Notice how the locations of the representatives change with updating. The darker circles indicate the <i>updated</i> Q -factors. When a Q -factor at a location, where no Q -factor exists, has to be updated, its nearest neighbor is <i>eliminated</i> . It is <i>not</i> necessary to start out with <i>uniformly-spaced</i> Q -factors as shown in this figure.	266

9.6	Schematics showing a 2-dimensional state space. The top schematic shows equal sized compartments and the bottom one shows an unequal sized compartments. Within each compartment, a neuron may be placed.	270
9.7	A Grid World	275
11.1	The thin line represents vector \vec{a} , the dark line represents the vector \vec{b} , and the dotted line the vector \vec{c} . This is before applying G .	310
11.2	The thin line represents vector \vec{a} , the dark line represents the vector \vec{b} , and the dotted line the vector \vec{c} . This is the picture after one application of G . Notice that the vectors have come closer.	311
11.3	This is the picture after 11 applications of F . By now the vectors are almost on the top of each other and it is difficult to distinguish between them.	311
12.1	Strict local optima and saddle points	321
12.2	Local optima and global optima	321
14.1	A schematic showing classification of customers based on the time of arrival.	414
14.2	A schematic showing classification of customers based on the origin (circle) and the destination, in one particular leg of an airline flight.	415
14.3	The graph shows that there is an optimal time to maintain.	417
14.4	A production-inventory system	418
14.5	A transfer line with buffers	421
14.6	An Automated Guided Vehicle System with one AGV	425

List of Tables

2.1	A List of acronyms and abbreviations used in the book	13
7.1	Table shows values of objective function	132
8.1	Table showing calculations in policy iteration for average reward MDPs on Example A	164
8.2	Table showing calculations in value iteration for average reward MDPs. Note that the values get unbounded. Although, we do not show the calculations in the table, the span semi-norm <i>does</i> decrease with every iteration.	167
8.3	Table showing calculations in <i>Relative</i> value iteration for average reward MDPs. The value of ϵ is 0.001. At the 11th iteration, the ϵ -optimal policy is found.	169
8.4	Table showing calculations in policy iteration for discounted MDPs	175
8.5	Table showing calculations in value iteration for discounted reward MDPs. The value of ϵ is 0.001. The norm is checked with $0.5\epsilon(1 - \lambda)/\lambda = 0.00125$. At the 52nd iteration, the ϵ -optimal policy is found.	177
8.6	Table showing calculations in value iteration for discounted reward MDPs. The value of ϵ is 0.001. The norm is checked with $0.5\epsilon(1 - \lambda)/\lambda = 0.00125$. At the 32nd iteration, the ϵ -optimal policy is found.	179
8.7	Table showing calculations in policy iteration for average reward SMDPs (Example B)	191
9.1	A comparison of RL, DP, and heuristics. Note that both DP and RL use the MDP model.	214
11.1	Table showing the change in values of the vectors \vec{a} and \vec{b} after repeated applications of G	310

Acknowledgments

I would like to acknowledge a number of people without whose support this book could never have been written. I am very grateful to Gary Folven at Kluwer Academic for his encouragement of this project. I would also like to express my deep appreciation for the enthusiasm and support extended to me by my former department Chair, Jane M. Fraser, at Colorado State University, Pueblo.

Over the last few years, a number of individuals have helped me either by pointing to useful references, sending their work and helping me understand their work better, or reviewing the manuscript. I thank Jayant Rajgopal (University of Pittsburgh), Ramesh Sharda (Oklahoma State University), James Spall (Johns Hopkins University), Kumapati Narendra (Yale University), Alice Smith (Auburn University), Janet Barnett (Colorado State University, Pueblo), Prasad Tadepalli (Oregon State University), Tito Homem-de-Mello (the Ohio State University), Vivek Borkar (the Tata Institute of Fundamental Research, Mumbai), Sigrún Andradóttir (Georgia Institute of Technology), and Barry Nelson (Northwestern University).

Parts of this book are based on notes prepared for material needed in graduate-level courses that I have taught in the last four years. The material was developed for teaching stochastic processes, simulation, classical optimization theory, heuristic optimization, and statistics. Many students have patiently struggled through my notes, and many have made useful suggestions for which I am thankful. A number of my graduate students have read parts of the manuscript and have made helpful comments; I thank Tolga Tezcan, Lisa Haynes, and Emrah Ozkaya.

I would also like to thank some of my former teachers: Tapas K. Das, who taught me the value of good research, Sridhar Mahadevan (now at the University of Massachusetts, Amherst), who introduced me to reinforcement learning, Michael X. Weng, Suresh K. Khator, Sudeep Sarkar, Geoffrey O. Okogbaa, and William A. Miller.

Many of the verses, placed at the beginning of chapters, have come from Wayne Dyer's *Wisdom of the Ages*. I would like to thank the owners of Numerical Recipes in C for allowing me to use one of their computer programs in this book.

Finally, and most importantly, I thank my parents, and other family members, Anuradha, Rajeev, Sucheta, Aparna and Milind, who have been robust pillars of support.

Neither the publisher nor myself is responsible for errors or any misuse of the computer programs in this book. I take full responsibility for all other errors in the book. If you find any errors or have any suggestions for improvement of this book, please send an email to me at agosavi@buffalo.edu

Abhijit Gosavi

**June, 2003
Buffalo, New York**

Preface

This book is written for students and researchers in the field of industrial engineering, computer science, operations research, management science, electrical engineering, and applied mathematics. The aim of the book is to introduce the reader to the newly-emerging and exciting topic of simulation-based optimization.

The reader interested in solving complex and large-scale problems of optimization in random (stochastic) systems should find this book useful. The book introduces the ideas of parametric optimization and reinforcement learning — in the context of stochastic optimization problems.

If you are working on a problem that involves a stochastic system, and if the problem is one of optimization, you are likely to find useful material here. Furthermore, the book is self-contained; all that is expected of you is: some background in elementary college calculus (basics of differential and integral calculus) and linear algebra (matrices and vectors). No training in real analysis or topology is required.

Any stochastic-optimization problem that can be solved with computer simulation is referred to as a **simulation-optimization** problem in this book. This book focuses on simulation-optimization techniques for solving stochastic-optimization problems. Today, simulation-based optimization is on the **cutting edge** of stochastic optimization — thanks to some recent path-breaking work in this field!

Although this science has a rigorous mathematical foundation, our development in the initial chapters is based on intuitively appealing explanations of the major concepts. Only from Chapter 11, do we adopt a more (mathematically) rigorous approach.

Broadly speaking, the book has two parts:

1. Parametric (static) optimization
2. Control (dynamic) optimization

By *parametric* optimization, we refer to static optimization in which the goal is to find the values of *parameters* that maximize or minimize a function — usually a function of those parameters. By *control* optimization, we refer to those dynamic optimization problems in which the goal is to find an optimal *control* (action) in each state visited by a system. The book describes these models and the associated optimization techniques — in the context of stochastic (random) systems — in some detail. While the book presents some classical paradigms to develop the background, the focus is on recent research in both parametric and control optimization. For example, exciting, recently-developed techniques such as *simultaneous perturbation* (parametric optimization) and *reinforcement learning* (control optimization) are two of the main topics.

A common thread running through the book is that of simulation. Optimization techniques considered in this book require simulation — as opposed to explicit mathematical models. Some special features of this book are:

- An accessible introduction to **reinforcement learning** and parametric optimization techniques.
- A step-by-step description of several algorithms of simulation-based optimization.
- A clear and simple introduction to the methodology of **neural networks**.
- A gentle introduction to convergence analysis of some of methods discussed in this book.
- **Computer programs** for many algorithms of simulation-based optimization.

In a one-semester course of 14-16 weeks, I believe the following chapters can be covered.

- Chapter 1: Background.
- Chapter 5: An introduction to simulation-based optimization.
- Chapter 6: Response surface methods and neural networks.
- Chapter 7: Parametric optimization.
- Chapter 8: Control optimization with dynamic programming.
- Chapter 9: Control optimization with reinforcement learning.
- Chapter 14: Case Studies.

The material on neural networks is not critical for understanding reinforcement learning, nor is it an integral part of Chapter 6. So neural networks could

be covered depending on whether time is available. The necessary background of probability and simulation can be covered from Chapters 3 and 4 — if students are not already familiar with it. The codes in Chapter 15 can be used in various exercises. If time permits, Chapter 10 should also be covered.

The remaining material, which is mostly convergence-related, is for those interested in research in this field. It could also be taught in the second semester in a two-semester course — perhaps at an advanced graduate level.

- Chapter 11.
- Chapter 12.
- Chapter 13.

Chapter 1

BACKGROUND

Do not believe what you have heard.

Do not believe in anything that has been spoken of many times.

Do not believe because the written statements come from some old sage.

Do not believe in authority or teachers or elders.

But after careful observation and analysis, when it agrees with reason and it will benefit one and all, then accept it and live by it.

— Buddha (563 B.C. - 483 B.C.)

1. Why this book was written

This book seeks to introduce the reader to the rapidly evolving subject that is called simulation-based optimization. This is not a very young topic. From the time computers started making an impact on scientific research and from the time it became possible to analyze a random system using a computer program that generated random numbers, scientists and engineers have always wanted to *optimize* systems using simulation models. However, it is only recently that noteworthy success in realizing this objective has been seen in practice.

Path-breaking work in computational operations research in areas such as non-linear programming (simultaneous perturbation), dynamic programming (reinforcement learning), and game theory (learning automata) has made it possible for us now to use simulation in conjunction with optimization techniques. This has given simulation the kind of power that it did not have in the bad old days when simulation optimization was usually treated as a synonym for the relatively sluggish (although robust) response surface method. Fortunately

things have changed, and simulation optimization today has acquired the status of a powerful tool in the toolkit of the stochastic programmer.

Advances in simulation optimization have been brought about not so much by any fundamental changes in the methods we use to simulate a system. Of course, the power of computers has increased dramatically over the years, it and continues to increase. This has helped increase the speed of running computer programs, but the over-riding factor that has turned things around for simulation optimization is the remarkable research that has taken place in various areas of computational operations research — research that has either given birth to new optimization techniques that are more compatible with simulation, or in many cases research that has generated modified versions of old optimization techniques that can be combined more elegantly with simulation. In any case, the success of these new techniques has to be, obviously, attributed to their firm mathematical roots in operations research. These strong mathematical roots have helped lead simulation optimization onto a new path where it has attacked problems which were previously considered intractable.

Surprisingly, the success stories have been reported in widely different, albeit related, areas of operations research. Not surprisingly, all of these success stories have a natural connection — the connecting thread being that of an adroit integration of computer simulation with an optimization technique. Hence in the opinion of this author, there is a need for a book that presents this subject in a unified manner.

Simulation-based optimization now has a finger in almost every pie of modern stochastic optimization. This phenomenon can be traced to the fact that as a rule it hinges on one central and surprisingly simple concept — that of computing the expected value (the mean or the average) of a random variable from its samples. This elementary idea finds application in many a stochastic optimization scenario. One may wonder: what is so wonderful about this notion? It is after all not very new. The answer to this question is that even though this may be a straightforward concept, it cannot always be used in a straightforward manner because a naive application of this idea can make the procedure computationally burdensome. Considerable research has gone into finding ways to devise optimization methods, in which such a computation (mean from the sample) can be used directly by an optimization procedure, with a relatively light computational burden. This book focuses on this very idea.

In view of this focus, one must first deal with the fundamental concepts underlying simulation optimization. The primary goal in writing this book is to lay bare the mathematical roots of the governing principles of this science. It is hoped that the reader will gain the ability to model relevant real-life problems using the simulation-optimization techniques discussed in this book. This is the **primary** objective.

The other objective is to expose the reader to the **technology** underlying this science — the knowledge of which is essential for its successful application to real-world problems. This is almost invariably true of any numerical technique. It forms an important ingredient of mastering its use, and a book that ignores this aspect remains somewhat incomplete. In plain English, a major objective of this book is to present some of the computational issues related to getting this science to work on real-world problems. To this end, we have supplied some computer codes for some of the algorithms discussed in this book.

And last but not the least, we have tried to expose the reader to the use of mathematically rigorous arguments for showing that the methods discussed in this book *do* work. This is loosely referred to in the existing literature as “convergence.”

Satisfactory experimental results with a particular algorithm in a specific problem scenario (which form a major component of this book) are never sufficient to ensure that the same algorithm performs equally well in other scenarios. Hence the importance of the study of theoretical convergence properties Of algorithms cannot be downplayed. Not only is this study important for theoretical reasons but also from a computational standpoint because a convergence analysis frequently explains why certain algorithms work in certain domains but fare poorly in others. As a result of this belief, we have analyzed convergence properties of some algorithms discussed in this book.

Overwhelming the beginner with mathematical details is certainly not the intention here, which is why discussion on convergence has been relegated to separate chapters towards the end of this book. As such, a major portion of the book can be understood by a reader with an elementary knowledge of linear algebra and calculus. Even the chapters related to convergence do not require any prior knowledge of mathematical analysis. Effort has been made to explain the convergence issues to a reader who has taken an elementary course in calculus but is unfamiliar with advanced calculus or real analysis.

To understand and use the computer programs presented in the book, one must have some background in computer programming.

2. Simulation-based optimization and modern times

A large number of factors of practical and applied origin, which play a major role in today’s industrial world, have stimulated research in simulation optimization. The biggest factor is the ever-increasing complexity of systems that modern engineers need to optimize. Traditional tools in stochastic optimization rapidly break down in the face of the complexity and the sheer size of the problems posed by modern service and manufacturing industries. By such tools, we mean a) traditional non-linear-programming methods, e.g., gradient descent with finite differences and b) classical algorithms in stochastic dynamic programming, e.g., value iteration and policy iteration.

Optimization problems in large-scale stochastic scenarios belong to two categories:

1. parametric optimization (also called **static** optimization) and
2. control optimization (also called **dynamic** optimization).

Parametric optimization is optimization performed to find the values for *a set of parameters* that optimize some performance measure (minimize a cost or maximize a reward). On the other hand, control optimization refers to finding *a set of actions* to be taken in the different states that a system can visit in order to optimize some performance measure of the system (minimize a cost or maximize a reward).

Parametric optimization can be performed using mathematical programming methods such as linear, non-linear and, integer programming. Control optimization may be performed via dynamic programming.

Parametric optimization is often called *static* optimization because the solution is a set of “static” parameters for all states. Control optimization is called *dynamic* optimization because the solution depends on the state; since the state changes “dynamically,” so does the solution.

The focus of this book is on *complex, large-scale, stochastic* systems, where a naive application of classical methods does not work. In such stochastic scenarios, if the problem is one of parametric optimization, formulating the objective function can often prove to be difficult. More often than not the objective function is non-linear and has multiple integrals and probabilistic elements. It goes without saying that the larger the number of random variables in a system the more complex the system gets for one to be able to express the objective function in the form of an algebraic expression. Naturally, under such circumstances, one is forced to resort to a simulation-based evaluation of the objective function. And if the only method to evaluate the objective function is simulation, one has to use a class of non-linear programming methods that rely *only* on function evaluations. Gradient descent with finite differences and the Nelder-Mead downhill simplex search are examples of such methods.

The dependence on function evaluations has generated research interest in non-linear programming methods that can work without needing an excessive number of function evaluations. This is so because even a one-time numerical evaluation of the objective function of a complex stochastic system via simulation is, computationally, a very expensive affair, and a method, such as gradient descent with regular finite differences, which requires a very large number of function evaluations, takes a prohibitively large amount of computational time to generate a solution. On the other hand, Spall’s simultaneous perturbation, which is a non-linear programming method, in comparison to other non-linear programming methods, requires few function evaluations, and as such has a

relatively low computational burden. This book will focus on methods that can produce good solutions with a light computational burden.

As far as control optimization is concerned, it is well known that stochastic dynamic programming, which happens to be one of its primary tools, suffers from the curses of *modeling* and *dimensionality*.

The curse of modeling can be interpreted as follows. When exposed to a problem with very complex stochastics, stochastic dynamic programming breaks down. This is because it requires the computation of the so-called *transition probabilities*, and for **complex** systems, it can be very difficult to compute them.

The curse of dimensionality can be interpreted as follows. When the number of states in the system becomes very large (say of the order of millions), the number of transition probabilities goes through the roof. For example, for a problem with one thousand states, which is incidentally a very small problem in the real-world context, one has to store one million transition probabilities just for one action. Since these probabilities are difficult to store, it is difficult to process them for generating a solution.

These two factors have inspired researchers to devise methods that generate optimal or near-optimal solutions *without having to compute or store transition probabilities*. Reinforcement learning is one such method. Not surprisingly reinforcement learning is simulation-based. Reinforcement learning uses a version of stochastic dynamic programming That does not require transition probabilities. The interesting and exciting fact about it is that even without using transition probabilities, reinforcement learning can be shown to generate optimal solutions. In recent literature, not only have theoretical guarantees appeared for reinforcement learning, but a significant amount of supportive *empirical* evidence has also surfaced.

As mentioned previously, this book concentrates on two different areas of stochastic optimization — parametric optimization and control optimization. Our focus will be on relatively new methods such as simultaneous perturbation, reinforcement learning and meta-heuristics. These methods have revolutionized the way optimization can be put to work in large-scale, complex, stochastic scenarios. The practical relevance of these methods is undeniable. At the same time, it must be emphasized that a majority of the methods presented here have roots in classical optimization theory and have proven convergence properties.

As a matter of fact, these new methods have succeeded where their traditional counterparts have failed. Globalization of the economy has increased the level of competition in the industry forcing industries to optimize areas previously run in heuristic modes. As a consequence, the modern engineer finds herself (himself) bombarded by very large-sized problems for many of which traditional optimization methods have no answers. These factors have generated an interest in optimization of large-scale systems that have uncertainty in their behavior.

This is evident from the ever-increasing number of special issues of well-known journals devoted to large-scale problems.

Operations researchers and industrial engineers working in the field of stochastic optimization are often criticized for the high degree of theory in their recommendations, the difficulty experienced in the implementation of the methods they suggest, and if not for anything else for the fact that they make too many simplifying assumptions in their models. The latter is often interpreted to imply that their approaches are not useful in real life. For various reasons some of the academic research in this field has not translated into applications in industry. The theory of Markov decision processes, for instance, has evolved significantly along theoretical lines, but is yet to make any substantial impact in solving industrial problems.

Much of the literature in stochastic optimization from the sixties and the seventies focuses on developing exact mathematical models for the problem at hand, *after making simplifying assumptions about the system*. A commonly made assumption is the use of the exponential distribution for random variables in the system. This assumption is often made because of its convenience but not always due to any real-life considerations; it keeps the mathematics tractable, paving the way for an elegant closed-form expression in the model. However in the last decade or so, a large number of researchers have concentrated on developing simulation-based solutions, *which usually do not require such assumptions*, to stochastic-optimization problems. What is more exciting is that it is becoming clear that simulation can actually be used for optimization.

And despite this, stochastic optimization is still in its early childhood. There are many indicators of this. For instance, there is very little in the literature by way of studying systems in which distributions tend to change over time. A major difficulty in any stochastic study is that one has to first collect data related to the random variables that govern the system's behavior and then fit distributions to it. What if the distributions change by the time data collection is completed?

Obviously such issues still loom large over this science. Also it is true that only in the last few years have we started seeing serious attempts to model realistic, large-scale systems. For several years, the curses of modeling and dimensionality were considered unbreakable barriers, and problems that fell under their influence were considered too difficult to be solved using any optimal-seeking method. This is changing.

This book is meant to achieve three goals, which are:

Goal I. To introduce the reader to the topic of simulation-based optimization, concentrating on two methodologies:

- Parametric optimization and
- Control optimization.

Goal II. To introduce the reader to some elementary but important mathematical results, which prove that the optimization algorithms presented in the book do actually produce optimal or in some cases near-optimal solutions.

Goal III. To present computer programs for simulation-based optimization.

The prerequisites for reading and understanding material presented in this book depend on what you are interested in. If you are interested in Goals I and II, all that is needed is a course in linear algebra and a college level calculus course. It has been our intention to present material to a reader, who is unfamiliar with mathematical analysis and measure theory. If you are interested in the Goal III, you should also have some knowledge of computer programming.

Theoretical presentation of almost all the algorithms presented in the book has been supplemented by case studies. The case studies have been drawn from various areas of operations research. The idea is to demonstrate the use of simulation-based optimization on real-life problems.

From reading the chapters related to convergence, the reader, it is hoped, will gain an understanding of the theoretical methods used to establish that a given algorithm works (or does not work). Regardless of how successful an algorithm is in one particular scenario, at the end of the day, what remains with us is the fact that an algorithm without mathematical roots is likely to be swept away by the winds of time.

3. How this book is organized

The rest of this book is organized as follows. Chapter 2 defines much of the notation used in this book. Chapter 3 is a refresher on probability theory and the material covered is elementary. Chapter 4 covers basic concepts related to discrete-event simulation. Chapter 5 is meant to present an overview of optimization with simulation.

Chapter 6 deals with the response surface methodology, which is used in conjunction with simulation optimization. This chapter also presents the topic of neural networks in some detail.

Chapter 7 covers the main techniques for parametric optimization with simulation. Chapter 8 discusses the classical theory of stochastic dynamic programming. Chapter 9 focuses on reinforcement learning. Chapter 10 covers automata theory in the context of solving Markov and Semi-Markov decision problems.

Chapter 11 deals with some fundamental concepts from mathematical analysis. These concepts will be needed for understanding the subsequent chapters on convergence. Convergence issues related to parametric optimization methods are presented in Chapter 12, while Chapter 13 discusses the convergence theory of stochastic dynamic programming and reinforcement learning.

Chapter 14 presents an overview of some case studies of simulation optimization from the existing literature. Chapter 15 contains some computer programs (codes) for some of the algorithms and case studies described in this book. Chapter 16 concludes this book.

Chapter 2

NOTATION

1. Chapter Overview

This chapter defines most of the notation used in this book. The discussion is in general terms and refers to some conventions that we have followed in this book. Vector notation has been avoided as much as possible; although it is more compact and elegant in comparison to component notation, we believe that component notation, in which all quantities are scalar, is usually easier to understand.

2. Some Basic Conventions

The symbol $\forall i$ will denote: for all possible values of i . The notation

$$\sum_i p(i)$$

will denote a *summation* over all values of i , while

$$\prod_i p(i)$$

will denote a *product* over all values of i .

$E[\cdot]$ will denote the expectation of the quantity in the square brackets. Also, in almost all places, $\{\dots\}$ will denote a set of the elements inside the curly braces.

3. Vector notation

In this book, a vector quantity will always have an arrow (\rightarrow) above it. This convention distinguishes between a scalar and a vector. From our experience, not making this distinction can create a great deal of confusion to the beginner.

Hence, for example, \vec{x} will denote a vector whereas x will denote a scalar. The i th component of a vector \vec{x} will be denoted by $x(i)$. A vector will also be denoted, at several places in the book, with the following notation. For example,

$$(x(1), x(2), x(3))$$

will denote a vector with three elements. Some other examples are:

$$(1, 4, 6) \text{ and } (3, 7).$$

Now, (a, b) may also denote the open interval between scalars a and b . However, if that is the case, and (a, b) is not a vector with 2 elements, we will make that clear. A closed interval will be denoted by:

$$[a, b].$$

The notation $\|\vec{x}\|$ will denote a **norm** of the vector \vec{x} .

3.1. Max norm

The notation $\|\vec{x}\|_\infty$ will denote the **max** norm of the vector \vec{x} . It is defined as follows:

$$\|\vec{x}\|_\infty = \max_i x(i).$$

The max norm is also called the **sup** norm or the **infinity** norm.

3.2. Euclidean norm

The notation $\|\vec{x}\|_E$ will denote the **Euclidean** norm of the vector \vec{x} . It is defined as follows:

$$\|\vec{x}\|_E = \sqrt{\sum_i [x(i)]^2}.$$

4. Notation for matrices

A matrix will be printed in **boldface**. For example:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 7 \\ 4 & 2 & 1 \end{bmatrix}. \quad (2.1)$$

The transpose of a matrix \mathbf{A} will be denoted by \mathbf{A}^T . Thus, using the definition given in Equation (2.1),

$$\mathbf{A}^T = \begin{bmatrix} 2 & 4 \\ 3 & 2 \\ 7 & 1 \end{bmatrix}.$$

\mathbf{I} will denote the identity matrix .

5. Notation for n -tuples

The notation that we have used for n -tuples is distinct from that used for vectors. For instance \hat{x} will denote an n -tuple. An n -tuple may or may not be a vector. An n -tuple may also be denoted with the following notation:

$$(a_1, a_2, \dots, a_n).$$

6. Notation for sets

Calligraphic letters, such as $\mathcal{A}, \mathcal{B}, \mathcal{C}, \dots, \mathcal{X}$, will *always* denote sets in this book. The notation $|\mathcal{A}|$ will denote the **cardinality** of the set \mathcal{A} (that is the number of elements in the set), but $|a|$ will denote the **absolute value** of the scalar a . Sets will also be denoted by using curly braces in the following form:

$$\{1, 2, 3, 4\} \text{ and } \{1, 3, 5, 7, \dots, \}.$$

In the above, both representations are sets.

The notation L^2 may denote L raised to the second power if L is a scalar quantity; it may denote something very different if L is a transformation operator. The actual meaning will be made clear in the context. Also, the superscript has been used in a number of places to denote the iteration in a given algorithm. In such cases, also, it does not represent the power. In other words, in this book, the superscript should not be assumed to be the power unless it is stated to be a power.

The notation, n th, will denote n^{th} . For instance, if $n = 5$ then n th will denote 5^{th} .

7. Notation for Sequences

The notation $\{a^n\}_{n=1}^{\infty}$ will represent an infinite sequence whose n th term is a^n . For instance, if a sequence is defined as follows:

$$\{a^n\}_{n=1}^{\infty} = \{10, 20, 30, \dots\},$$

then $a^3 = 30$.

8. Notation for Transformations

A transformation is an operation carried out on a vector. The result of a transformation is a vector or a scalar. An example of a transformation (or transform, as it is sometimes referred to,) is:

$$x^{k+1} \leftarrow 3x^k + 2y^k, \quad y^{k+1} \leftarrow 4x^k + 6y^k. \quad (2.2)$$

Suppose a vector starts out at $(x^0, y^0) = (5, 10)$. When the transformation defined in (2.2) is applied on this vector, we have:

$$x^1 = 35, \text{ and } y^1 = 80.$$

The equation in (2.2) is often abbreviated as:

$$(x^{k+1}, y^{k+1}) = T((x^k, y^k)),$$

where T denotes the transform. T is understood as T^1 . Two applications of T is written as T^2 , and so on. For example:

$$(x^2, y^2) = T^2((x^0, y^0)),$$

and

$$(x^5, y^5) = T((x^4, y^4)) = T^2((x^3, y^3)).$$

9. Max, min, and arg max

The following notation will be encountered frequently.

$$x = \max_{i \in S}[a(i)].$$

This means that x equals the *maximum* of the values that a can assume. So if set S is defined as below:

$$S = \{1, 2, 3\},$$

and

$$a(1) = 1, a(2) = 10, \text{ and } a(3) = -2,$$

then $x = 10$. Similarly, \min will be used in the context of the *minimum* value. Now read the following notation carefully.

$$y = \arg \max_{i \in S}[a(i)].$$

Here, y denotes the *argument* or the *element index* associated with the maximum value. So, if set S and the values of a are defined as above, then

$$y = 2.$$

so that $a(y)$ is the maximum value for a . It is to be noted that $\arg \min$ has a similar meaning in the context of the minimum.

10. Acronyms and Abbreviations

In Table 10 (see page 13), we present a list of acronyms and abbreviations that we have used in this book.

11. Concluding Remarks

It is important to pay attention to the notation defined in this chapter. It will be needed throughout the book; in particular for the chapters related to convergence, a great deal of mathematical notation is necessary.

Table 2.1. A List of acronyms and abbreviations used in the book

<i>Acronym / Abbreviation</i>	<i>Full name</i>
AGV	Automated (Automatic) Guided Vehicle
Backprop	Backpropagation
cdf	cumulative distribution function
CTMDP	Continuous Time Markov Decision Problem
DMP	Decision-Making Process
DP	Dynamic Programming
DTMDP	Deterministic Time Markov Decision Problem
e.g.	for example
etc.	etcetera
EMSR	Expected Marginal Seat Revenue
i.e.	that is
LAST	Learning Automata Search Technique
LP	Linear Programming
MATLAB	A software
MCAT	Markov Chain Automata Theory
MDP	Markov Decision Problem
NP	Natural Process
pdf	probability density function
pmf	probability mass function
PDN	Parameter Dependence Network
POMDP	Partially Observable Markov Decision Problem
<i>Q</i> -Learning	Learning involving <i>Q</i> factors
<i>Q</i> - <i>P</i> -Learning	Learning involving <i>Q</i> and <i>P</i> factors
Relaxed-SMART	Relaxed version of SMART (see below for SMART)
RL	Reinforcement Learning
RSM	Response Surface Method (Methodology)
SMART	Semi-Markov Average Reward Technique
SMDP	Semi-Markov Decision Problem
TD	Temporal Difference(s)
TPM	Transition Probability Matrix
TRM	Transition Reward Matrix
TTM	Transition Time Matrix
URS	Unrestricted in Sign
WH	Widrow Hoff

Chapter 3

PROBABILITY THEORY: A REFRESHER

1. Overview of this chapter

To understand the fundamentals underlying computer simulation, one must have some knowledge of the theory of probability. The goal of this chapter is to introduce some basic notions related to this theory. We will discuss the following concepts: random variables, probability of an event, some basic laws of probability, probability distributions, the mean and variance of random variables, and some “limit” theorems. The discussion here is at a very elementary level. If you are familiar with these concepts, you may skip this chapter.

1.1. Random variables

Before defining a *random* variable, let us understand what kind of a variable is *not* random. A quantity whose value never changes is the *opposite* of what is meant by a random variable and is known as a **deterministic** quantity. For example, consider the number of suns in our solar system. This quantity is always equal to 1, and is therefore a deterministic quantity and not a random one. On the other hand, consider the number of telephone calls you receive in any given day. This quantity usually takes a different value every day and is a nice example of a random variable. A **random** variable may hence be defined, loosely speaking, as a quantity that is capable of taking different values. Some other examples of random variables are: the amount of time you spend in bed every day after waking up, the number of car accidents on Interstate-25 in one day, the number of American tourists in Hawaii in one day, and the amount of electricity consumed by the city of Pueblo, Colorado in one year.

2. Laws of Probability

We will next define the concept of **probability**. To define probability, we must have a clear idea of what is meant, in the context of the theory of probability, by an ‘experiment.’ Let us consider some examples. Taking a test is an experiment. Rolling a die is also an experiment. So is playing a tennis match or tossing a coin into air.

An experiment, always, has an outcome. We are interested, here, in experiments that have *random* outcomes. In other words, we are interested in those experiments whose outcomes are random variables that can take multiple values.

If you take a test, you either pass it or fail it. When you roll a 6-sided die, it either turns up a 1, 2, 3, 4, 5, or 6. When you play a tennis match, you either win or lose. Let us denote the outcome of an experiment by X which in case of the test-taking experiment takes two values from the set — $\{\text{Pass}, \text{Fail}\}$. In case of the tennis match example, X equals either *win* or *lose*. In case of rolling the die X can take 6 values — any one from the set $\{1, 2, 3, 4, 5, \text{ and } 6\}$. Similarly if you toss a coin, the outcome is either *Heads* or *Tails*.

Probability of an event (such as an outcome of the experiment) is the **likelihood** (or chance) of the occurrence of the event. An event in the die-rolling experiment could be that the die turns up a 3. An event could also be more complex than this. An example of a more complex event is the die turning up an even value. In this experiment, an even value would mean 2, 4, or 6.

Regardless of the nature of the experiment, in order to determine the probability of an event, one must make several trials of the related experiment.

So if we conduct m trials of an experiment, and an event A occurs n out of m times in that particular experiment, the probability of the event A is mathematically defined as:

$$P(A) = \lim_{m \rightarrow \infty} \frac{n}{m}. \quad (3.1)$$

According to this definition, to obtain the probability of an event, we must perform infinitely many trials. In practice, this implies that we must perform a large number of trials to obtain the probability of an event. So do the following experiment at home. Take a fair coin, and toss it in air a large number of times to obtain the probability of the event of the coin turning up *Heads*. The longer you repeat the experiment, the closer will your estimate be to the theoretical value of 0.5. The probability of the coin showing *Tails* is also 0.5.

The probability that the die rolls up 3 is theoretically 1/6 (if it’s a fair die). The probability that the die turns up 2 is also 1/6. It is clear that the sum of the probabilities of all events in any given experiment is 1. This should also be clear from Equation (3.1). In addition, Equation (3.1) tells us that the probability of

an event is a number between 0 and 1. It cannot exceed 1 or be less than 0. Consider the following examples:

Example 1 An example of an event whose probability is 0: a 6-sided die, whose sides are numbered 1 through 6, rolls up a 9.

Example 2 An event whose probability is 1: a 6-sided die, whose sides are numbered 1 through 6, rolls up on a number (integer) between 1 and 6 (both inclusive).

What is the probability that a 6-sided die turns up an even value? The probability should equal the sum of the probabilities of the die turning up 2, 4, and 6, which is equal to $(1/6 + 1/6 + 1/6) = 1/2$. This brings us to an important rule in probability theory, namely, the addition law of probabilities.

2.1. Addition Law

Two events are said to be **mutually exclusive** if the occurrence of one rules out the possibility of occurrence of the other *in the same trial*. The two cases that follow can serve as examples.

Example 1 A coin turning up *Heads* and the same coin turning up *Tails* in the same toss. These two events are mutually exclusive.

Example 2 A die turning up a 1 and the same die turning up a 5 in the same roll. These two events are also mutually exclusive. A **compound** event is an event made up of mutually exclusive events. To find the probability of a compound event, E , which is made up two mutually exclusive events called A and B , the following relation can be used:

$$P(E) = P(A) + P(B). \quad (3.2)$$

From a sum of two events, this relation can be generalized to a sum of n events. The probability of the compound event, E , composed of n mutually exclusive events, A_1, A_2, \dots, A_n , can be expressed as:

$$P(E) = P(A_1) + P(A_2) + \dots + P(A_n). \quad (3.3)$$

Notice that in the previous section, we had actually used this law to calculate the probability of a die turning up an even number. The probability of this event is the sum of the probabilities of three mutually exclusive events, which are: the probability that the die turns up a 2, the probability that the die turns up a 4, and the probability that the die turns up a 6. Then using Equation (3.3)

$$P(E) = P(2) + P(4) + P(6) = 1/6 + 1/6 + 1/6 = 1/2.$$

A **complex** event (unlike the compound event discussed above) is composed of events that are *not* mutually exclusive.

Example 3 A six-sided die turns up either a prime number *or* an even number. Let us define A to be the set of prime number outcomes in the die-rolling experiment and B be the set of even number outcomes in the same experiment. Thus $A = \{2, 3, 5\}$ and $B = \{2, 4, 6\}$.

As a result of these definitions, A and B are not mutually exclusive since if the die turns up a 2, both A and B occur. Hence this event can be defined by $A \cup B$. Consequently, Equation (3.2) will not work here for calculating the probability of the complex event defined in Example 3. To find the relation for the probability of a complex event, we need to use some ideas from the theory of sets. In Example 3, we are interested in finding $P(A \cup B)$.

Let $n(S)$ denote the number of times the event S occurs, if we do m trials. Now we know from elementary set theory that

$$n(A \cup B) = n(A) + n(B) - n(A \cap B). \quad (3.4)$$

Dividing both sides of Equation (3.4) by the number of trials, m , and then taking the limit with m tending to infinity, we have:

$$\lim_{m \rightarrow \infty} \frac{n(A \cup B)}{m} = \lim_{m \rightarrow \infty} \frac{n(A)}{m} + \lim_{m \rightarrow \infty} \frac{n(B)}{m} - \lim_{m \rightarrow \infty} \frac{n(A \cap B)}{m}, \quad (3.5)$$

which using the definition of probability (Equation (3.1)) becomes

$$P(A \cup B) = P(A) + P(B) - P(A \cap B). \quad (3.6)$$

Hence, in Example 3,

$$P(A) = P(2) + P(3) + P(5) = 1/6 + 1/6 + 1/6 = 1/2$$

and

$$P(B) = P(2) + P(4) + P(6) = 1/6 + 1/6 + 1/6 = 1/2.$$

Now $P(A \cup B) = P(2) = 1/6$. Then using Equation (3.6)

$$P(A \hat{\cup} B) = 1/2 + 1/2 - 1/6 = 5/6.$$

This should be intuitively obvious since the event $A \hat{\cup} B$ denotes the event that the die turns up a 2, 3, 4, 5, or 6, the probability of which should equal $P(2) + P(3) + P(4) + P(5) + P(6) = (5)(1/6) = 5/6$.

2.2. Multiplication Law

Sometimes, we have to compute probabilities in complicated situations. Consider the following example.

Example 4 An unfair coin is tossed twice. The probability of obtaining *Heads* with this coin is 0.3 and that of obtaining *Tails* is 0.7. If A denotes the event of

obtaining *Heads* in the first toss, and B that of obtaining *Tails* in the second, then what is the probability that A and B will occur in successive tosses?

Note that A and B are events that occur at different times. Under such situations, we say that the events are **independent of each other**. In other words, if the probability of the occurrence of one event is in no way dependent on that of the other, we say the two events are independent. Similarly, when the probability of one event depends on that of the other, we say the events are **dependent**. In the example above, what happens in the first toss has nothing to do with what happens in the next, and hence it makes sense to treat events A and B as “independent.” Under such circumstances, the event E in which the first toss results in *Heads* and the second in *Tails* is $A \cap B$ is $A \cap B$. If A and B are independent, we can write :

$$P(A \cap B) = P(A) \cdot P(B). \quad (3.7)$$

This is known as the multiplication law of probability. Then, for the event in Example 4, $P(E) = (0.3)(0.7) = 0.21$.

The events A and B , as described in the event above, are independent but not mutually exclusive. Why? Because the occurrence of A does not rule out the occurrence of B . *This does not mean that events that are not mutually exclusive are necessarily independent.* However, Equation (3.6) always holds.

Thus far we have seen scenarios in which we have:

1. two events (A and B) that are *mutually exclusive*, and hence the intersection $P(A \cap B) = 0$. The question of dependence or independence does not arise here.
2. two events that are *not* mutually exclusive, but they are *independent*, that is, $P(A \cap B) = P(A) \cdot P(B)$.

Now, let us examine an even more complicated scenario, where the two events are neither mutually exclusive nor are they independent.

Example 5 Suppose a medical analyst has computed the probability of acquiring a certain type cancer from a certain geographical location to be 0.2. She has also computed the probability that a human being from the same area is exposed to radiation from a nearby factory to be 0.4. In addition, she has discovered that exposure to radiation and cancer are both seen in 14 percent of the people.

Let us denote by A the event of a person suffering from the cancer in question and by B the event that a person is exposed to radiation. Here, we cannot assume that A and B are independent because of their nature. Also, obviously they are not mutually exclusive. Our objective here is to develop an expression for the so-called **conditional probability**. For instance, we may be interested in computing the probability that a person develops cancer, given that the person is exposed to radiation. Here to compute the required probability, we must first perform an infinite (large) number of trials (i.e., collect data). So if

- $m (= 10000)$ denotes the number of pieces of data collected,
- $n(B) (= 4000)$ denotes the number of persons exposed to radiation, and
- $n(A \cap B) (= 1400)$ denotes the number of persons who carry both attributes,

then, *as m tends to infinity* (assuming that 10000 approximates infinity), the probability that a person exposed to radiation develops cancer should be

$$\frac{n(A \cap B)}{n(B)} \quad (3.8)$$

(which is $1400/4000$ and not $1400/10000$). This probability is denoted by $P(A|B)$ (read as probability of A given B, i.e., the probability of A if B is true) and may be also written as :

$$P(A|B) = \frac{n(A \cap B)}{n(B)}. \quad (3.9)$$

Therefore,

$$\frac{n(A \cap B)}{n(B)} = \frac{\frac{n(A \cap B)}{m}}{\frac{n(B)}{m}},$$

and hence

$$\frac{n(A \cap B)}{n(B)} = \lim_{m \rightarrow \infty} \frac{n(A \cap B)}{m} / \lim_{m \rightarrow \infty} \frac{n(B)}{m}. \quad (3.10)$$

But by the definition of probability,

$$\lim_{m \rightarrow \infty} \frac{n(A \cap B)}{m} / \lim_{m \rightarrow \infty} \frac{n(B)}{m} = \frac{P(A \cap B)}{P(B)}. \quad (3.11)$$

Hence combining Equations (3.9), (3.10), and (3.11), we have that:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \quad (3.12)$$

which is an important relation that we will use frequently. Similarly,

$$P(B|A) = \frac{P(B \cap A)}{P(A)}.$$

Of course, $P(A \cap B) = P(B \cap A)$ (from the definition of the intersection of two sets).

It may be noted that if A and B are independent,

$$P(A \cap B) = P(A) \cdot P(B),$$

which would imply, using Equation (3.12), that

$$P(A|B) = \frac{P(A) \cdot P(B)}{P(B)} = P(A),$$

which is as it should be since A and B are independent, and therefore $P(A|B)$ should mean the same thing as $P(A)$.

3. Probability Distributions

We have said above that a random variable assumes multiple values. Now, a random variable may assume **discrete** values such as the elements of the set \mathcal{A} :

$$\mathcal{A} = \{1, 4.5, 18, 1969\}$$

or **continuous** values such as the elements of the interval $(1, 5)$. We will first consider the discrete case.

3.1. Discrete random variables

A **discrete** random variable assumes values from a finite or a countably infinite subset of the set: $(-\infty, \infty)$. We have seen that we associate a probability with the event that a random number takes on a given value. For example in the die-rolling experiment, the probability that the outcome X takes on a value of 1 is $1/6$. We can represent this by

$$P(X = 1) = 1/6,$$

which means that the probability that the random variable X assumes a value equal to 1 is $1/6$. Similarly $P(X = 2) = 1/6$, and so on. In general, we use the notation

$$P(X = k) = l$$

to denote that l is the probability of X equaling the value k .

In the equation, $P(X = k) = l$, we can think of P as a function whose “input” is k and “output” is l . This function is called the *probability mass function* and is abbreviated as **pmf**. A more frequently used notation, which does not show the input and the output, is $f(x)$. It denotes the probability that X equals the value x , i.e., $P(X = x)$.

Since the output of the pmf (or rather the pmf itself) is a probability, it cannot assume any arbitrary value. First of all, the pmf must always be between 0 and 1. Secondly, the sum of probabilities must always equal 1. Hence if the random variable X can take 6 different values, then the sum of the probabilities for each of those 6 different values should equal 1 (like in the die-rolling experiment, we had $1/6+1/6+1/6+1/6+1/6+1/6=1$). This must always be true.

Let us consider the following example.

Example 6 X is a random variable that can take on the following values: 2, 4, 7, and 8 with the respective probabilities of $1/4$, $1/8$, $1/5$, and p .

Then $(1/4 + 1/8 + 1/5 + p)$ must equal 1 (and so $p = ?$). We may therefore write $P(X = 2) = 1/4$ and $P(X = 4) = 1/8$ and so on. Notice that $P(X \leq 2)$ is hence $1/4$. Also $P(X \leq 4) = 1/4 + 1/8$, $P(X \leq 7) = 1/4 + 1/8 + 1/5$ and $P(X \leq 8) = 1$.

Notice that $P(X \leq k) = l$ denotes that the probability that the random variable X does not exceed k is equal to l . This function is different from the pmf and is called the cumulative distribution function. It is usually abbreviated as cdf. It should be clear from the way we calculated the cdf that it is possible to obtain the value of the cdf at every point, if the value of the pmf at every point is known. The reverse is also possible, i.e., the pmf can be obtained from the cdf.

Let us assume that we know the cdf in the Example 6. We will now obtain the pmf from our knowledge of the cdf. Notice that:

$$P(X = 8) = P(X \leq 8) - P(X \leq 7) = 1 - (1/4 + 1/8 + 1/5).$$

Similarly,

$$P(X = 7) = P(X \leq 7) - P(X \leq 6) = (1/4 + 1/8 + 1/5) - (1/4 + 1/8),$$

$$P(X = 4) = P(X \leq 4) - P(X \leq 2) = (1/4 + 1/8) - (1/4) = 1/8,$$

and

$$P(X = 2) = P(X \leq 2) = 1/4.$$

The cdf is also represented as $F(X = k)$ or $F(X)$. The calculations shown above can be generalized to obtain the following formula:

$$F(X = k) = \sum_{x=a}^k f(x). \quad (3.13)$$

This means that the cdf at any point $X = k$ is the sum of the values of the pmf at its (i.e. the pmf's) possible values, starting from its lowest possible value, a , to k .

It should be clear now that both pmf and cdf in the discrete case actually denote probabilities, and that knowledge of one is sufficient to obtain the other. Let us now next discuss the continuous random variable.

3.2. Continuous random variables

Time between successive telephone calls is an example of a continuous random variable. Unlike the discrete random variable, this random variable can assume infinitely many values. For example, a continuous random variable may assume values between 0 and 12, or from 5 to infinity, etc.

The quantity corresponding to the pmf of the discrete case is called the probability *density* function (pdf). It is denoted, like in the discrete case, by $f(x)$. The cdf is defined for the continuous case — also in a manner similar to that for the discrete case. The difference with the discrete case, in calculating the cdf, is that the summation value in Equation (3.13) will have to be replaced by an integral. As a result, the corresponding equation is:

$$F(X = k) = \int_a^k f(x)dx, \quad (3.14)$$

where $f(x) = 0$ for $x < a$. Here X denotes the continuous random variable.

Just as the sum of the probabilities in the discrete case has to be 1, the same is true for the continuous case. Therefore, if a denotes the lowest value the random variable X can assume (i.e., $f(x) = 0$ for $x < a$) and b denotes the highest value that the random variable can assume (i.e., $f(x) = 0$ for $x > b$), then the following condition has to be satisfied by the pdf and the cdf:

$$F(X = b) = \int_a^b f(x)dx = 1. \quad (3.15)$$

The condition in Equation (3.15) places a restriction on the pdf (and the cdf) in the sense that any **arbitrary** function may not qualify to be called a pdf (or a cdf). Let us explain, next, what we mean by this.

For a random variable that assumes values between 1 and 2, the function defined by

$$g(x) = \frac{x^2}{3}$$

cannot serve as a valid pdf. Here is why. Remember, the condition given in Equation (3.15) must be satisfied by every pdf. But:

$$F(X = 2) = \int_1^2 g(x)dx = \left[\frac{x^3}{9} \right]_1^2 \neq 1.$$

In the continuous case, to obtain the pdf from the cdf, the operation required should be differentiation, since integration was performed to obtain the cdf from the pdf. The necessary mechanism is shown by the following formula:

$$f(x) = \frac{dF(x)}{dx}. \quad (3.16)$$

4. Expected value of a random variable

We like to obtain a deterministic characterization of any random variable, because 1) we are comfortable with deterministic characterizations of variables

and 2) because such a characterization is very useful in analyzing processes that contain random variables.

Several parameters have been developed by statisticians to help us make such characterizations. Each parameter has its own limitations and applies to a different aspect of the randomness present in a random variable. Examples of such parameters are the *mean* and the *standard deviation*. We will take a close look at these parameters, next.

Let us first examine the **mean** value of the random variable. This also goes by other names such as *average* or *expected value*. The mean (as the name *average* suggests) is the average value of the random variable. To find the mean of a random variable, one must perform infinitely many trials, then sum the values obtained in each trial, and then divide the sum by the number of trials (m) performed. As m tends to infinity, we get the right estimate of the mean. In terms of the pmf, the mean of a discrete random variable, X , can be defined as:

$$E(X) = \sum_x xf(x). \quad (3.17)$$

For the continuous case, the definition, in terms of the pdf, is:

$$E(X) = \int_x xf(x)dx. \quad (3.18)$$

It is not hard to understand why these two relations make sense intuitively. Imagine that we conduct 10 trials for obtaining the mean and the pmf of a random variable, and that the number 10 can be treated as infinity for our situation. And the results are:

$$2, 4, 2, 6, 3, 4, 6, 6, 3, 2.$$

Now one way to calculate the mean of this sample (that we treat as a representative of the whole population) is to add the values and divide the sum by the number of trials. Hence the mean is

$$E(X) = (2 + 4 + 2 + 6 + 3 + 4 + 6 + 6 + 3 + 2)/10 = 3.8.$$

From our experiments, we know that the random variable can take one of the following values: 2, 3, 4 and 6. Now what is the probability (based on our experiments), that the random variable X equals the value 2? Out of the ten trials, X assumed the value of 2 in three trials. And hence $P(X = 2) = 3/10$ (assuming that 10 is close to infinity). Similarly, $P(X = 3) = 2/10$, $P(X = 4) = 2/10$, and $P(X = 6) = 3/10$. Using these values, it is now easy to see why the definition of the expected value of a random variable, as given in Equation (3.17), is true. You can verify that the value, 3.8, of the mean which we obtained from our simple calculations in the paragraph above, is indeed

what we obtain if we use the formula in Equation (3.17).

$$(2)(3/10) + (3)(2/10) + (4)(2/10) + (6)(3/10) = 3.8.$$

The above discussion not only verifies the equations above, but also throws light on what the pdf means, intuitively, for the discrete case. It demonstrates that the pdf at any point k is the probability that the random variable assumes the value of k . Also notice that: $P(X = 6) = P(X \leq 6) - P(X \leq 4) = F(6) - F(4)$. This shows that the cdf may be used directly to obtain the probability of the random variable taking on a particular value.

The situation changes slightly in the case of continuous random variables. In the continuous case, when it comes to finding the probability at a particular point, *using the cdf* we have two options:

1. We can differentiate the cdf *analytically* as shown below

$$f(x) = \frac{dF(x)}{dx}$$

to obtain the pdf, and then use the pdf directly ($P(X = k) = f(k)$).

2. We can differentiate the cdf *numerically*. Select a small value for h , and then use the following formula:

$$P(X \rightarrow k) \approx \frac{F(k+h) - F(k-h)}{2h}. \quad (3.19)$$

The above it must be understood is an approximation that improves as p is made smaller. (Why does this work? The reason is: as p tends to 0, we have the derivative at that point.) One may have to use this formula to find the pdf from the cdf in the normal distribution tables.

So in any case, our method hinges on the *differentiation* process.

5. Standard deviation of a random variable

The mean of a random variable does not tell us everything about the random variable. If it did, the random variable would cease to be a random variable and would be a deterministic quantity instead.

Together, the mean and the *standard deviation* tell us quite a lot about the nature of the random variable, although they do not tell us everything. All the information is contained in the pdf (or the cdf).

The standard deviation is an estimate of the *average deviation* of the variable from its mean. So let us assume that the random variable assumes the following values:

$$8, 11, 4, \text{ and } 5.$$

The mean of these numbers is 7. Now, we can calculate the deviation of each random variable from the mean. The deviations are:

$$(8 - 7), (11 - 7), (4 - 7) \text{ and } (5 - 7),$$

i.e.,

$$1, 4, -3 \text{ and } -2.$$

Now if one were to calculate the average of these four quantities, it would turn out to be 0. (This can be called the “mean” deviation).

One way out of this difficulty is to square the deviations and then average the squared quantities. The squared deviations would be :

$$1, 16, 9, \text{ and } 4.$$

Now, even though we have four quantities here, in reality, we have only three independent quantities since we can take any three of the deviations and obtain the fourth. ($-2 = -[(1) + (-3) + (4)]$ and $-3 = -[(1) + (-2) + (4)]$ and so on).

If the squared deviations are summed, and the sum is divided by three (the number of independent quantities), then the square root of the quotient (since the deviations were squared) should be an estimate of the average deviation. So in the example that we are considering, an estimate of the average deviation would be

$$\sqrt{\frac{(1)^2 + (4)^2 + (-3)^2 + (-2)^2}{3}} = 5.47.$$

The estimate that we have just calculated is also known as the **standard deviation**. It is defined as follows:

$$\sigma(X) = \sqrt{\frac{\sum_{i=1}^n (x_i - a)^2}{n - 1}}, \quad (3.20)$$

where X is the random variable that takes n values and $\sigma(X)$ denotes the standard deviation of X . The square of the standard deviation is also called the **variance** of the random variable.

The reason for dividing the sum of the squared deviations by $(n - 1)$ and not by n has been demonstrated above by the numerical example.

Let us consider the following scenario. We have access to *the information about the mean of a random variable* (we know it to be m), and we have k samples of the random variables. An estimate of the standard deviation may be obtained as follows:

$$\sigma^*(X) = \sqrt{\frac{\sum_{i=1}^k (x_i - m)^2}{k}}. \quad (3.21)$$

It should be clear that since the individual deviations here do not sum to 0, we actually have k independent quantities. Hence it is not necessary to subtract 1 from the number of samples.

Recall that the mean of a random variable can be obtained from its cdf or its pdf (pmf). The same holds true for the variance (and hence for the standard deviation) as well. In terms of its pmf, the variance of a discrete random variable, X , is given as:

$$\sigma^2(X) = \sum_i f(x)(x_i - a)^2 \quad (3.22)$$

where a is the mean. For the continuous case, the variance is given by:

$$\sigma^2(X) = \int_{-\infty}^{\infty} f(x)(x - a)^2 dx, \quad (3.23)$$

where, as before, a denotes the mean of the random variable.

6. Limit Theorems

We present two important theorems from statistics in this last section. We do not present their proofs; however their statements need to be understood.

THEOREM 3.1 (The Strong Law of Large Numbers) *Let X_1, X_2, \dots be a sequence (a set with infinitely many elements) of independent random variables having a common distribution with mean $E(X) = \mu$. Then, with probability 1,*

$$\lim_{n \rightarrow \infty} \frac{X_1 + X_2 + X_3 + \dots + X_n}{n} = \mu.$$

This is an important theorem. We will use it in simulation-based evaluation of parameters and in several other places.

THEOREM 3.2 (Central Limit Theorem) *Let X_1, X_2, \dots be a sequence (a set with infinitely many elements) of independent random variables having a common distribution with mean μ and variance σ^2 . Let us define Y_n as follows:*

$$Y_n = \frac{X_1 + X_2 + \dots + X_n}{n}.$$

As n becomes large (that is, $n \rightarrow \infty$), the distribution of Y_n becomes (asymptotically) normal with mean μ and variance σ^2/n , regardless of the distribution of X_1, X_2, \dots, X_n .

As a result of this theorem, we can often construct “confidence intervals” on the estimates of parameters.

7. Review Questions

1. If you have a telephone machine at home that records the number of calls you received while you were gone, do the following experiment. Keep track of this number for a week. Is this number a constant or a random variable? Can you identify its distribution (perhaps with data over one month)?
2. Explain why in computing the variance of a sample, we use $(n - 1)$ in the formula and not n . Is there any scenario in which we should use n ?
3. When we compute the variance, we *square* the deviations. What is the difficulty with adding the deviations without squaring them first?
4. Find the value of the cdf at $x = 1$, if the pdf is given by:

$$f(x) = kx^2 \exp(-x^3), x \geq 0.$$

The random variable only assumes non-negative values.

5. "The pdf of a continuous random variable is always 0." Comment.

6. The cdf of a function is given as

$$F(x) = 1 - \exp(-5x) \quad x \geq 0.$$

Calculate the pdf at the following points, *without analytical differentiation*.

$$x = 0.1, x = 1, \text{ and } x = 10.$$

Chapter 4

BASIC CONCEPTS UNDERLYING SIMULATION

True randomness can be found in radioactive decay; inside computers we must do with artificial random numbers

1. Chapter Overview

This chapter has been written to introduce the topic of *discrete-event* simulation. To comprehend the material presented in this chapter, some background in the theory of probability is needed. Alternatively, the reader should read Chapter 3). The two major topics covered in this chapter are random number generation and simulation modeling of random systems.

2. Introduction

According to one definition [102], a system is a collection of entities that interact with each other. An example of a system is the queue that forms in front of a bank-teller (See Figure 4.1). The entities in this system include the people who come to get service and the teller who provides service.

The *behavior* of a system can usually be described in terms of a number of *traits* of the system. What is frequently of interest to us is how a system changes over time, that is, how these traits change as time passes.

A more commonly used word for ‘trait’ is *state*. In our queuing system, the state could very well be defined by the length of the queue, that is, the number of people in the queue. Generally, in a real banking queue, the queue length fluctuates, and as such we may conclude that the behavior of the system changes with time. Systems that change with time are called *dynamic* systems.

The behavior of any dynamic system is usually *governed* by some *variables*. Such variables are often called the **governing variables** of a system. The

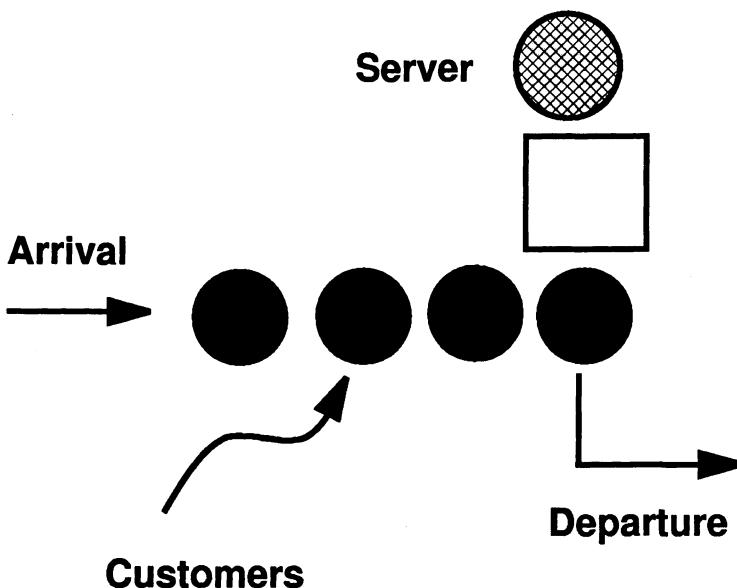


Figure 4.1. A Schematic of a queue that forms in a bank or supermarket.

governing variables in our banking system are the time taken by the teller in giving service to a customer and the time between the arrival of successive customers to the bank. An appropriate question at this point is:

Why do we consider these two quantities to be the governing variables of the queuing system?

The answer is: it can be shown easily, using laws from queuing theory, that the behavior of the system (an indicator of which is the length of the queue) depends on the values assumed by these two variables.

When the governing variables are random variables, the system is referred to as a *random system* or a *stochastic system*. Therefore the reader should always keep in mind that the behavior of a random system is governed by one or more *random variables*. Knowledge of the distributions of these random variables helps us analyze the behavior of the system.

3. Models

To understand, to analyze, and to predict the behavior of systems (both random and deterministic), operations researchers construct **models**. These models are abstract models unlike say a miniature airplane, which is a physical model that one can touch. Abstract models may take the form of equations, functions, inequations, and computer programs etc.

To understand how useful abstract models can be, let us consider a familiar model from Newtonian physics:

$$v = u + gt.$$

This equation gives us the speed of a freely-falling body that has been in the air for t time units after starting its descent at a speed of u . Here $g = 9.8 \text{ m/s}^2$. This mathematical model gives us the ability to *predict* the speed of the body at any time in its descent.

The literature in stochastic operations research is full of similar mathematical models that help us analyze and predict the behavior of stochastic systems. In fact, stochastic operations research can be described as the **physics** of stochastic systems. Queuing theory, renewal theory, and Brownian motion theory (all three have several overlapping areas) have been exploited by researchers to construct powerful mathematical models.

Although these mathematical models enjoy an important place in operations research, they are often tied to the assumptions made about the system while constructing the models. These assumptions may be related to the structure of the system or to the distributions of the governing random variables. For instance, many models in queuing theory are limited in use to exponential service and inter-arrival times. Some models related to Brownian motion are restricted to the so-called "heavy traffic" conditions. As such, there is a need for models that generalize beyond such narrow assumptions.

For generating more powerful models that can analyze and predict the behavior of large and complex systems, one may use a computer program that mimics the behavior of the system. The program usually achieves this by generating random numbers for the governing random variables. Such a program is also called a *simulation model*.

For a long time, simulation models did not quite receive the respect that they deserve. The primary reason for this was that although these models could be used for analysis of the behavior of random systems, their use in **optimizing** systems was not well understood. On the other hand, a major strength of mathematical models is that, among other things such as elegance, they can be used for optimizing systems using mathematical programming. Fortunately, things have changed and simulation models too, can now be used in optimizing systems. Furthermore, more importantly, they can be used to optimize *complex, large-scale, stochastic systems*, where it is difficult to construct mathematical models.

Before embarking on the journey towards understanding computer simulation, let us turn our attention to a fundamental question:

Why do we study stochastic systems?

The answer is

1. many real-life systems are random systems and
2. by studying them, we can often make changes in them to run them better.

In this book, we will be keenly interested in being able to control the stochastic system in a manner such that it runs more efficiently and/or the cost of running the system is reduced (or profits gained from running it are increased).

In what follows, we have made a serious attempt to explain the inner workings of a simulation model. A thorough understanding of this topic is invaluable for understanding several topics in the remainder of this book.

4. Simulation Modeling of Random Systems

Finding out the distributions of the governing random variables is the first step towards modeling a stochastic system, regardless of whether mathematical models or simulation models are used. In mathematical models, the pdfs (or cdfs) of the governing random variables are used in the formulations (formulas if you like). In simulation models, on the other hand, the pdfs (or cdfs) are used to generate random numbers for the variables concerned. These random numbers can be used to imitate, within a computer, the behavior of the system. Imitating the behavior of a system essentially means *re-creating* the events that take place in the real system that is being imitated.

How does one determine the distribution of a random variable? There is usually only one way to do that. One has to actually collect data from the real-life system. Data should be collected on the values that the random variable actually takes on in the real system. Then using this data, it is usually possible to fit a distribution to that data, provided sufficient data is collected. This is also known as distribution fitting. This, we repeat, is invariably the first step in any kind of stochastic analysis, and the importance of careful distribution fitting should never be downplayed. For a good discussion on distribution fitting, the reader is referred to Chapter 6 of Law and Kelton [102]. As stated above, the knowledge (related to the distribution) of the governing random variables plays a crucial role in stochastic analysis.

An important issue in stochastic analysis that we must address is related to the *number* of random variables in the system. It is generally true that the larger the number of governing random variables in a system the more complicated is its analysis (study of its properties). This is especially true of analysis with mathematical models. On the other hand, with the advent of structure in the field of computer programming, *simulating* a system with several governing random variables has become a trivial task.

Real-life random systems are often governed by not one or two but rather several random variables. Naturally, simulation models give us a reason for hope, when faced with complex, large-scale, stochastic systems.

With this motivation, we are now ready to discuss the steps needed in order to *simulate* a real-life system. Our strategy will be to *re-create*, within a computer program, the events that take place in the real-life system. The re-creation of events is based on using suitable values for the governing random variables. For this, one has to have a mechanism for generating values of the governing variables, and if the governing variables are random variables, one has to generate random numbers. In what follows, we will first discuss how to create random numbers and then discuss how to use the random values of the variables to re-create the events that occur in the real-life system.

4.1. Random Number Generation

In this section, we will discuss a random number generation scheme for generating *uniformly distributed* random numbers. This scheme is usually exploited to generate random numbers for other distributions.

4.1.1 Uniformly Distributed Random Numbers

An important element of a simulation model is the scheme used to generate values for the governing random variables in the system. We must make it clear at the outset that the random numbers that we will discuss here are in some sense *artificial*. This is because “true” random numbers cannot be generated by a computer but must be generated by a human brain or must be obtained from a real system. Having said that, we need to make it clear that for all practical purposes, these *artificial* random numbers that our “inferior” computers generate are usually good enough for us. This is an important concern, however, and the reader is strongly encouraged to pursue this idea till any misgivings he or she has about these computer-generated numbers are dispelled from the mind. After all, without an error-free random number generation method, there would be no point in pursuing the topic of simulation-based optimization any further!

Many of the earlier random number generation schemes were unacceptable due to a variety of reasons. However, one of these “unacceptable” schemes, although imperfect, contained a gem of an idea and has now evolved into a good random number generation scheme. We will focus our discussion on this scheme. There are several statistical tests that can be performed on computer-generated random numbers to determine, if the numbers generated are *truly* random (although the shadow of artificialness will always lurk in the background, alas!). For a good discussion on tests for random numbers, the reader is referred to Knuth [95].

Let us consider the so-called **linear congruential generator** of random numbers [126], which is given by the following equation:

$$I_{j+1} \leftarrow (aI_j \bmod m), \quad (4.1)$$

where a and m are positive integers. The notation in Equation (4.1) is to be understood to mean the following. If aI_j is divided by m , the *remainder* obtained is denoted by $(aI_j \bmod m)$. So, if one starts with a positive value, which is less than or equal to m , for I_0 , this scheme will produce integers between 0 and m (both 0 and m excluded).

Let us demonstrate the above-described idea with an example. Let $a = 2$, $m = 20$ and $I_0 = 12$. (By the way, these numbers are used here only for the sake of demonstrating how this scheme works but are not recommended for use in the computer programs for generating random numbers.). The sequence it generated will be:

$$(12, 4, 8, 16, 12, 4, 8, 16, 12, 4, 8, 16, \dots) \quad (4.2)$$

This sequence has integers between 1 and $m - 1 = 19$ (both 1 and 19 included). It cannot have an integer equal to 20 or greater than 20 because each integer is obtained after a division by 20. Then we can conclude that in the sequence defined in (4.2), the terms

$$12, 4, 8, 16$$

form a set that has integers ranging from 0 to 20 (both 0 and 20 excluded).

By a suitable choice of a and m , it is possible to generate a set that contains *all* the different integers in the range from 0 to m . The number of elements in such a set will clearly be $m - 1$. If we divide each integer in this set by m , we will obtain a set of numbers in the interval $(0, 1)$. In general, if the i th integer is denoted by x_i and the i th random number is denoted by y_i , then

$$y_i = x_i/m.$$

Now each number (y_i) in $(0, 1)$, is equally likely, because each associated integer (x_i), between 0 and m , occurs *once* at some point in the original set. Then, for large values of m , this set of random numbers from 0 to 1 that *approximates* a set of natural random numbers from a uniform distribution. Remember that the pdf of the random variable has the same value at every point in the uniform distribution.

Three important remarks related to the linear congruential generator are in order.

Remark 1: The *maximum* number of integers that may be generated in this process before it starts repeating itself is $m - 1$.

Remark 2: If I_0 equals 0, the sequence will only contain zeros.

Remark 3: I_0 is also called the *seed*.

Remark 4: A suitable choice of a and m will yield a set of $m - 1$ numbers where each integer between 0 and m occurs at some point.

An important question that we must address is:

Is it okay to use a sequence of random numbers which has repeating subsets, e.g., (12, 4, 8, 16, 12, 4, 8, 16, . . .)?

The answer to this question is an emphatic no. Here is why. These numbers are not really random. The numbers (12, 4, 8, 16) repeat after every four numbers. They repeat after a *regular* interval and hence are not random. Using numbers like these is like inviting the ghost of artificialness. This is a very serious issue and should not be taken lightheartedly.

Now random numbers from the linear congruential generator that we have discussed above, unfortunately, must repeat after a *finite* period. So the only way out of artificialness is to generate a sequence of a *sufficiently long* period such that we are done with the sequence before it repeats itself. If m is a number like $2^{32} - 1$, then it is possible with a suitable choice of a to generate a sequence with a period of $m - 1$. Thus, if the number of random numbers we need is less than $m - 1 = 2147483646$, (for many applications this will be sufficient) then we have a set of random numbers that have no repetitions. Besides, then we cannot be accused of using artificial random numbers.

This scheme is not used properly in many systems (we mean compilers of C etc), and this results in random numbers that have a small *period*. When such schemes are used, several repetitions occur causing erroneous results. Furthermore, when numbers repeat, they are not *independent*. Repetitions imply that the numbers stray away, considerably, from the uniform distribution.

If the largest number in your computer's memory is $2^{31} - 1$, then a legal value for a that goes with $m = 2^{31} - 1$ is 16807. These values of m and a cannot be implemented naively in the computer program. The reason is easy to see. In the multiplication of a by I_j , where the latter can be of the order of m , we will run into trouble as the product will often be larger than m , the largest number that the computer can store. A clever trick due to Schrage [154] will help us circumvent this difficulty. In the equations below,

$$[x/y]$$

will denote the *integer* part of the quotient obtained after dividing x by y .

Using Schrage's approximate factorization, if

$$Q = a(I_j \bmod q) - r[I_j/q],$$

the random number generation scheme is given by

$$I_{j+1} \leftarrow \begin{cases} Q & \text{if } Q \geq 0 \\ Q + m & \text{otherwise.} \end{cases}$$

In this scheme, q and r are positive numbers. As is clear from this approximate factorization (see the definition of Q), multiplication of a and I_j , which is

necessary in Equation 4.1, is avoided. For $a = 7^5 = 16807$ and $m = 2^{31} - 1$, suggested values for q and r are: $q = 127773$ and $r = 2836$. Other values can be found in Park and Miller [126]. This scheme is not the last word on random number generation. The period of this scheme is $m - 1 = 2^{31} - 2$. When the number of calls to this scheme becomes of the order of the period, it starts repeating numbers and is not recommended. L'Ecuyer [103] has shown how two sequences of different periods can be combined to give a sequence of a longer period. The reader is referred to [103] for further reading. Also, see Press *et al.* [139] for useful codes in C, one of which implements the scheme given in [103].

We must point out that the above-described scheme can be used to generate random numbers for any uniform distribution (a, b) . It is very easy to generate a random number between a and b from a random number between 0 and 1. Here is how. If y denotes the random number between 0 and 1, and x that between a and b , then

$$x = a + (b - a)y.$$

The computer program for the scheme described above is given in Chapter 14.

4.1.2 Other Distributions

In this section, we will discuss how random numbers for distributions other than the uniform distribution may be generated. We will limit our discussion to the “inverse function method.” For other methods, please consult Law and Kelton [102].

Inverse function Method The inverse function method relies on manipulating the cdf of a function. This often requires the cdf to be “nice.” In other words, the cdf should be of a form that can be manipulated. We will show how this method works with the example of the exponential distribution. The cdf of the exponential distribution is given by :

$$F(x) = 1 - e^{-\lambda x}.$$

Note that $\lambda > 0$. It is known that for any given value of x , $F(x)$, the cdf, assumes a value between 0 and 1. Conversely, it may be said that when $F(x)$ assumes a value between 0 and 1, that particular value (of $F(x)$) corresponds to some value of x . Hence to find a random number in this distribution, one should first find a *random* value, say y , for $F(x)$ between 0 and 1 from the uniform distribution $(0,1)$, and then find the value of x that corresponds to y . We, next, show how this works for the exponential distribution.

We generate a random number y from the uniform distribution $(0,1)$, equate it to the cdf, and then solve for x .

$$F(x) = y,$$

that is,

$$1 - e^{-\lambda x} = y.$$

which implies that

$$e^{-\lambda x} = 1 - y.$$

Taking the natural logarithm of both sides, we obtain after some simplification:

$$x = -\frac{\ln(1 - y)}{\lambda}.$$

Thus, we can find the value of x corresponding to y . In fact, by replacing $1 - y$ by y , one may use the following simplified rule:

$$x = -\frac{\ln(y)}{\lambda}.$$

The replacement is legal because y is also a number between 0 and 1. The code for doing the above-described scheme is given in Chapter 14.

After some reflection on the method presented above, you will see that it is called the inverse function method because the cdf is manipulable and we are able to solve for the value of x for a given value of $F(x)$. This is not always possible when the cdf takes a more formidable closed form, which cannot be manipulated. Then one must use other method such as the rejection method [102].

We will next turn our attention to a very important mechanism that lies at the heart of computer simulation. A clear understanding of it is vitally essential.

4.2. Re-creation of events using random numbers

The best way to explain how values for random variables can be used to re-create events is to use an example. We will use the single server queuing example that has been discussed above.

In the single-server queuing example, there are two governing variables:

1. the time between successive arrivals to the system — let us denote it by t_a
2. the time taken by the server to give service to one customer — let us denote it by t_s .

It will be easy to see from the example under consideration, how the values assumed by the two governing variables will help us identify the time of occurrence of the different events. We assume here that both t_a and t_s are random variables.

So let us generate values for the two sequences. Since we know the distributions, it is possible to generate values for them. For example, let the first 7 values for t_a be:

$$10.1, 2.3, 1, 0.9, 3.5, 1.2, 6.4$$

and those for t_s , be:

$$0.1, 3.2, 1.19, 4.9, 1.1, 1.7, 1.5.$$

We need to introduce some notation here. We will use $\{t(n)\}_{n=1}^{\infty}$ to denote a sequence defined by: $\{t(1), t(2), \dots\}$.

A question should rise in your mind, now. If you decide to go and observe the system and collect data for these two sequences from the real life system, will your values be identical to those shown above? The answer is no, not necessarily. So how are we re-enacting the real-life system in our simulation model, with these sequences, then? Well, the elements of the sequence we generated come from the distribution of the random variable in the real-life system, and the distribution, it is assumed, is known to us. However, this argument is not enough, and we will come back to this important point shortly. Let us assume, for the time being, that we have created a sequence for each random variable that could very well be the sequence in the real-life system.

Now, from the two sequences (one for inter-arrival times and the other for service times) one can construct a sequence of happenings or events. Events in this system are of two types and can be described as follows:

1. a customer enters the system (**arrival**)
2. a customer is serviced and leaves the system (**departure**)

Our task of re-creating events boils down to the task of finding the time of occurrence of each event. In the single server queuing system, (see Fig. 4.2) when a customer, who arrives to find that the server is busy giving service to somebody, either becomes the first person in the queue or joins the queue's end. When a customer arrives to find that the server is idle, he or she directly goes to the server without waiting. The arrivals in this case, we will assume, occur regardless of the number of people waiting in the queue.

To analyze the behavior of the system, the first task is to determine the clock time of each event as it occurs. This task, as stated above, lies at the heart of stochastic simulation. If one can accomplish this task, namely, the task of identifying the clock time of each event, various aspects of the system can be analyzed. We will discuss how this task can be performed in the queuing example.

Before delving into its details, let us strive to get a clear understanding of what the elements of the sequences $\{t_a(n)\}_{n=1}^{\infty}$ and $\{t_s(n)\}_{n=1}^{\infty}$ stand for. The first element in each sequence is associated with the first customer who arrives in the system, the second is associated with the second customer to arrive, and so on. Keeping this in mind, note that in the queuing example, there are two types of events: arrivals and departures. The task in each case can be accomplished as described below:

1. For arrivals: The clock time of any arrival in the system will be equal (in this case) to 1) the clock time of the previous arrival plus 2) the inter-arrival time of the current arrival. (Remember the second quantity is easy to find. The inter-arrival time of the k th arrival is $t_a(k)$.)
2. For departures:
 - a. If an arriving customer finds that the server is idle, the next departure will occur at a clock time equal to the arrival clock time of the arriving customer plus the service time of the arriving customer. (Again, the second quantity is easy to find. The service time of the k th arrival is $t_s(k)$.)
 - b. If an arriving customer finds that the server is busy, the next departure will occur at a time equal to 1) the clock time of the *previous* departure plus 2) the service time of the customer *currently being served*.

Now, let us illustrate these ideas with the values generated for the sequence. See Figure 4.2. The first arrival takes place at clock time of 10.1, the second at clock time of $10.1 + 2.3 = 12.4$, the third at clock time of $12.4 + 1 = 13.4$, the fourth at clock time of $13.4 + 0.9 = 14.3$, the fifth at a clock time of $14.3 + 3.5 = 17.8$, the sixth at a clock time of $17.8 + 1.2 = 19.0$, and the seventh at a clock time of $19 + 6.4 = 25.4$.

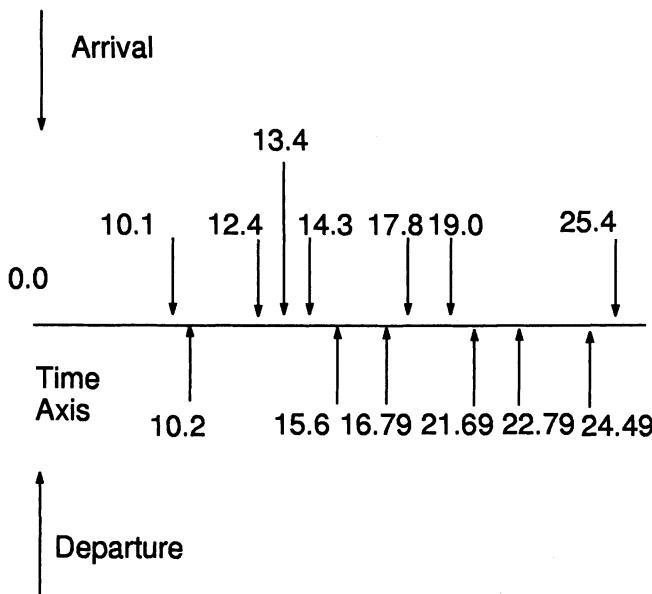


Figure 4.2. The event clock showing arrivals and departures.

When the first arrival occurs, there is nobody in the queue and hence the first departure occurs at a clock time of $10.1 + 0.1$ (service time of the first customer) = 10.2. Now, from the clock time of 10.2 till the clock strikes 12.4, when the second arrival occurs, the server is idle. So when the second arrival occurs, there is nobody in the queue and hence the time of the second departure is $12.4 + 3.2$ (the service time of the second arrival) = 15.6. The third arrival takes place at a clock time of 13.4, but the second customer departs much later at a clock time of 15.6. Hence the second customer must wait in the queue till 15.6 when he/she joins service. Hence the third departure will occur at a clock time of 15.6 plus the service time of the third customer, which is 1.19. Therefore, the departure of the third customer will occur at a clock time of $15.6 + 1.19 = 16.79$. The fourth customer arrives at a clock time of 14.3 but the third departs only at 16.79. It is clear that the fourth customer will depart at a clock time of $16.79 + 4.9 = 21.69$. In this way, we can find that the fifth departure will occur at a clock time of 22.79 and the sixth at a clock time of 24.49. The seventh arrival occurs at a clock time of 25.4, which is after the sixth customer has departed. Hence when the seventh customer enters the system, there is nobody in the Queue, and the seventh customer departs some time after the clock time of 25.4. We will analyze the system till the time when the clock strikes 25.4.

Now, from the sequence of events constructed, we can collect data related to some system parameters of interest to us. Let us begin with server utilization. From our observations, we know that the server was idle from a clock time of 0 till the clock struck 10.1, that is, for a *time interval* of 10.1. Then again it was idle from the time 10.2 (the clock time of the first departure) till the second arrival at a clock time of 12.4, that is, for a time interval of 2.2. Finally, it was idle from the time 24.49 (the clock time of the sixth departure) till the seventh arrival at a clock time of 25.4, that is, for a time interval of 0.91. Thus based on our observations, we can state that the system was idle for a total time of $10.1 + 2.2 + 0.91 = 13.21$ out of the total time of 25.4 time units for which we observed the system. Thus the server utilization (fraction of time for which the server was busy) is $1 - \frac{13.21}{25.4} = 0.4799$.

If one were to create very *long* sequences for the inter-arrival times and the service times, one could then obtain estimates of the utilization of the server *over a long run*. Of course, this kind of a task should be left to the computer, but the point is that computer programs are thus able to collect estimates of parameters measured over a long run.

It should be clear to the reader that although the sequences generated may not be identical to sequences obtained from actual observation of the original system, what really concern us are estimates of parameters such as long-run server utilization. As long as the sequence of values for the governing random variables are generated from the right distributions, *with some work*, the right

values of these parameters can be obtained. For instance, an estimate such as long-run utilization will approach some constant value as the simulation period (time of observing the system, which is 25.4 in the example considered above) approaches infinity.

Some other parameters (or performance measures) for the queuing system can also be measured from the simulation. If we define W to be the average waiting time of the customer in the queue, then it is intuitively clear that the average *long-run* waiting time can be found by summing the waiting times of a very large number (infinity) of customers and dividing the sum by the number of customers. A precise mathematical definition, where w_i will denote the waiting time (in the queue) of the i th customer, is with probability 1,

$$W = \lim_{n \rightarrow \infty} \frac{\sum_{i=1}^n w_i}{n}. \quad (4.3)$$

almost surely. Similarly, the average *long-run* number in the queue can be defined, with probability 1, as

$$Q = \lim_{T \rightarrow \infty} \frac{\int_0^T Q(t) dt}{T}. \quad (4.4)$$

Here $Q(t)$ denotes the number in the queue at time t . Now, to use these definitions with the lim terms, one has to treat ∞ as a large number. How large is “large enough”? The answer to this is that a large number is one which if increased does not significantly affect the quantity being estimated. For example, if we use Equation (4.3) to come up with one estimate of the average long-run waiting time, we can (from our simulation) generate a large number of customers, sum the waiting times of the customers, and then divide the sum by the number of customers. Let us say that we simulate 1000 customers, and the average waiting time turns out to be 1.34. Now if we increase the number of customers simulated, and the average waiting time is still 1.34 in the accuracy of two digits after the decimal point, then it can be concluded that the estimate has *converged* to the limit; we need not increase the number of customers any further.

For *estimating* the average waiting time in the *long run*, we can use the following formula:

$$\tilde{W} = \frac{\sum_{i=1}^n w_i}{n}. \quad (4.5)$$

In the example given in Figure 4.2,

$w_1 = w_2 = 0, w_3 = 15.6 - 13.4, w_4 = 16.79 - 14.3, w_5 = 21.69 - 17.8$, and

$$w_6 = 22.79 - 19.0, \text{ i.e., } \tilde{W} = \frac{0 + 0 + 2.2 + 2.49 + 3.89 + 3.79}{6} = 2.06$$

For estimating the average *long-run* number in the queue, we use a similar mechanism. Whenever a customer arrives or departs, that is, an event occurs, a new *epoch* will be assumed to have begun. The formula for the estimate is:

$$\tilde{Q} = \frac{\sum_{i=1}^n t_i Q_i}{T}, \quad (4.6)$$

where t_i is the time taken in the i th epoch, Q_i is the number of people in the queue during the i th epoch, and $T = \sum_{i=1}^n t_i$ is the amount of time for which simulation is conducted, that is, the sum of the time spent in each epoch.

Ideally, instead of using a large value for n in Equation (4.3) and for T in Equation (4.4), one should check whether the quantity being estimated has remained unchanged, within the desired level of accuracy, in the last several estimates.

It is important to remember that to obtain *long-run* estimates of quantities, one has to run the simulation for a *long* time period. This eliminates any error (bias) due to starting conditions (also called initialization). From one run, one typically obtains one sample.

The other issue is that of independence. Recall that the strong law of large numbers (see Theorem 3.1 in Chapter 3) states that if a large number of independent samples of a random variable are collected, then, almost surely, the sample mean tends to the population mean as n tends to infinity.

In light of this law, we must pay attention to whether the samples being used in estimating a mean are independent *of each other*. This important issue is discussed next.

4.3. Independence of samples collected

In simulation analysis, often, we run the simulation a number of times using a different set of random numbers in each run. When we do one run of the simulation with a given set of random numbers, we are said to have done one **replication** of the simulation. This actually yields only one sample for the quantity we are trying to estimate. Hence, a number of replications have to be done because each replication generates **independent** samples. Why? Because each replication runs with a *unique* set of random numbers. (By changing the seed of a random number generator to a different value, one obtains a new set of random numbers.) The reason for using a large number of replications is that this way we can guarantee that we have a large number of samples, *which are independent*.

Each replication provides one estimate of the quantity being estimated. Averaging over several such estimates, usually, provides a good estimate for the true mean of the parameter.

It is entirely possible that the samples collected within one replication are not independent but *correlated*. The need for independence, of course, follows from the strong law of large numbers.

Doing multiple replications is not the only way to generate independent samples. There are other methods such as the *batching* method and the *regenerative* method. For a discussion on batching, the reader is referred to Pollatschek [137].

Sometimes, it is possible to cleverly combine output from multiple replications to avoid problems related to using co-related data. We, next, look at one example.

Consider the computation of waiting time in the single server queuing system. Since the waiting time of one customer is usually affected by the waiting time of the previous customers, it is *incorrect* to assume that the sample waiting times within one replication are independent of each other. To avoid this difficulty, we can record the waiting time of the i th customer, say the 10th customer, from each replication. The samples collected in this style are not dependent on each other (since they come from different replications, and each replication uses a different set of random numbers). The mean obtained from these samples is closer to the true mean in comparison to the mean obtained from *one replication* in which the service of several customers is simulated.

For quantities such as utilization of the server or the number of people waiting in the queue (in the queuing example), it is not easy to combine output from several replications. Hence other methods may have to be used. These are important issues, and any standard simulation text will discuss them in some detail.

4.4. Terminating and non-terminating systems

A distinction is often made between *terminating* and *non-terminating* systems. Terminating systems are those whose simulations naturally produce independent samples. For example, consider the simulation of a bank that opens at 9 AM and closes at 9 PM. The data collected on any given day from this system is independent of the data collected from any other day. However if the system is non-terminating, for example an ATM center that is open 24 hours, this is not the case. For a non-terminating system, one approach, as described above, is to simulate the system many times using a *different* set of random numbers each time, that is, do several replications. (Notice that in a terminating system we automatically get several runs of the system if the system is run for a very long time.)

For estimating *long-run* estimates of quantities in non-terminating systems, care has to be exercised on two counts: 1) the run length (simulation period) and 2) the number of replications. Unless the run length is sufficient, effects of initializing the system can cause the estimate to have error (bias). Also,

unless, a sufficient number of replications are chosen, we do not have enough independent samples.

Reduction of error (bias) in the estimates is called *variance-reduction*. Several techniques for variance reduction have been suggested in the literature [102]. Variance-reduction techniques can help reduce the error in the data used for estimation, and can bring the variance estimates close to their true values.

The effect of initializing the system is related to the fact that when a system starts, it exhibits several features that are not characteristic of the long run. As such, either the data in the start-up period (also called the warm-up period) has to be eliminated, or the system has to be run for a very long time so that the initial effect is eliminated in the averaging.

In non-terminating systems, the warm-up period's data can be eliminated easily, but in terminating systems, this loss can be significant. The reason for the latter is that often the warm-up period is a significant fraction of the time between start and termination.

Regardless of whether the system is terminating or non-terminating, it is important to base our estimates on independent samples and reduce the noise present in them.

5. Concluding Remarks

Simulation of dynamic systems using random numbers was the main topic covered in this chapter. However, it does not form the main focus of this book, and hence the discussion was not comprehensive. Nevertheless, it is an important topic in the context of simulation-based optimization, and the reader is strongly urged to get a clear understanding of it. For further reading, we refer you to: Law and Kelton [102] and Pollatschek [137].

6. Historical Remarks

Simulation using random numbers can be traced back to 1733. George Louis Leclerc (later known as Buffon) was the first to use simulation to determine the value of π . Random numbers can be used to determine values of deterministic quantities. In the early days, the word Monte-Carlo was associated with simulation. The name Monte Carlo no doubt comes from the fact that Monte Carlo, a city in Europe, was a city known for gambling, much like Las Vegas in the US, and that random numbers are associated with gambling machines. The name Monte Carlo simulation was restricted, originally, to generating random numbers in a static system (e.g., finding the value of π and complicated definite integrals). However, the name is now used to refer to any simulation, including the simulation of a dynamic system.

It was with the advent of computers in the industry during the sixties that *computer simulation* was born. It is not difficult to see that the queuing simula-

tion presented above is best left to a computer program. The power of computers has increased dramatically in the last few decades and continues to increase. This has enabled it to play a major role in analyzing stochastic systems.

We will have more to say about the history of computer simulation in the following chapters, but we must conclude this chapter by pointing out that the initial contributions to what has now emerged as a powerful technique in operations research occurred when nobody could ever have dreamed that there would be computers one day. Not surprisingly, these contributions came from people with a deep understanding of mathematics.

7. Review Questions

1. What is the difference between a mathematical model and a simulation model?
2. Under what scenarios is it a good idea to simulate a system?
3. When is simulation modeling preferable to mathematical modeling? When is the converse true?
4. Why are simulation models usually implemented within computer programs?
5. When do machine-generated *pseudo*-random numbers run into trouble?
(Hint: Read the section that discusses the period of a random number generator.)
6. Can the inverse transform method be used on any given distribution?
7. Find the random number from the distribution (defined below) using the inverse transform method. The corresponding uniform random number is 0.4. The pdf is given by:

$$f(x) = kx \exp(-x^2) \quad x \geq 0.$$

The pdf is 0 in other places.

8. Name the law from Chapter 3 that is used in simulation estimation of parameters. Explain how this law is used in the context of simulation. Explain which condition in this law often makes it necessary for us to do multiple replications. Read the proof of this important law. (Finding a book that discusses its proof will require some library work and possibly some help from your instructor.)
9. Using the data for the single server queue in Figure 4.2, compute the mean and standard deviation of the queue length. Also, determine the probability that the customer has to wait for a non-zero length of time in the queue.
10. Visit the cafeteria in your university. Observe any one counter where customers pay the store for the food they are buying. There are several random variables in this system. Some of them are: 1) the waiting time for a customer, 2) the time between successive arrivals, 3) the time spent in the breaks taken by the server, and 4) the time for service. From this set, identify the governing random variables in the system, and those whose values are determined by the values of the governing random variables.

Chapter 5

SIMULATION-BASED OPTIMIZATION: AN OVERVIEW

The beginning is the most important part of the work.

— Plato (427 BC - 347 BC)

1. Chapter Overview

The purpose of this chapter is to discuss the role that can be played by computer simulation in stochastic optimization. To this end, we will begin by defining stochastic optimization. We will then discuss the usefulness of simulation in the context of stochastic optimization. In this chapter, we will provide a broad description of stochastic optimization problems rather than describing their solution methods.

Stochastic optimization is a class of methods that deal with maximizing (minimizing) the rewards (costs) obtained from a *random* system. We will be concerned with two types of optimization problems for random systems. The problems are:

1. parametric optimization (also called static optimization) and
2. control optimization (also called dynamic optimization).

Section 2 deals with parametric optimization and Section 3 with control optimization. Section 4 provides some historical remarks.

2. Stochastic parametric optimization

Parametric optimization is related to finding the optimal values of decision variables (**parameters**). It can be mathematically defined as follows.

Optimize $f(x(1), x(2), \dots, x(k))$,

subject to some linear or non-linear constraints involving the decision variables $x(1)$, $x(2)$, ..., and $x(k)$.

Here f denotes a function of the decision variables. The problem is one of finding a set of values for these decision variables that optimize, that is, maximize or minimize, this function. Operations research discusses a *class* of methods that can be used to solve this problem.

Methods in this class are called “parametric-optimization” methods.

(Remember, that any maximization (minimization) problem can be converted to a minimization (maximization) problem by multiplying the objective function by -1 .)

Consider the following example.

Example 1. Minimize $f(x(1), x(2), x(3)) = 2x(1) + 4x(2) - 3x(3)$

such that:

$$x(1) + x(2) \leq 4, \text{ and}$$

$$x(1) + x(3) = 5$$

$$x(1), x(2), x(3) \geq 0.$$

Usually f is referred to as the **objective function**. It also goes by other names such as the performance metric, the cost function, the loss function or the penalty function. It is well known that if

- the function f is linear and
- so are the constraints, (as in the Example 1),

one can use **linear programming** (see Taha [168] or any undergraduate text on operations research) to solve this problem. Linear programming methods work **only when the closed form of the function is known**. If the function or its constraints are non-linear, one has to use **non-linear programming** (see Winston [188] or any undergraduate text on operations research). For example, consider the following problem.

Example 2. Minimize $f(x, y) = (x - 2)^2 + (y - 4)^2$,
where x and y take values in the interval $(-\infty, \infty)$.

Using calculus, it is not hard to show that the optimal point (x^*, y^*) is $(2, 4)$. Here is how. One can calculate the partial derivative of the function with respect

to x , and then set it to 0. That is,

$$\frac{\partial f(x, y)}{\partial x} = 0.$$

This will yield:

$$2(x - 2) = 0,$$

which implies that x should equal 2. Similarly, it can be shown that y should equal 4. The optimal point must be (2,4) where the objective function equals 0 because 0 is the least value that the function can take (note that the function is a sum of two squared terms). Finding the optimal point in this case was straightforward because the **closed form** of the objective function was known.

Although the optimization process may not always be this straightforward — just because the closed form is available — it is the case that the availability of the closed form often simplifies the optimization process.

Now let us turn our attention to an objective function with *stochastic elements*. Such a function contains elements of the probability mass function (pmf) or the cumulative distribution function (cdf) of one or more random variables.

Example 3. a. Minimize

$$f(x(1), x(2)) = 0.2[x(1) - 3]^2 + 0.8[x(2) - 5]^2.$$

b. Minimize

$$f(x(1)) = \int_{-\infty}^{\infty} 8[x(1) - 5]^{-0.3} g(x(1)) dx(1)$$

Let us assume the objective function in each case to be the expected value of some random variable, which is associated with a random system. Let us further assume that 0.2, and 0.8 are the elements of the pmf (case a) of the random variable and that g is the pdf (case b) of the random variable.

Example 3 is a non-linear programming problem in which the *closed* form of the objective function is known. Hence it can be solved with standard non-linear programming techniques.

Now consider the following scenario. It is difficult to obtain the elements of the pmf or pdf in the objective function (f) given above, but the value of f — for any set of given values for the decision variables — can be estimated via simulation. In other words, the following holds.

1. The *closed* form of f is *not* known. It may be the case that it is difficult to obtain the closed form or that we may not be interested in spending the time needed to obtain an expression for the closed form.
2. We wish to optimize the function.

3. The function is an objective function associated with a *random system*.

It is in this kind of a situation that simulation may be helpful in optimizing the function. It is a good idea to discuss why we may want to do something like this, that is, optimize a function **without finding its closed form**. There are two reasons.

- a. The first reason stems from the fact that in many real-world stochastic problems, the objective function is too complex to be obtained in its closed form. Under these circumstances, operations researchers frequently employ a theoretical model (such as renewal theory, Brownian motion, and exact or approximate Markov chain approaches) to obtain an **approximate** closed form. In the process, usually, it becomes necessary to make some assumptions about the system to keep the model tractable. If such assumptions are related to the system structure or to the distributions of the relevant random variables, the model generated may be too simplistic for use in the real world. However, if optimization can be performed without obtaining the closed form, one can make many **realistic** assumptions about the system and come up with useful solutions.
- b. The other motivating factor is purely theoretical. Most non-linear programming methods from the early days of operations research require the closed form. Hence there tends to be an interest, among researchers, in methods that can work without the closed form.

Fortunately, the existing literature provides optimization methods that only need the *value of the function* at any given point. These are referred to as *numerical* methods because they depend only on the *numeric value of the function* at any given point. This is in contrast to analytic methods that need the closed form. Examples of numerical techniques are the simplex method of Nelder and Mead [123], the finite difference gradient descent, and the simultaneous perturbation method[162].

The advantage of a numerical method lies in its ability to perform without the closed form. Thus, even if the closed form is unknown, *but some mechanism that provides the function value is available*, we can optimize the function. Consequently, numerical methods form the natural choice in solving complex stochastic optimization problems in the real world, where the objective function's closed form is frequently unknown, but the function can be evaluated numerically.

2.1. The role of simulation in parametric optimization

It is often true that a stochastic system can be easily *simulated*, and an objective function related to the system can then be evaluated at known values of the decision variables. Hence simulation *in combination with numerical meth-*

ods of optimization can be an effective tool for attacking difficult stochastic optimization problems of the parametric variety.

In many stochastic problems, the objective function is the expected (mean) value of a random variable. Then simulation can be used to generate samples of this random variable at any given point in the solution space. Subsequently, these samples can be used to find an estimate of the objective function at the given point. This “estimate” plays the role of the objective function value. This is precisely how simulation provides a mechanism to “evaluate” the objective function and aids in the optimization process.

Combining simulation with numerical parametric-optimization methods is easier said than done. There are many reasons for this. First, the estimate of the objective function is not perfect and contains “noise.” Fortunately, there are techniques that can be used to minimize the effect of noise. Second, a parametric-optimization method that requires a very large number of function evaluations to generate a good solution may be infeasible since even *one* function evaluation via simulation usually takes a considerable amount of computer time (one function evaluation in turn requires several samples, i.e., replications). This is a more serious challenge that simulation-based optimization faces. We will keep this point in mind in our analysis in the remainder of this book.

We would like to reiterate that the role simulation can play in parametric optimization is limited to estimating the function value. Simulation on its own is not an optimization technique. But, as stated above, combining simulation with optimization is possible in many cases, and this throws open an avenue along which many real-life systems may be optimized. In subsequent chapters, we will deal with a number of numerical parametric-optimization techniques that can be combined with simulation to obtain solutions in a reasonable amount of computer time.

We will next discuss the problem of control optimization.

3. Stochastic control optimization

The problem of control optimization is different from the problem of parametric optimization in many respects. Hence considerable work in operations research has been devoted to developing *specialized* techniques for tackling the control optimization problem.

Recall that a system was defined in Chapter 4 as a collection of entities (such as people and machines) that interact with each other. A *dynamic* system is one in which the system *changes* in some way from time to time. To detect changes, we should be able to describe the system using either a *numerical* (quantitative) or a *non-numerical* (qualitative) attribute. Then a change in the value of the attribute can be *interpreted* as a change in the system. This attribute is referred to as the *state* of the system.

A stochastic system is a dynamic system in which the state changes **randomly**. For example, consider a queue that builds up in a counter in a supermarket. Let the state of the system be denoted by the number of people waiting in the queue. Such a description of the queue makes it a stochastic system because the number of people in the queue fluctuates randomly. The randomness in the queuing system could be due to the random inter-arrival time of customers or the random service time of the servers (the service providers at the counters).

The science of **stochastic control optimization** deals with methods that can be used to control a stochastic system. Hence the goal of a control-optimization goal is to **control** the system under consideration, that is, to obtain desirable behavior from the system. Desirable behavior is usually synonymous with improved efficiencies, lowered costs, and increased revenues.

The area of stochastic control optimization studies several problems, some of which are:

1. the Markov decision problem (MDP),
2. the Semi-Markov decision problems (SMDP),
3. the Partially observable Markov decision problems, and
4. the Competitive Markov decision problem (also called a stochastic game.)

In this book, we will concern ourselves with stochastic problems only. Further, we will focus on only the first two problems in this list.

In any control optimization problem, the goal is to move the system along a desirable path — that is, to move it along a desirable trajectory of states. In most states, one has to select from more than one **action**. The actions in each state, essentially, dictate the trajectory of states followed by the system. The problem of control optimization, hence, revolves around selecting the right actions in all the states visited by the system. The performance metric in control optimization is a function of the actions selected in all the states. In general, the control optimization problem can be mathematically described as:

$$\text{Optimize } f(\mu(1), \mu(2), \dots, \mu(N)),$$

where $\mu(i)$ denotes the action selected in state i , f denotes the objective function, and N denotes the number of states. In large problems, N may be of the order of thousands or millions.

Dynamic programming is a well-known and efficient technique for solving many control optimization problems. This technique requires the computation of a so-called **value function** for every state. We will next discuss the role that simulation can play in estimating the elements of the value function of dynamic programming.

3.1. The role of simulation in control optimization

It turns out that each element of the value function can be expressed as an *expectation* of a random variable. Therefore, simulation can be used to generate samples of this random variable. Let us denote the i th sample, of the random variable, X , by X_i . Then the value function at each state can be estimated by using:

$$E(X) = \frac{X_1 + X_2 + \dots + X_n}{n}.$$

This approach can find the expected value of X as n tends to ∞ (see the strong law of large numbers in Chapter 3). Using a sufficiently large sample size, we have in simulation a robust mechanism to evaluate the value function.

It must be understood that the role of simulation in stochastic control optimization is limited to generating samples of the value function, and for optimization purposes, we need to turn to dynamic programming.

There are some real advantages to the use of simulation in combination with dynamic programming. A major drawback of dynamic programming is that it requires the so-called transition probabilities of the system. The probabilities may be hard to obtain.

Theoretically, these transition probabilities can be generated from the distributions of the governing random variables of the system. However, in many complex problems which have a large number of random variables, this may prove to be a difficult task. Hence a computational challenge created by real-world problems is to evaluate the value function without having to compute the transition functions. This is where simulation can play a useful role.

Simulation can be used to first estimate the transition probabilities in a simulator, and thereafter one can use dynamic programming with the transition probabilities generated by the simulator. However, this is usually an inefficient approach.

In **reinforcement learning**, which combines simulation and dynamic programming, simulation is used to generate samples of the value function, and then the samples are averaged to obtain the expected value of the value function. This approach does not require a prior computation of the transition probabilities. This approach is the focus of the chapters dealing with control optimization, and hence we will not delve into any methodological details here.

Compared to finding the transition probabilities of a complex stochastic system, simulating the same system is relatively “easy.” As such, a combination of simulation and dynamic programming can help us attack problems that were considered intractable in the past — problems whose transition probabilities are hard to find.

A large volume of the literature in operations research is devoted to analytical methods of finding exact expressions for transition probabilities of complex random systems (in manufacturing and service). The expressions for the transition

probabilities can get quite involved — with multiple integrals and complicated algebra (see Das and Sarkar [39] for one example). And even these expressions are usually obtained after making simplifying assumptions about the system — to keep the mathematics tractable. *Many of these assumptions can be easily relaxed in a simulator*, and more importantly, in reinforcement learning, *the transition probability — the major stumbling block of the analytical approach — may not be needed!* This gives us the ability to deal with complex problems.

4. Historical Remarks

The simplest and perhaps the oldest example of an operations research model is a linear model, which can be coupled with a linear programming method. Linear programming was developed around World War II. Operations research has come a long way since the days of linear programming. A great deal of research has occurred in the areas of non-linear programming and stochastic programming.

Stochastic dynamic programming was also developed around World War II. The theory of stochastic processes and game theory have also seen a great deal of research in the last five decades.

Simulation-based optimization, as a field, derives strength from research in optimization theory and simulation, and the research that has occurred in the interface of the two. The finite difference gradient descent approach and response surface methods were some of the traditional tools of simulation optimization for several years. Some of the latest path-breaking research in simulation optimization has occurred in areas such as simultaneous perturbation and reinforcement learning. These methods have been developed for use on complex stochastic systems with several decision variables and / or several decision-making states.

The globalization of the economy and the intense competition that has ensued as a result of it have stimulated a great deal of interest in large-scale problems in business applications. Naturally, questions are being asked about how complex stochastic systems may be optimized. It seems like simulation-based optimization should have some useful answers.

5. Review Questions

1. Is there any need for simulation-based optimization techniques when we actually have classical stochastic optimization techniques?
2. What is simulation-based parametric (static) optimization? What is the role played by simulation in it?
3. What is meant by a *numerical* technique for parametric optimization? How is it helpful in simulation-based parametric optimization?

4. How is simulation-based control (dynamic) optimization different from the parametric (static) version? What is the role played by simulation in control optimization?
5. Go to your library and read about the Hooke-Jeeves procedure.

Chapter 6

PARAMETRIC OPTIMIZATION: RESPONSE SURFACES AND NEURAL NETWORKS

Go some distance away because the work appears smaller and more of it can be taken in a glance, and a lack of harmony or proportion is more readily seen.

— Leonardo da Vinci (1452-1519)

1. Chapter Overview

This chapter will discuss one of the oldest simulation-based methods of parametric optimization — namely, the response surface method. For simulation-optimization purposes, the response surface method (RSM) is admittedly primitive. But it will be some time before it moves to the museum because it is a very robust technique that often works well when other methods fail. It hinges on a rather simple idea — that of obtaining an approximate form of the objective function by simulating the system at a finite number of points, which are carefully sampled from the function space. Traditional RSM usually uses regression over the sampled points to find an approximate form of the objective function.

We will also discuss a more powerful alternative to regression — namely, neural networks. Our analysis of this method will concentrate on exploring its roots, which lie in the principles of gradient descent and least square error minimization. Our goal is to expose the “operations research” roots of this “artificial intelligence” method.

We will first discuss the theory of regression-based traditional response surfaces. Thereafter, we will present a response surface technique that uses neural networks — we call it **neuro-response surfaces**.

2. RSM: An Overview

The problem considered in this chapter is the “parametric-optimization problem” discussed in Chapter 5. For the sake of convenience, we reproduce the problem statement here.

Maximize $f(\vec{x})$,

subject to some linear or non-linear constraints involving the vector

$$\vec{x} = (x(1), x(2), \dots, x(k)).$$

Here f denotes the objective function, which may be a linear or non-linear function. The elements of vector, \vec{x} , which are $\{x(1), x(2), \dots, x(k)\}$, are the decision variables. In this book, our interest lies in an objective function with the following traits:

1. It is difficult to obtain an expression for its closed form.
2. The closed form contains elements of pdfs or cdfs, and its value can be estimated via simulation.

As discussed previously in Chapter 5, usually, the value of such functions can be estimated at any given point using simulation. Hence, not surprisingly, simulation can prove to be a useful tool for optimizing such functions — via optimization techniques that rely solely on function evaluation. Although, no attempt is made to find the exact closed form in RSM (we will try to do without the exact closed form throughout this entire book), we will make a **guess** of the structure of the closed form. This guess is usually called the **metamodel**. The metamodel is a term that is distinguishing it from the term “closed form.”

Let us explain this idea with an example. If we assume the structure of the objective function to be linear in one independent variable — x , the decision variable, the metamodel assumes the equation of a straight line, which is:

$$y = a + bx.$$

Here a and b are unknowns that define the metamodel. We will try to estimate their values using the available data related to the function. This is essentially what RSM is all about.

The strategy underlying RSM consists of the following three steps:

1. Select a finite number of points, and evaluate the function at a finite number of points.
2. Assume a **metamodel** for the objective function, and use regression (or some other approach) to fit the metamodel equation to the data; the data is comprised of the selected points and their function values.

Using statistical tests, determine whether the assumed metamodel is acceptable.

3. If the assumed metamodel is acceptable, use it to determine the optimal point; otherwise, go back to the second step.

Remark: We hope that once we estimate the values of the unknowns in the metamodel (e.g., a and b in the straight line), what we have in the metamodel is a *good* approximation of the actual closed form (which is unknown).

Let us, next, look at some more examples of metamodels. If the function is linear with two decision variables (x, y), the metamodel assumes the equation of a plane:

$$z = ax + by + c.$$

The function could be non-linear (a very large number of real-world problems tend to have non-linear objective functions) with one decision variable, and the metamodel could be:

$$y = a + bx^2,$$

or it could be:

$$y = a + bx + cx^2,$$

and so on.

Clearly, there is an infinite number of metamodels for a non-linear objective function, and therefore when the closed form is unknown, the structure of the metamodel is also unknown. Fortunately, there are statistical tests that can be used to determine whether an assumed metamodel is acceptable.

However, notice that the third step in the RSM strategy may reveal that the metamodel assumed in the second step was incorrect. When this happens, one has to guess another metamodel, based on the knowledge of the function, and get back to work! This means that the method may need several iterations if the third step keeps showing that the assumed metamodel is, in fact, not acceptable.

Now, this does not sound very exciting, but it turns out that in practice, very often, we can make pretty good guesses that can closely approximate the actual closed form. Moreover, oftentimes, there is more than one metamodel that is acceptable. In other words, we are rarely in quest of a *unique* metamodel.

In Section 4, we will see a *neural network based* response surface method that does not need the knowledge of the metamodel.

3. RSM: Details

As stated above, RSM consists of several steps. In what follows, we will discuss each step in some detail.

3.1. Sampling

Sampling of points (data pieces) from the function space is an issue that has been studied by statisticians for several years. A comprehensive source for material on this is Myers and Montgomery [119]. Proper sampling of the function space requires a good **design of experiment**. In simulation optimization, the experiment has to be designed properly. The reason is obtaining the function value, at even one point, can be time consuming. As a consequence, one must make an *economic* choice of the number of points to be sampled.

A rough guideline for sampling is as follows: Divide the solution space into a finite number of zones, and select the corner points (or points very close to the corner points) of each zone as samples. Additional samples may be collected from the central points in each zone. Fishing for other points by sampling uniformly in between the corner points is a frequently used strategy when nothing is known about the function.

3.2. Function Fitting

Regression is usually used to fit a function when the coordinates and function values of some points are known. To use standard regression, one must assume the metamodel of the function to be known. We will begin with the simplest possible example, and then move on to more complicated scenarios.

3.2.1 Fitting a straight line

The problem of fitting a straight line belongs to \mathcal{R}^2 space. In other words, the data related to the linear function in one variable is available in the form of (x, y) pairs (or **data pieces**), where y is the objective function value and x is the decision variable. The metamodel is hence:

$$y = a + bx, \quad (6.1)$$

with unknown a and b . See Figure 6.1.

Regression is one of the many ways available to fit a straight line to given (x, y) pairs for obtaining the values of a and b . Some other methods are: Chebyshev fitting, minimax error fitting, absolute mean error fitting, etc. We will not pursue these topics here; we will limit our discussion to regression.

Regression, in comparison to most of the other methods mentioned above, happens to have a low computational burden. It is important to understand the mechanism of regression to appreciate the philosophy underlying RSM and neural networks.

Regression minimizes the *the total squared error between the actual data and the data predictions from the model (metamodel equation) assumed*. For instance, we assumed above that the model is linear. Then using regression, we can come up with values for a and b to predict the value of y for any given

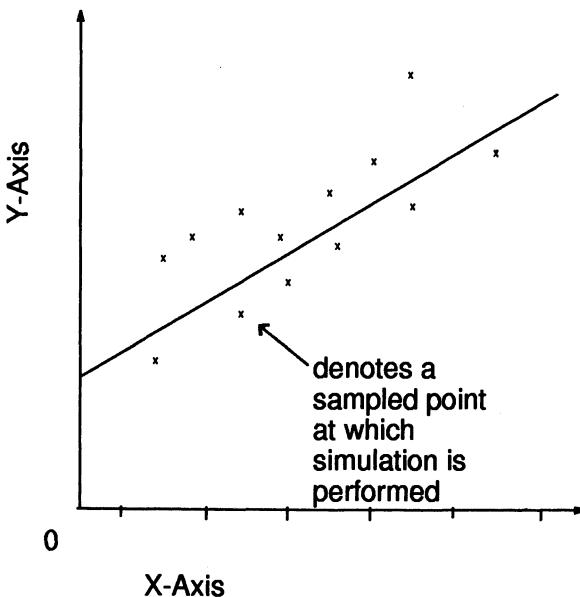


Figure 6.1. Fitting a straight line.

value of x . Clearly, the predicted value may differ from the actual value unless the data is perfectly linear. In regression, our objective is to find those values of a and b that minimize the sum of the square of these differences. Let us state this in more formal terms.

Let $(x_p, y_p), p = 1, 2, \dots, n$ represent n data-pairs available to us. The x_p values are selected by the analyst from the optimization space and the corresponding y_p values are the objective function values obtained from simulation. Let us define e_p as the error term for the p th data piece. It is the difference between the actual value of the objective function, which is y_p , and the predicted value, which is $a + bx_p$. Hence

$$e_p = y_p - (a + bx_p). \quad (6.2)$$

As stated above, the goal in regression is to find the values of a and b that minimize the sum of the squares of the e_p terms. We will denote this sum by SSE . In other words, the goal is to

$\text{minimize } SSE \equiv \sum_{p=1}^n (e_p)^2.$

Now, using Equation (6.2), we have that

$$SSE \equiv \sum_{p=1}^n (e_p)^2 = \sum_{p=1}^n (y_p - a - bx_p)^2.$$

To find a value of a that minimizes SSE, we can find the partial derivative of SSE with respect to a and then equate it to 0, as shown next.

$$\frac{\partial}{\partial a}(SSE) = 0.$$

Calculating the partial derivative, the above becomes:

$$2 \sum_{p=1}^n (y_p - a - bx_p)(-1) = 0,$$

which simplifies to:

$$na + b \sum_{p=1}^n x_p = \sum_{p=1}^n y_p, \quad (6.3)$$

noting that $\sum_{p=1}^n 1 = n$.

Like in the preceding operations, to find the value of b that minimizes SSE, we can calculate the partial derivative with respect to b and then equate it to 0. Thus

$$\frac{\partial}{\partial b}(SSE) = 0$$

which implies that:

$$2 \sum_{p=1}^n (y_p - a - bx_p)(-x_p) = 0.$$

This simplifies to:

$$a \sum_{p=1}^n x_p + b \sum_{p=1}^n x_p^2 = \sum_{p=1}^n x_p y_p. \quad (6.4)$$

Equations (6.3) and (6.4) can be solved simultaneously to find the values of a and b . Let us illustrate the use of these two equations with an example.

Example A. Consider the four pieces of data (x_p, y_p) shown below. The goal is to fit a straight line. The values of x_p have been chosen by the analyst and the values of y_p have been obtained from simulation with decision variable x_p . The values are

$$(50, 12), (70, 15), (100, 21), \text{ and } (120, 25).$$

Then $\sum_{p=1}^4 x_p = 340$, $\sum_{p=1}^4 y_p = 73$, $\sum_{p=1}^4 x_p y_p = 6750$, $\sum_{p=1}^4 x_p^2 = 31800$. Then using Equations (6.3) and (6.4), we have:

$$4a + 340b = 73$$

and

$$340a + 31800b = 6750$$

which when solved yield $a = 2.2759$, and $b = 0.1879$. Thus the metamodel is:

$$y = 2.2759 + 0.1879x.$$

3.2.2 Fitting a plane

The mechanism of fitting a plane is very similar to that of fitting a straight line. We will find the partial derivatives of the sum of the squared error terms with respect to the unknowns in the metamodel, and then set the derivatives to 0.

The metamodel equation for a plane is:

$$z = a + bx + cy. \quad (6.5)$$

The error term is defined as:

$$e_p = z_p - (a + bx_p + cy_p).$$

Hence the SSE is:

$$SSE \equiv \sum_{p=1}^n (e_p)^2 = \sum_{p=1}^n (z_p - a - bx_p - cy_p)^2.$$

Setting the partial derivative of SSE with respect to a to 0, we have:

$$2 \sum_{p=1}^n (z_p - a - bx_p - cy_p)(-1) = 0.$$

This simplifies to:

$$na + b \sum_{p=1}^n x_p + c \sum_{p=1}^n y_p = \sum_{p=1}^n z_p, \quad (6.6)$$

noting that $\sum_{p=1}^n 1 = n$.

Similarly, setting the partial derivative of SSE with respect to b to 0, we have:

$$2 \sum_{p=1}^n (z_p - a - bx_p - cy_p)(-x_p) = 0.$$

This simplifies to:

$$a \sum_{p=1}^n x_p + b \sum_{p=1}^n x_p^2 + c \sum_{p=1}^n x_p y_p = \sum_{p=1}^n x_p z_p. \quad (6.7)$$

As before, setting the partial derivative of SSE with respect to c to 0, we have:

$$2 \sum_{p=1}^n (z_p - a - bx_p - cy_p)(-y_p) = 0.$$

This simplifies to:

$$a \sum_{p=1}^n y_p + b \sum_{p=1}^n x_p y_p + c \sum_{p=1}^n y_p^2 = \sum_{p=1}^n y_p z_p. \quad (6.8)$$

The three linear Equations: (6.6), (6.7), and (6.8) can be solved to obtain the regression (least-square) estimates of a , b , and c .

3.2.3 Fitting hyper-planes

When we have data from \mathcal{R}^k spaces, where $k \geq 4$, it is not possible to *visualize* what a linear form (or any form for that matter) will look like. A linear form in a space such as this is called a **hyper-plane**. When we have three or more decision variables and a linear metamodel to tackle, it is the hyper-plane that needs to be fitted. An example of a hyper-plane with three decision variables is:

$$w = a + bx + cy + dz.$$

In a manner analogous to that shown in the previous section, the following four equation can be derived.

$$\begin{aligned} na + b \sum_{p=1}^n x_p + c \sum_{p=1}^n y_p + d \sum_{p=1}^n z_p &= \sum_{p=1}^n w_p, \\ a \sum_{p=1}^n x_p + b \sum_{p=1}^n x_p^2 + c \sum_{p=1}^n x_p y_p + d \sum_{p=1}^n x_p z_p &= \sum_{p=1}^n x_p w_p, \\ a \sum_{p=1}^n y_p + b \sum_{p=1}^n y_p x_p + c \sum_{p=1}^n y_p^2 + d \sum_{p=1}^n y_p z_p &= \sum_{p=1}^n y_p w_p, \end{aligned}$$

and

$$a \sum_{p=1}^n z_p + b \sum_{p=1}^n z_p x_p + c \sum_{p=1}^n z_p y_p + d \sum_{p=1}^n z_p^2 = \sum_{p=1}^n z_p w_p.$$

Here, as before, n denotes the number of data-pieces. In general, a hyper-plane with k decision variables needs $(k+1)$ linear equations since it is defined by $(k+1)$ unknowns.

3.2.4 Piecewise regression

Sometimes, the objective function is non-linear, but we wish to approximate it by a piecewise *linear* function. A piecewise linear function is not a continuous function; rather, it is defined by a unique linear function in each domain (piece). See Figure 6.2. When we have a non-linear function of this kind, we divide the function space into finite areas or volumes or hyper-spaces (depending on the dimension of the space), and then fit, respectively, a straight line, a plane, or a hyper-plane in each. For example, consider a function, in \mathcal{R}^2 space, defined by:

$$y = 6x + 4 \text{ when } 0 \leq x \leq 4,$$

$$y = 2x + 20 \text{ when } 4 < x \leq 7,$$

$$y = -2x + 48 \text{ when } 7 < x \leq 10,$$

and

$$y = -11x + 138 \text{ when } x > 10.$$

(Please note that Figure 6.2 represents a *similar* function.)

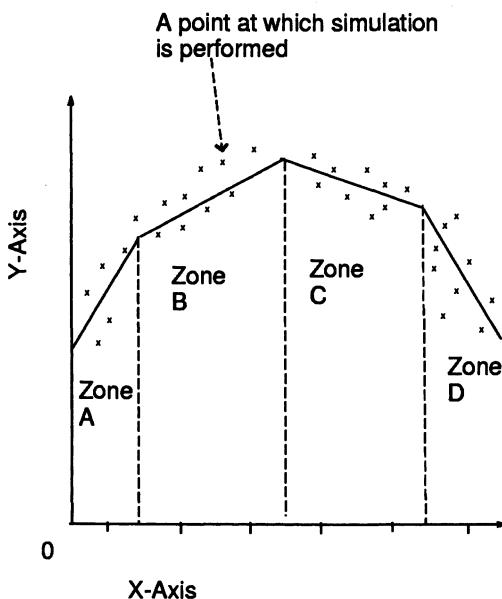


Figure 6.2. Fitting a piecewise linear function.

Fitting this function would require the obtaining of a straight-line fit in each of the four zones: $(0, 4]$, $(4, 7]$, $(7, 10]$, and $(10, \infty)$.

In piecewise regression, one can also use non-linear pieces such as quadratics or higher-order non-linear forms.

3.2.5 Fitting non-linear forms

In this section, we address the issue of how to tackle a non-linear form using regression. Regardless of the form of the objective function, our mechanism is analogous to what we have seen above.

Consider the function given by:

$$y = a + bx + cx^2. \quad (6.9)$$

This form can be expressed in the form of a plane

$$y = a + bx + cz,$$

by setting:

$$z = x^2.$$

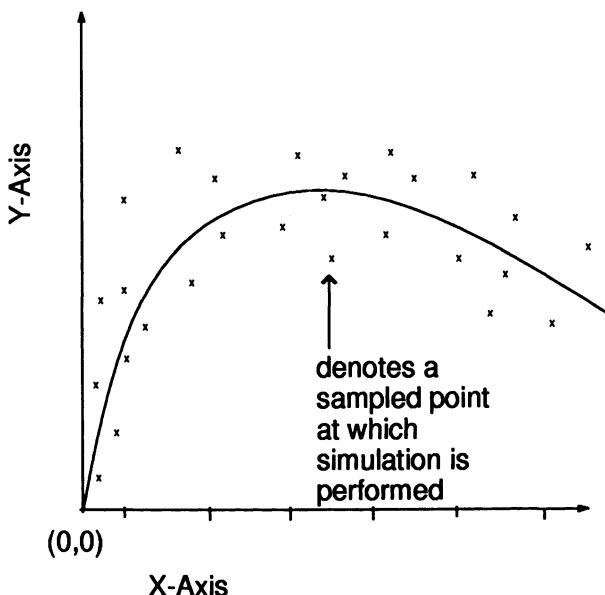


Figure 6.3. Fitting a non-linear equation with one independent variable

With this replacement, the equations of the plane can be used for the meta-model. See Figure 6.3 for a non-linear form with one independent variable and Figure 6.4 for a non-linear form with two independent variables. Other non-linear forms can be similarly obtained by using the mechanism of regression explained in the case of a straight line or plane.

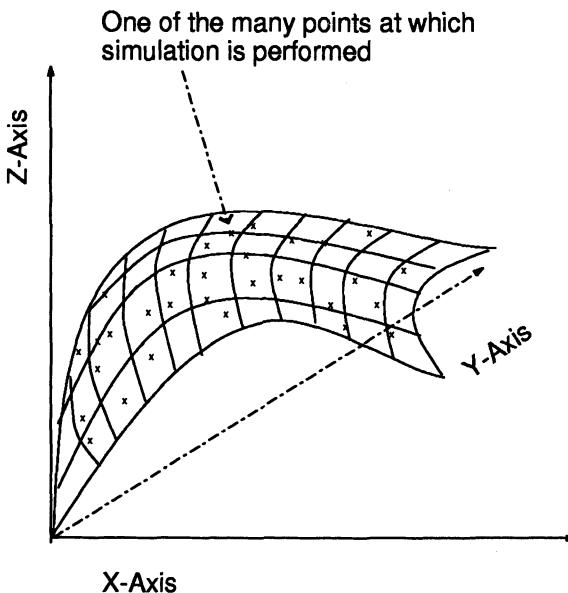


Figure 6.4. Fitting a non-linear equation with two independent variables

3.3. How good is the metamodel?

An important issue in RSM is to determine whether the guessed metamodel is indeed a good fit. Testing whether the fit is reliable has deep statistical ramifications. We will not delve into these issues. Here, we will restrict ourselves to naming some ideas related to this issue. For more details, the reader is referred to Montgomery, Runger, and Hubele [117].

A parameter that is often used in a preliminary test for the goodness of a fit in regression goes by the name: **coefficient of determination**. Its use may be extended to neural network models. It is defined as follows:

$$r^2 = 1 - (SSE/SST),$$

where SST is given by:

$$SST = \sum_{p=1}^n (y_p - \bar{y})^2,$$

and SSE is defined as:

$$SSE = \sum_{p=1}^n (y_p - y_p^{predicted})^2.$$

In the above, \bar{y} is the mean of the y_p terms and $y_p^{predicted}$ is the value predicted by the model for the p th data piece. We have defined SSE during the discussion on fitting straight lines and planes. Note that those definitions were special cases of the definition given here.

Now, r^2 , denotes the proportion of variation in the data that is explained by the metamodel assumed in calculating SSE . Hence a large value of r^2 (i.e., a value close to 1) usually indicates that the metamodel assumed is a good fit. However, the r^2 parameter can be misleading, especially when dealing with several variables. *In a very rough sense*, the reliability of the r^2 parameter increases with the value of n .

3.4. Optimization with a metamodel

Once a satisfactory metamodel is obtained, it is usually easy to find the optimal point on the metamodel. With most metamodels one can use calculus to find the minima or maxima. When a piecewise linear form is used, usually the endpoints of each piece are candidates for minima or maxima. The following example should serve as an illustration.

Example B. Consider a function that has been fitted with four linear functions. The function is to be maximized. The metamodel is given by:

$$y = 5x + 6 \text{ when } 0 \leq x \leq 5,$$

$$y = 3x + 16 \text{ when } 5 < x \leq 10,$$

$$y = -4x + 86 \text{ when } 10 < x \leq 12,$$

and

$$y = -3x + 74 \text{ when } x > 12.$$

It is not hard to see that the peak is at $x = 10$ around which the gradient (slope) changes its sign. Thus $x = 10$ is the optimal point.

The approach used above is quite crude. It can be made more sophisticated by adding a few stages to it. The response surface method is often used in a bunch of stages to make it more effective. One first uses a rough metamodel (possibly piecewise linear) to get a general idea of the region in which the optimal point(s) may lie (as shown above via Example B). Then one zeroes in on that region and uses a more non-linear metamodel in that region. In Example B, the optimal point is likely to lie in the region close to $x = 10$. One can now take the next step, which is to use a non-linear metamodel *around* 10. This form can then be used to find a more precise location of the optimum. It makes a lot of sense to use more replications in the second stage than in the first. A multi-stage approach can become quite time-consuming, but more reliable.

Remark: Regression is often referred to as a **model-based** method because it assumes the knowledge of the metamodel for the objective function.

In the next section, we will study a **model-free** mechanism for function-fitting — the neural network. Neural networks are of two types — linear and non-linear. It is the non-linear neural network that is model-free.

4. Neuro-Response Surface Methods

One of the most exciting features of the non-linear neural network is its ability to approximate *any given function*. It is for this reason that neural networks are used in a wide range of areas ranging from cutting force measurement in metal-cutting to cancer diagnosis. No matter where neural networks are used, they are used for function fitting.

Neural networks are used heavily in the area of pattern recognition. In *many* pattern recognition problems, the basic idea is one of function fitting. Once one is able to fit a function to data, a “pattern” is said to have been recognized. This idea can be generally extended to a large number of scenarios. However, just because a problem has a pattern recognition flavor but does not need function fitting, neural networks should not be used on it.

The open literature reports the failure of neural networks on many problems. Usually, the reasons for this can be traced to the misuse of the method in some form. For example, as mentioned above, neural networks should not be used simply because the problem under consideration has a pattern recognition flavor but has nothing to do with function fitting.

As we will see shortly, the theory of neural networks is based on very sound mathematics. It is, in fact, an alternative way of doing regression. It makes clever use of the chain rule of differentiation and a well-known non-linear programming technique called gradient descent. In what follows, we will first discuss **linear neural networks** — also called **neurons** and then **non-linear neural networks**, which use the famous **backpropagation** algorithm.

4.1. Linear Neural Networks

The linear neural network (also called a neuron) is not model-free; it is model-based and assumes a linear model. The neuron is run by an algorithm that performs **linear regression without solving** any linear systems of equations. Convergence of this algorithm to an optimal solution can be proved. Also, there is strong empirical backing for this algorithm. Although the algorithm has several names such as delta, adaline, and least mean square, we will call it the Widrow-Hoff algorithm in honor of its inventors [187]. We will, next, derive the Widrow-Hoff (WH) algorithm.

Recall that the goal underlying a regression problem for a hyper-plane of the order k is to obtain a fit for the linear equation of the form:

$$y = w(0) + w(1)x(1) + w(2)x(2) + \cdots + w(k)x(k).$$

To obtain a straight line, we would have to set $k = 1$. Recall the definitions of a and b from Equation 6.1. Then, $w(0)$ would correspond to a and $w(1)$ to b . Similarly, for the plane, $k = 2$, $w(2)$ would correspond to c in Equation 6.5.

We will now introduce a subscript in the following terms: $x(i)$ and y . The notation $x_p(i)$ will denote the value of $x(i)$ in the p th data piece. Similarly, y_p will denote the function value in the p th data piece. Now, using this notation, the SSE can be written as:

$$SSE \equiv \sum_{p=1}^n [y_p - w(0) - w(1)x_p(1) - w(2)x_p(2) - \cdots - w(k)x_p(k)]^2. \quad (6.10)$$

For obtaining neural network algorithms, we minimize $SSE/2$ rather than minimizing SSE . The reason will be clear shortly. It amounts to the same thing since minimization of one clearly ensures the minimization of the other.

Now, the WH algorithm is essentially a non-linear programming algorithm in which the function to be minimized is $SSE/2$ and the decision variables are the $w(i)$ terms, for $i = 0, 1, \dots, k$. The topic of non-linear programming will be discussed in more detail in Chapter 7. Here, we present an important and popular gradient-based algorithm for solving a non-linear program. If the non-linear programming problem is described as follows:

Minimize $f(\vec{x})$ where $\vec{x} = \{x(1), x(2), \dots, x(k)\}$ is a k -dimensional vector,

then the main transformation in the gradient-descent algorithm is given by:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}, \quad \text{for each } i, \quad (6.11)$$

where μ is a step size that diminishes to 0.

Thus to derive a gradient-descent algorithm for $SSE/2$, we need to find the partial derivatives of $SSE/2$ with respect to each of the $w(i)$ terms. This is precisely what Widrow and Hoff [187] did. Let us see how it was done.

Now, using Equation (6.10), we have

$$\begin{aligned} \frac{\partial E}{\partial w(i)} &= \frac{1}{2} \sum_{p=1}^n \frac{\partial}{\partial w(i)} [y_p - w(0)x_p(0) - w(1)x_p(1) - \cdots - w(k)x_p(k)]^2 \\ &= \frac{1}{2} \sum_{p=1}^n 2(y_p - o_p) \frac{\partial}{\partial w(i)} [y_p - w(0)x_p(0)] \end{aligned}$$

$$\begin{aligned}
& -w(1)x_p(1) - \cdots - w(k)x_p(k)] \\
& [\text{ by change of notation with } o_p \equiv w(0)x_p(0) \\
& + w(1)x_p(1) + \cdots + w(k)x_p(k)] \\
= & \sum_{p=1}^n (y_p - o_p)[-x_p(i)] \\
= & -\sum_{p=1}^n (y_p - o_p)[x_p(i)].
\end{aligned}$$

Thus,

$$\frac{\partial E}{\partial w(i)} = -\sum_{p=1}^n (y_p - o_p)x_p(i). \quad (6.12)$$

Using Equation (6.12) and transformation (6.11), the WH algorithm becomes:

$$w(i) \leftarrow w(i) + \sum_{p=1}^n (y_p - o_p)x_p(i), \quad (6.13)$$

where $o_p = w(0)x_p(0) + w(1)x_p(1) + \cdots + w(k)x_p(k)$.

Advantages of the WH Algorithm. The advantages of the WH algorithm over regression are not immediately obvious. First and foremost, one does not have to solve linear equations unlike regression. For hyper-planes with large values of k , this can mean considerable savings in the memory needed for the computer program. For instance, to obtain a regression fit for a problem with 100 variables, one would have to set up a matrix of the size of 101×101 and then use a linear equation solving algorithm, whereas the WH algorithm would need to store only 101 $w(i)$ terms. The WH algorithm is guaranteed to converge as long as μ is sufficiently small.

As we will see later, the WH algorithm can also be used for incremental purposes — that is, when the data pieces become available *one by one* and not at the same time. This is seen in reinforcement learning (control optimization). In other words, (6.13) can be used with $n = 1$. We will see this shortly.

We must note here that regression too can be done incrementally and if the order of the hyper-plane is not big, (i.e., for small k), the Widrow-Hoff algorithm does not seem to possess any advantage over regular regression discussed in previous sections.

We next present a step-by-step description of the Widrow-Hoff algorithm.

4.1.1 Steps in the Widrow-Hoff Algorithm

Let us assume that n data pieces are available. Some termination criterion has to be used for stopping the algorithm. One termination criterion assumes that if the absolute value of the difference between the values of SSE computed in successive iterations is “negligible,” the algorithm has converged. How small is negligibly small is left to the user. This quantity is usually referred to as *tolerance*. Another possible termination criterion runs the algorithm till the step size becomes negligibly small.

Step 1: Set all the $w(i)$ terms (for $i = 0, 1, \dots, k$) to small random numbers (preferably between 0 and 1). Set SSE_{old} to a large number. The available data for the p th piece is (y_p, \vec{x}_p) where \vec{x}_p is a vector with k components. Set $x_p(0) = 1$ for all p . Set *tolerance* to a small value.

Step 2: Compute o_p for $p = 1, 2, \dots, n$ using

$$o_p = \sum_{j=0}^k w(j)x_p(j).$$

Step 3: Update each $w(i)$ for $i = 0, 1, \dots, k$ using:

$$w(i) \leftarrow w(i) + \mu \sum_{p=1}^n (y_p - o_p)x_p(i).$$

Step 4: Calculate SSE_{new} using

$$SSE_{new} = \sum_{p=1}^n (y_p - o_p)^2.$$

Reduce the value of μ . One possible approach to determine μ is as follows:

$$\mu = \frac{A}{m},$$

where A is a pre-specified small quantity and m is the number of visits to Step 4.

If $|SSE_{new} - SSE_{old}| < \text{tolerance}$, STOP. Otherwise, set $SSE_{old} = SSE_{new}$, and then go back to Step 2.

4.1.2 Incremental Widrow-Hoff

As discussed in Section 4.1, an incremental version of the Widrow-Hoff algorithm is sometimes useful. The algorithm presented in the previous subsection

is usually called a *batch* algorithm because it uses the entire batch of data simultaneously. The incremental algorithm is obtained from the batch algorithm. This is done by replacing the sum over all values of p by just one quantity. The incremental algorithm is also convergent as long as we use small values for the step size μ . We present a step-by-step description, next.

Step 1: Set all the $w(i)$ terms (for $i = 0, 1, \dots, k$) to small random numbers (between 0 and 1). The available data for the p th piece is (\vec{x}_p, y_p) where \vec{x}_p is a vector with k components. Set $x_p(0) = 1$ for all values of p . Set iter_{\max} to the max number of iterations for which the algorithm is to be run. Set *tolerance* to a small value, and set both p and *iter* to 0.

Step 2: For each value of p from 1 to n , execute the following steps.

Step 2a: Compute o_p using

$$o_p = \sum_{j=0}^k w(j)x_p(j).$$

Step 2b: Update each $w(i)$ for $i = 0, 1, \dots, k$ using:

$$w(i) \leftarrow w(i) + \mu(y_p - o_p)x_p(i).$$

Step 3: Increment *iter* by 1. Reduce the value of μ . One possible approach to determine μ is as follows:

$$\mu = \frac{A}{\text{iter}},$$

where A is a pre-specified small quantity.

If $\text{iter} < \text{iter}_{\max}$, return to Step 2. Otherwise, STOP.

4.1.3 Pictorial Representation of a Neuron

The neuron can be represented pictorially as shown in Figure 6.5. The circles denote the *nodes*. There is one node for each independent variable and one node for the constant term — the term a in regression (See Equation (6.1) or (??). The input to the i th node is denoted by $x(i)$.

4.2. Non-linear Neural Networks

In this section, we will discuss the backpropagation (often abbreviated as *backprop*) algorithm that helps us perform function fitting for non-linear meta-models. To visualize how a non-linear neural network works, consider Figure 6.6. It shows a neural network with 3 layers - an *input* layer, a *hidden* layer

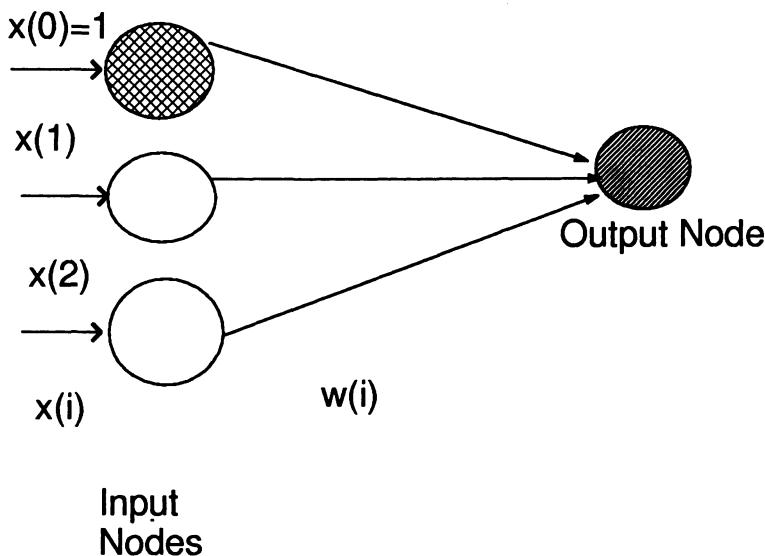


Figure 6.5. A linear network — a neuron with three input nodes and one output node. The approximated function is a plane with two independent variables: $x(1)$ and $x(2)$. The node with input $x(0)$ assumes the role of the constant a in regression.

and an *output* layer. The output layer contains one node. The input nodes are connected to each of the nodes in the hidden layer, and each node in the hidden layer is connected to the output node. All connections are made via unidirectional links.

Neural networks with more than one output node are not needed for regression purposes. In artificial intelligence, we must add here, neural networks with more than one output are used regularly, but their scope is outside of regression. We stick to a regression viewpoint in this book, and therefore avoid multiple outputs. One output in a neural network implies that only one function can be fitted with that neural network.

Neural networks with multiple hidden layers have also been proposed in the literature. They make perfect mathematical sense. Multiple hidden layers are often needed to model highly non-linear functions. We do not discuss their theory. We would also like to add that multiple hidden layers increase the computational burden of the neural network and make it slower.

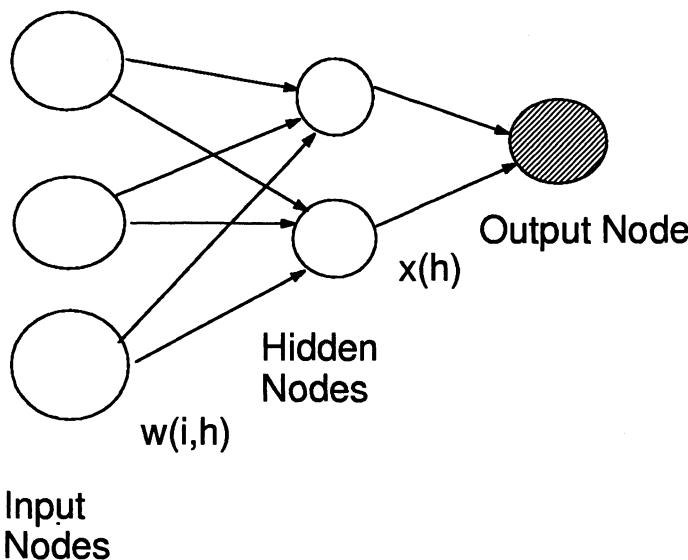


Figure 6.6. A non-linear neural network with an input layer, one hidden layer and one output node. The term $w(i,h)$ denotes a weight on the link from the i th input node to the h th hidden node. The term $x(h)$ denotes the weight on the link from the h th hidden node to the output node.

4.2.1 The Basic Structure of a Non-Linear Neural Network

The input layer consists of a finite number of nodes. One input node is usually associated with each independent variable. Hence, the number of nodes usually equals the number of variables in the function-fitting process.

Usually, we also use one extra node in the input layer. This is called the **bias** node. It takes care of the constant term in a metamodel. It is directly connected to the output node. In our discussion in this section, we will not take this node into consideration. We will deal with it in Section 4.2.4.

There is no fixed rule on how to select the number of nodes in the hidden layer. In a rough sense, the more non-linear the function to be fitted, the larger the number of hidden nodes needed. As we will see, the hidden layer is the entity that makes a neural network non-linear.

Before understanding how the neural network implements the backprop algorithm, we must understand how it predicts function values at any given point *after* the backprop algorithm has been used on it. We also need to introduce some notation at this point.

The connecting arrows or links have numbers associated with them. These scalar values are called **weights** (see Figure 6.6). The neural network backprop algorithm has to derive the right values for each of these weights. Let us denote

by $w(i, h)$ the weight on the link from the i th input node to the h th hidden node. Each input node is fed with a value equal to the value of the associated independent variable at that point. Let us denote the value fed to the j th input node by $u(j)$. Then the raw value of the h th hidden node is given by:

$$v^*(h) = \sum_{j=1}^I w(j, h)u(j),$$

where I denotes the number of input nodes.

However, this is not the actual value of the hidden node that the backprop algorithm uses. This *raw* value is converted to a so-called “thresholded” value. The thresholded value lies between 0 and 1, and is generated by some function. An example of such a function is:

$$\text{Thresholded value} = \frac{1}{1 + e^{-\text{Raw value}}}.$$

The above function goes by the name *sigmoid* function. There are other functions that can be used for thresholding. We will understand the role played by functions such as these when we derive the backprop algorithm.

Thus the actual value of the h th hidden node, $v(h)$, using the sigmoid function, is given by:

$$v(h) = \frac{1}{1 + e^{-v^*(h)}}.$$

Let $x(h)$ denote the weight on the link from the h th hidden node to the output node. Then the output node’s value is given by:

$$o = \sum_{h=1}^H x(h)v(h), \quad (6.14)$$

where $v(h)$ denotes the actual (thresholded) value of the h th hidden node and H denotes the number of hidden nodes.

Now we will demonstrate these ideas with a simple example.
Example C: Let us consider a neural network with three input nodes, two hidden nodes, and one output node, as shown in Figure 6.7. Let the input values be: $u_1 = 0.23$, $u_2 = 0.43$, and $u_3 = 0.12$. Let the weights from the input node to the hidden nodes be: $w(1, 1) = 1.5$, $w(1, 2) = 4.7$, $w(2, 1) = 3.7$, $w(2, 2) = 8.9$, $w(3, 1) = 6.7$ and $w(3, 2) = 4.8$. Let the weights from the hidden nodes to the output node be $x(1) = 4.7$ and $x(2) = 8.9$. Then using the formulas given above:

$$v^*(1) = \sum_{i=1}^3 w(i, j)u(i) = (1.5)(0.23) + (3.7)(0.43) + (0.67)(0.12) = 2.74$$

Similarly, $v^*(2) = 5.484$. Then:

$$v(1) = \frac{1}{1 + e^{-v^*(1)}} = \frac{1}{1 + e^{-2.74}} = 0.9393.$$

Similarly, $v(2) = 0.9959$. Then:

$$o = x(1)v(1) + x(2)v(2) = (4.7)(0.9393) + (8.9)(0.9959) = 13.2782.$$

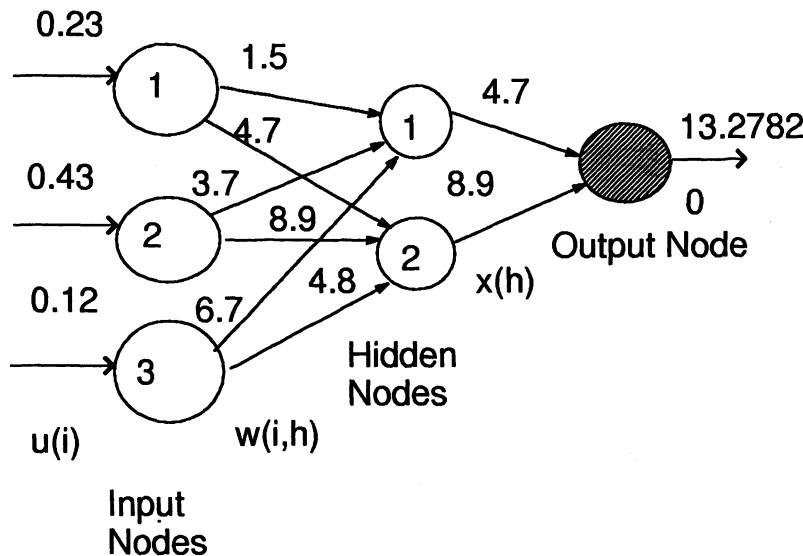


Figure 6.7. Deriving the value of the output for given values of inputs and weights.

IMPORTANT REMARKS.

Remark 1. From the above example, one can infer that large values for inputs and the $w(i, h)$ terms will produce $v(h)$ values that are very close to 1. This implies that for large values of inputs (and weights), the network will lose its discriminatory power. It turns out that even for values such as 50, we have $\frac{1}{1+e^{-50}} \approx 1$. If all data pieces are in this range, then all of them will produce the same output o for a given set of weights. And this will not work. One way out of this problem is to use the following trick. Normalize the raw inputs to values between 0 and 1. Usually the range of values that the input can take on is known. So if the minimum possible value for the i th

input is $a(i)$ and the maximum is $b(i)$ then we should first normalize our data using the following principle:

$$u(i) = \frac{u_{\text{raw}}(i) - a(i)}{b(i) - a(i)}.$$

So for example, if values are: $u_{\text{raw}}(1) = 2$, $a(1) = 0$, and $b(1) = 17$, then the value of $u(1)$ to be fed into the neural network should be:

$$u(1) = \frac{2 - 0}{17 - 0} = 0.117647.$$

Remark 2. An alternative way to work around this difficulty is to modify the sigmoid function as shown below:

$$v = \frac{1}{1 + e^{-v^*/M}},$$

where $M > 1$.

This produces a somewhat similar effect but then one must choose M carefully.

$$M = \max_i b(i) - \min_i a(i)$$

will work.

Remark 3. The $w(i, h)$ terms should also remain at small values for retaining the discriminatory power of the neural network. As we will see later, these terms are in the danger of becoming too large. We will discuss this issue later again.

The subscript p will be now used as an index for the data piece. (If the concept of a “data piece” is not clear, we suggest you review Example A in Section 6.1.) Thus y_p will denote the function value of the p th data piece obtained from simulation. For the same reason, $v_p(h)$ will denote the value of the h th hidden node when the p th data piece is used as an input to the node. So also, $u_p(i)$ will denote the value of the input for the i th input node when the p th data piece is used as input to the neural network. The notations $w(i, h)$ and $x(h)$, however, will never carry this subscript because they do not change with every data piece.

4.2.2 The Backprop Algorithm

Like in regression, the objective of the backprop algorithm is to minimize the sum of the squared differences between actual function values and the predicted values. Recall that in straight line fitting, the regression error was defined as:

$$e_p = y_p - (a + bx_p). \quad (6.15)$$

Now, in the context of backprop, we assume that we do not know the model. So clearly, we do not have access to quantities such as $a + bx_p$ in the equation given above. Notice that this quantity is the *predicted value* of the function. Hence we will replace it by a variable o_p . Hence the error term, regardless of what function is to be fitted, can be expressed as:

$$e_p = y_p - o_p. \quad (6.16)$$

We repeat: the reason for using the term o_p instead of the actual form is: we want to fit the function without assuming any metamodel.

Therefore SSE becomes:

$$SSE = \sum_{p=1}^n (y_p - o_p)^2 \quad (6.17)$$

where n is the total number of data pieces available. For the backprop algorithm, instead of minimizing SSE , we will minimize $SSE/2$. (If $SSE/2$ is minimized, SSE will be minimized too.) In the following section, we will discuss the derivation of the backprop algorithm.

The following star-marked section can be skipped without loss of continuity in the first reading.

4.2.3 Deriving the Backprop Algorithm *

The backprop algorithm is essentially a non-linear programming algorithm in which the function to be minimized is $SSE/2$ and the decision variables of the non-linear program are the weights – the $w(i, h)$ and $x(h)$ terms. If the non-linear programming problem is described as follows:

Minimize $f(\vec{x})$ where $\vec{x} = \{x(1), x(2), \dots, x(k)\}$ is a k -dimensional vector,

then the main transformation in a gradient-descent algorithm is given by:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}, \quad \text{for all } i, \quad (6.18)$$

where μ denotes a step size that diminishes to 0.

For the backprop algorithm, $f(\vec{x})$ is $SSE/2$, and \vec{x} is made up of the $w(i, h)$ and $x(h)$ terms. Using the transformation defined in (6.18), for the $w(i, h)$ terms, we have:

$$w(i, h) \leftarrow w(i, h) - \mu \frac{\partial}{\partial w(i, h)} (SSE/2), \quad (6.19)$$

and for the $x(h)$ terms we have:

$$x(h) \leftarrow x(h) - \mu \frac{\partial}{\partial x(h)} (SSE/2). \quad (6.20)$$

We will denote $SSE/2$ by E . We now need to derive expressions for the partial derivatives: $\frac{\partial E}{\partial x(h)}$ and $\frac{\partial E}{\partial w(i,h)}$ in Equations (6.20) and (6.19) respectively. This is what we do next.

Using Equation (6.17), we have

$$\begin{aligned}\frac{\partial E}{\partial x(h)} &= \frac{1}{2} \sum_{p=1}^n \frac{\partial}{\partial x(h)} (y_p - o_p)^2 \\ &= \frac{1}{2} \sum_{p=1}^n 2(y_p - o_p) \frac{\partial}{\partial x(h)} (y_p - o_p) \\ &= \sum_{p=1}^n (y_p - o_p) \left(-\frac{\partial o_p}{\partial x(h)} \right) \\ &= - \sum_{p=1}^n (y_p - o_p) (v_p(h)).\end{aligned}$$

The last equation follows from the fact that $o_p = \sum_{i=1}^H x(i)v_p(i)$, where H is the number of hidden nodes.

Thus we can conclude that:

$$\frac{\partial E}{\partial x(h)} = - \sum_{p=1}^n (y_p - o_p) v_p(h). \quad (6.21)$$

Similarly, the derivative with respect to $w(i, h)$ can be derived as follows:

$$\begin{aligned}\frac{\partial E}{\partial w(i, h)} &= \frac{1}{2} \sum_{p=1}^n \frac{\partial}{\partial w(i, h)} (y_p - o_p)^2 \\ &= \frac{1}{2} \sum_{p=1}^n 2(y_p - o_p) \frac{\partial (y_p - o_p)}{\partial w(i, h)} \\ &= \sum_{p=1}^n (y_p - o_p) \left(-\frac{\partial o_p}{\partial w(i, h)} \right) \\ &= \sum_{p=1}^n (y_p - o_p) \left(-\frac{\partial o_p}{\partial v_p(h)} \cdot \frac{\partial v_p(h)}{\partial w(i, h)} \right) \\ &= - \sum_{p=1}^n (y_p - o_p) \left(x(h) \cdot \frac{\partial v_p(h)}{\partial w(i, h)} \right) \text{ since } o_p = \sum_{i=1}^H x(i)v_i(p) \\ &= - \sum_{p=1}^n (y_p - o_p) \left(x(h) \cdot \frac{\partial v_p(h)}{\partial v_p^*(h)} \cdot \frac{\partial v_p^*(h)}{\partial w(i, h)} \right)\end{aligned}$$

$$= - \sum_{p=1}^n (y_p - o_p) x(h) v_p(h) (1 - v_p(h)) u_p(i).$$

The last equation follows from the fact that

$$v_p^*(h) = \sum_{j=1}^I w(j, h) u_p(j)$$

implies:

$$\frac{\partial v_p^*(h)}{\partial w(i, h)} = u_p(i),$$

and

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}}$$

implies:

$$\frac{\partial v_p(h)}{\partial v_p^*(h)} = v_p(h)[1 - v_p(h)].$$

Thus in conclusion,

$$\frac{\partial E}{\partial w(i, h)} = - \sum_{p=1}^n [y_p - o_p] x(h) v_p(h) [1 - v_p(h)] u_p(i). \quad (6.22)$$

The backprop algorithm can now be defined by the following two main transformations.

1. Transformation (6.20), which can now be written, using Equation (6.21), as:

$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^n (y_p - o_p) v_p(h). \quad (6.23)$$

2. And transformation (6.19), which can now be written, using Equation (6.22), as:

$$w(i, h) \leftarrow w(i, h) + \mu \sum_{p=1}^n (y_p - o_p) x(h) v_p(h) (1 - v_p(h)) u_p(i). \quad (6.24)$$

Remark 1: A nice thing about this algorithm is that closed form expressions for the partial derivatives could be derived.

Remark 2: Any gradient-descent algorithm can only be guaranteed to converge to *local optima*. If multiple optima exist, convergence to the global optimum

cannot be ensured with a gradient-descent algorithm. Naturally, backprop, being a gradient-descent algorithm, suffers from this drawback. However, in practice, the problem of getting trapped in local optima has not been found to be too menacing. Furthermore, there are ways of working around this problem.

Remark 3: The algorithm derived above uses a sigmoid function for thresholding. Another function that has also been used by researchers is the *tanh* function.

We next discuss how we can deal with a bias node in the backprop algorithm.

4.2.4 Backprop with a Bias Node

The idea underlying the so-called bias node is to assume that the function has a constant term — a term that corresponds to the term, a , in regression theory, which can be found in Equation 6.1 or Equation 6.5. (It is always acceptable to do so because if the true function to be fitted does not have such a term, the weight associated with the bias node will converge to 0.) This is taken care of by assuming that there is an extra node — the bias node — that is connected directly to the output node. See Figure 6.8.

Let us denote the bias weight by b . The input to the bias node is always 1 or some constant value. Hence the output node o should now be defined as:

$$o = (b)(1) + \sum_{h=1}^H x(h)v(h). \quad (6.25)$$

The following star-marked section can be skipped without loss of continuity in the first reading.

4.2.5 Deriving the algorithm for the bias weight *

We will next derive a gradient-descent transformation for the bias weight. Notice the difference between Equation 6.25 and Equation (6.14). Change in the definition of o will not alter $\frac{\partial E}{\partial x(h)}$ that we have derived earlier. Of course $\frac{\partial E}{\partial w(i,h)}$ will not change either. Now, $\frac{\partial E}{\partial b}$ can be derived in a manner identical to that of $\frac{\partial E}{\partial x(h)}$. We will show the details, next.

$$\begin{aligned} \frac{\partial E}{\partial b} &= \frac{1}{2} \sum_{p=1}^n \frac{\partial}{\partial b} (y_p - o_p)^2 \\ &= \frac{1}{2} \sum_{p=1}^n 2(y_p - o_p) \frac{\partial}{\partial b} (y_p - o_p) \\ &= \sum_{p=1}^n (y_p - o_p) \left(-\frac{\partial o_p}{\partial b} \right) \end{aligned}$$

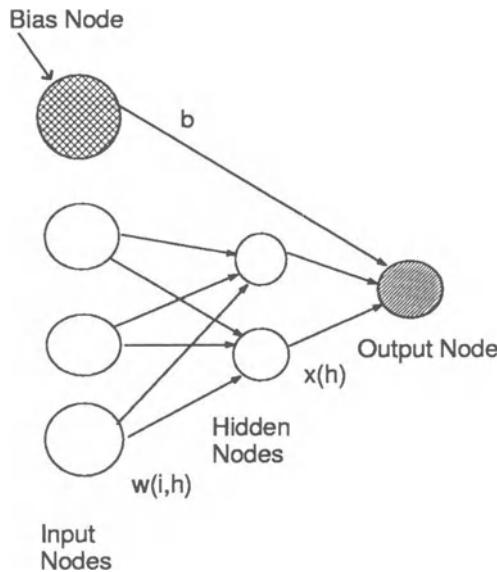


Figure 6.8. A neural network with a bias node. The topmost node is the bias node. The weight on the direct link to the output node is b .

$$= - \sum_{p=1}^n (y_p - o_p) 1.$$

The last equation follows from the fact that $o_p = b + \sum_{i=1}^H x(i)v_p(i)$. Thus,

$$\frac{\partial E}{\partial b} = - \sum_{p=1}^n (y_p - o_p). \quad (6.26)$$

The gradient-descent transformation for the bias weight, which is given by

$$b \leftarrow b - \mu \frac{\partial E}{\partial b},$$

can be written, using (6.26), as:

$$b \leftarrow b + \mu \sum_{p=1}^n (y_p - o_p). \quad (6.27)$$

The backprop algorithm with a bias node is defined by three transformations:

1. transformation (6.23),

2. transformation (6.24), and
3. transformation (6.27).

In the next section, we present a step-by-step description of a version of the backprop algorithm that uses the sigmoid function.

4.2.6 Steps in Backprop

The backprop algorithm is like any other iterative gradient-descent algorithm. We will start with arbitrary values for the weights — usually small values close to 1. Then we will use the transformations defined by (6.23), (6.24), and (6.27) repeatedly till the *SSE* is minimized to an acceptable level. One way to terminate this iterative algorithm is to stop when the difference in the value of *SSE* obtained in successive iterations (step) becomes negligible.

What we present below is also called the “batch-updating” version of the algorithm. The reason underlying this name is that all the data pieces are simultaneously (batch mode) used in the transformations. When we use a transformation and change a value, we can think of the change as an *update* of the value. The step-by-step description is presented below.

Let us define some quantities and recall some definitions. H will denote the number of hidden nodes, and I will denote the number of input nodes. (Note that I will include the bias node.) The algorithm will be terminated when the absolute value of the difference between the *SSE* in successive iterations is less than *tolerance* — a pre-specified small number, e.g., 0.001.

Step 1: Set all weights — that is, $w(i, h)$, $x(h)$, and b for $i = 1, 2, \dots, I$, and $h = 1, 2, \dots, H$, to small random numbers between 0 and 0.5. Set the value of SSE_{old} to a large value. The available data for the p th piece is (\vec{u}_p, y_p) where \vec{u}_p denotes a vector with I components.

Step 2: Compute each of the $v_p^*(h)$ terms for $h = 1, 2, \dots, H$ and $p = 1, 2, \dots, n$ using

$$v_p^*(h) = \sum_{j=1}^I w(j, h) u_p(j).$$

Step 3: Compute each of the $v_p(h)$ terms for $h = 1, 2, \dots, H$ and $p = 1, 2, \dots, n$ using

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}}.$$

Step 4: Compute each of the o_p terms for $p = 1, 2, \dots, n$ where n is the number of data pieces using

$$o_p = b + \sum_{h=1}^H x(h) v_p(h).$$

Step 5:

- Update b using:

$$b \leftarrow b + \mu \sum_{p=1}^n (y_p - o_p).$$

- Update each $w(i, h)$ using:

$$w(i, h) \leftarrow w(i, h) + \mu \sum_{p=1}^n (y_p - o_p) x(h) v_p(h) (1 - v_p(h)) u_p(i).$$

- Update each $x(h)$ using

$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^n (y_p - o_p) v_p(h).$$

Step 6: Calculate SSE_{new} using

$$SSE_{new} = \sum_{p=1}^n (y_p - o_p)^2.$$

Reduce the value of μ . One approach to this is to use the following rule:

$$\mu \leftarrow A/m$$

where A is a small number and m is the number of visits to Step 6.

If $|SSE_{new} - SSE_{old}| < tolerance$, STOP. Otherwise, set $SSE_{old} = SSE_{new}$, and then return to Step 2.

Remark 1: If one wishes to ignore the bias node, b should be set to 0 in Step 4, and the first transformation in Step 5 should be disregarded.

Remark 2: Clearly, the value of *tolerance* should not very low; otherwise, the algorithm may require a very large number of iterations.

Remark 3: Setting a very low value for *tolerance* and running the algorithm for too many iterations can cause *overfitting*. This is undesirable as it can lead to a fit that works well for the data over which the function has been fitted but predicts poorly at points at which the function is unknown.

Remark 4: The parameter r^2 (see Section 3.3) used to evaluate the goodness of a fit can be computed in the case of a neural network too. It can shed some light on how good or bad the neural network fit is.

Remark 5: A good reference to neural networks in practice is the text written by Haykin [70]. Another nice text is Mitchell [116].

Remark 6: As mentioned previously, the version presented above is also called the batch version because all the data pieces are used simultaneously in the updating process.

4.2.7 Incremental Backprop

Like in the case of the linear neural network, we will also discuss the **incremental** variant of the backprop algorithm. The incremental algorithm uses one data piece at a time. We will present the necessary steps in the incremental version. The behavior of the incremental version can be shown to become arbitrarily close to that of the batch version. Incremental backprop is usually not used for response surface optimization. It can be useful in control optimization. We will refer to it in the chapters related to reinforcement learning.

The incremental version is obtained from the batch version, presented in the previous section, by removing the summation over all data pieces. In other words, in the incremental version, the quantity summed in the batch version is calculated for one data piece at a time. The relevant steps are shown next.

Step 1: Set all weights – that is, $w(i, h)$, $x(h)$, and b , for $i = 1, 2, \dots, I$, $h = 1, 2, \dots, H$, to small random numbers. The available data for the p th piece is (\vec{u}_p, y_p) where \vec{u}_p denotes a vector with I components. Set $iter$ to 0 and $iter_{max}$ to the maximum number of iterations for which the algorithm is to be run.

Step 2: For each value of p , i.e., for $p = 1, 2, \dots, n$, execute the following steps.

1. Compute $v_p^*(h)$ for $h = 1, 2, \dots, H$ using

$$v_p^*(h) = \sum_{j=1}^I w(j, h) u_p(j).$$

2. Compute $v_p(h)$ for $h = 1, 2, \dots, H$ using

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}}.$$

3. Compute o_p using

$$o_p = b + \sum_{h=1}^H x(h) v_p(h).$$

4. Update b using:

$$b \leftarrow b + \mu(y_p - o_p).$$

Update each $w(i, h)$ using:

$$w(i, h) \leftarrow w(i, h) + \mu(y_p - o_p) x(h) v_p(h) (1 - v_p(h)) u_p(i).$$

Update each $x(h)$ using

$$x(h) \leftarrow x(h) + \mu(y_p - o_p)v_p(h).$$

Step 3: Increment $iter$ by 1, and reduce the value of μ .

Step 4: If $iter < iter_{max}$, go back to step 2; otherwise stop.

The advantage of incremental backprop over batch backprop lies in its lower computational burden. Since many quantities are summed over n points in batch backprop, the calculations can take considerable amount of computer time. However, with present-day computers, this is not much of an issue. The gradient in the incremental version is not accurate, as it is not summed over all the n data pieces. However, with a **small enough value for the step size μ** , the incremental version should closely approximate the behavior of the batch version.

Some Neural Network Tricks.

1. In practice, the bias node is placed in the hidden layer and an additional bias node (virtual bias node) is placed in the input layer. It is connected to all the other nodes in the hidden layer. The bias node does not have any connection to the input layer. However, it is connected to the output node. The input to each bias node is 1. This trick makes the net stable. See Figure 6.9 for a picture. The C codes in Chapter 15 use this trick.
2. Since backprop is a gradient-descent algorithm and the error function, SSE , may contain multiple optima, getting trapped in local optima cannot be ruled out. This is a commonly experienced difficulty with gradient descent. One way around this is to start at a new point when the net does not perform well. This is a well-known trick. In the case of a neural network, the starting point is generated by the random values (between 0 and 1, typically) given to all the weights. Hence one should use a different SEED (see Chapter 4), and generate new values for the weights.
3. Choice of the number of hidden nodes needs some experimentation. Any arbitrary choice may not yield the best results. It is best to start with a small number of hidden nodes.
4. After performing the backprop for a very long time, the weights can become large. This can pose a problem for the net. (It loses its discriminatory power). One way out of this is to multiply each weight in each iteration by $(1 - \frac{\mu\gamma}{2})$, where γ is another step size less than 1. This update should be performed after the regular backprop update. This idea has good theoretical backing; see Mitchell [116] for more on this.

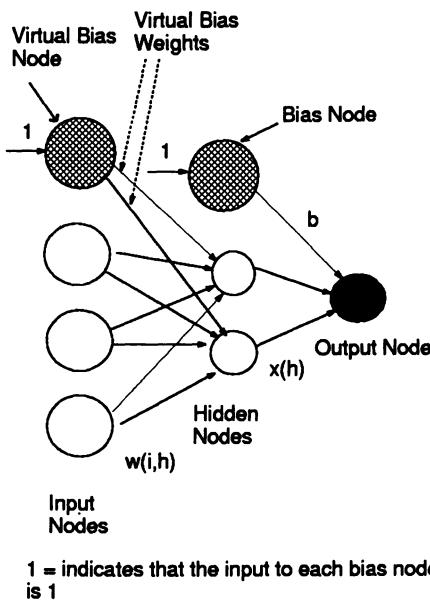


Figure 6.9. Actual implementation with a bias node

4.2.8 Example D

Let us next consider a function the closed form of which is known. We will generate 10 points on this function and then use them to generate a neuro-response surface. Since the function is known, it will help us test the function at any arbitrary point.

Consider the function:

$$y = 3 + x_1^2 + 5x_1x_2.$$

We generate 10 points on this function. The points are tabulated below.

Point	x_1	x_2	y
1	0.000783	0.153779	3.000602
2	0.560532	0.865013	5.738535
3	0.276724	0.895919	4.316186
4	0.704462	0.886472	6.618693
5	0.929641	0.469290	6.045585
6	0.350208	0.941637	4.771489
7	0.096535	0.457211	3.230002
8	0.346164	0.970019	4.798756
9	0.114938	0.769819	3.455619
10	0.341565	0.684224	4.285201

A neural network with backpropagation was trained on this data. 3 hidden nodes were used and a bias node along with a virtual bias node (in the input layer) was used. See Figure 6.9 to see how a virtual bias node is placed in a neural network. We will denote the weight on the link connecting the virtual bias node to the h th hidden node by $vb(h)$. As before, $x(h)$ will denote the weight from the h th hidden node to the output node and $w(i, h)$ will denote the weight from the i th input to the h th hidden node. The bias nodes have inputs of 1 and b will denote the bias weight (the weight on the link from the bias node in the hidden layer to the output node — see Figure 6.9). The neural network was trained for 500 iterations. The best SSE was found to be 0.785247. The codes are given in Chapter 15. The $w(i, h)$ values are:

i	h	$w(i, h)$
1	1	2.062851
1	2	2.214936
1	3	2.122674
2	1	0.496078
2	2	0.464138
2	3	0.493996

The $x(h)$ and the $vb(h)$ weights are:

h	$x(h)$	$vb(h)$
1	2.298128	-0.961674
2	2.478416	-1.026590
3	2.383986	-0.993016

The bias weight is 0.820535. Since the function form is known, we can test how well the function has been approximated by the backprop algorithm. Some comparisons are tabulated below.

Point	x_1	x_2	y	$y^{predicted}$
1	0.2	0.5	3.54	3.82
2	0.3	0.9	4.44	4.54
3	0.1	0.9	3.46	3.78

Note that we have tested the function on input values between 0 and 1. As discussed previously, before training, it is best to normalize the input values to the range (0, 1).

4.2.9 Validation of the neural network

Like in the case of regression-based response surface methods, it is important to “validate” the weights generated by the neural network — that is, check if the network predicts well on the data that was *not* used in generating the

weights. Unless validation is performed, there is no guarantee of the network's performance. Unlike regression-based models, parameters such as r^2 may not be sufficient, and more sophisticated validation tests are needed.

In a commonly-used strategy, called **data splitting**, a data set of n points at which simulation has been performed is split into 2 sets — S_1 and S_2 . S_1 is used to generate the neural network weights and S_2 is used to test the ability of the net to predict well. Then the absolute error, which is reflective of the net's performance, can be calculated as:

$$\text{Absolute Error} \equiv \max_{i \in S_2} |y_i - y_i^{\text{predicted}}|.$$

Several other statistical tests have been described in the literature (see Twomey and Smith [178]). Some of these tests are:

- re-substitution,
- cross-validation,
- jackknife, and
- bootstrap, .

(These tests can also be used in testing the validity of regression metamodels.)

In many of these tests, the *absolute error* between the value predicted by the net and the actual value (defined above) is used, instead of the *squared error* discussed previously.

4.2.10 Optimization with a neuro-RSM model

The neuro-RSM model, once generated and proved to be satisfactory, can be used to evaluate the function at any given point. Then, it may be combined with any numerical non-linear algorithm to find the optimal solution. (Note that the word "optimal" here implies near-optimal since the response surface approach itself introduces considerable approximations.)

5. Concluding Remarks

The chapter was meant to serve as an introduction to the technique of response surfaces in simulation-based optimization. A relatively new topic of combining response surfaces with neural networks was also introduced. The topic of neural networks will surface one more time in this book in the context of control optimization.

6. Bibliographic Remarks

The technique of response surfaces is now widely used in the industry. The method was developed around the end of the Second World War. For a comprehensive survey of RSM, the reader is referred to Myers and Montgomery

[119]. Use of neural networks in RSM is a relatively recent development (see Kilmer, Smith, and Shuman [90] and the references therein.)

The idea of neural networks, however, is not very new. Widrow and Hoff's work [187] on linear neural networks appeared in 1960. Research in non-linear neural networks was triggered by the pioneering work of Werbös [183] — a Ph.D. dissertation in the year 1974. These concepts received widespread attention with the publication of a book by Rumelhart, Hinton, and Williams [149] in 1986. Since then countless papers have been written on the methodology and uses of neural networks. Haykin's text [70] contains nice discussions on several important technological aspects. Our account in this chapter follows Law and Kelton [102] and Mitchell [116]. Neural networks remain, even today, a topic of ongoing research.

7. Review Questions

1. What is the similarity between a neural network and a regression model? Under what scenarios would one prefer a neural network to regression? Discuss some disadvantages of neural networks.
2. Consider the following non-linear function:

$$f(x) = 5x^2 - 2x + 1.$$

Generate 20 points on this function with the center at $x = 0$.

- a. Use regression to estimate the function from the data and then find the point at which the function is minimized.
- b. Use a backpropagation algorithm to predict the function value at any point. (Use codes from Chapter 15). Then minimize the function. (Hint: Test the function values at each training point. This should give you an idea of the region in which the optimal point lies. Then zero in on that region and evaluate the function with a finer grid around the first estimate of the optimal point. Do this a few times.)
3. Evaluate the function $f(x)$ at the following twenty points.

$$x = 2 + 0.4i, \text{ where } i = 1, 2, \dots, 20, \text{ where}$$

$$f(x) = 2x^2 + \frac{\ln(x^3)}{x-1}, \text{ where } 1 \leq x \leq 10.$$

Now, using this data, train the batch version of backprop. Then, with the *trained* network, predict the function at the following points.

$$x = 3 + 0.3i \text{ where } i = 1, 2, \dots, 10.$$

Test the difference between the actual value and the value predicted by the network. (Use codes from Chapter 15).

Chapter 7

PARAMETRIC OPTIMIZATION: SIMULTANEOUS PERTURBATION & META-HEURISTICS

Looking at small advantages prevents great affairs from being accomplished.

— Confucius (551 B.C. — 479 B.C.)

1. Chapter Overview

This chapter discusses simulation-based methods for stochastic parametric optimization. The problem of parametric optimization has been discussed in Chapter 5.

At the very outset, we would like to make an important point. We will concentrate on **model-free** methods — that is, methods which do not require the analytical form of the objective function. The reason for this is: in this book, our interest lies in complex stochastic-optimization problems whose objective functions are rarely known. Furthermore, we will focus on problems with either a very large or an uncountably infinite set of solutions.

Model-free methods are sometimes also called *black-box* methods. Essentially, most model-free methods are numerical — that is, they rely on the (objective) function value and not on the closed form.

Taking into account the major focus of the book, methods requiring knowledge of the analytical form of the objective function (model-based methods) have not been discussed. We must add that when the analytical form is available, it is prudent to use model-based methods. The necessity of model-free methods in simulation optimization arises in those complex, stochastic scenarios, in which the closed form is inaccessible, but may be numerically evaluated at any given point.

We will first discuss methods for *continuous* parametric optimization. The main method that we will cover are simultaneous perturbation and the Nelder-Mead simplex method. For the *discrete* case, we will discuss simulated annealing, genetic algorithms, tabu search, and the learning automata search technique. Convergence of *some* of these methods is discussed in Chapter 12.

2. Continuous Optimization

The problem of continuous parametric optimization can be described as:

$$\text{minimize } f(x(1), x(2), \dots, x(k))$$

subject to some constraints on the vector

$$\vec{x} = (x(1), x(2), \dots, x(k)).$$

In the above, $x(i)$ denotes the i th decision variable, and f is the objective function.

In the model for continuous parametric optimization, we assume that each decision variable can assume any value. We will also assume that there possibly exist some constraints which can restrict the values that the decision variables can assume.

(Any maximization problem can be converted to a minimization problem by reversing the sign of the objective function f , i.e., maximize $f(x) = \text{minimize } -f(x)$.)

As discussed in Chapter 5, we are interested in functions with *stochastic* elements, whose analytical expressions are unknown because it is difficult to obtain them. As a result, simulation may have to be used to find estimates of the function.

Next, we will turn our attention to an approach that uses the gradient of the function in optimizing it.

2.1. Gradient Descent

The method of gradient descent is often used to solve parametric optimization (non-linear programming) problems. What we loosely refer to as gradient descent is often called the method of **steepest descent**. Outside of steepest descent, there are approaches that use the gradient in the optimization process. Since we will limit our discussion to the steepest-descent method, we will use the two names interchangeably.

It is perhaps appropriate to explain why we are discussing the gradient-descent method in a chapter on optimizing “without the closed form of the objective function.” It is the case that the gradient can often be computed, *numerically*, even when the closed form is unknown. When this is the case, one can use the gradient-descent method for optimization without bothering to find the closed form.

The gradient-descent method operates according to the following rule, when proceeding from one iterate to the next:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)},$$

when one is interested in minimizing the function f . In case of maximization (also called hill climbing), the rule becomes:

$$x(i) \leftarrow x(i) + \mu \frac{\partial f(\vec{x})}{\partial x(i)}.$$

In the above:

- $\frac{\partial f(\vec{x})}{\partial x(i)}$ denotes the partial derivative of f with respect to $x(i)$, which is evaluated at \vec{x} , and
- μ denotes what is usually referred to as the step size.

In this section, we will make use of the following notation frequently.

$$\frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{a}}.$$

This denotes a scalar, and it is the numerical value of the partial derivative at $\vec{x} = \vec{a}$.

In what follows, we present a step-by-step description of the gradient-descent method. We will assume that the analytical form of the function is known. The goal is to **minimize** the function.

Steps in Gradient Descent. Set m , the iteration number in the algorithm, to 1. Let \vec{x}^m denote the solution vector at the m th iteration. Initialize \vec{x}^1 to an arbitrary feasible solution (point).

Step 1. For each i , where $i = 1, 2, \dots, k$ (k denotes the number of decision variables), obtain the expression for the partial derivative:

$$\frac{\partial f(\vec{x})}{\partial x(i)}.$$

Step 2. For each i , update $x^m(i)$, using the following rule:

$$x^{m+1}(i) \leftarrow x^m(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{x}^m}.$$

Notice that in the above, one needs the expressions derived in Step 1. The value of the partial derivative, in the above, is calculated at \vec{x}^m .

Step 3. If all the partial derivatives are 0, STOP. Otherwise increment m by 1, and return to Step 2.

When all the partial derivatives are 0, the algorithm *cannot* update the decision variables any further; the value of the update is zero. In any *local optimum*, the partial derivatives are 0, and this is why it is often said that the gradient-descent method gets “trapped” in local optima.

The next example is provided as a simple reminder — from your first course in calculus — of how the gradient can be used directly in optimization.

Example. Let the function to be minimized be:

$$f(x, y) = 2x^2 + 4y^2 - x - y - 4.$$

Clearly, the partial derivatives are:

$$\frac{\partial f(x, y)}{\partial x} = 4x - 1$$

and

$$\frac{\partial f(x, y)}{\partial y} = 8y - 1.$$

Equating the partial derivatives to 0, we obtain the following local optimum:

$$x = 1/4 \text{ and } y = 1/8.$$

Now that we know a local optimum for this problem, next, we demonstrate how the same local optimum can be obtained via the method of gradient descent.

Let the starting point for the method be $(x = 2, y = 3)$. We will use 0.1 for the value for μ . The step size will not be changed. The expressions for the partial derivative are already known.

The results obtained from using gradient descent (described above) are summarized below:

Iteration	x	y	function value
1	1.300000	0.700000	-0.660000
2	0.880000	0.240000	-3.340800
3	0.628000	0.148000	-3.899616
.	.	.	.
.	.	.	.
30	0.250000	0.125000	-4.187500

The above demonstrates that the gradient-descent method yields the same local optimum, that is,

$$x = 1/4 \text{ and } y = 1/8.$$

Increasing the step size, it is intuitively obvious, will accelerate the rate of convergence of this procedure. Suppose $\mu = 0.9$. It can be seen below that the method diverges!

Iteration	x	y	function value
1	-4.3	-17.700000	1308.14
2	12.08	110.64	49129.97
3	80.22	4248.32	72201483.88
.	.	.	.
.	.	.	.
.	.	.	.

The reason for this is: it has been proved that the step size must be *sufficiently small* for the method to work (see Chapter 12). As a result, one seeks the value of the upper limit for the step size with which convergence can be achieved. When the closed form is unknown, the upper limit has to be found, usually, with experimentation.

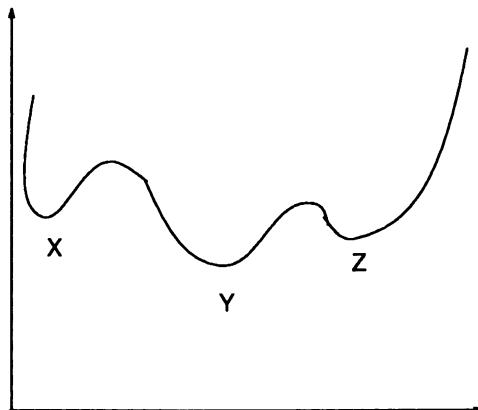
The gradient-descent method is guaranteed to generate the optimal solution only when (i) the step size is sufficiently small (in other words, arbitrary values of the step size may not work) and (ii) the function satisfies certain conditions related to the “*convexity*,” “*continuity*,” and the “*differentiability*” of the function. These conditions have been discussed in Chapter 12 and can be found in any standard text on non-linear programming [18]. Usually, in the absence of the closed form, conditions such as these cannot be verified.

Sometimes, these conditions are not met. In particular, a non-convex cost function contains multiple local minima — in any one of which the gradient-descent method can get trapped. In other words, with non-convex functions, the gradient method may yield solutions that are far from optimal. The gradient method starts from an arbitrary point in the solution space. It then moves from one point to another in its attempt to seek better points where the cost function has lower values. When it reaches an “optimum” (local or not), it stops moving. An “optimum,” roughly speaking, is a point surrounded by worse points. See Figure 7.1 for a pictorial representation of multiple optima.

It is the case that when a function has multiple optima (such as the function in Figure 7.1), the gradient-descent algorithm can get trapped in any one optimum *depending on where it started*. When we say that the gradient-descent algorithm “moves,” we mean: it changes the value of \vec{x} . This continues until it strikes a point where the gradient is 0, that is, a local optimum.

One way to circumvent the problem of getting trapped in local optima is to run the algorithm a number of times, starting at a different point each time, and declare the best local optimum obtained to be the solution. Of course, this does not guarantee the best solution found to be an optimum, but this is frequently the best that we can do in practice.

Function Minimization



X, Y, and Z are local optima. Y is the global optimum

Figure 7.1. A surface with multiple minima

2.1.1 Simulation and Gradient Descent

In this book, we are interested in functions that have a) stochastic elements and b) unknown closed forms. With unknown closed forms (that is, closed forms that cannot be expressed in nice analytical expressions), it is difficult to verify if conditions such as convexity, continuity, and differentiability are satisfied. Moreover, since the function form is unknown, the derivative, if it is to be calculated, must be calculated *numerically*.

The classical definition of the derivative is:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x - h)}{2h},$$

which, of course, you know from your first course in calculus. This suggests the following formula for calculating the derivative numerically. Using a “small” value for h ,

$$\frac{df(x)}{dx} \approx \frac{f(x + h) - f(x - h)}{2h}. \quad (7.1)$$

The above is called the **central differences formula**.

Classically, the derivative can also be defined as:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h},$$

which suggests

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h}. \quad (7.2)$$

The above is called the **forward** differences formula.

Formulas in both (7.1) and (7.2) yield *approximations* of the actual value of the derivative. The actual value is obtained in the limit with h tending to 0.

Note that, in the context of simulation, $f(x+h)$, $f(x)$, and $f(x-h)$ will have to be estimated by simulation. Simulation-based estimates themselves have errors, and therefore this approach is approximate; one has to live with this error in simulation-based optimization.

We state two important facts, next.

- It has been seen via experiments and otherwise (see Andradóttir [8] and references therein) that Equation (7.1) (central differences) has statistical properties superior to those of Equation (7.2); in other words, the error produced due to the approximation of h by a positive quantity is less with the central differences formula. (We will discuss this in Chapter 12.)
- The function evaluations at $(x+h)$ and $(x-h)$ must be performed using common random numbers. This means that both function evaluations should use the *same* set of random numbers in the replications. For instance, if a set of random numbers is used in replication 3 of $f(x+h)$, then the same set should be used in replication 3 of $f(x-h)$. Using common random numbers has been proven to be a “good” strategy — through the viewpoint of statistics. For more on this, see Andradóttir [8].

Let us illustrate, with a simple example, the process of numerically computing the derivative. The derivative of the function

$$f(x) = 2x^3 - 1$$

is $6x^2$. Therefore the actual value of the derivative when $x = 1$ is 6. Now from Equation (7.1), using $h = 0.1$, the derivative is found to be:

$$\frac{[2(6+0.1)^3 - 1] - [2(6-0.1)^3 - 1]}{(2)(0.1)} = 6.02,$$

and using $h = 0.01$, the derivative is found to be:

$$\frac{[2(6+0.01)^3 - 1] - [2(6-0.01)^3 - 1]}{(2)(0.01)} = 6.0002.$$

As h becomes smaller, we approach the value of the derivative. The above demonstrates that the value of the derivative *can* be approximated with small values for h . When the analytic function is unavailable, as is the case with

objective functions of complex stochastic systems, one has to use numerical approximations — such as these — for computing derivatives.

The so-called “finite difference” formula for estimating the derivative is the formula in Equation (7.1) or Equation (7.2). In problems with *many* decision variables, the finite difference method runs into trouble since its computational burden becomes overwhelming. Here is why. Consider the case with k decision variables:

$$x(1), x(2), \dots, x(k).$$

In each iteration of the gradient-descent algorithm, one then has to calculate k partial derivatives of the function. Note that the general expression using central differences is:

$$\frac{\partial f(\vec{x})}{x(i)} =$$

$$\frac{f(x(1), x(2), \dots, x(i) + h, \dots, x(k)) - f(x(1), x(2), \dots, x(i) - h, \dots, x(k))}{2h}$$

The i th partial derivative requires two function evaluations; one evaluation is at

$$(x(1), x(2), \dots, x(i) + h, \dots, x(k))$$

and the other is at

$$(x(1), x(2), \dots, x(i) - h, \dots, x(k)).$$

This implies that in each iteration of the gradient-descent algorithm, one would require $2k$ function evaluations — 2 evaluations per decision variable. Since each function evaluation is via simulation, each function evaluation in turn needs several replications. The computer time taken for just one replication can be significant. Clearly, as k increases, the number of simulations needed increases, and consequently just one iteration of gradient descent may take significant time. This is a major stumbling block of gradient descent. Is there a way out?

The answer is yes, sometimes. A major breakthrough was provided by the work of Spall [162]. Spall, via what he called the *simultaneous* perturbation method, showed that regardless of the number of decision variables, the gradient could be estimated via only 2 function evaluations. Now, it is the case that it is not possible to tamper with the definition of the derivative. The significance of Spall’s derivative is that although it differs significantly from the classical definition of a derivative, it *can* be used in a gradient-descent method to find the local optimum. In other words, it is not a very accurate method for estimating the derivative, but works well in the gradient-descent method. We will discuss his method, next.

2.1.2 Simultaneous Perturbation

The word “perturbation” is related to the fact that the function is evaluated at $(x - h)$ and $(x + h)$. In other words, the function is moved slightly (perturbed) from x — the point at which the derivative is desired. This, of course, is the central idea underlying numerical evaluation of a derivative, and this stems from the classical definition of a derivative.

In simulation optimization, this idea can be found in Kiefer and Wolfowitz [88] — perhaps for the first time. This idea is elementary in the sense that it stems from the fundamental work of Newton (1642-1727) and Leibniz (Latin spelling is Leibnitz) (1646-1716), who (independently) invented the framework of differential and integral calculus. Much of the early work in gradient-based simulation optimization is essentially an application of the fundamental definition of a derivative, which has been around since the seventeenth century. It is Spall’s work [162] that actually, for the first time, breaks away from this path and paves the way for an efficient method for numerical non-linear optimization.

When used in a gradient-descent algorithm, Spall’s definition for a derivative provides us with a method that is not only very efficient in terms of function evaluations needed, but also one that has been shown to converge. As such, it is not surprising that the method is extremely attractive for problems in which

- the objective function’s analytical form is unknown,
- function estimation can be done via simulation, and
- the number of decision variables is large.

Examples of such problems, as mentioned previously, are optimization problems related to complex stochastic systems.

In the finite difference method, when calculating a partial derivative with respect to a variable $x(i)$, it is $x(i)$ that is perturbed — in other words, we evaluate the function at $x(i) + h$ and $x(i) - h$, keeping the other variables unchanged. To illustrate this idea, we present the case with two variables: $x(1)$ and $x(2)$. Then:

$$\frac{\partial f(\vec{x})}{\partial x(1)} = \frac{f(x(1) + h, x(2)) - f(x(1) - h, x(2))}{2h}$$

and

$$\frac{\partial f(\vec{x})}{\partial x(2)} = \frac{f(x(1), x(2) + h) - f(x(1), x(2) - h)}{2h}.$$

The above should make it clear that each variable is perturbed (with h) separately, and as a result, one needs to evaluate the function 4 times. The 4 evaluations in the two preceding expressions are:

- (i) $f(x(1) + h, x(2))$,

- (ii) $f(x(1) - h, x(2))$,
- (iii) $f(x(1), x(2) + h)$, and
- (iv) $f(x(1), x(2) - h)$.

In the simultaneous perturbation method, we perturb all variables *simultaneously*. So in the above 2-variable example, using simultaneous perturbation, we would need to evaluate the function at only two points. The function evaluations needed would be

- (i) $f(x(1) + h(1), x(2) + h(2))$, and (ii) $f(x(1) - h(1), x(2) - h(2))$.

These two evaluations would then be used to find the two partial derivatives. The formula for the “derivative” is provided below in Step 3 of the algorithm description.

Steps in Simultaneous Perturbation. Set m , the iteration number, to 1. Let k denote the number of decision variables. Initialize \vec{x}^1 to an arbitrary feasible point. We will need to use a sequence whose m th term will be denoted by c^m . This has to be a decreasing sequence that converges to 0. An example is $c^m = 1/m^t$, with a fixed t , where $0 < t < 1$. The term μ will denote the step size. The step size will decrease with every iteration of the algorithm. We will terminate the algorithm when the step size becomes smaller than μ_{\min} — a user-specified value. Initialize A to a small value. Set $\mu = A$.

Step 1. Assume that $H(i)$, for $i = 1, 2, \dots, k$, is a Bernoulli distributed random variable, whose two permissible, *equally likely*, values are 1 and -1 . Using this distribution, assign values to $H(i)$ for $i = 1, 2, \dots, k$. Then compute $h(i)$, for all values of i , using

$$h(i) \leftarrow H(i)c^m.$$

Step 2. Calculate F^+ and F^- using the following formulas:

$$F^+ = f(x^m(1) + h(1), x^m(2) + h(2), \dots, x^m(k) + h(k)),$$

and

$$F^- = f(x^m(1) - h(1), x^m(2) - h(2), \dots, x^m(k) - h(k)).$$

Step 3. For $i = 1, 2, \dots, k$, obtain the value for the partial derivative using

$$\frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{x}^m} \approx \frac{F^+ - F^-}{2h(i)}.$$

Step 4. For each i , update $x^{m+1}(i)$ using the following rule.

$$x^{m+1}(i) \leftarrow x^m(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{x}^m}.$$

Notice that, in the above, one needs the values of the derivatives, which were obtained in Step 3.

Step 5. Increment m by 1 and update μ using

$$\mu \leftarrow A/m.$$

If $\mu < \mu_{\min}$ then STOP; otherwise, return to Step 1.

Regarding the above description, we will make a few remarks.

Remark 1. The formula used above for estimating the derivative, it must be noted, is different from the classical finite difference formula. The difference should become clearer from the discussion (see below) on the finite difference approach.

Remark 2. The initial value of μ should be sufficiently small. Gradient-descent rules do not permit arbitrary values for μ . The value, typically, must be less than some threshold value. Moreover, the bias (error) in the updating rule, which is produced by the use of the simultaneous perturbation estimate in place of the gradient estimate, also requires that the step size be small enough. (These issues will be discussed in some more detail in Chapter 12.) At the same time, if μ is too small, no updating will occur in the values of the decision variables. Finding the exact value for the upper limit for μ generally requires knowledge of the analytical form of the function — something that we do not have access to. Hence trial and error must be used to find the initial value of μ .

Remark 3. The algorithm, as mentioned above, cannot guarantee the finding of the global optimum. Hence it is best to run the algorithm a few times starting at a different point in each run.

Remark 4. Common random numbers should be used when evaluating F^+ and F^- .

Remark 5. If the solution space is **constrained**, then one can convert the problem into a problem of unconstrained minimization by using a so-called *penalty* function. An alternative is to “project” the solution onto the feasible region. The latter means that when the algorithm suggests a solution beyond the feasible region, the solution is adjusted so that it lies just inside the boundary.

A Finite Difference Version. If one were to use a finite difference estimate of the derivative (and not a simultaneous perturbation estimate), some steps in the description given above would change. In Step 1, $H(i)$ would be assigned the value of 1 for each $i = 1, 2, \dots, k$. The other changes would be:

Step 2. Calculate $F^+(i)$ and $F^-(i)$ for each $i = 1, 2, \dots, k$, using the following formulas:

$$F^+(i) =$$

$$f(x^m(1) + h(1)I(i=1), x^m(2) + h(2)I(i=2), \dots, x^m(k) + h(k)I(i=k)),$$

and

$$F^-(i) =$$

$$f(x^m(1) - h(1)I(i=1), x^m(2) - h(2)I(i=2), \dots, x^m(k) - h(k)I(i=k)).$$

Here $I(\cdot)$ is an identity function that equals 1 when the condition inside the round brackets is satisfied and equals 0 otherwise.

Step 3. For each $i = 1, 2, \dots, k$, obtain the value for the partial derivative using:

$$\frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{x}^m} \approx \frac{F^+(i) - F^-(i)}{2h(i)}.$$

The actual values of the derivatives can be quite different from the estimates produced by simultaneous perturbation; in other words, there is an error (or bias) in the simultaneous perturbation estimate. As mentioned earlier, we will discuss these issues in Chapter 12 where we present a convergence analysis of simultaneous perturbation. Codes for simultaneous perturbation are provided in Chapter 15.

2.2. Non-derivative methods

We will conclude our discussion on continuous optimization with a method that does not require the derivatives. The method we are referring to is due to Nelder and Mead [123]. It goes by various names, some of which are: the Nelder-Mead method, the downhill simplex method, and the flexible polygon search.

This method is **immensely popular** because it has been widely used in the real world with considerable success. It is regarded as a heuristic because it does not have satisfactory convergence properties. See however [101].

This method can be used in simulation optimization because it only needs numeric values of the function. Furthermore, the number of function evaluations needed in one iteration of the algorithm does *not* depend on the number of decision variables in the problem. The number of function evaluations needed is at most 5.

The philosophy underlying this method is quite simple. We start with a set of feasible solutions; the set is referred to as a “simplex” or a “polygon.” In every iteration, a bad solution in the simplex is dropped in favor of a good solution. This method is not to be confused with the simplex method of Dantzig, which is used for solving linear programs.

We next provide the steps in the algorithm. The algorithm is written in terms of *minimizing* the objective function value. In the description below and in some other algorithms in this chapter, we will use the following notation frequently:

$$\vec{a} \leftarrow \vec{b}.$$

The above implies that, if \vec{a} and \vec{b} are k -dimensional,

$$a(j) \leftarrow b(j) \text{ for } j = 1, 2, \dots, k.$$

Steps in the Nelder-Mead method. Let k denote the number of decision variables. Select, arbitrarily, $(k + 1)$ solutions in the feasible solution space. Let us denote the i th solution by $\vec{x}(i)$ and the set of these solutions by \mathcal{P} . These solutions, as stated above, together form a so-called *polygon* or a *simplex*.

Step 1. From the set \mathcal{P} , select the following three solutions: the solution with the maximum value, to be denoted by \vec{x}_{\max} , the solution with the second largest function value, to be denoted by \vec{x}_{sl} , and the solution with the lowest function value, to be denoted by \vec{x}_{\min} . Now compute a so-called *centroid* of all the points but \vec{x}_{\max} as follows:

$$\vec{x}_c \leftarrow \frac{1}{k} \left[-\vec{x}_{\max} + \sum_{i=1}^{k+1} \vec{x}(i) \right].$$

Then compute a so-called *reflected* point as follows:

$$\vec{x}_r \leftarrow 2\vec{x}_c - \vec{x}_{\max}.$$

Step 2.

- If $f(\vec{x}_{\min}) > f(\vec{x}_r)$, go to Step 3.
- If $f(\vec{x}_{sl}) > f(\vec{x}_r) \geq f(\vec{x}_{\min})$, go to Step 4.
- If $f(\vec{x}_r) \geq f(\vec{x}_{sl})$, go to Step 5.

Step 3. We come here if the reflected point is better than the best point in \mathcal{P} . The operation performed here is called **expansion**. The idea is to determine if a point better than the reflected point can be obtained. Compute the expanded solution as follows:

$$\vec{x}_{exp} \leftarrow 2\vec{x}_r - \vec{x}_c.$$

If $f(\vec{x}_{exp}) < f(\vec{x}_r)$, set

$$\vec{x}_{new} \leftarrow \vec{x}_{exp}.$$

Otherwise, set

$$\vec{x}_{new} \leftarrow \vec{x}_{exp}.$$

Go to Step 6.

Step 4. We come here if the reflected point is better than \vec{x}_{sl} . Set

$$\vec{x}_{new} \leftarrow \vec{x}_r,$$

and go to Step 6.

Step 5. We come here if the reflected point is worse than \vec{x}_{sl} . The operation performed here is called **contraction**.

- If $f(\vec{x}_r) < f(\vec{x}_{\max})$, set: $\vec{x}_{new} \leftarrow 0.5(\vec{x}_r + \vec{x}_c)$, and go to Step 6. Otherwise, compute:

$$\vec{x}_{new} \leftarrow 0.5(\vec{x}_{\max} + \vec{x}_c).$$

- If $f(\vec{x}_{new}) < f(\vec{x}_{\max})$, go to Step 6. Otherwise, that is, if the new solution is worse than the worst (\vec{x}_{\max}), then we have to squeeze the old simplex towards the best point. Compute, for each $i \in \mathcal{P}$,

$$\vec{x}(i) \leftarrow \frac{\vec{x}(i) + \vec{x}_{\min}}{2},$$

and go to Step 1.

Step 6. Replace the old \vec{x}_{\max} by \vec{x}_{new} . Set

$$\vec{x}_{\max} \leftarrow \vec{x}_{new},$$

and return to Step 1.

The algorithm can be run for a user-specified number of iterations. The algorithm works well only on problems with up to 10 decision variables [18]. Another popular non-derivative method, which we do not discuss, is that of Hooke and Jeeves [80].

We next turn our attention to discrete optimization.

3. Discrete Optimization

The problem of discrete parametric optimization is actually harder than continuous parametric optimization since the function has gaps, and so derivatives may be of little use.

We will assume in this section that the solution space is finite. Now, if the solution space is small, say composed of 100 points, then the problem can often be solved by an exhaustive search of the solution space. In discrete (stochastic) parametric optimization problems, like in the continuous case, we assume that it is possible to estimate the function at any given point using simulation. Thus an exhaustive search *should* be conducted — if it can be done in a reasonable amount of time.

If the solution space is large (several thousand or more points), it becomes necessary to use algorithms that can find **good** solutions without having to search exhaustively. We have said above that 100 points constitute a manageable space for an exhaustive search. We must add that the actual number may depend on how complex the system is. For an M/M/1 queuing simulation written in C (see Law and Kelton [102] for a code), testing the function even at 500 points may not take too much time since M/M/1 is perhaps the simplest stochastic system.

However, if the simulation is more complex (even if it is written in a language such as C), the time taken on the computer to evaluate the function at one point can be significant, and hence what constitutes a *manageable* space may be smaller. With the increasing power of computers, the size of the manageable space is likely to increase. But it is the case that when one has several thousand or million points to evaluate, one has to look for a method which can do without exhaustive evaluation.

When we have a small search space (say of the size 100), we need to compare the solutions and identify the best, or rank them. This can be done with the help of the **ranking and selection** techniques (see Goldsman and Nelson [56]) or the multiple comparison procedure (see Hochberg and Tamhane [76]). These methods have good mathematical backing and serve as robust methods for comparison purposes. As we will see later, they can be used in combination with other methods also. Next, we will discuss ranking and selection.

3.1. Ranking and Selection

Ranking and selection methods are statistical methods designed to select the best solution from among a set of competing candidates. These methods have a great deal of theoretical (statistical theory) and empirical backing and can be used when one has up to 20 (candidate) solutions. In recent times, the use of these methods has been shown on much larger problems.

A strong feature of the ranking methods is that they can *guarantee*, as long as certain conditions are met, that the probability of selecting the best solution from the candidate set exceeds a user-specified value. These methods are primarily useful in a careful statistical comparison of a finite number of solutions.

In this section, we will discuss two types of ranking and selection methods, namely, the Rinott method and the Kim-Nelson method. The problem considered here is one of finding the best solution from a set of candidate solutions. We may also want to rank them. We will discuss the comparison problem as one in which the solution with the *greatest* value for the objective function is declared as the best. Before plunging into the details of ranking and selection methods, we need to introduce some notation.

- Let r denote the total number of solutions to be evaluated and compared.
- Let $X(i, j)$ represent the j th independent observation from the i th solution. This needs explanation. Recall from Chapter 4 that to estimate steady-state (that is, long-run) performance measures, it becomes necessary to obtain several independent observations of the performance measure under consideration. (Usually, these observations are obtained from the independent replications of the system.) In the context of this book, the performance measure is the objective function. If we have r solutions to compare, then

clearly, $i = 1, 2, \dots, r$. Also, $j = 1, 2, \dots, m$, if m denotes the total number of independent observations.

- Let $\bar{X}(i, m)$ represent the sample mean obtained from averaging the first m samples from solution i . Mathematically, this can be expressed as:

$$\bar{X}(i, m) = \frac{\sum_{j=1}^m X(i, j)}{m}.$$

- Let δ denote the so-called “indifference zone” parameter. If the absolute value of the difference in the objective function values of two solutions is less than δ , we will treat the two solutions to be equally good (or bad). In other words, we will not distinguish between those two solutions. Clearly, δ will have to be set by the user and will depend on the situation.
- Let α represent the significance level in the comparison. In the context of our definition of δ above, we can say that a ranking and selection method will guarantee with a probability of $1 - \alpha$ that the solution selected by it as the best *does* have the largest mean, if the *true* mean of the best solution is at least δ better than the second best.

We will assume, throughout the discussion on ranking and selection methods, that $X(i, j)$, for every j , is normally distributed and that its mean and variance are unknown.

We first discuss the Rinott method [141].

3.1.1 Steps in the Rinott method

After reviewing the notation defined above, select suitable values for α , δ , and the sampling size m , where $m \geq 2$. For each $i = 1, 2, \dots, r$, simulate the system associated with the i th solution. Obtain m independent observations of the objective function value for every system. $X(i, j)$, as defined above, denotes the j th observation (objective function value) of the i th solution.

Step 1. Find the value of Rinott’s constant h_R from [13]. The value will depend on m, r , and α . For each $i = 1, 2, \dots, r$ compute the sample mean using:

$$\bar{X}(i, m) = \frac{\sum_{j=1}^m X(i, j)}{m},$$

and the sample variance using:

$$S^2(i) = \frac{1}{m-1} \sum_{j=1}^m [X(i, j) - \bar{X}(i, m)]^2.$$

Step 2. Compute, for each $i = 1, 2, \dots, r$,

$$N_i = \max \left(m, \left[\frac{h_R^2 S^2(i)}{\delta^2} \right]^+ \right),$$

where $[a]^+$ denotes the *smallest* integer greater than a . If $m \geq \max_i N_i$, declare the solution with the maximum $\bar{X}(i, m)$ as the best solution. STOP.

Otherwise, obtain $\max(0, N_i - m)$ *additional* independent observations of the objective function value for the i th solution, for $i = 1, 2, \dots, r$. Then declare the solution(s) with the maximum value for $\bar{X}(i, N_i)$ as the best.

We next discuss the Kim-Nelson method [91], which is a very recent development. This method may require fewer observations in comparison to the Rinott method.

3.1.2 Steps in the Kim-Nelson method

After reviewing the notation defined above, select suitable values for α , δ , and the sampling size m , where $m \geq 2$. For each $i = 1, 2, \dots, r$, simulate the system associated with the i th solution. Obtain m independent observations of the objective function value for every system. $X(i, j)$, as defined above, denotes the j th observation (objective function value) of the i th solution.

Step 1. Find the value of the Kim-Nelson constant using:

$$h_{KN}^2 = \left[[2\{1 - (1 - \alpha)^{1/(r-1)}\}]^{-2/(m-1)} - 1 \right] [m - 1].$$

Step 2. Let $\mathcal{I} = \{1, 2, \dots, r\}$ denote the set of candidate solutions. For each $i = 1, 2, \dots, r$ compute the sample mean as:

$$\bar{X}(i, m) = \frac{\sum_{j=1}^m X(i, j)}{m}.$$

For all $i \neq l$, compute:

$$S^2(i, l) = \frac{1}{m-1} \sum_{j=1}^m [X(i, j) - X(i, l) + \bar{X}(l, m) - \bar{X}(i, m)]^2.$$

Step 3. Compute:

$$N_{il} = \left(\frac{h_{KN}^2 S^2(i, l)}{\delta^2} \right)^-,$$

where $(a)^-$ denotes the largest integer smaller than a . Let

$$N_i = \max_{i \neq l} N_{il}.$$

If $m \geq (1 + \max_i N_i)$, declare the solution with the maximum value for $\bar{X}(i, m)$ as the best solution, and STOP.

Otherwise, set $p \leftarrow m$, and go the next step.

Step 4. Let

$$\mathcal{I}_s = \{i : i \in \mathcal{I} \text{ and } \bar{X}(i, p) \geq \bar{X}(l, p) - W_{il}(p) \quad \forall l \in \mathcal{I}, l \neq i\},$$

where

$$W_{il}(p) = \max \left(0, \frac{\delta}{2p} \left[\frac{h_{KN}^2 S(i, l)}{\delta^2} - p \right] \right).$$

Then set:

$$\mathcal{I} \leftarrow \mathcal{I}_s.$$

Step 5. If $|\mathcal{I}| = 1$, declare the solution whose index is still in \mathcal{I} as the best solution, and STOP.

Otherwise, go to Step 6.

Step 6. Take one *additional* observation for each system in \mathcal{I} and set

$$p \leftarrow p + 1.$$

If $p = 1 + \max_i N_i$, declare the solution whose index is in \mathcal{I} and has the maximum value for $\bar{X}(i, p)$ as the best solution, and STOP.

Otherwise, go to Step 4.

3.2. Meta-heuristics

When we have several hundred (or thousand) solutions in the solution space, ranking and selection methods cannot be used directly. In the last twenty-five years or so, a great deal of research has focused on developing methods that can produce “good” solutions on large-scale discrete optimization problems. These methods are generally called **meta-heuristics**. One of the first meta-heuristics that appeared in the literature is the *genetic algorithm*. Other popular meta-heuristics are *simulated annealing* and *tabu search*. Two other useful meta-heuristics are the *learning automata search technique* and the *stochastic ruler method*.

These methods rely on numeric function evaluations and as such can be combined with simulation. It must be understood that with large-scale problems, the optimal solution may be difficult to characterize. Therefore, the performance of a meta-heuristic on a large problem, usually, cannot be calibrated with reference to an optimal solution. The calibration has to be done with other available heuristic methods. Also, it must be understood that these methods

are not guaranteed to produce optimal solutions, but only good solutions in a reasonable time interval on the computer.

In the last few years, the popularity of meta-heuristics has steadily increased; the reason for this is that sometimes, on large-scale real-world problems, meta-heuristics are the only viable tools. Also, some analytical proofs of convergence have been developed for these methods.

We will concentrate on the use of meta-heuristics in discrete optimization of objective functions related to stochastic systems — in particular of those objective functions whose closed forms are not available, but whose values may be estimated via simulation with ease.

Before plunging into the details of meta-heuristics, we must discuss what is meant by a “neighbor” of a solution. We will explain this with an example.

Example. Consider a parametric optimization problem with two decision variables, both of which can assume values from the set:

$$\{1, 2, \dots, 10\}.$$

Now consider a solution $(3, 7)$. A *neighbor* of this solution is $(4, 6)$, which is obtained by making the following changes in the solution $(3, 7)$.

$$3 \rightarrow 4 \text{ and } 7 \rightarrow 6.$$

It is not difficult to see that these changes produced a solution $-(4, 6)$ — that lies in the “neighborhood” of a given solution $(3, 7)$. Neighbors can also be produced by more complex changes.

We will begin our discussion of meta-heuristics with simulated annealing.

3.2.1 Simulated Annealing

The mechanism of simulated annealing is straightforward. An arbitrary solution is selected to be the starting solution. (It helps to start at a good solution, but this is not necessary.) In each iteration, the algorithm tests one of the neighbors of the starting solution (current solution) for its function value. Now, the neighbor is either better or worse / equally good in comparison to the current solution. (In case of minimization, a worse solution would mean a higher value for the objective function.) If the neighbor is better, the algorithm moves to the neighbor. If the neighbor is worse / equally good, the algorithm moves to it with a low probability. This is said to complete one iteration of the algorithm. The phenomenon of moving to a worse solution is also called *exploration*.

Several iterations of the algorithm are typically needed. In every iteration, there is a positive probability of moving to a worse solution; in other words, the algorithm *does* have the choice of moving to a worse neighbor. But as the algorithm progresses, the probability of moving to a worse neighbor *must be*

reduced. The algorithm terminates when the probability of moving to a worse neighbor approaches 0. At each iteration, the best solution obtained—until that iteration — is stored separately. As a result, the best of all the solutions tested by the algorithm is returned as the “near-optimal” solution from the algorithm.

Will this strategy work? There are two points to be made in this context.

- In practice, this algorithm has been shown to return an **optimal** solution on small problems; remember on small problems, one can determine the optimal solution by an exhaustive evaluation of the solution space. On large problems, in many cases, the algorithm has been *reported* to outperform *other heuristics* (usually the optimum is unknown).
- Markov chains have been used to establish that the simulated annealing algorithm has the potential to return the optimal solution provided the probability of moving into a worse neighbor is decreased *properly*.

Do note though that we do not really have a definitive answer to the question posed above.

The simulated annealing algorithm is so named because of its similarity with the “annealing” process in metals. The latter (process) requires that the temperature of the metal be raised and then lowered *slowly* so that the metal acquires desirable characteristics such as hardness. The algorithm is said to be analogous to the metallurgical process because the algorithm starts with a relatively high probability (read temperature) of moving to worse neighbors, but the probability has to be properly *reduced*. Just as the metal does not get the desired features unless the temperature is lowered at the right rate, the simulated annealing algorithm does not produce the optimal solution unless the probability in question is reduced properly. The analogy can be taken a little further. At high temperatures, the atoms vibrate a good deal. These vibration levels gradually reduce with the temperature. Similarly, in the beginning, the simulated annealing algorithm moves out to what seem like poor solutions quite a bit, but the level of this activity gets reduced (or rather should be reduced) as the algorithm progresses.

Although this analogy serves to help our intuition, we should not take it too seriously because associated with such analogies is the inherent danger of overlooking the actual reason for the success (or failure) of an algorithm. Furthermore, almost all meta-heuristics have such metallurgical (simulated annealing), biological (genetic algorithms), or artificial intelligence (learning automata) connections. We will use an operations research perspective in this book and treat each of these methods as an optimization technique. When it comes to studying their convergence properties, mathematically, these analogies are seldom of any use.

Steps in Simulated Annealing. Choose an initial solution; denote it by \vec{x}_{current} . Let $f(\vec{x})$ denote the value of the objective function (obtained via simulation) at \vec{x} . Let \vec{x}_{best} denote the best solution so far. Set:

$$\vec{x}_{\text{best}} \leftarrow \vec{x}_{\text{current}}.$$

Set T , also called the “temperature,” to a pre-specified value. The temperature will be gradually reduced in discrete steps. But at each temperature, we will perform Steps 2 and 3 for a number of iterations. This is called a phase. Thus each phase will consist of several iterations. The number of iterations in each phase should generally increase with the number of phases. The algorithm is written for *minimizing* the objective function.

Step 1. Set the phase number, P , to 0.

Step 2. Randomly select a neighbor of the current solution. (Selection of neighbors will be discussed below in Remark 1.) Denote the neighbor by \vec{x}_{new} .

Step 3. If $f(\vec{x}_{\text{new}}) < f(\vec{x}_{\text{best}})$, then set:

$$\vec{x}_{\text{best}} \leftarrow \vec{x}_{\text{new}}.$$

Let

$$\Delta = f(\vec{x}_{\text{new}}) - f(\vec{x}_{\text{current}}).$$

- If $\Delta \leq 0$, set:

$$\vec{x}_{\text{current}} \leftarrow \vec{x}_{\text{new}}.$$

- Otherwise, that is, if $\Delta > 0$, generate a uniformly distributed random number between 0 and 1, and call it U .

If

$$U \leq \exp(-\frac{\Delta}{T}),$$

then set:

$$\vec{x}_{\text{current}} \leftarrow \vec{x}_{\text{new}},$$

otherwise

$$\text{do not change } \vec{x}_{\text{current}}.$$

Step 4. One execution of Steps 2 and 3 constitutes one iteration of a phase. Repeat Steps 2 and 3 until the number of iterations associated with the current phase are performed. When these iterations are performed, go to Step 5.

Step 5. Increment the phase number P by 1. If $P < P_{\max}$, then reduce T (a temperature reduction scheme will be discussed below) and go back to Step

2 for another phase. Otherwise, terminate the algorithm and declare \vec{x}_{best} to be the best solution obtained.

A few remarks on the algorithm are in order.

Remark 1. There are no “accepted” rules for selecting neighbors. Perhaps this deficiency is an important issue for researchers to pursue. To give an example for selecting neighbors, consider a problem with two decision variables: x and y . The decision variables, let us assume, could take on any one of the following values:

$$1, 2, 3, 4, 5, 6, 7, 8.$$

So if a point is defined as $(3, 5)$, some of its *adjacent* neighbors are:

$$(2, 6), (4, 6), \text{ and } (4, 4).$$

But neighbors do not have to be *adjacent* neighbors. For instance, $(1, 8)$ can also be viewed as a neighbor of $(3, 5)$.

To select an adjacent neighbor for a decision variable (say x), a possible strategy is to produce a uniformly distributed random number in the interval $(0, 1)$. Then, the left hand neighbor is to be selected if the random number is less than 0.5 and the right hand neighbor, otherwise. For instance, in the example considered above, when $x = 3$,

the left hand neighbor = 2 and the right hand neighbor = 4.

This strategy will have to be used separately for each of x and y to produce a neighbor for (x, y) . The neighbor selection strategy can depend on the problem structure and usually affects the amount of time taken to produce a good solution.

Remark 2. The issue of temperature reduction has caused a great deal of debate. It is the case that the algorithm can perform rather poorly when the temperature is not reduced “properly.” Furthermore, finding the right strategy for temperature reduction needs experimentation. Two issues are relevant there.

- i. What should the number of iterations in one phase (remember, in each phase, the temperature is kept at a constant value) be?
- ii. How should the temperature be reduced?

The answer to the first question can vary from one iteration to a number that increases with the phase number P . One iteration would imply that the temperature is changed with every iteration, and that the phase has only one iteration. A simple linear rule for the number of iterations (n_P) in a given phase is:

$$n_P = a + bP,$$

where $a \geq 1$ and $b \geq 0$. This results an increasing number of iterations per phase.

To answer the second question, a general strategy to reduce temperature is to make the temperature a decreasing function of P . A possible rule, with $P = 0$ in the beginning, is

$$T_P = \frac{C}{\ln(2 + P)}. \quad (7.3)$$

A similar rule was analyzed in Hajek [67], where each phase had only 1 iteration. But there is no hard and fast rule for selecting C . Small values of C can cause the algorithm to get trapped in local optima, while large values of C can cause increased computer time. Why? (We will explain below). Alrefaei and Andradóttir [5] have proposed a simulated annealing algorithm where the temperature is not changed at all.

Remark 3. The expression $\exp(-\frac{\Delta}{T})$, with which U (the random number between 0 and 1) is compared, needs to be studied carefully. As T becomes small, this expression also becomes small. As the expression becomes small, so does the probability of accepting a worse solution. Equation (7.3) should also make it clear that as the number of iterations increases, T decreases. One should select a value for C in Equation (7.3) such that when P is 0, the expression (which is compared to U) is a number significantly larger than 0. If this number is too close to 0, then, as stated above, the probability of selecting a worse solution is very small at the start itself; this can cause the algorithm to get trapped in the nearest local optimum, which may be a poor solution. On the other extreme, a number close to 1 will result in an algorithm that wanders a great deal and needs increased computer time. If the temperature is not reduced, one essentially sees a “wanderer.”

Remark 4. When we use simulated annealing with a simulator, we assume that the estimate produced by the simulator is “close” to the actual function value. A difficulty with this assumption is: it is well-known that simulation estimates are not perfect; in other words they have noise, and so the estimates may not be close *enough*. For an analysis of the presence of noise in simulated annealing, the reader is referred to Chapter 12.

Remark 5. The reason for allowing the algorithm to move to *worse* solutions is to provide the algorithm with the opportunity of moving away from a local optimum and finding the global optimum. See Figure 7.1 (see page 98). A simulated annealing algorithm that finds X in Figure 7.1 may still come out of it and go on to find Y — the global optimum. Remember, if the algorithm moves out of a local optimum, and that local optimum happens to be the global optimum, that particular solution is not lost from our radar screen. The best solution is always retained; the steps in the algorithm ensure that. It may be a good idea to increase the accuracy of the function estimation process as the algorithm progresses. The accuracy can be improved by increasing the number of replications.

Remark 6: An important question is: how many phases to perform? The answer depends on how the temperature is reduced. When the temperature approaches small values at which no exploration occurs, the algorithm should be stopped. The rate at which the temperature is reduced depends on how much time is available to the user.

Example. We will demonstrate a few steps in the simulated annealing algorithm with an example. The example will be one of minimization. Consider a problem with two decision variables, x and y , each taking values from the set:

$$\{1, 2, 3, 4, 5, 6\}$$

We will assume that one iteration is allowed per phase. The temperature is decayed using the following rule:

$$T = 100/\ln(2+P), \text{ where } P \text{ denotes the number of phases performed so far.}$$

Let the current solution be:

$$\vec{x}_{current} = (3, 4).$$

The same solution is also the best solution; in other words:

$$\vec{x}_{best} = (3, 4).$$

Let $f(\vec{x}_{current})$ be 1400.

Step 1. P is equal to 0. Hence

$$T = 100/\ln(2) = 144.27.$$

Step 2. Let the selected neighbor of the current solution be:

$$\vec{x}_{new} = (2, 5),$$

where $f(\vec{x}_{new}) = 1350$.

Step 3. Since $f(\vec{x}_{new}) < f(\vec{x}_{best})$,

$$\vec{x}_{best} = \vec{x}_{new} = (2, 5).$$

Now:

$$\Delta = f(\vec{x}_{new}) - f(\vec{x}_{current}) = 1350 - 1400 = -50.$$

Since $\Delta < 0$, set:

$$\vec{x}_{current} = \vec{x}_{new} = (2, 5).$$

Step 4. Since only one iteration is to be done per phase, we move to Step 5.

Step 5. Increment P to 1. Since $P < P_{\max}$, re-calculate T :

$$T = 2/\ln(3) = 91.02.$$

Step 2. Let the selected neighbor of the current solution be:

$$\vec{x}_{new} = (1, 6),$$

where $f(\vec{x}_{new}) = 1470$.

Hence:

$$\Delta = f(\vec{x}_{new}) - f(\vec{x}_{current}) = 1470 - 1350 = 120.$$

Since $\Delta > 0$, generate U , a uniformly distributed random number between 0 and 1. Let the number U be 0.1. Now:

$$\exp\left(-\frac{\Delta}{T}\right) = \exp(-120/91.2) = 0.268.$$

Since $U = 0.1 < 0.268$, we will actually move into the worse solution. Hence:

$$\vec{x}_{current} = \vec{x}_{new} = (1, 6).$$

The algorithm can be terminated after a finite number of iterations.

3.2.2 The Genetic Algorithm

Survival of the fittest is a widely believed theory in evolution. It is believed that the reproduction process favors the fittest individual; in other words, the fittest individual reproduces more. As such, the fittest individuals get more opportunities — via mating — to pass their genes to the next generation. And an accepted belief is that this, in the next generation, produces individuals who are especially capable of reproduction. In other words those genes are passed that can produce healthy individuals capable of reproduction. We are talking of operations research, of course.

The genetic algorithm is a *very* popular meta-heuristic, although theoretically its convergence or success is hard to prove. It has been inspired by the evolutionary phenomenon that favors reproduction of individuals with certain traits. The genetic algorithm has been applied extensively in the industry with a good deal of success. Like other meta-heuristics, it needs only the value of the objective function. We will first present a *highly-simplified* version of the algorithm, and then discuss how the algorithm is motivated by the science of genetics. The algorithm is presented in terms of *minimization* of the objective function value.

Steps in the Genetic Algorithm. Let m denote the iteration number in the algorithm. Let m_{\max} denote the maximum number of iterations to be performed. This number has to be pre-specified, and there is no rule to find an optimal value for it. Usually, as discussed in the context of simulated annealing, the number is dictated by the permissible amount of computer time.

Step 1. Set $m = 1$. Select r initial solutions. The actual value of r is specified by the user. Of course r must be greater than 1. Preferably, all the r solutions should be relatively “good,” although this is not a requirement.

Step 2. Identify the best and the worst among the r solutions. Denote the best by \vec{x}_{best} and the worst by \vec{x}_{worst} . Select randomly a neighbor of \vec{x}_{best} and call it \vec{x}_{new} . Now, replace the worst solution by the new solution. In other words:

$$\vec{x}_{worst} \leftarrow \vec{x}_{new}.$$

Do not change any other solution, and go to Step 3. Notice that the solution set of r solutions has now changed.

Step 3. Increment m by 1. If $m > m_{\max}$, return \vec{x}_{best} as the best solution and STOP. Otherwise, go back to Step 2.

The value of r depends on the size of the problem. It seems intuitively that a large value for r may lead to better performance. A *practical* modification of Step 2 is to check if the new solution is better than the worst. If it is, the new solution should replace the worst; if not, one should return to Step 2 and generate a new neighbor of the best solution.

The format described above for the genetic algorithm is one *out of a very large number of variants* proposed in the literature [131]. In another format, in Step 2, the worst and the second worst solutions are replaced by neighbors of the best and the second best solutions. We must answer the question that is probably on your lips now. Why is this algorithm called a “genetic” algorithm?

In each generation (iteration), one has a set of individuals (solutions). The genetic algorithm makes use of the phenomenon seen in natural evolution. It allows only the fit individuals to reproduce. Fitness, in our context, is judged by how good the objective function is. The genetic algorithm also assumes that a good solution (read a potential *parent* with *strong reproductive features*) is likely to produce a good or a better neighbor (read a *child* that has good or even a better capability of reproducing).

In Step 2, the algorithm selects the best solution (the individual fittest to reproduce), selects its neighbor (allows it to reproduce — produce a child), and then replaces the worst solution (the individual least fit for reproduction dies) by the selected neighbor (child of a fit individual). In the process, in the next iteration (generation), the individuals generated are superior in their objective function values. This continues with every iteration producing a better solution.

Example. We next show the steps in the genetic algorithm via a simple example with two decision variables. The problem is one of minimization. Let the set, from which these decision variables assume values, be:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

Step 1. Set $m = 1$. Let us set r , the population size, to 4. Let us select the 4 solutions to be:

$$(2, 4), (1, 5), (4, 10), \text{ and } (3, 2),$$

and assume the respective function values to be:

$$34, 12, 45, \text{ and } 36.$$

Step 2. Clearly:

$$\vec{x}_{best} = (1, 5)$$

and

$$\vec{x}_{worst} = (4, 10).$$

Let the randomly selected neighbor (\vec{x}_{new}) of the best solution be $(2, 6)$. Replacing the worst solution by the new solution, our new population becomes:

$$(2, 4), (1, 5), (2, 6), \text{ and } (3, 2).$$

Then we go back to Step 2, and then perform more iterations.

A more refined version of the genetic algorithm uses the notions of *cross-over* and *mutation* to produce neighbors. In a “cross-over,” \vec{x}_{new} is generated by combining the best and the second best solutions; the latter are the two “parents” and the new solution is the “child.” Each solution is assumed to be composed of “genes.” So in (a, b) , a is the first gene and b the second. An example of a cross-over-generated child, whose parents are $(2, 4)$ and $(1, 5)$, is $(1, 4)$. In this child, the first gene, i.e., 1, comes from the second parent, while the second, i.e., 4, comes from the first. In a so-called mutation, some solution (often the best) is “mutated” by swapping “genes.” An example of a mutant of $(1, 5)$ is $(5, 1)$. In one variant of the genetic algorithm [71], several mutant and cross-over-generated children are produced, and much of the older generation (ninety percent) is replaced by the progeny.

3.2.3 Tabu Search

The tabu search algorithm, which originated from the work of Glover [51], has emerged as a widely-used meta-heuristic. It has been adapted to solving a large number of combinatorial-optimization problems.

A distinctive feature of the tabu search algorithm is the so-called tabu list. This is a list of *mutations* that are prohibited in the algorithm. Let us illustrate the idea of *tabu-search* mutations with a simple example.

As is perhaps clear from our discussions on simulated annealing and the genetic algorithm, in any meta-heuristic, we *move* from one solution to another. So if a problem with 2 decision variables is considered, in which both decision variables can assume values from the set {1, 2, 3}, a possible move is:

$$(2, 1) \xrightarrow{\text{to}} (3, 2).$$

In the above move, for the first decision variable, the “mutation” is

$$2 \xrightarrow{\text{to}} 3,$$

and for the second decision variable, the “mutation” is

$$1 \xrightarrow{\text{to}} 2.$$

The tabu list is a finite-sized list of mutations that keeps changing over time. We next present step-by-step details of a tabu search algorithm. The algorithm is presented in terms of *minimization* of the objective function value.

Steps in Tabu Search. Let m denote the iteration number in the algorithm. Let m_{\max} denote the maximum number of iterations to be performed. Like in the genetic algorithm, m_{\max} has to be pre-specified, and there is no rule to find an optimal value for it. Also, as stated earlier, this number is based on the available computer time to run the algorithm.

Step 1. Set $m = 1$. Select an initial solution \vec{x}_{current} randomly. Set:

$$\vec{x}_{\text{best}} \leftarrow \vec{x}_{\text{current}},$$

where \vec{x}_{best} is the best solution obtained so far. Create k empty lists. Fix the maximum length of each of these lists to r . Both k and r are pre-specified numbers. *One* list will be associated with *each* decision variable.

Step 2. Select a neighbor of \vec{x}_{current} , arbitrarily. Call this neighbor \vec{x}_{new} . If all the tabu lists are empty, go to Step 4. Otherwise, go to Step 3.

Step 3. Consider the move

$$\vec{x}_{\text{current}} \xrightarrow{\text{to}} \vec{x}_{\text{new}}.$$

For this move, there is one mutation associated with each decision variable. Check if any of these mutations is present in its respective tabu list. If the answer is no, go to Step 4. Otherwise, the move is considered tabu (illegal); go to Step 5.

Step 4. Enter each mutation associated with

$$\vec{x}_{\text{current}} \xrightarrow{\text{to}} \vec{x}_{\text{new}}$$

at the top of the respective tabu list. Then push down, by one position, all the entries in each list. If the tabu list has more than r members, as a result of this addition, *delete* the bottommost member. Then set

$$\vec{x}_{\text{current}} \leftarrow \vec{x}_{\text{new}}.$$

If the new solution is better than the best obtained so far, replace the best obtained so far by the current. That is if

$$f(\vec{x}_{\text{current}}) < f(\vec{x}_{\text{best}})$$

set:

$$\vec{x}_{\text{best}} \leftarrow \vec{x}_{\text{current}}.$$

Step 5. Increment m by 1. If m is greater than m_{\max} , STOP, and return \vec{x}_{best} as the solution. Otherwise, go to Step 2.

The tabu list is thus a list of mutations that have been made recently. Maintaining the list avoids the re-evaluation of solutions that were examined recently. This is perhaps a distinguishing feature of this algorithm. (It must be added, however, that in simulation optimization even if a solution is re-examined, we do not have to re-simulate the system. All the evaluated solutions can be kept in a binary tree — a computer programming construct. Once a solution is simulated, its objective function is fetched from the binary tree, and so re-simulation is not necessary.)

Examples of tabu lists. Consider a problem with two decision variables: p and q , where each decision variable can assume values from:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

Then the following listing is an example of what gets stored in a tabu list of length 3. Each element in this list is a move.

$$\begin{array}{ccc} p & & q \\ 2 \xrightarrow{\text{to}} 3 & 11 \xrightarrow{\text{to}} 12 \\ 4 \xrightarrow{\text{to}} 3 & 2 \xrightarrow{\text{to}} 3 \\ 3 \xrightarrow{\text{to}} 2 & 7 \xrightarrow{\text{to}} 8 \end{array}$$

At this point, if the mutation, $3 \xrightarrow{\text{to}} 2$, is generated for p in Step 3, it will not get stored since it is already in the tabu list. On the other hand, the mutation $3 \xrightarrow{\text{to}} 4$, is not tabu, and hence the new list for p will be:

$$\begin{array}{c} p \\ 3 \xrightarrow{\text{to}} 4 \\ 2 \xrightarrow{\text{to}} 3 \\ 4 \xrightarrow{\text{to}} 3 \end{array}$$

We need to make a few additional remarks.

Remark 1. An alternative interpretation of what is tabu — an interpretation commonly found in the literature — is to store the entire move as a mutation. In such an implementation, only one tabu list is maintained for the entire problem, and the entire move is treated as a mutation. An example of a tabu list for this implementation is:

$$\begin{aligned}(3, 4) &\xrightarrow{\text{to}} (4, 5) \\ (2, 6) &\xrightarrow{\text{to}} (3, 1) \\ (4, 2) &\xrightarrow{\text{to}} (2, 5)\end{aligned}$$

Remark 2. In yet another interpretation of what is tabu, whenever a move is selected, the reverse move is entered in the tabu list. For instance, if the algorithm makes the following move

$$(3, 4) \xrightarrow{\text{to}} (4, 5),$$

then the move

$$(4, 5) \xrightarrow{\text{to}} (3, 4),$$

is stored in the tabu list. This prevents cycling.

Remark 3. Our strategy for declaring a move to be tabu (in Step 3) may be overly restrictive. One way to work around this is to add a so-called aspiration criterion. A simple aspiration criterion, cited in Heragu [71], determines if the selected neighbor is actually better than the best solution so far. If the answer is yes, the tabu list consultation steps are skipped, the newly selected neighbor is treated as \vec{x}_{current} , and then one goes to Step 5. This would require a slight modification of Step 2, as shown below.

Step 2. Select a neighbor of \vec{x}_{current} arbitrarily. Call this neighbor \vec{x}_{new} .
If

$$f(\vec{x}_{\text{new}}) < f(\vec{x}_{\text{best}})$$

set:

$$\vec{x}_{\text{best}} \leftarrow \vec{x}_{\text{new}},$$

$$\vec{x}_{\text{current}} \leftarrow \vec{x}_{\text{new}},$$

and go to Step 5.

Otherwise, check to see if all the tabu lists are empty. If yes, go to Step 4 and if no, go to Step 3.

Of course, the aspiration criterion could be less strong. For instance, an aspiration criterion could determine how many of the new mutations are in

their respective tabu lists. If this number is less than k , we could consider the new neighbor as non-tabu and accept it.

Remark 4. The length of the tabu list, r , should be a fraction of the problem size. In other words, for larger problems, larger tabu lists should be maintained. Very small tabu lists can cause cycling, i.e., the same solution may be visited repeatedly. Very large tabu lists can cause the algorithm to wander too much!

Remark 5. Tabu search has already appeared in some commercial simulation-optimization packages [53].

As is perhaps obvious from these remarks, there are various ways of implementing tabu search. There is a vast literature on tabu search, and the reader should consult Glover and Laguna [54] for more details.

3.2.4 A Learning Automata Search Technique

In this section, we present a meta-heuristic based on the theory of games and artificial intelligence. We refer to it as the Learning Automata Search Technique (acronym LAST).

Unlike many other meta-heuristics, LAST is *not* a local-search technique. In other words, it does not search for a new solution in the *neighborhood* of the current solution. Rather, it jumps around a great deal retaining the best solution in its memory. *In the beginning, the jumping around is random, but gradually it starts zeroing in on what it perceives as the optimal solution(s).*

See Narendra and Thathachar [120] for an excellent introduction to the theory of learning automata and the general framework. Narendra and Wheeler [121] and Thathachar and Ramakrishnan [172] have obtained important results in the context of identical payoff games that form the basis for automata theory. The algorithm that we present here is due to Thathachar and Sastry [173].

Let $(x(1), x(2), \dots, x(k))$ denote k decision variables (parameters), where $x(i)$ takes values from the finite set $\mathcal{A}(i)$. Thus $\mathcal{A}(i)$ denotes the finite set of values that are permitted for decision variable i . The algorithm maintains a database of probabilities. In each iteration of the algorithm, a solution is selected and evaluated. The selection is done with the help of the probabilities. Let $p^m(i, a)$ denote the probability of selecting the value a for decision variable i in the m th iteration of the algorithm.

The probabilities can be viewed as the knowledge base of the algorithm. The objective function value of the selected solution is used to *update* the probabilities. This kind of updating is interpreted as the “algorithm getting smarter.” The idea underlying the updating is quite simple. Those values that produce “good” objective function values are rewarded via an increase in their probabilities (of getting selected in the future), and those values that

produce “poor” objective function values are punished by a reduction in their probabilities. In automata theory parlance, the objective function value goes by the name *feedback*. . Figure 7.2 is a schematic that shows the working mechanism of the algorithm. It also reinforces the connection with artificial intelligence.

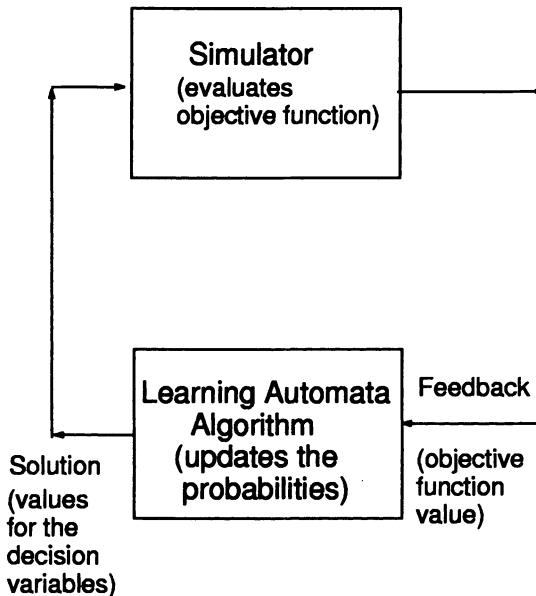


Figure 7.2. The Mechanism Underlying LAST

At the beginning, the algorithm starts with an open mind. This literally means that the same probability is assigned to each value associated with a given decision variable. Mathematically, this means that:

$$p^1(i, a) = \frac{1}{|\mathcal{A}(i)|} \text{ for } i = 1, 2, \dots, k, \text{ and } a \in \mathcal{A}(i).$$

In the above, $|\mathcal{A}(i)|$ denotes the number of elements (cardinality) of the set $\mathcal{A}(i)$. The updating scheme of the algorithm ensures that:

$$\sum_{a \in \mathcal{A}(i)} p^m(i, a) = 1 \text{ for all } i, a, m.$$

Since the probabilities are updated using the objective function values, the objective function value has to be *normalized* between 0 and 1. This is achieved as follows:

$$F = \frac{G - G_{\max}}{G_{\max} - G_{\min}}.$$

In the above, G denotes the actual objective function value, F denotes the *normalized* objective function value, G_{\max} denotes the maximum objective function value that can possibly be obtained, and G_{\min} denotes the lowest objective function value that can be obtained. Knowledge of G_{\max} and G_{\min} is necessary for this algorithm.

The algorithm also stores in its database the best (normalized) objective function value — obtained so far — for each (i, a) pair with $i = 1, 2, \dots, k$ and $a \in \mathcal{A}(i)$. This value will be denoted by $B(i, a)$. The probabilities are updated gradually with a step size, which will be denoted by μ . The algorithm is presented in terms of maximizing the objective function value.

Steps::

Step 1. Set the number of iterations, m , to 0. Let k denote the number of decision variables. Set $p^m(i, a) = 1/|\mathcal{A}(i)|$ and $B(i, a) = 0$ for $i = 1, 2, \dots, k$ and every $a \in \mathcal{A}(i)$. Assign suitable values to $\mu, G_{\max}, G_{\min}, m_{\max}$, and F_{best} . F_{best} will denote the best *normalized* objective function value obtained so far, and should equal a small quantity. G_{\max} and G_{\min} should denote, respectively, the maximum and minimum possible objective function values (before normalization) that can be obtained from the problem. The maximum number of iterations to be performed is denoted by m_{\max} . The symbol μ will denote a step size used to update the probabilities.

Step 2. For $i = 1, 2, \dots, k$, select a value $x(i)$ from $\mathcal{A}(i)$ with probability $p^m(i, x(i))$. Let the solution be denoted by \vec{x} .

Step 3a. Evaluate the objective function value associated with \vec{x} . Let the value obtained be G .

Step 3b. Calculate the normalized objective function value using

$$F = \frac{G - G_{\min}}{G_{\max} - G_{\min}}.$$

If $F > F_{best}$, set

$$\vec{x}_{best} \leftarrow \vec{x}.$$

Step 4. Set $i = 1$.

Step 5. For $a = 1, 2, \dots, |\mathcal{A}(i)|$, do:

If $B(i, a) < B(i, x(i))$, set

$$p^{m+1}(i, a) \leftarrow p^m(i, a) - \mu[B(i, x(i)) - B(i, a)]p^m(i, a).$$

If $B(i, a) > B(i, x(i))$, set

$$p^{m+1}(i, a) \leftarrow p^m(i, a) + \mu[B(i, a) - B(i, x(i))] \frac{[1 - p^m(i, a)]p^m(i, x(i))}{|\mathcal{A}(i)| - 1}.$$

Step 6. Set

$$p^{m+1}(i, x(i)) \leftarrow 1 - \sum_{\substack{a=|\mathcal{A}(i)| \\ a \neq x(i); a=1}} p^{m+1}(i, a).$$

If $i < k$, increment i by 1, and go back to Step 5. Otherwise, increment m by 1, and go to Step 7.

Step 7. If $m < m_{\max}$, set $i = 1$, and go to Step 8a; otherwise STOP returning \vec{x}_{best} as the solution.

Step 8a. (Updating **B**) If $F \leq B(i, x(i))$, go to Step 8b. Otherwise set

$$B(i, x(i)) \leftarrow F,$$

and then go to Step 8b.

Step 8b. If $i < k$, increment i by 1 and go to Step 8a; else return to Step 2.

Remark 1. The algorithm is likely to re-evaluate the same solution a number of times. While this is not a problem, a practical difficulty with the description given above is that one may have to re-simulate a solution that was simulated earlier. This difficulty can be circumvented by maintaining a dynamic memory structure which stores all the solutions that were evaluated in the past. When a solution that was tried earlier is re-visited by the algorithm, its objective function value can then be fetched from the memory. This is critical for simulation optimization, since each function evaluation may take considerable computer time.

Remark 2: The size of the **B** matrix and the number of probabilities are very small compared to the solution space. For instance, in a problem with 10 decision variables with 2 values for each variable, the solution space is 2^{10} , but the number of elements in the **B** matrix is only $(10)(2) = 20$, and the number of probabilities is also 20. The **B** matrix can also be stored as a binary tree — thereby reducing the memory requirement further. This is possible because as long as a pair (i, a) is not tried, its value is 0, and hence the matrix is sparse.

Remark 3. The problem of having to store too many probabilities (for large problems) can sometimes be overcome by using what is called a Parameter Dependence Network (PDN). The latter uses Bayesian Learning theory. The PDN was developed in Sarkar and Chavali [152]. The PDN can be integrated with the simulator to *adaptively* search the decision variable space. Becoming

self-adjusting can perhaps be viewed as the highest goal of an “artificial intelligence” method. Treatment of this topic in more detail requires discussion of several topics such as Bayesian Learning, Gibbs sampling, and the so-called entropy measures. All this is beyond the scope of this book, but the interested reader is referred to the work of Sarkar and Chavali [152].

Remark 4. G_{\min} and G_{\max} are actually the greatest lower bound and the least upper bound, respectively, for the objective function value that is to be maximized. If these are not known, any upper bound and any lower bound for the objective function can be used. What is essential is that we restrain the value of the objective function value to the interval $(0, 1)$. However, if the G_{\min} and G_{\max} are much smaller and much larger respectively than the greatest lower bound and the least upper bound, the updating can become very slow. The following example should illustrate the steps of the algorithm.

Example. Consider a small problem with two decision variables where each decision variable has 3 values. We join the “learning process” after m iterations. As a result, the \mathbf{B} matrix will not be empty. Let us assume that the \mathbf{B} matrix after m iterations is:

$$\begin{bmatrix} 0.1 & 0.2 & 0.4 \\ 0.1 & 0.3 & 0.4 \end{bmatrix}.$$

Let the value vector, \vec{x} , selected in the $(m + 1)$ th iteration, be $\{2, 1\}$. In other words, the second value for the first decision variable and the first value for the second have been selected. Let the objective function value, F , be 0.1. F_{best} , it should be clear from the \mathbf{B} matrix, is assumed to be 0.4. We next show all the calculations to be performed at the end of this iteration.

Now from Step 5, since $B(1, 1) < B(1, 2)$, $p(1, 1)$ will decrease and will be updated as follows.

$$p^{m+1}(1, 1) = p^m(1, 1) - \mu[B(1, 2) - B(1, 1)]p^m(1, 1).$$

And $p(1, 3)$ will increase since $B(1, 3) > B(1, 2)$, and the updating will be as follows:

$$p^{m+1}(1, 3) = p^m(1, 3) + \mu[B(1, 3) - B(1, 2)]\frac{[1 - p^m(1, 3)]p^m(1, 2)}{3 - 1}.$$

And finally from Step 6, $p(1, 2)$ will be updated as follows:

$$p^{m+1}(1, 2) = 1 - p^{m+1}(1, 1) - p^{m+1}(1, 3).$$

Similarly, we will update the probabilities associated with the second decision variable. Here both $p(2, 2)$ and $p(2, 3)$ will increase since both $B(2, 2)$ and $B(2, 3)$ are greater than $B(2, 1)$. The updating equations are as follows:

$$p^{m+1}(2, 2) = p^m(2, 2) + \mu[B(2, 2) - B(2, 1)]\frac{[1 - p^m(2, 2)]p^m(2, 1)}{3 - 1}$$

and

$$p^{m+1}(2, 3) = p^m(2, 3) + \mu[B(2, 3) - B(2, 1)] \frac{[1 - p^m(2, 3)]p^m(2, 1)}{3 - 1}.$$

The third probability will be normalized as follows:

$$p^{m+1}(2, 1) = 1 - p^{m+1}(2, 2) - p^{m+1}(2, 3).$$

Since both $B(1, 2)$ and $B(2, 1)$ are greater than F , the new response will not change the \mathbf{B} matrix. Thus the new \mathbf{B} matrix will be identical to the old. And now we are ready to begin the $(m + 2)$ th iteration.

The game-theoretic connection of this algorithm, although not essential for understanding it, is quite interesting. One can think of an iteration as a “game” between the decision variables. Each decision variable — in this setting — is a “player” trying to find its best “action” (value), and the objective function value is the “common payoff.” We will not pursue this analogy any further, although we should point out that this analogy has actually been used to prove the convergence of this algorithm in [173].

3.2.5 Other Meta-Heuristics

The literature talks of other meta-heuristics such as the stochastic ruler method (see Yan and Mukai [190] and Alrefaei and Andradóttir [6]) and the stochastic comparison method of Gong, Ho, and Zhai [58]. Use of multiple meta-heuristics on the same problem is not uncommon; the reason is that it is difficult to rely on one algorithm for any given problem.

3.2.6 Ranking and selection & meta-heuristics

A careful statistical comparison of a few solutions is an integral part of some meta-heuristics such as the genetic algorithm, LAST, and simulated annealing. In each of these, one has to compare two or more solutions. We know of one instance where these features were incorporated in a meta-heuristic approach. See Boesel and Nelson [22] for details. Combination of this kind increases the power of the meta-heuristic and makes its results more reliable.

Marriage of meta-heuristics with such statistical methods may be the key to developing robust meta-heuristics. In fact, it is likely that these methods may dramatically alter the number of replications needed and thereby reduce the computational time of a meta-heuristic.

4. Hybrid solution spaces

Oftentimes, in the real world, we face problems with a hybrid solution space, i.e., a solution space in which some decision variables are continuous and some are discrete. In tacking such problems, one has two options:

Option 1. Discretize the solution space of the continuous decision variables using a reasonably-fine grid, and then use discrete optimization algorithms. For instance if x is a continuous decision variable, where $x \in [1, 2]$, a “discretized” x would assume values from the following set:

$$\{1, 1.1, 1, 2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2\}.$$

Option 2. Treat the discrete decision variables as though they were continuous decision variables, and use continuous optimization algorithms. The values of the discrete decision variables would have to be rounded off to the nearest discrete value in the optimization process.

Both methods have drawbacks. Notably, Option 2 is known to have problems in practice. The size of the solution space in Option 2 can become quite large depending on how fine the grid is, and depending on the range of acceptable values for the continuous decision variable. In practice, both options must be tried in an effort to seek the best results.

5. Concluding Remarks

The importance of parametric *simulation* optimization stems from the fact that sometimes it is the only approach to solving optimization problems in complex *stochastic* systems, whose objective functions are unknown / difficult to characterize.

We would like to note that almost all the approaches discussed in this chapter are semi-classical. In the continuous case, one may have to run the algorithm many times — *starting* at a different point on each occasion —, and in the discrete case, one may have to use more than one meta-heuristic to obtain high-quality solutions. The advice of Confucius (in the beginning of this chapter) should serve us well.

6. Bibliographic Remarks

Some good references for review material on simulation optimization are Andradottir [8], Fu [47], Carson and Maria [33], and Kleijnen [93]. The material presented in this chapter comes from a large number of sources. We were not able to discuss a number of related works, some of which are: sample path optimization (Plambeck *et al.* [135]), perturbation-related work (Ho and Cao [75]), weak derivative estimation (Pflug [130]), the score function method (Rubinstein and Shapiro [146]), and the frequency-domain method for estimating derivatives and Hessians (Jacobson and Schruben [82]). Some of these methods are not completely model-free.

Continuous parametric optimization using gradient descent was discussed. The finite difference method of computing the derivative, which stems from the classical definition of the derivative in the seventeenth century, is very time

consuming — when used in a gradient-descent algorithm on a large number of decision variables. The number of function evaluations per iteration is directly proportional to the number of decision variables in finite difference approaches.

A breakthrough was provided by Spall [162], who showed that in problems with multiple decision variables, the gradient-descent method could actually use a different mechanism to calculate the derivative — a mechanism that has a lower computational burden in comparison to the finite difference method. Spall's method of simultaneous perturbation works even in the presence of noise in the objective function. Noise, it must be noted, cannot be avoided in simulation-based optimization.

This is naturally an exciting area for further research. In this chapter, we have concentrated on model-free methods, which do not need any knowledge of the form of the objective function.

Parametric simulation optimization for the discrete case is actually harder than the continuous case. Much research in this area has resulted from the adaptation of meta-heuristics used for deterministic systems. For a general discussion on meta-heuristics, a nice text is Pham and Karaboga [131]. It is important to study the effect of simulation noise in the objective function evaluation in such an adaptation. In case of simulated annealing, Homem-de-Mello [79], Fox and Heine [46], Gelfand and Mitter [50], Alrefaei and Andradóttir [5], and Gosavi [60] have studied the effects of noise.

Each meta-heuristic has its own distinctive traits. Simulated annealing mimics a metallurgical process, while the genetic algorithm uses a biological evolution strategy. The learning automata search technique relies on learning theory and game theory, but it is not a local search technique. The tabu search method is not a stochastic search, unlike some other methods, and it uses a distinctive mechanism of maintaining a tabu-list of solutions.

Simulated annealing has its origins in the work of Metropolis *et al.* [115] in the fifties, but it was only in the eighties that it was used as an optimization method by Kirkpatrick, Gelatt, and Vecchi [92]. A very large number of papers have resulted from this work. Homem-de-Mello [79] has proposed a “variable sampling.” Fox and Heine [46] have suggested a “quicker” version of simulated annealing, while Alrefaei and Andradóttir [5] have proposed a “constant-temperature” simulated annealing algorithm. Textbook treatment to this topic can be found in Van Laarhoven and Aarts [179].

The genetic algorithm originated from the work of Holland [77], while tabu search was conceived by Glover [51]. The learning automata search technique originated from the work of Thathachar and Sastry [173]. Thathachar and Sastry analyzed the method on problems whose objective functions are non-stationary. The method was used as an optimization tool in Tezcan and Gosavi [171]. Sarkar and Chavali [152] have suggested a Parameter Dependence Network that can be combined with the method to improve its performance and efficiency.

Meta-heuristics have become popular in the applied world because of their impressive success on large-scale combinatorial optimization problems, on which theoretically convergent methods such as branch-and-bound and dynamic programming break down. Although these methods were met with disbelief in the beginning, they are now finding wider acceptance perhaps because of these success stories. Furthermore, convergence analysis of many of these methods has increased their stature somewhat. One sees sessions in INFORMS (Institute for Operations Research and Management Science) meetings devoted to meta-heuristic related topics. Several papers related to meta-heuristics have started appearing in journals such as *Operations Research*, the *INFORMS Journal on Computing*, *Management Science*, and *Institute of Industrial Engineering Transactions*. Simulation-optimization modules have started appearing in commercial packages such as PROMODEL [68] and ARENA. OPTQUEST [53] is a package that uses tabu search and neural networks in simulation optimization.

It cannot be denied that research related to meta-heuristics will continue to thrive because they have the potential to solve large-scale problems. In case of simulation optimization, there is an acute need for methods that need fewer function evaluations because each function evaluation is a time-consuming affair. At the same time, these methods must ensure that the noise levels do not disturb the normal path of the algorithm.

Combining meta-heuristics with statistically-meaningful procedures such as ranking and selection (Goldsman *et al.* [57], Nelson [124], and Goldsman and Nelson [55]) may prove to be an important topic for further research. Another important issue is the sensitivity of these methods to tuning decision variables (such as the temperature in simulated annealing etc). High sensitivity implies that unless the right values are selected, the algorithm may perform poorly. Every algorithm has its own tunable decision variable(s). The search for robust meta-heuristics continues.

7. Review Questions

1. Name the researchers who invented: a) Simultaneous perturbation, b) Tabu Search, and c) Simulated Annealing.
2. Discuss an advantage of simultaneous perturbation over finite difference perturbation?
3. Can simultaneous perturbation get stuck in local optima?
4. Define what is meant by a model-free method of parametric optimization?
5. Given two parents (1, 4, 5) and (2, 7, 8), generate 4 children using cross-over. Generate two mutants each from the two parents.

6. Devise your own mechanism for generating mutants and doing cross-over. See [131, 71] for some ideas.
7. Define a neighbor in the context of meta-heuristics.
8. “The learning automata search technique different from other techniques discussed in this chapter.” Explain the above in the context of neighborhoods.
9. What role can the aspiration criterion play in the tabu search method?
10. Simulate a single server single channel queuing system using C or any programming language that you are comfortable with (see Law and Kelton [102]), and then optimize for the maximum number of customers to be allowed into the system — using a meta-heuristic. This is commonly known as the *admissions control problem*. Assume the cost of refusing one customer to be \$10 and the cost of having an average queue length of 1 to be \$5. Assume distributions and parameters for arrival rates and service rates. Make sure that the arrival rate is smaller than the service rate.
11. Use each meta-heuristic discussed in this chapter to minimize the objective function value. The two decision variables assumes values from the set:

$$\{0, 1, 2, 3, 4\}.$$

The objective function values, obtained from a simulator, are given in Table 7.1.

12. Use simultaneous perturbation to minimize the objective function given below.

$$f(x, y) = 0.1(x - 3)^2 + 0.9(y - 8)^2 + w$$

where w , the simulation noise, is a random variable. Any time the function is evaluated, w should be generated from the uniform distribution $U(1, 1.5)$.

Table 7.1. Table shows values of objective function

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 0$	13.1	10.2	9.3	10.4	13.1
$y = 1$	8.2	5.2	4.1	5.2	8.1
$y = 2$	5.2	2.0	1.1	2.0	5.3
$y = 3$	4.1	1.1	0	1.1	4.1
$y = 4$	5.1	2.4	1.2	2.3	5.1

Chapter 8

CONTROL OPTIMIZATION WITH STOCHASTIC DYNAMIC PROGRAMMING

You can never plan the future by the past.

— Edmund Burke (1729-1797)

1. Chapter Overview

This chapter focuses on a problem of control optimization — namely, the Markov decision problem. Our discussions will be at a very elementary level, and we will not attempt to prove any theorems.

The major aim of this chapter is to introduce the reader to *classical* dynamic programming in the context of solving Markov decision problems. In the next chapter, the same ideas will be presented in the context of *simulation-based* dynamic programming.

The main concepts presented in this chapter are 1) Markov chains, 2) Markov decision problems, 3) semi-Markov decision problems, and 4) classical dynamic programming methods.

We recommend that you read Chapter 2 — for a review of the notation used in this book — before reading this chapter.

2. Stochastic processes

We begin with a discussion on *stochastic processes*. A stochastic (or random) process, roughly speaking, is something that has a property which changes randomly with time. We refer to this changing property as the **state** of the stochastic process. A stochastic process is usually associated with a stochastic **system**. Read Chapter 4 for a definition of a stochastic system . The concept of a stochastic process is best understood with an example.

Consider a queue of persons that forms in a bank. Let us assume that there is a single server (teller0 serving the queue. See Figure 8.1. The queuing system is an example of a stochastic system. We need to investigate further into the nature of this queuing system to identify properties, associated with the queue, that change randomly with time.

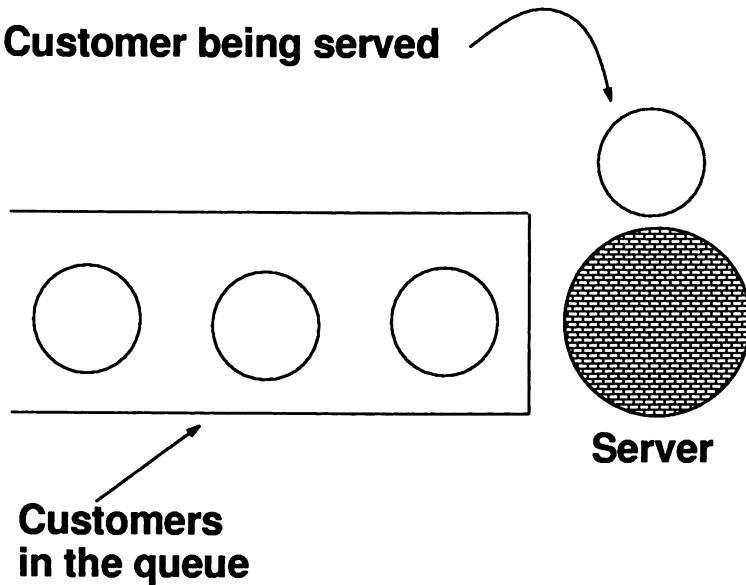


Figure 8.1. A single-server queue

Let us denote

- the number of customers in the queue at time t by $X(t)$ and
- the number of busy servers at time t by $Y(t)$.

Then, clearly, $X(t)$ changes its value from time to time and so does $Y(t)$. By its definition, $Y(t)$ equals 1 when the teller is busy serving customers, and equals 0 when it is idle.

Now if the state of the system is recorded after *unit time*, $X(t)$ could take on values such as:

$$3, 3, 4, 5, 4, 4, 3 \dots$$

The set $\{X(t)|t = 1, 2, \dots, \infty\}$, then, defines a stochastic process. Mathematically, the *sequence* of values that $X(t)$ assumes in this example is a stochastic process.

Similarly, $\{Y(t)|t = 1, 2, \dots, \infty\}$ denotes another stochastic process underlying the *same* queuing system. For example, $Y(t)$ could take on values

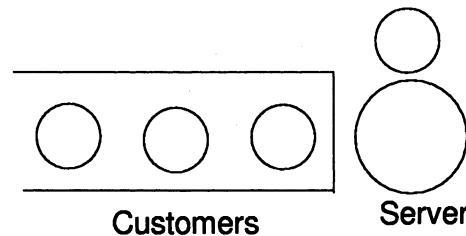
such as:

$$0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, \dots$$

This example should make it clear that with a *given* stochastic system, more than one stochastic process may be associated. The stochastic process X and the stochastic process Y differ in their definition of the **state** of the system. For X , the state is defined by the number of customers in the queue and for Y , the state is defined by the number of busy servers.

An analyst selects the stochastic process that is of interest to her. For instance, an analyst interested in studying the *utilization* of the server (that is, proportion of time the server is busy) will choose Y , while the analyst interested in studying the *length of the queue* will choose X . See Figure 8.2 for a pictorial explanation of the word “state.”

State = 3



State = 2

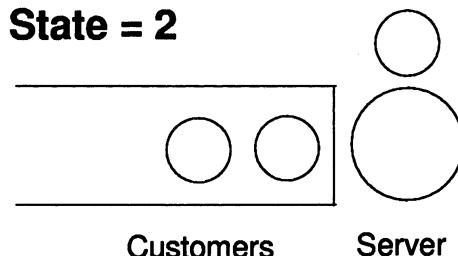


Figure 8.2. The figure shows the queue in two different *states*. The state is defined by the number in the queue.

In general, choosing the appropriate definition of the state of a system is a part of “modeling.” The state must be defined in a manner suitable for the optimization problem under consideration. To understand this better, consider the following definition of state.

Let $Z(t)$ denote the total number of persons in the queue with *black hair*. Now, although $Z(t)$ is a mathematically perfect example of a stochastic process, this definition may contain very little information of use — when it comes to controlling systems in a cost-optimal manner!

We have defined the state of a stochastic process. Now, it is time to closely examine an important stochastic process, namely, the Markov process.

3. Markov processes, Markov chains and semi-Markov processes

A Markov process is of special interest to us because of its widespread use in studying real-life systems. In this section, we will study some of its salient features.

A stochastic process, usually, visits more than one state. We will assume throughout this book that the set of states visited by the stochastic process is a finite set denoted by \mathcal{S} .

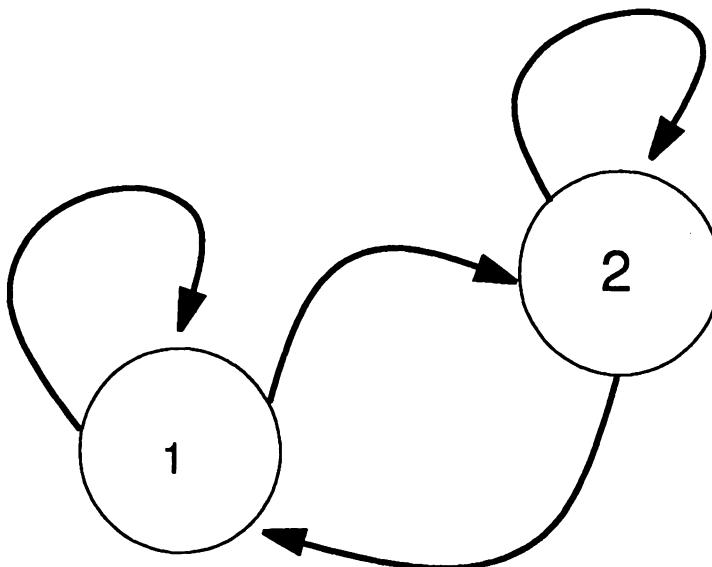


Figure 8.3. Schematic of a two-state Markov chain, where circles denote states.

An important property of a Markov process is that it *jumps* regularly — in fact it jumps after **unit time**. (Some authors do not use this “unit time” convention, and we will discuss this matter in detail later.) What this means is that, after unit time, the system either transitions (moves) to a new state or else the system returns to the current state.

To understand this jumping phenomenon, consider Figure 8.3. The figure shows two states, which are denoted by circles, numbered 1 and 2. The arrows show the possible ways of jumping. This system has two states: 1 and 2. Assuming that we first observe the system when it is in state 1, it may follow

the trajectory given by:

$$1, 1, 2, 1, 1, 1, 2, 2, 1, 2, \dots$$

A state transition in a Markov process is usually a probabilistic, i.e., random affair. Consider the Markov process in Figure 8.3. Let us further assume that in its first visit to state 1, from state 1 the system jumped to state 2. In its next visit to state 1, the system may not jump to state 2 again; it may jump back to state 1. This should make it clear that the transitions in a Markov chain are “random” affairs.

We now need to discuss our convention regarding the time needed for one jump (transition). It is the case that how much time is spent in one transition, in a Markov process, is irrelevant to its analysis. As such, even if the time is not always unity, or even if it is not a constant, we **assume** it to be unity for our analysis — for the sake of convenience.

When we study real-life systems using Markov processes, it usually becomes necessary to define a performance metric for the real-life system. It is in this context that one has to be careful with how this convention is interpreted. A common example of a performance metric is: average reward *per unit time*. In the case of a Markov process, the phrase “per unit time” in the definition of average reward actually means “per jump” or “per transition.” (In the so-called Semi-Markov process that we will study later, one has to be more careful.)

In a Markov process, the probability that the process jumps from a state i to a state j does **not** depend on the states visited by the system *before* coming to i . This is called the **memoryless** property. This property distinguishes a Markov process from other stochastic processes and as such needs to be understood clearly.

Because of the memoryless property, one can associate a probability with a transition from a state i to a state j , that is,

$$i \longrightarrow j.$$

We denote the probability of this transition by $P(i, j)$. We explain this idea with an example.

Consider a Markov chain with three states, numbered 1, 2, and 3. The system starts in state 1 and traces the following trajectory:

$$1, 3, 2, 1, 1, 1, 2, 1, 3, 1, 1, 2, \dots$$

Let us assume that the probability $P(3, 1) = 0.2$ and $P(3, 2) = 0.8$.

When the system visits 3 for the first time, let us assume that it jumps to 2. Now, from the data supplied above, the probability of jumping to 2 is 0.8, and that of jumping to 3 is 0.2. Now, when it *revisits* 3, the probability of jumping to 2 still has to be 0.8, and that of jumping to 3 still has to be 0.2.

The point we are trying to make here is when the system comes to a state i , where it jumps to depends on the transition probabilities: $P(i, 1)$, $P(i, 2)$ and $P(i, 3)$ — quantities which are not affected by the states visited before coming to i . Thus, when it comes to jumping to a new state, the process does not “remember” what states it has had to go through in the past. The next state to which it jumps depends only on the current state (say i) and the probabilities of jumping from that state to other states — $P(i, 1)$, $P(i, 2)$ and $P(i, 3)$.

In general, when the system is ready to leave state i , to which state it jumps depends only on $P(i, j)$ if j is the state to which it jumps. Furthermore, $P(i, j)$ is completely independent of where the system has been — before coming to i .

We now give an example of a process that is non-Markovian. Assume that a process has three states, numbered 1, 2, and 3. $X(t)$, as before, denotes the system state at time t . Let us assume that the law governing this process is given by:

$$\text{Prob}\{X(t + 1) = j | X(t) = i, X(t - 1) = k\} = f(i, k, j), \quad (8.1)$$

where $f(i, k, j)$ is a probability that depends on i , k , and j . This implies that if the process is in state i at time t (notice that $X(t) = i$ in the equation above is supposed to mean exactly this), and if it was in state k at time $(t - 1)$, then the probability that the next state (i.e., the state visited at time $(t + 1)$) will be j is a function of i , k , and j . In other words, at any point of time, the past (i.e., $X(t - 1)$) will affect its future course.

The process described above is not a Markov process. In this process, the state of the process at time $(t - 1)$ *does* affect the probabilities, at time $(t + 1)$, of going to other states. The path of this stochastic process is thus dependent on its past — not a whole lot of the past, but some of its past.

The Markov process, on the other hand, is governed by the following law:

$$\text{Prob}\{X(t + 1) = j | X(t) = i\} = f(i, j), \quad (8.2)$$

where $f(i, j)$ is the probability that the next state is j given that the current state is i . Also $f(i, j)$ is a constant for given values of i and j .

Carefully note the difference between Equation (8.2) and Equation (8.1). Where the process resides *one step* before its current state is immaterial in a Markov process. It should be obvious that in the Markov-process case, the transition probability (probability of going to one state to another in the stochastic process **in one step**) depends on two quantities — the present state (i) and the next state (j). In a non-Markovian process, such as the one defined by Equation (8.1), the transition probability depended on the current state (i), the next state (j), and the previous state (k). An implication is that even if both the processes have the same number of states, we will have to deal with more probabilities in the two-step stochastic process than in the Markov process.

The quantity $f(i, j)$ is an element of a two-dimensional matrix. Note that $f(i, j)$ is actually $P(i, j)$ — the one-step transition probability of jumping from i to j — that we have defined earlier.

All the transition probabilities of a Markov process can be conveniently stored in the form of a matrix. We refer to this matrix as the one-step transition probability matrix or simply the transition probability matrix, which is usually abbreviated as TPM. An example of a TPM, \mathbf{P} , with 3 states is:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.4 & 0.2 & 0.4 \\ 0.6 & 0.1 & 0.3 \end{bmatrix}. \quad (8.3)$$

$P(i, j)$ here denotes the (i, j) th element of the matrix, \mathbf{P} , that is, the element in the i th row and the j th column of \mathbf{P} . In other words, $P(i, j)$ denotes the one-step transition probability of jumping from state i to state j . Thus, for example, $P(3, 1)$, which is 0.6 in the *TPM* given above, denotes the one-step transition probability of jumping from state 3 to state 1.

We will also assume that a finite amount of time is taken in any transition and that *no time is actually spent in a state*. This is one convention (there are others), and we will stick to it in this book. Also, by our convention, the time spent in a transition is unity (1).

In summary, a Markov process possesses three important properties:

1. the jumpy property,
2. the memoryless property, and
3. the unit time property (by our convention).

3.1. Markov chains

A Markov chain can be thought of as an entity that accompanies a stochastic process. Examples of stochastic processes that are kept company by Markov chains are the Markov process, the semi-Markov process, and the partially observable Markov process.

We associate a unique TPM with a given Markov chain. It should be kept in mind that the Markov chain (not the Markov process) contains no information about how much time is spent in a given transition. It contains information about the transition probabilities, however.

Example 1. Consider Figure 8.4. It shows a Markov chain with two states, shown by circles, numbered 1 and 2. The arrow indicates a possible transition, and the number on the arrow denotes the probability of that transition. The figure depicts the following facts. If the process is in state 1, it goes to state 2 with a probability of 0.3, and with a probability of 0.7, it stays in the same state

(i.e., 1). Also, if the process is in state 2, it goes to state 1 with a probability of 0.4 and stays in the same state (i.e., 2) with a probability of 0.6. The TPM of the Markov chain in the figure is therefore

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

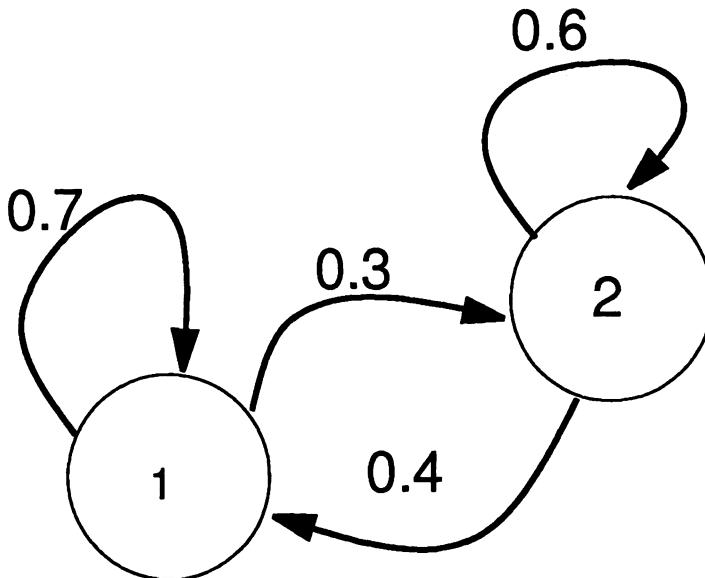


Figure 8.4. Schematic of a two-state Markov chain, where circles denote states, arrows depict possible transitions, and the numbers on the arrows denote the probabilities of those transitions.

Figures 8.5 and 8.6 show some more examples of Markov chains with 3 and 4 states respectively. In this book, we will consider Markov chains with a finite number of states.

Estimating the values of the elements of the TPM is often quite difficult. This is because, in many real-life systems, the TPM is very large, and evaluating any given element in the TPM requires the setting up of complicated expressions, which may involve multiple integrals. In subsequent chapters, this issue will be discussed.

3.1.1 n -step transition probabilities

The n -step transition probability of going from state i to state j is defined as the probability of starting at state i and being in state j after n steps (or jumps/transitions). From the one-step transition probabilities, it is possible to construct the two-step transition probabilities, the three-step transition proba-

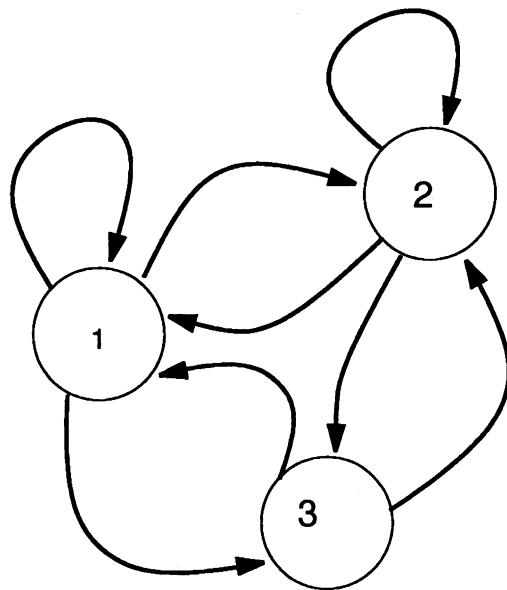


Figure 8.5. Schematic of a Markov chain with 3 states

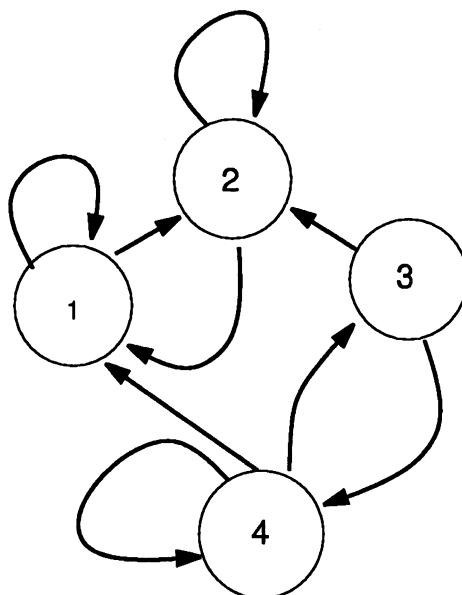


Figure 8.6. Schematic of a Markov chain with 4 states

bilities, and the n -step transition probabilities, in general. The n -step transition probabilities are very often of great importance to the analyst.

The so-called Chapman-Kolmogorov theorem helps us find these probabilities. The theorem states that the n -step transition probabilities can be obtained by raising the one-step transition probability matrix to the n th power. We next state the theorem without proof.

THEOREM 8.1 *If \mathbf{P} denotes the one-step transition probability matrix of a Markov chain and $\mathbf{Q} = \mathbf{P}^n$, then $Q(i, j)$ denotes the n -step transition probability of going from state i to state j .*

Basically, the theorem states says that the n th power of a TPM is also a transition probability matrix, whose elements are the n -step transition probabilities. Let us illustrate the meaning of this result with an example. Consider the TPM given by:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

Now,

$$\mathbf{P}^2 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.61 & 0.39 \\ 0.52 & 0.48 \end{bmatrix}.$$

Here the value of $P^2(1, 1)$ is 0.61. The theorem says that $P^2(1, 1)$ is the *two-step* transition probability of going from 1 to 1. Let us verify this from the basic principles of probability theory.

By its definition, $P^2(1, 1)$ is the probability that the system starts in state 1 and ends up in state 1 after two transitions. Let C_{x-y-z} denote the probability of going from state x to state z in two transitions with y as the intermediate state. Then, clearly, $P^2(1, 1)$ is equal to

$$C_{1-1-1} + C_{1-2-1}.$$

From the values of \mathbf{P} , $C_{1-1-1} = (0.7)(0.7)$ and $C_{1-2-1} = (0.3)(0.4)$, and therefore the required probability should equal:

$$(0.7)(0.7) + (0.3)(0.4) = 0.61,$$

which is equal to $P^2(1, 1)$. The verification is thus complete.

3.2. Regular Markov chains

A regular Markov chain is one whose TP satisfies the following property. *There exists a finite positive value for n such that $P^n(i, j) > 0$ for all i and j , n greater than 1.*

In other words, by raising the TPM of a regular Markov chain to some positive power , one obtains a matrix in which each element is greater than 0.

An example of a regular Markov chain is:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.0 & 0.3 \\ 0.4 & 0.6 & 0.0 \\ 0.2 & 0.7 & 0.1 \end{bmatrix}.$$

It is not hard to verify that P can be raised to a suitable power to obtain a matrix in which each element is strictly greater than 0. An example of Markov chain that is not regular is:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

This is not regular because:

$$\mathbf{P}^n = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ for odd values of } n \text{ and}$$

$$\mathbf{P}^n = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ for even values of } n.$$

3.2.1 Limiting probabilities

If one raises the TPM of a regular Markov chain to higher powers, the elements in any given column start *converging* to (that is, approaching) the same number. For example, consider \mathbf{P} of the previous section, raised to the 8th power:

$$\mathbf{P}^8 = \begin{bmatrix} 0.5715 & 0.4285 \\ 0.5714 & 0.4286 \end{bmatrix}$$

Notice how close the elements of a given *column* in \mathbf{P}^8 are to each other. It can be proved that as the power n tends to infinity, $P^n(i, j)$ starts converging (to a finite number) for a given value of j . In other words, for a given value of j ,

$$\lim_{n \rightarrow \infty} P^n(i, j) \text{ exists.}$$

In the example under consideration, the limit appears to be 0.57 for state 1 and 0.43 for state 2. We will denote the limit for state j by $\pi(j)$. Mathematically,

$$\pi(j) \equiv \lim_{n \rightarrow \infty} P^n(i, j).$$

The quantity $\pi(j)$ will also be referred to as the **limiting probability** of the state j .

Now, $\pi(j)$, it must be understood, is the **long-run** probability of entering the state j from **any given state**. For instance, in the example given above, regardless of which state the Markov chain is in, the long-run (that is, when $n \rightarrow \infty$) probability of entering state 1 is 0.57. Similarly, the long-run probability of entering state 2 is 0.43. From this, we can make an important inference:

Since the transitions are assumed to take unit time, 57% of the time will be spent by the process in transitions to state 1 and 43% of the time in transitions to state 2.

For the Markov process, the time taken in any transition is equal, and hence the limiting probability of a state also denotes the proportion of time spent in transitions to that particular state.

We will now discuss a method for obtaining the limiting probabilities from the TPM — without having to raise the TPM to large powers. The following important theorem forms a very convenient way for obtaining the limiting probabilities.

THEOREM 8.2 *Let $P(i, j)$ denote the transition probability from state i to state j , let $\pi(i)$ denote the limiting probability of state i , and let \mathcal{S} denote the set of states in the Markov chain. Then the limiting probabilities for all the states in the Markov chain can be obtained from the transition probabilities by solving the following set of linear equations:*

$$\sum_{i=1}^{|\mathcal{S}|} \pi(i)P(i, j) = \pi(j), \text{ for every } j \in \mathcal{S} \quad (8.4)$$

and

$$\sum_{j=1}^{|\mathcal{S}|} \pi(j) = 1, \quad (8.5)$$

where $|\mathcal{S}|$ denotes the number of elements in the set \mathcal{S} .

Equation (8.4) is often expressed in the matrix form as:

$$[\pi(1), \pi(2), \dots, \pi(|\mathcal{S}|)]\mathbf{P} = [\pi(1), \pi(2), \dots, \pi(|\mathcal{S}|)].$$

This is an important theorem from many standpoints. We do not present its proof here.

If you use this theorem to find the limiting probabilities of a Markov chain, you will notice that there is one *extra* equation in the linear system of equations defined by the theorem. You can eliminate one equation from the system defined by (8.4), and then solve the remaining equations to obtain a unique solution. We demonstrate this idea with the TPM given in (8.3).

From equations defined by (8.4), we have:

$$\text{For } j = 1 : \quad 0.7\pi(1) + 0.4\pi(2) + 0.6\pi(3) = \pi(1). \quad (8.6)$$

$$\text{For } j = 2 : \quad 0.2\pi(1) + 0.2\pi(2) + 0.1\pi(3) = \pi(2). \quad (8.7)$$

$$\text{For } j = 3 : \quad 0.1\pi(1) + 0.4\pi(2) + 0.3\pi(3) = \pi(3). \quad (8.8)$$

With some transposition, we can re-write these equations as:

$$-0.3\pi(1) + 0.4\pi(2) + 0.6\pi(3) = 0. \quad (8.9)$$

$$0.2\pi(1) - 0.8\pi(2) + 0.1\pi(3) = 0. \quad (8.10)$$

$$0.1\pi(1) + 0.4\pi(2) - 0.7\pi(3) = 0. \quad (8.11)$$

Now from Equation (8.5), we have:

$$\pi(1) + \pi(2) + \pi(3) = 1. \quad (8.12)$$

Thus we have four Equations: (8.9), (8.10), (8.11), and (8.12) and three unknowns: $\pi(1)$, $\pi(2)$, and $\pi(3)$. Notice that the system defined by the three Equations (8.9), (8.10), and (8.11) actually contains only two independent equations because any one can be obtained from the knowledge of the other two. Hence we select *any* two equations from this set. The two along with Equation (8.12) can be solved to find the unknowns. The values are: $\pi(1) = 0.6265$, $\pi(2) = 0.1807$, and $\pi(3) = 0.1928$.

Before concluding this section on Markov chains, we would like to summarize the main ideas presented.

1. From the TPM, it is possible to find the n -step transition probabilities.
2. The limiting probabilities of any Markov chain can be found by solving

$$[\pi(1), \pi(2), \dots, \pi(|\mathcal{S}|)]\mathbf{P} = [\pi(1), \pi(2), \dots, \pi(|\mathcal{S}|)],$$

$$\sum_{i=1}^{|\mathcal{S}|} \pi(i) = 1.$$

Remark. The history-independent property of the Markov chain is misleading. You can incorporate as much history as you want into the state space to convert a history-dependent process into a Markov process. This, however, comes with a downside which is that the number of states in the synthesized Markov process is much larger than that in the history-dependent process.

3.3. Ergodicity

A state in a Markov chain is said to be **recurrent** if it is visited repeatedly (again and again). In other words, if one views a Markov chain for an infinitely long period of time, one will see that a recurrent state is visited infinitely many

times. A **transient state** is one which is visited only a finite number of times in such an “infinite viewing.”

An example of a transient state is one to which the system does not come back from any recurrent state in one transition. In Figure 8.7, 1 is a transient state in this figure because once the system enters 2 or 3, it cannot come back to 1.

There may be more than one transient state in a Markov chain. See Figure 8.8. In this figure 1a and 1b are transient states. If the system starts in any one of these two states, it can visit both but once it goes to 2, it can never come back to 1a or 1b. A state that is not transient is called a recurrent state. Thus 2 and 3 are recurrent states in both Figures 8.7 and 8.8.

Another type of state is the *absorbing state*. Once the system enters any absorbing state, it can never get out of that state. In such a state, the system keeps coming back to itself.

An **ergodic chain** is one in which all states are recurrent and no absorbing states are present. Ergodic chains are also called irreducible chains. All regular Markov chains are ergodic, but the converse is not true. (Regular chains were defined in Section 3.2.) For instance, a chain that is not regular may be ergodic. Consider the Markov chain with the following TPM:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

This chain is not regular, but ergodic. It is ergodic because both states are visited infinitely many times in an infinite viewing.

We will next discuss the semi-Markov process.

3.4. Semi-Markov processes

A stochastic process that spends a random amount of time (which is not necessarily unity) in each transition, but is otherwise similar to a Markov process, is called a semi-Markov process. Consequently, underlying a semi-Markov process, there lurks a Markov *chain*. Roughly speaking, the only difference between the semi-Markov process and the Markov process lies in the time taken in transitions.

In general, when the distributions for the transition times are arbitrary, the process goes by the name semi-Markov. If the time is an exponentially distributed random variable, the stochastic process is referred to as a *continuous time Markov process*.

Some authors refer to what we have called the continuous time Markov process as the “Markov process,” and by “Markov chain” they mean what we have referred to as the Markov process.

If the time spent in the transitions is a deterministic quantity, the semi-Markov process has a transition time matrix analogous to the TPM. It should be clear

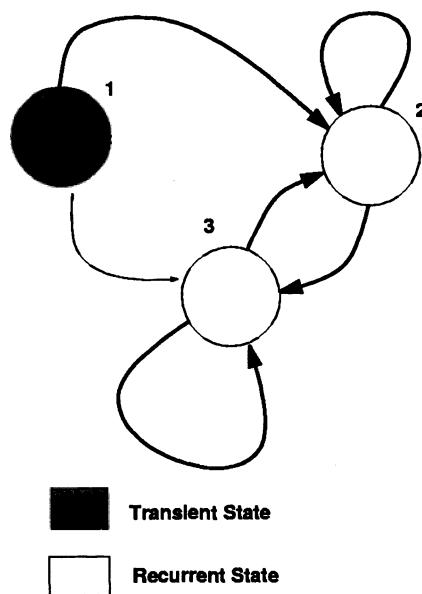


Figure 8.7. A Markov chain with one transient state

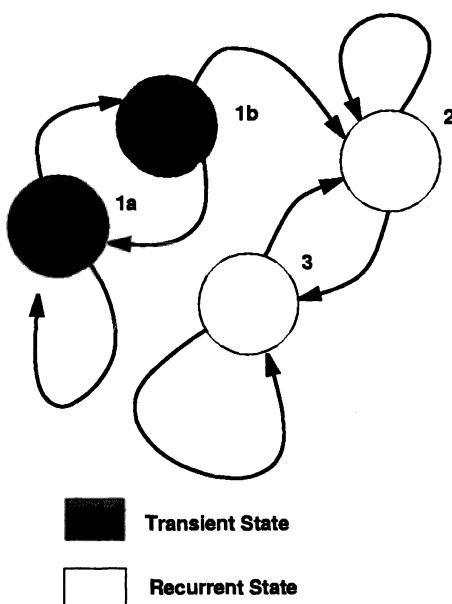


Figure 8.8. A Markov chain with two transient states

that the Markov process can be viewed as a special case of the semi-Markov process in which the time spent is a deterministic quantity that always equals 1.

There is, however, a difference between the Markov chain underlying a Markov process and that underlying a semi-Markov process. In a semi-Markov process, the system jumps, but not necessarily after unit time, and when it jumps, it jumps to a state that is *different* than the current state. In other words, in a semi-Markov process, the system does not jump back to the same state. However, in a semi-Markov *decision* process, which we will discuss later, jumping back can occur.

When we analyze a semi-Markov process, we begin by analyzing the Markov chain embedded in it. The next step is to analyze the time spent in each jump. As we will see later, the semi-Markov process is more powerful than the Markov process in modeling real-life systems, although very often its analysis can prove to be more difficult.

4. Markov decision problems

We will now discuss the topic that forms the central point of control optimization in this book. Thus far, we have considered Markov chains in which the transition from one state to another is governed by only one transition law, which is contained in the elements of the TPM. Such Markov chains are called *uncontrolled* Markov chains, essentially because in such chains there is no external agency that can control the path taken by the stochastic process.

We also have systems that can be run with different control mechanisms — where each control mechanism has its own TPM. In other words, the routes dictated by the control mechanisms are not the same. The control mechanism specifies the “action” to be selected in each state. When we are faced with the decision of choosing from *more than one* control mechanism, we have what is called a **Markov decision problem**, which is often abbreviated as an **MDP**.

The MDP is a problem of control optimization — that is, the problem of finding the optimal **action** to be selected in each state. Many real-world problems can be set up as MDPs, and before we discuss any details of the MDP framework, let us study a simple example of an MDP.

Example. Consider a queuing system, such as the one you see in a supermarket, with a maximum of 3 counters (servers). You have probably noticed that when queues become long, more counters (servers) are opened. The decision-making problem we will consider here is to find the number of servers that should be open at any given time.

The people who function as servers also have other jobs to perform, and hence it does not make business sense to have them wait on the counters when there are no customers at the counters. At the same time, if very long queues

build up but more counters are not opened when there is capacity, customers do not feel very happy about it, and may actually go elsewhere the next time. Hence in this situation, one seeks an optimal strategy (that is a control mechanism) to control the system. Next, we will discuss the idea of control mechanisms or policies with some examples.

Consider a system which has a maximum of 3 counters. Let us assume that the state of this system is defined by the number of people waiting for service. Let us further assume that associated with this state definition, a Markov process exists. (See Figure 8.9 for a picture of the underlying Markov chain.) Thus when the system enters a new state of the Markov chain, one out of the following 3 actions can be selected:

1. Open 1 counter.

2. Open 2 counters.

3. Open 3 counters.

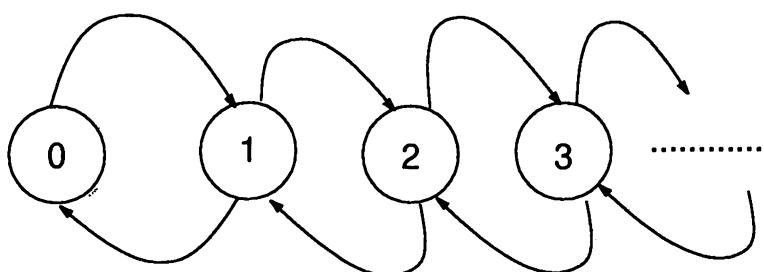


Figure 8.9. A Markov chain underlying a simple single server queue.

One possible control mechanism (or policy) in this situation would look like this:

State = Number waiting for service	Action
0	Open 1 counter
1	Open 1 counter
2	Open 1 counter
3	Open 1 counter
4	Open 2 counters
5	Open 2 counters
6	Open 2 counters
7	Open 3 counters
8	Open 3 counters
.	Open 3 counters
.	Open 3 counters
.	Open 3 counters

Another possible control mechanism could be:

State = Number waiting for service	Action
0	Open 1 counter
1	Open 1 counter
2	Open 1 counter
3	Open 1 counter
4	Open 1 counter
5	Open 1 counter
6	Open 2 counters
7	Open 2 counters
8	Open 3 counters
.	Open 3 counters
.	Open 3 counters
.	Open 3 counters

Note that the two control mechanisms are different. In the first, two counters are opened in state 4, while in the second, the same is done in state 6. From these two examples, it should be clear that there are, actually, several different control mechanisms that can be used, and the effect on the system — in terms of the net profits generated — may differ with the control mechanism used. For instance, consider the following.

- A control mechanism that allows big queues to build up may lead to a cost reduction in some sense since less employees may be necessary to run the system. But it may also lead to reduced profits because in the future customers may choose to go elsewhere where queues are shorter.
- On the other hand, a control mechanism which is over-designed with a large number of servers — and where customers hardly ever have to wait — may

be expensive to maintain because of the costs incurred in hiring a large number of servers; eventually the high costs of running this system will be transmitted to the customers through higher-priced products. The latter is also likely to drive the customers away.

It naturally makes business sense to use the control mechanism that produces the greatest net profits. Formulation of this problem as a Markov decision problem will help us identify the best control mechanism.

From our discussion above, we can conclude that:

- Each control mechanism is likely to have unique costs and profits.
- The system considered above is stochastic, and associated with each control mechanism, we may see a distinctive pattern of behavior.
- The problem is one of finding the right control mechanism.

The Markov decision framework is a sophisticated operations research model designed to solve this problem. Let us discuss it in some detail, next.

4.1. Elements of the Markov decision framework

The Markov decision framework is designed to solve the so-called Markov decision problem (MDP). The framework is made up of five important elements. They are:

1. A decision maker,
2. policies,
3. transition probability matrices,
4. transition reward matrices, and
5. a performance metric (objective function).

Let us discuss the role of these elements, next.

Decision maker. The decision maker is a fictitious entity that *selects* the control mechanism. It is also called the *agent* or *controller*.

Policies. The control mechanism is usually referred to as a **policy**. A policy for an MDP with n states is an n -tuple. Each element of this n -tuple specifies the action to be selected in the state associated with that element. For example, consider a 2-state MDP in which two actions are allowed in each state. An example of a policy for this MDP is: $(2, 1)$. This means that by adhering to this policy, the following would occur: In state 1, action 2 would be selected, and in state 2, action 1 would be selected. Thus in general, if $\hat{\mu}$ denotes a policy,

the i th element of $\hat{\mu}$, that is, $\mu(i)$, denotes the action selected in the i th state for the policy $\hat{\mu}$.

In this book, unless otherwise stated, the word “policy” will imply a **stationary, deterministic** policy. The word stationary means that the policy does *not* change with time. This implies that if a policy dictates an action a be taken in a state x , then no matter how long the Markov chain has been operating, every time the system visits state x , it is action a that will be selected.

The word deterministic implies that in any given state, we can choose **only one (1)** action. In other words, with a probability of 1, a given action is selected. We will deal with stochastic policies in the context of learning automata for control optimization. However, in this chapter, we will only consider stationary, deterministic policies.

We will assume throughout this book that the set of actions allowed in each state is a finite set. The set of actions allowed in state i will be denoted by $\mathcal{A}(i)$. We will also assume the set of states in the Markov chain to be a finite set — which will be denoted by \mathcal{S} .

Since the number of actions allowed in each state and the number of states themselves are finite quantities, we must have a finite number of policies. For instance in an MDP with 2 states and two actions allowed in each state, we have $2^2 = 4$ policies, which are:

$$(1, 1), \quad (1, 2), \quad (2, 1), \quad \text{and} \quad (2, 2).$$

An MDP, let us reiterate, revolves around finding the most *suitable* policy.

A few more words about the term “state” are in order. Sometimes, we deal with Markov chains in which decisions are not made in all states. That is, in some states, there is only one allowable action. As such, there is no decision making involved in these states. These states are called non-decision-making states. A state in which one has to choose from more than one action is hence called a **decision-making** state.

In this book, by “state,” we refer to a *decision-making* state. When we have models in which some states are not decision-making, we will distinguish between the two by the qualifiers: decision-making and non-decision-making.

Transition probability matrices. The decision maker executes the action to be used in each state of an MDP. Associated with **each action**, we usually have a transition probability matrix (TPM). Associated with each policy, also, we have a unique TPM. The TPM for a policy can be constructed from the TPMs associated with the individual actions in the policy.

To understand how this construction can be performed, let us consider a 2-state MDP with 2 actions allowed in each state. Let the TPM associated with action 1 be

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix},$$

and that associated with action 2 be

$$\mathbf{P}_2 = \begin{bmatrix} 0.1 & 0.9 \\ 0.8 & 0.2 \end{bmatrix}.$$

Now consider a policy $\hat{\mu} = (2, 1)$. The TPM associated with this policy will contain the transition probabilities of action 2 in state 1 and the transition probabilities of action 1 in state 2. The TPM of policy $\hat{\mu}$ is thus

$$\mathbf{P}_{\hat{\mu}} = \begin{bmatrix} 0.1 & 0.9 \\ 0.4 & 0.6 \end{bmatrix}.$$

See Figure 8.10 for a pictorial demonstration of the construction process.

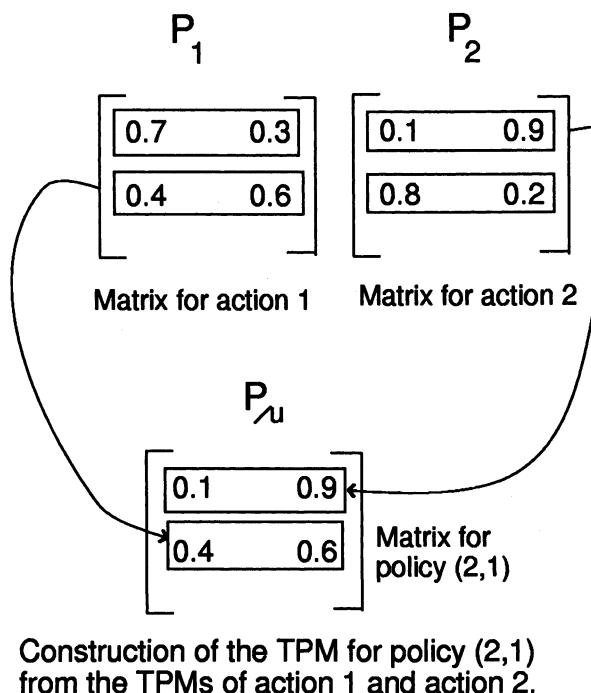


Figure 8.10. Schematic showing how the TPM of policy $(2, 1)$ is constructed from the TPMs of action 1 and 2.

In general, we will use the following notation to denote a transition probability:

$$p(i, a, j).$$

This term will denote the one-step transition probability of going from state i to state j when action a is selected in state i . Now, if policy $\hat{\mu}$ is followed, then the action selected in state i will be denoted by $\mu(i)$, and as a result the transition probability of going from state i to state j will be denoted by

$$p(i, \mu(i), j).$$

Then $p(i, \mu(i), j)$ will define the element in the i th row and the j th column of the matrix $P_{\hat{\mu}}$ — the TPM associated with the policy $\hat{\mu}$.

In this section, we have used the phrase: “a transition probability under the influence of an action.” The significance of this must be noted with care. In our previous discussions, we spoke of transition probabilities without reference to any action. That was because we were dealing with **uncontrolled** Markov chains — which had **only one action** in each state. Now, with the introduction of multiple actions in each state, we must be careful with the phrase “transition probability,” and must specify the action along with a transition probability.

Now, as stated previously, the MDP is all about identifying the optimal policy (control mechanism). The TPM, we will see shortly, will serve an important purpose in evaluating a policy and will be essential in identifying the best policy. The other tool that we need for evaluating a policy is discussed in the next paragraph.

Transition reward matrices. With each transition in a Markov chain, we can associate a reward. (A negative value for the reward is equivalent to a cost.) We will refer to this quantity as the **immediate reward** or transition reward. The immediate reward helps us incorporate reward and cost elements into the MDP model. The immediate reward matrix, generally called the transition reward matrix (TRM), is very similar to the TPM. Recall that the (i, j) th element (the element in the i th row and j th column) of the TPM denotes the transition probability from state i to state j . Similarly, the (i, j) th element of the TRM denotes the immediate reward earned in a transition from state i to state j . Just as we have TPMs associated with individual actions and policies, we have TRMs associated with actions and policies. Let us examine some examples from a 2-state MDP, next.

In a 2-state MDP, the TRM associated with action 1 is

$$\mathbf{R}_1 = \begin{bmatrix} 11 & -4 \\ -14 & 6 \end{bmatrix},$$

and that associated with action 2 is

$$\mathbf{R}_2 = \begin{bmatrix} 45 & 80 \\ 1 & -23 \end{bmatrix}.$$

Now consider a policy $\hat{\mu} = (2, 1)$. Like in the TPM case, the TRM associated with this policy will contain the immediate reward of action 2 in state 1 and the immediate rewards of action 1 in state 2. Thus the TRM of policy $\hat{\mu}$ can be written as

$$\mathbf{R}_{\hat{\mu}} = \begin{bmatrix} 45 & 80 \\ -14 & 6 \end{bmatrix}.$$

The TPM and the TRM of a policy together contain all the information one needs to evaluate the policy in an MDP. In terms of notation, we will denote the immediate reward, earned in going from state i to state j , under the influence of action a , by:

$$r(i, a, j).$$

When policy $\hat{\mu}$ is followed, the immediate reward earned in going from state i to state j will be denoted by:

$$r(i, \mu(i), j)$$

because $\mu(i)$ is the action that will be selected in state i when policy $\hat{\mu}$ is used.

Performance metric . To compare policies, one must define a performance metric (objective function). Naturally, the performance metric should involve reward and cost elements. To give a simple analogy, in a linear programming problem, one judges each solution on the basis of the value of the associated objective function. Any optimization problem has a performance metric, which is also called the **objective function** . In this book, the MDP will be studied with respect to two performance metrics. They are:

1. Expected reward *per unit time* calculated over an infinitely long trajectory of the Markov chain — we will refer to this metric as: the **average reward**.
2. Expected *total* discounted reward calculated over an infinitely long trajectory of the Markov chain — we will refer to it as the **discounted reward**.

It is the case that of the two performance metrics, average reward is easier to understand, although the average reward MDP is more difficult to analyze for its convergence properties. Hence, we will begin our discussion with the average reward performance criterion. Discounted reward will be defined later.

We first need to define the **expected immediate reward** of a state under the influence of a given action. Consider the following scenario. An action a is selected in state i . Under the influence of this action, the system can jump to three states: 1, 2, and 3 with probabilities of

$$0.2, 0.3, \text{ and } 0.5,$$

respectively. The immediate rewards earned in these three possible transitions are, respectively,

10, 12, and -14.

Then the **expected** immediate reward that will be earned, when action a is selected in state i , will clearly be:

$$0.2(10) + 0.3(12) + 0.5(-14) = 1.6.$$

The expected immediate reward is calculated in the style shown above. (Also see Figure 8.11). In general, we can use the following expression to calculate the expected immediate reward.

$$\bar{r}(i, a) = \sum_{j \in S} p(i, a, j)r(i, a, j). \quad (8.13)$$

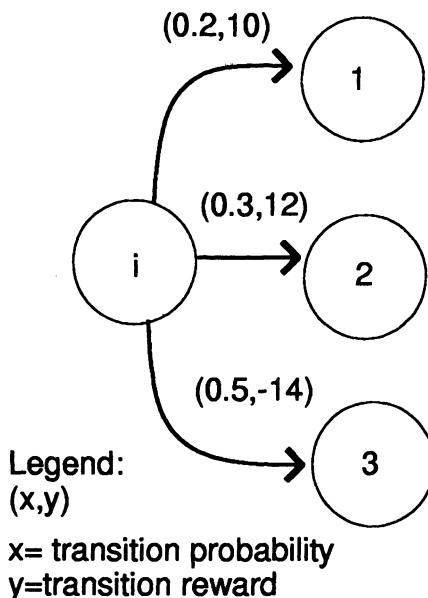


Figure 8.11. Calculation of expected immediate reward

The average reward of a given policy is the expected reward earned per unit time by running the Markov chain with the policy for an infinitely long period of time. It turns out that there is a very convenient way for representing the average reward of a policy — using the limiting probabilities associated with the policy. We will explain this with an example, and then generalize from there to obtain a generic expression for the average reward.

Consider a 3-state MDP with states numbered 1, 2, and 3. Let us assume that the system follows a fixed policy $\hat{\mu}$ and that the limiting probabilities of the three states with respect to this policy are: $\pi_{\hat{\mu}}(1) = 0.3$, $\pi_{\hat{\mu}}(2) = 0.5$, and $\pi_{\hat{\mu}}(3) = 0.2$. Let us further assume that the expected immediate rewards earned in the 3 states are:

$$\bar{r}(1, \mu(1)) = 10, \bar{r}(2, \mu(2)) = 12, \text{ and } \bar{r}(3, \mu(3)) = 14.$$

Now from our discussion on limiting probabilities, we know that the limiting probability of a state denotes the proportion of time spent in transitions to that particular state in the long run. Hence if we observe the system over k transitions, $k\pi_{\hat{\mu}}(i)$ will equal the number of transitions to state i in the long run. Now, the expected reward earned in each visit to state i under policy $\hat{\mu}$ is $\bar{r}(i, \mu(i))$. Then the total long-run expected reward earned in k transitions for this MDP can be written as:

$$k\pi_{\hat{\mu}}(1)(\bar{r}(1, \mu(1))) + k\pi_{\hat{\mu}}(2)(\bar{r}(2, \mu(2))) + k\pi_{\hat{\mu}}(3)(\bar{r}(3, \mu(3))).$$

Consequently the average reward associated with policy $\hat{\mu}$ can be written as:

$$\begin{aligned} \rho_{\hat{\mu}} &= \frac{k\pi_{\hat{\mu}}(1)(\bar{r}(1, \mu(1))) + k\pi_{\hat{\mu}}(2)(\bar{r}(2, \mu(2))) + k\pi_{\hat{\mu}}(3)(\bar{r}(3, \mu(3)))}{k} \\ &= \sum_{i=1}^3 \pi_{\hat{\mu}}(i)\bar{r}(i, \mu(i)) \end{aligned}$$

Then, in general, the average reward of a policy $\hat{\mu}$ can be written as:

$$\rho_{\hat{\mu}} = \sum_{i \in \mathcal{S}} \pi_{\hat{\mu}}(i)\bar{r}(i, \mu(i)), \quad (8.14)$$

where

- $\pi_{\hat{\mu}}(i)$ denotes the limiting probability of state i when the Markov chain is run with the policy $\hat{\mu}$,
- \mathcal{S} denotes the set of states visited in the Markov chain,
- and $\bar{r}(i, a)$ denotes the expected immediate reward earned in the state i when action a is selected in state i .

In the next section, we will discuss a simple method to solve the MDP.

5. How to solve an MDP using exhaustive enumeration

The method that we will discuss in this section goes by the name: exhaustive enumeration. Conceptually, this is the easiest method to understand, although

in practice we can use it only on small problems. The method is based on the following idea.

Enumerate every policy that can possibly be selected, evaluate the performance metric associated with each policy, and then declare the policy that produces the best value for the performance metric to be the optimal policy.

[If you are familiar with linear programming, you will see an analogy to the evaluation of all extreme points in the simplex (although this problem, of course, has nothing to do with a linear program).]

Let us illustrate how this method works with a simple example MDP that has just two states and two actions in each state. This example will be used repeatedly throughout the remainder of this book.

5.1. Example A

There are two states numbered 1 and 2 in an MDP, and two actions — also numbered 1 and 2 — are allowed in each state. The transition probability matrix (TPM) associated with action 1 is

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix},$$

and that for action 2 is

$$\mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

The TRM for action 1 is

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix},$$

and the same for action 2 is

$$\mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

Pictorially, the MDP is represented in Figure 8.12.

In this MDP, there are 4 possible policies that can be used to control the Markov chain. They are:

$$\hat{\mu}_1 = (1, 1), \hat{\mu}_2 = (1, 2), \hat{\mu}_3 = (2, 1), \text{ and } \hat{\mu}_4 = (2, 2).$$

The TPMs and TRMs of these policies are constructed from the individual TPMs and TRMs of each action. The TPMs are:

$$\mathbf{P}_{\hat{\mu}_1} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix},$$

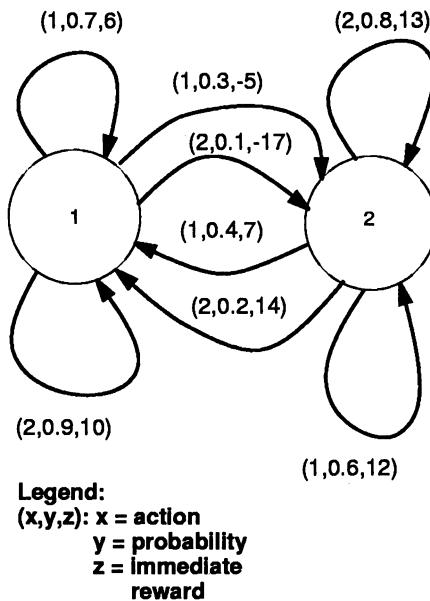


Figure 8.12. A two state MDP

$$\mathbf{P}_{\hat{\mu}_2} = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix},$$

$$\mathbf{P}_{\hat{\mu}_3} = \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix},$$

and

$$\mathbf{P}_{\hat{\mu}_4} = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

The TRMs are

$$\mathbf{R}_{\hat{\mu}_1} = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix},$$

$$\mathbf{R}_{\hat{\mu}_2} = \begin{bmatrix} 6 & -5 \\ -14 & 13 \end{bmatrix},$$

$$\mathbf{R}_{\hat{\mu}_3} = \begin{bmatrix} 10 & 17 \\ 7 & 12 \end{bmatrix},$$

and

$$\mathbf{R}_{\hat{\mu}_4} = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

From the TPMs, using Equations (8.4) and (8.5), one can find the limiting probabilities of the states associated with each policy. They are:

$$\pi_{\hat{\mu}_1}(1) = 0.5714 \text{ and } \pi_{\hat{\mu}_1}(2) = 0.4286,$$

$$\begin{aligned}\pi_{\hat{\mu}_2}(1) &= 0.4000 \text{ and } \pi_{\hat{\mu}_2}(2) = 0.6000, \\ \pi_{\hat{\mu}_3}(1) &= 0.8000 \text{ and } \pi_{\hat{\mu}_3}(2) = 0.2000,\end{aligned}$$

and

$$\pi_{\hat{\mu}_4}(1) = 0.6667 \text{ and } \pi_{\hat{\mu}_4}(2) = 0.3333.$$

We will next find the average reward of each of these four policies. Let us first evaluate the average immediate reward in each possible transition in the MDP using Equation (8.13). For this, we need the TPMs and the TRMs of each policy.

$$\begin{aligned}\bar{r}(1, \mu_1(1)) &= p(1, \mu_1(1), 1)r(1, \mu_1(1), 1) + p(1, \mu_1(1), 2)r(1, \mu_1(1), 2) \\ &= 0.7(6) + 0.3(-5) = 2.7.\end{aligned}$$

$$\begin{aligned}\bar{r}(2, \mu_1(2)) &= p(2, \mu_1(2), 1)r(2, \mu_1(2), 1) + p(2, \mu_1(2), 2)r(2, \mu_1(2), 2) \\ &= 0.4(7) + 0.6(12) = 10.\end{aligned}$$

$$\begin{aligned}\bar{r}(1, \mu_2(1)) &= p(1, \mu_2(1), 1)r(1, \mu_2(1), 1) + p(1, \mu_2(1), 2)r(1, \mu_2(1), 2) \\ &= 0.7(6) + 0.3(-5) = 2.7.\end{aligned}$$

$$\begin{aligned}\bar{r}(2, \mu_2(2)) &= p(2, \mu_2(2), 1)r(2, \mu_2(2), 1) + p(2, \mu_2(2), 2)r(2, \mu_2(2), 2) \\ &= 0.2(-14) + 0.8(13) = 7.6.\end{aligned}$$

$$\begin{aligned}\bar{r}(1, \mu_3(1)) &= p(1, \mu_3(1), 1)r(1, \mu_3(1), 1) + p(1, \mu_3(1), 2)r(1, \mu_3(1), 2) \\ &= 0.9(10) + 0.1(17) = 10.7.\end{aligned}$$

$$\begin{aligned}\bar{r}(2, \mu_3(2)) &= p(2, \mu_3(2), 1)r(2, \mu_3(2), 1) + p(2, \mu_3(2), 2)r(2, \mu_3(2), 2) \\ &= 0.4(7) + 0.6(12) = 10.\end{aligned}$$

$$\begin{aligned}\bar{r}(1, \mu_4(1)) &= p(1, \mu_4(1), 1)r(1, \mu_4(1), 1) + p(1, \mu_4(1), 2)r(1, \mu_4(1), 2) \\ &= 0.9(10) + 0.1(17) = 10.7.\end{aligned}$$

$$\begin{aligned}\bar{r}(2, \mu_4(2)) &= p(2, \mu_4(2), 1)r(2, \mu_4(2), 1) + p(2, \mu_4(2), 2)r(2, \mu_4(2), 2) \\ &= 0.2(-14) + 0.8(13) = 7.6.\end{aligned}$$

Now, using these quantities, we can now calculate the average reward of each individual policy. We will make use of Equation (8.14).

Thus:

$$\begin{aligned}\rho_{\hat{\mu}_1} &= \pi_{\hat{\mu}_1}(1)\bar{r}(1, \mu_1(1)) + \pi_{\hat{\mu}_1}(2)\bar{r}(2, \mu_1(2)) \\ &= 0.5741(2.7) + 0.4286(10) = 5.83,\end{aligned}$$

$$\begin{aligned}\rho_{\hat{\mu}_2} &= \pi_{\hat{\mu}_2}(1)\bar{r}(1, \mu_2(1)) + \pi_{\hat{\mu}_2}(2)\bar{r}(2, \mu_2(2)) \\ &= 0.4(2.7) + 0.6(7.6) = 5.64,\end{aligned}$$

$$\begin{aligned}\rho_{\hat{\mu}_3} &= \pi_{\hat{\mu}_3}(1)\bar{r}(1, \mu_3(1)) + \pi_{\hat{\mu}_3}(2)\bar{r}(2, \mu_3(2)) \\ &= 0.8(10.7) + 0.2(10) = 10.56,\end{aligned}$$

and

$$\begin{aligned}\rho_{\hat{\mu}_4} &= \pi_{\hat{\mu}_4}(1)\bar{r}(1, \mu_4(1)) + \pi_{\hat{\mu}_4}(2)\bar{r}(2, \mu_4(2)) \\ &= 0.6667(10.7) + 0.3333(7.6) = 9.6667.\end{aligned}$$

It is clear that policy $\hat{\mu}_3 = (2, 1)$ is the best policy, since it produces the highest average reward.

5.2. Drawbacks of exhaustive enumeration

It should be clear that exhaustive enumeration can only be used on small problems. For example, consider the scenario with 10 states and 2 actions in each state. This is a very small problem but if we were to use exhaustive enumeration, we would have to evaluate 2^{10} different policies. As such the computational burden of this method can overwhelm even small problems. In the next section, we will study a method that is considerably more efficient than exhaustive enumeration.

6. Dynamic programming for average reward

The method of **dynamic programming (DP)**, in the context of solving MDPs, was developed in the late fifties and early sixties with the pioneering work of Bellman [15] and Howard [81]. Dynamic programming has been shown to have a considerably lower computational burden — in comparison to exhaustive enumeration. The theory of DP has evolved a great deal in the last five decades, and a voluminous amount of literature now exists on this topic. In fact, it continues to be the main pillar of stochastic control optimization.

Although many highly technical papers have been written on this topic, its theory rests on a simple linear system of equations. As such, it is very easy to understand its basic principles; the analysis of its convergence properties, however, can get quite complicated.

In this chapter and in the next, we will endeavor to present the main equations underlying DP — without worrying about how the equations were derived. Also, although we will present some DP-related algorithms, we will not attempt to *prove* that they generate optimal solutions. That kind of analysis has been relegated to Chapter 13. Computer codes for some DP algorithms are presented in Section 6 of Chapter 15.

The linear system of equations, to which we are referring here, is often called the Bellman equation. The Bellman equation has several forms. In this section, we will concentrate on the average reward MDP. In the next section, we will be examining MDPs related to discounted reward. Even for the average reward MDP, there are at least two different useful forms of the Bellman equation.

In this section, we will study two different DP methods to solve average reward MDPs. These methods go by the following names.

- Policy iteration (uses the Bellman equation for a policy).
- Value iteration (uses the Bellman optimality equation).

Policy iteration requires a form of the Bellman equation that we will present next.

6.1. Average reward Bellman equation for a policy

We have seen above that associated with a given policy, there is a scalar quantity called the average reward of the policy. Similarly, associated with a policy there exists a vector called the **value function** vector for the policy. The dimension of this vector is equal to the number of elements in the set (\mathcal{S}) of decision-making states in the Markov chain. The values of the components of this vector can be found by solving a linear system of equations, which is collectively called the Bellman equation for a policy.

A natural question at this stage is: what purpose will be served by finding the values of the elements of this vector? It is best that we answer this question at the outset, because much of DP revolves around this vector. The answer is that the values of these elements can help us find a *better* policy.

The Bellman equation for a given policy in the average reward context is:

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) - \rho_{\hat{\mu}} + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \quad (8.15)$$

The above is a system of linear equations; the number of equations is equal to the number of elements in the set \mathcal{S} , that is, $|\mathcal{S}|$. The unknowns in the equation are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$. The other terms are described below:

- $\mu(i)$ denotes the action selected in state i under the policy $\hat{\mu}$. Since the policy is known, each $\mu(i)$ is known.
- $\bar{r}(i, \mu(i))$ denotes the expected immediate reward in state i under policy $\mu(i)$. Each of these terms can be calculated from the TPMs and the TRMs.
- $p(i, \mu(i), j)$ denotes the one-step transition probability of jumping from state i to state j under the policy $\hat{\mu}$. Again, these terms can be obtained from the TPMs.
- $\rho_{\hat{\mu}}$ denotes the average reward associated with the policy $\hat{\mu}$, and it can be obtained, when the policy is known, from the TPMs and the TRMs as discussed in the context of exhaustive enumeration.

We will next discuss the policy iteration algorithm to solve the average reward MDP.

6.2. Policy iteration for average reward MDPs

The basic idea underlying policy iteration is to start with an arbitrary Policy, and then move to a better policy in every iteration. This will continue until no further improvement is possible.

When a policy is selected, the Bellman equation for a policy is used to obtain the value function vector for that policy. This is called the policy evaluation stage because in this stage we evaluate the value function vector associated with a policy. Then the value function vector is used to find a *better* policy. This step is called the policy improvement step.

The value function vector of the new (better) policy is then obtained. In other words, one goes back to the policy evaluation step. This continues until the value function vector obtained from solving the Bellman equation cannot produce a better policy.

If you are familiar with the simplex algorithm of linear programming, you will see an analogy. In simplex, we start at a solution (corner point) and then move to a better point. This continues till no better point can be obtained.

The advantage of policy iteration over exhaustive enumeration is that usually one obtains the solution with a comparatively small number of computations.

6.2.1 Steps

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be \mathcal{S} . Select an arbitrary policy. Let us denote the policy selected in the k th iteration by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation:) Solve the following linear system of equations.

For $i = 1, 2, \dots, |\mathcal{S}|$,

$$h^k(i) = \bar{r}(i, \mu_k(i)) - \rho^k + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h^k(j). \quad (8.16)$$

Here one linear equation is associated with each value of $i \in \mathcal{S}$. In this system, the unknowns are the h^k terms and ρ^k . The number of unknowns exceeds the number of equations by 1. Hence to solve the system, one should set any one of the h^k terms to 0, and then solve for the rest of the unknowns. The term ρ^k should not be set to 0.

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} \bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j).$$

If possible, one should set $\mu_{k+1}(i) = \mu_k(i)$ for each i . The significance of \in in the above needs to be understood clearly. There can be more than one action that satisfies the arg max operator. Thus there may be multiple candidates for $\mu_{k+1}(i)$. However, the latter is selected in a way such that $\mu_{k+1}(i) = \mu_k(i)$ if possible.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i , then stop and set $\mu^*(i) = \mu_k(i)$ for every i . Otherwise, increment k by 1, and go back to the second step.

Policy iteration on Example A. We used policy iteration on Example A from Section 5.1. The results are shown in Table 8.1. The optimal policy is $(2, 1)$.

We will next discuss an alternative method, called value iteration, to solve the average reward MDP.

Table 8.1. Table showing calculations in policy iteration for average reward MDPs on Example A

Iteration (k)	Policy Selected	Values	ρ^k
1	$\hat{\mu}_1 = (1, 1)$	$h^1(1) = 0$ $h^1(2) = 10.56$	5.86
2	$\hat{\mu}_2 = (2, 1)$	$h^2(1) = 0$ $h^2(2) = -1.4$	10.56

6.3. Value iteration and its variants: average reward MDPs

The value iteration algorithm is another very useful algorithm that can solve the MDP. It uses a form of the Bellman equation that is different than the form used in policy iteration.

Another major difference between value iteration and policy iteration is that in value iteration, one does not have to solve any equations unlike policy iteration. This makes the algorithm easy to code (that is, write a computer program for it); this has contributed to its popularity. However, the real advantage of value iteration will be realized in reinforcement learning — a topic that will be discussed in Chapter 9.

We first present the Bellman **optimality** equation. A variant of this will be used in the value iteration and the relative value iteration algorithms.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) - \rho^* + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^*(j)] \text{ for each } i \in \mathcal{S}. \quad (8.17)$$

The following remarks will explain the notation.

- $\mathcal{A}(i)$ is the set of actions allowed in state i .
- The J^* terms are the unknowns. They are the components of the **optimal** value function vector \vec{J}^* . The number of elements in the vector \vec{J}^* equals the number of states in the MDP.
- The term $\bar{r}(i, a)$ denotes the expected immediate reward in state i when action a is selected in state i .
- The term $p(i, a, j)$ denotes the one-step transition probability of jumping from state i to state j when action a is selected in state i .
- The term ρ^* denotes the average reward associated with the optimal policy.

Equation (8.17) is called the Bellman **optimality** equation for average reward MDPs. Since this equation contains the max operator, it cannot be solved using linear algebra techniques such as Gauss elimination. Also, since the optimal policy is unknown at the start, one does not have access to the value of ρ^* . These difficulties can be easily circumvented, as we will see shortly.

We will next present the value iteration algorithm for average reward MDPs.

6.4. Value iteration for average reward MDPs

Value iteration uses a form of the Bellman optimality equation that does not have the ρ^* term. One starts with some arbitrary values for the value function

vector. And then a transformation — that is an updating mechanism — derived from the Bellman equation is applied on the vector repeatedly. This mechanism keeps updating (changing) the elements of the value function vector. The values never converge; hence one must identify some mechanism to terminate the algorithm. The termination mechanism in Step 3 of the algorithm uses the so-called *span seminorm* (often referred to as simply the span). Our notation for the span seminorm of a vector is sp . The definition is

$$sp(\vec{x}) = \max_i x(i) - \min_i x(i).$$

The span of a vector is non-negative and denotes the range of values in the different components of the vector. A vector may keep changing, and yet its span may not change.

6.4.1 Steps

Step 1: Set $k = 1$, and select an arbitrary vector \vec{J}^1 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|S|} p(i, a, j) J^k(j)].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1, and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|S|} p(i, a, j) J^k(j)],$$

and stop. The ϵ -optimal policy is \hat{d} .

Several important remarks are in order here.

- The implication of ϵ -optimality needs to be understood. The smaller the value of ϵ , the closer we get to the optimal policy. Usually, for small values of ϵ , one obtains policies that are very close to optimal. With some additional work, we can derive the optimal policy. We do not discuss how, but refer the interested reader to Puterman [140].
- In Step 2, we have used a “transformation” (for a definition of transformation see Chapter 2) derived from the Bellman equation. A transformation takes a vector and produces a new vector from the vector supplied to it.

- Value iteration may not work on Markov chains that are not regular (see Section 3.2 for a definition of regularity) under all policies.
- The span of the difference vector ($\vec{J}^{k+1} - \vec{J}^k$) keeps getting smaller and smaller (for a proof, you are referred to the convergence analysis of value iteration in Chapter 13), and hence for a given positive value of ϵ , the algorithm terminates in a finite number of steps.
- A major difficulty with value iteration is that the values themselves can become very large or very small. This difficulty can be overcome by using *relative* value iteration — the topic of discussion in the next section.
- Had we used the actual value of ρ^* in Step 2 (assuming that the value of ρ^* is known somehow), instead of replacing it by 0,
 - we would have obtained the same value for the span in Step 3 and
 - we would have still obtained the same sequence of maximizing actions in Step 2.

This implies that value iteration with the Bellman transformation in Step 2 should yield the same results as the value iteration algorithm discussed above. The Bellman transformation, it can be shown, leads one to an optimal solution.

Use of Value Iteration on Example A. See Table 8.2 for some sample calculations with value iteration on Example A (from Section 5.1). Note that the values are gradually becoming larger with every iteration.

We discuss relative value iteration next.

Table 8.2. Table showing calculations in value iteration for average reward MDPs. Note that the values get unbounded. Although, we do not show the calculations in the table, the span semi-norm *does* decrease with every iteration.

Iteration (k)	$J^k(1)$	$J^k(2)$
1	10.7	10.0
2	21.33	20.28
3	31.925	30.70
.
.
.
152	1605.40	1604.00
153	1615.96	1614.56

6.5. Relative value iteration

The major obstacle in using value iteration, as discussed above, is that some of the values themselves can become very large or very small. On problems with a small number of states, using a not-very-small value for ϵ , one can use value iteration without running into trouble. However, for larger problems, it is likely that one of the values may become very large or very small, and it is then best to use *relative* value iteration.

In relative value iteration, we select any arbitrary state in the Markov chain to be a *distinguished* state. Also we replace ρ^* in the Bellman equation by the value of the distinguished state. This keeps the values from becoming too large or too small. Remember, in value iteration, the updating mechanism (see Step 2 of value iteration) is derived from the Bellman equation with the replacement of ρ^* by 0.

6.5.1 Steps

Step 1: Select any arbitrary state from \mathcal{S} to be the distinguished state i^* . Set $k = 1$, and select an arbitrary vector \vec{J}^1 in which $J^1(i^*) = 0$. Specify $\epsilon > 0$.

Step 2: Compute:

$$J^{k+1}(i^*) = \max_{a \in \mathcal{A}(i^*)} [\bar{r}(i^*, a) + \sum_{j=1}^{|\mathcal{S}|} p(i^*, a, j) J^k(j)].$$

Step 3: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)] - J^{k+1}(i^*).$$

Step 4: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 5. Otherwise increase k by 1 and go back to Step 2.

Step 5: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)],$$

and stop. The ϵ -optimal policy is \hat{d} .

The only difference between relative value iteration and regular value iteration is in the subtraction of a value. And yet what a difference!

- a. The values remain bounded and
- b. the algorithm converges to the same policy that value iteration converges to.

What is the reason for (b)?

- The maximizing actions in the Step 2 of each algorithm are the same.

We can show that value iteration converges in the span seminorm (see Chapter 13), and hence so must relative value iteration.

Example A and relative value iteration. In Table 8.3, we show the calculations with using relative value iteration on Example A from Section 5.1.

6.6. A general expression for the average reward of an MDP

It is a good idea to remind the reader that we have assumed the time spent in a transition of a Markov process to be unity. The actual time spent may not be unity. This assumption makes no different to us in the MDP. But if the time spent in each transition is relevant to the performance metric, then one must model the problem with a semi-Markov process. We will treat this matter with more care in our discussions on semi-Markov decision problems (SMDPs).

In this section, we will present a general expression for the average reward per unit time in an MDP. Unlike the previous expression, this expression does not contain the limiting probabilities. Thus far, the expression used for average reward has used limiting probabilities. Here we will formulate a more general expression.

Table 8.3. Table showing calculations in *Relative* value iteration for average reward MDPs. The value of ϵ is 0.001. At the 11th iteration, the ϵ -optimal policy is found.

Iteration (k)	$J^k(1)$	$J^k(2)$	span
1	0	-0.7	0.7
2	0	-1.05	0.35
3	0	-1.225	0.175
4	0	-1.3125	0.0875
5	0	-1.3125	0.04375
6	0	-1.378125	0.021875
7	0	-1.389063	0.010938
8	0	-1.394531	0.005469
9	0	-1.397266	0.002734
10	0	-1.398633	0.001367
11	0	-1.399316	0.000684

If the time spent in a transition is not unity, you may view the following expression as one for the average reward per jump or transition.

The average reward per unit time calculated over an infinite time horizon, starting at state i and using a policy $\hat{\mu}$, can be expressed as:

$$\rho(i) = \lim_{k \rightarrow \infty} \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]}{k}$$

where

- k denotes the number of transitions (or time assuming that each transition takes unit time) over which the Markov chain is observed,
- x_s denotes the state from where the s th jump or transition of the Markov chain occurs under the policy μ , and
- E denotes the expectation operator.

It can be shown that for ergodic Markov chains, the average reward is independent of the starting state i . Hence $\rho(i)$ can be replaced by ρ . Intuitively, the above expression implies that the average reward is

the expected sum of rewards earned in a long trajectory of the Markov chain
the number of transitions experienced in the same trajectory

We will next turn our attention to discounted reward MDPs.

7. Dynamic programming and discounted reward

The discounted reward criterion is another popular performance metric that has been studied intensively by researchers in the context of MDPs. In this section, we will focus on the use of DP methods to find the policy in an MDP that optimizes discounted reward. Computer codes for some DP algorithms are presented in Section 6 of Chapter 15.

The idea of discounting is related to the fact that the value of money reduces with time. To give a simple example, a dollar tomorrow is worth less than a dollar today. The discounting factor is the rate by which money gets devalued. So for instance, if I earn \$3 today, \$5 tomorrow, \$6 the day after tomorrow, and if the discounting factor is 0.9 per day, then the present worth of my earnings will be:

$$3 + (0.9)5 + (0.9)^2 6.$$

The reason for raising 0.9 to the power of 2 is that tomorrow, the present worth of day-after-tomorrow's earning will be $0.9(6)$. Hence today, the present worth of this amount will be $0.9[0.9(6)] = (0.9)^2 6$.

In general, if the discounting rate per unit time (that is the discounting factor) is λ , and if $e(t)$ denotes the earning in the t th period of time, then the present worth of earnings over n periods of time can be denoted by

$$e(0) + \lambda e(1) + \lambda^2 e(2) + \cdots + \lambda^n e(n) + \cdots \quad (8.18)$$

Let us begin with the definition of the discounted reward of a policy.

7.1. Discounted reward

By the discounted reward of a policy, we mean the expected total discounted reward earned in the Markov chain when the policy is followed over an infinitely long trajectory of the Markov chain. A technical definition for the discounted reward earned with a given policy $\hat{\mu}$ starting at state i is

$$v_{\hat{\mu}}(i) = \lim_{k \rightarrow \infty} E\left[\sum_{s=1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \quad (8.19)$$

where

- λ denotes the discounting factor,
- k denotes the number of transitions (or time assuming that each transition takes unit time) over which the Markov chain is observed,
- x_s denotes the state from where the s th jump or transition of the Markov chain occurs under the policy μ , and
- E denotes the expectation operator.

In Equation (8.19)

$$\begin{aligned} E\left[\sum_{s=1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] = \\ E[r(i, \mu(i), x_2) + \lambda r(x_2, \mu(x_2), x_3) + \cdots + \lambda^{k-1} r(x_k, \mu(x_k), x_{k+1})]. \end{aligned}$$

This should make it obvious that the discounted reward of a policy is measured very much in the style discussed in Equation (8.18).

7.2. Discounted reward MDP

In the discounted reward MDP, we have more than *one* performance metric (objective function), and hence the performance metric is expressed in the form of a vector. Remember that in the average reward case, the performance metric is the unique scalar quantity (ρ) — the average reward of a policy. In the

discounted problem, associated with every state, there exists a scalar quantity; we seek to maximize each of these quantities. These quantities are the elements of the so-called value function vector. Of course, there is no such quantity as ρ in the discounted problem!

The value function for discounted reward was actually defined in Equation (8.19). The term $v_{\hat{\mu}}(i)$ in Equation (8.19) denotes the i th element of the value function vector. The number of elements in the value function vector is equal to the number of states.

The definition in Equation (8.19) is associated with a policy $\hat{\mu}$. The point to be noted is that the value function will depend on the policy. (Of course it will also depend on the discounting factor, the transition probabilities, and the transition rewards.)

Solving an MDP means identifying the policy that will return a value function vector \vec{v}^* such that

$$v^*(i) = \max_{\hat{\mu}} v_{\hat{\mu}}(i)$$

for each $i \in S$.

The above means that the optimal policy will have a value function vector which satisfies the following property: each element of the vector is greater than or equal to the corresponding element of the value function vector of any other policy. This concept is best explained with an example.

Consider a 2-state Markov chain with 4 allowable policies denoted by $\hat{\mu}_1, \hat{\mu}_2, \hat{\mu}_3$, and $\hat{\mu}_4$. Let the value function vector be defined by

$$\begin{array}{llll} v_{\hat{\mu}_1}(1) = 3 & v_{\hat{\mu}_2}(1) = 8 & v_{\hat{\mu}_3}(1) = -4 & v_{\hat{\mu}_4}(1) = 12 \\ v_{\hat{\mu}_1}(2) = 7 & v_{\hat{\mu}_2}(2) = 15 & v_{\hat{\mu}_3}(2) = 1 & v_{\hat{\mu}_4}(2) = 42 \end{array}$$

Now from our definition of an optimal policy, policy $\hat{\mu}_4$ should be the optimal policy since the value function vector assumes the maximum value for this policy for each state.

Now, the following question should rise in your mind at this stage. What if there is no policy for which the value function is maximized for each state? For instance consider the following scenario:

$$\begin{array}{llll} v_{\hat{\mu}_1}(1) = 3 & v_{\hat{\mu}_2}(1) = 8 & v_{\hat{\mu}_3}(1) = -4 & v_{\hat{\mu}_4}(1) = 12 \\ v_{\hat{\mu}_1}(2) = 7 & v_{\hat{\mu}_2}(2) = 15 & v_{\hat{\mu}_3}(2) = 1 & v_{\hat{\mu}_4}(2) = -5 \end{array}$$

In this scenario, there is no policy for which the value function is maximized in each state. Fortunately, it has been proved that there exists an optimal policy; in other words, there exists a policy for which the value function is maximized in *each state*. We will not prove it here but the interested reader is referred to Sennott [157] (pages 60-64), among other sources, for the proof. Thus, we should never face a situation such as the one shown above.

The important point that we need to address next is: how does one find the value function of any given policy? Equation (8.19) does not provide us with any mechanism for this purpose. Not surprisingly, we will have to turn to the Bellman equation of a given policy.

7.3. Bellman equation for a policy: discounted reward

The Bellman equation for a given policy is a system of linear equations in which the unknowns are the elements of the value function associated with the policy.

The Bellman equation for a given policy for the discounted reward MDP is

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) + \lambda \sum_{j=1}^{|S|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \quad (8.20)$$

The number of equations in this system is equal to $|\mathcal{S}|$ — the number of elements in the set \mathcal{S} . The unknowns in the equation are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$. The other terms are defined below.

- λ is the discounting factor,
- $\mu(i)$ denotes the action selected in state i under the policy $\hat{\mu}$. Since the policy is known, each $\mu(i)$ is known,
- $\bar{r}(i, \mu(i))$ denotes the expected immediate reward in state i under policy $\mu(i)$, and
- $p(i, \mu(i), j)$ denotes the one-step transition probability of jumping from state i to state j under the policy $\hat{\mu}$.

By solving the Bellman equation, one can obtain the value function vector associated with a given policy. Clearly, the value function vectors associated with each policy can be evaluated by solving the respective Bellman equations. Then from the value function vectors obtained, it is possible to determine the optimal policy. This method is called the method of exhaustive enumeration.

The method of exhaustive enumeration, as discussed in the case of average reward, is not a very efficient method to solve the MDP since its computational burden is enormous. For a problem of 10 states with two allowable actions in each, one would need to evaluate 2^{10} policies. The method of policy iteration is considerably more efficient.

7.4. Policy iteration for discounted reward MDPs

Like in the average reward case, the basic principle of policy iteration is to start from an arbitrary policy and then move to a better policy. This has to continue until no further improvement is possible.

When a policy is selected, the Bellman equation for a policy (Equation (8.20)) is used to find the value function vector for that policy. This is known as the policy evaluation stage because in this stage we evaluate the value function vector associated with a policy. Then the value function vector is used to find a better policy. This step is called the policy improvement step. The value function vector of the new policy is then found. In other words, one goes back to the policy evaluation step. This continues until the value function vector obtained from solving the Bellman equation cannot produce a better policy.

7.4.1 Steps

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select an arbitrary policy. Let us denote the policy selected in the k th iteration by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation:) Solve the following linear system of equations for the unknown h^k terms. For $i = 1, 2, \dots, |\mathcal{S}|$

$$h^k(i) = \bar{r}(i, \mu_k(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h^k(j). \quad (8.21)$$

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j)].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i , then stop and set $\mu^*(i) = \mu_k(i)$ for every i . Otherwise, increment k by 1, and go back to the second step.

Policy iteration on Example A. We used policy iteration on Example A from Section 5.1. The results are shown in Table 8.4. The optimal policy is $(2, 1)$.

Like in the average reward case, we will next discuss the value iteration method. The value iteration method is also called the method of successive approximations (in the discounted reward case). This is because the successive application of the Bellman operator in the discounted case does lead one to the optimal value function. Recall that in the average reward case, the value iteration operator may not keep the iterates *bounded*. Fortunately this is not the case in the discounted problem.

7.5. Value iteration for discounted reward MDPs

Like in the average reward case, the major difference between value iteration and policy iteration is that unlike in policy iteration, in value iteration, one does not have to solve any equations. This is a big plus — not so much in solving small MDPs, but in the context of simulation-based DP (reinforcement learning). We will discuss this issue in more detail in Chapter 9.

We begin by presenting the Bellman optimality equation for discounted reward.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^*(j)] \text{ for each } i \in \mathcal{S}. \quad (8.22)$$

The following remarks will explain the notation.

- $\mathcal{A}(i)$ is the set of actions allowed in state i .
- The J^* terms are the unknowns. They are the components of the optimal value function vector \vec{J}^* . The number of elements in the vector \vec{J}^* equals the number of states in the MDP.
- The term $\bar{r}(i, a)$ denotes the expected immediate reward in state i when action a is selected in state i . Each of these terms can be calculated from the TPMs and the TRMs.
- The term $p(i, a, j)$ denotes the one-step transition probability of jumping from state i to state j when action a is selected in state i . Again, these terms can be obtained from the TPMs.
- The term λ , as usual, denotes the discounting factor.

Equation (8.22), that is, the Bellman optimality equation for discounted reward contains the max operator; hence it cannot be solved using linear algebra techniques such as Gauss elimination. However, the value iteration method

Table 8.4. Table showing calculations in policy iteration for discounted MDPs

Iteration (k)	Policy Selected (μ^k)	Values
1	(1, 1)	$h^1(1) = 25.026$ $h^1(2) = 34.631$
2	(2, 1)	$h^2(1) = 53.03$ $h^2(2) = 51.87$

forms a convenient method to solve it. Solving the equation also yields an ϵ -optimal policy.

In value iteration, one starts with some arbitrary values for the value function vector. Then a transformation, derived from the Bellman optimality equation, is applied on the vector successively till the vector starts approaching a fixed value. The fixed value is also called a *fixed point*. We will discuss issues such as convergence to fixed points in Chapter 13 in a more mathematically rigorous framework. However, at this stage, it is important to get an intuitive feel of a fixed point.

If a transformation has a unique fixed point, then no matter what vector you start with, if you keep applying the transformation repeatedly, you will reach the fixed point. Several operations research algorithms are based on such transformations. The value iteration algorithm for discounted reward is of course one example. The Weiszfeld's algorithm, a popular algorithm in facilities designing (see Heragu [71]), is another example.

We will next present the step-by-step details of the value iteration algorithm. In Step 3, we will need to calculate the max norm of a vector. See Chapter 2 for a definition of max norm.

7.5.1 Steps

Step 1: Set $k = 1$ and select an arbitrary vector \vec{J}^1 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)].$$

Step 3: If

$$\|(\vec{J}^{k+1} - \vec{J}^k)\|_\infty < \epsilon(1 - \lambda)/2\lambda,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)]$$

and stop.

Several important remarks are in order here.

- In Step 2, we have used a transformation derived from the Bellman equation.
- The max-norm of the difference vector $(\vec{J}^{k+1} - \vec{J}^k)$ decreases with every iteration. The reason for the use of the expression $\epsilon(1 - \lambda)/2\lambda$ in Step 3 will be explained in Chapter 13. The condition in Step 3 ensures that when

the algorithm terminates, the max norm of the difference between the value function vector returned by the algorithm and the *optimal* value function vector is ϵ . Since the max norm is in some sense the “length” of the vector, the implication here is that the vector generated by the algorithm is different from the optimal value function vector by a length of ϵ . Thus by making ϵ small, one can obtain a vector that is as close to the optimal vector as one desires.

- Unlike the average reward case, the values do *not* become unbounded in this algorithm.
- The algorithm’s speed (which is inversely proportional to the number of iterations needed to terminate) can be increased by other methods that we will discuss in the next subsection.

Example A and relative value iteration. Table 8.5 shows the calculations with using value iteration on Example A from the Section 5.1. The discounting factor is 0.8.

7.6. Getting value iteration to converge faster

The regular value iteration algorithm that we discussed above may not be the best approach to do value iteration since it is a slow algorithm and takes many iterations to converge. Of course, the number of iterations needed for convergence of value iteration depends on the value of ϵ chosen by the user. However, even for a given value of ϵ , we know of some variants of regular value iteration that may converge in fewer iterations.

We will discuss three variants. They are

1. Gauss Siedel value iteration algorithm,

Table 8.5. Table showing calculations in value iteration for discounted reward MDPs. The value of ϵ is 0.001. The norm is checked with $0.5\epsilon(1 - \lambda)/\lambda = 0.00125$. At the 52nd iteration, the ϵ -optimal policy is found.

Iteration	$v(1)$	$v(2)$	norm
1	10.700	10.000	10.7
2	19.204	18.224	8.504
3	25.984	24.892	6.781
.
.
52	53.032	51.866	0.000121

2. the relative value iteration method (this was discussed in the average reward problem),
3. use of span semi-norm for termination purposes (We have seen this too in the context of average reward).

(Recall that in the average reward case, we *must* use the span for termination because the algorithm does not converge in any norm.) The span method may not yield an enhanced convergence rate. This is especially true of problems with sparse transition probability matrices. (A sparse matrix has many zeros.)

7.6.1 Gauss Siedel value iteration

The Gauss Seidel value iteration algorithm is the first **asynchronous** algorithm to be discussed in this book. It differs from the regular value iteration algorithm in that the updating mechanism uses “current” versions of the elements of the value function. This needs to be explained in more detail.

Review value iteration from the previous subsection. In Step 2, $J^{k+1}(i)$ is obtained based on the values of the vector \vec{J}^k . Consider a two-state example. When $J^{k+1}(2)$ will be calculated, one has already computed $J^{k+1}(1)$. But Step 2 says that we must use $J^k(1)$. This is where the Gauss-Seidel version of value iteration differs from regular value iteration. In Step 2 of Gauss Seidel, we will use $J^{k+1}(1)$.

In general, we will use $J^{k+1}(i)$ of every i for which the $(k + 1)$ th estimate is available. In other words, in the updating of the values, we will use the latest estimate of the value. A step-by-step description should make these ideas clear.

Steps.

Step 1: Set $k = 1$ and select an arbitrary vector \vec{J}^1 . Specify $\epsilon > 0$. Select any arbitrary state from \mathcal{S} to be the distinguished state i^* .

Step 2: Set $i = 1$.

Step 2a. For $l = 1, 2, \dots, i - 1$,

$$\text{set } w(l) = J^{k+1}(l).$$

Step 2b. For $l = i, i + 1, \dots, |\mathcal{S}|$

$$\text{set } w(l) = J^k(l).$$

Step 2c. Compute

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)w(j)] - w(i^*).$$

Increment i by 1. If $i > |\mathcal{S}|$, go to Step 3 else return to Step 2a.

Step 3: If

$$\|(\vec{J}^{k+1} - \vec{J}^k)\| < \epsilon(1 - \lambda)/2\lambda,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)]$$

and stop.

Example A and Gauss-Siedel value iteration. In Table 8.6, we show the calculations with using Gauss-Seidel value iteration on Example A from the section 5.1. The discounting factor is 0.8.

7.6.2 Relative value iteration for discounted reward

The relative value iteration algorithm in the average reward MDP keeps the iterates bounded and thus helps convergence of the values to the value function vector. In the discounted case, regular value iteration itself keeps values bounded and as such relative value iteration is not needed to keep iterates bounded. However, the relative value iteration algorithm can accelerate the rate of convergence in the norm. Hence we present its steps next.

Steps.

Step 1: Select any arbitrary state from \mathcal{S} to be the distinguished state i^* . Set $k = 1$ and select an arbitrary vector \vec{J}^1 with $J^1(i^*) = 0$. Specify $\epsilon > 0$.

Table 8.6. Table showing calculations in value iteration for discounted reward MDPs. The value of ϵ is 0.001. The norm is checked with $0.5\epsilon(1 - \lambda)/\lambda = 0.00125$. At the 32nd iteration, the ϵ -optimal policy is found.

Iteration (k)	$J^k(1)$	$J^k(2)$	norm
1	12.644	18.610	18.612500
2	22.447	27.906	9.803309
.
.
32	39.674	43.645	0.000102

Step 2: Compute:

$$J^{k+1}(i^*) \leftarrow \max_{a \in \mathcal{A}(i^*)} [\bar{r}(i^*, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i^*, a, j) J^k(j)].$$

Step 3: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)] - J^{k+1}(i^*).$$

Step 4: If

$$\|(\vec{J}^{k+1} - \vec{J}^k)\| < \epsilon,$$

go to Step 5. Otherwise increase k by 1 and go back to Step 2.

Step 5: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)]$$

and stop. The ϵ -optimal policy is \hat{d} .

7.6.3 Span seminorm termination

In Step 4, the span seminorm can be used to terminate the algorithm instead of using the norm. (For a definition of span, see the discussion on value iteration in the context of average reward.) Sometimes the span converges much faster than the norm. Hence, it is always a good idea to use the span rather than the norm. We next present the details of value iteration which uses the span for termination purposes rather than the norm. The termination criterion in Step 3 (see below) has been proved to generate an ϵ -optimal policy in Proposition 6.6.5 of Puterman [140] (Page 201).

Steps.

Step 1: Set $k = 1$ and select an arbitrary vector \vec{J}^1 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon(1 - \lambda)/\lambda,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)]$$

and stop. The ϵ -optimal policy is \hat{d} .

8. The Bellman equation: An intuitive perspective

The validity of the Bellman equation will be proved using mathematical arguments in Chapter 13. However, in this section, we will motivate the Bellman equation in an intuitive manner. The intuitive explanation will throw considerable insight on the *physical* meaning of the value function.

We consider the Bellman equation for a given policy in the discounted case. The value function of a given policy for a given state is the *expected total discounted reward* earned over an infinitely long trajectory, if the given policy is adhered to in *every* state visited. Keeping this in mind, let us strive to obtain an expression for the value function of a given state i for a policy \hat{d} . Denoting this element by $v_{\hat{d}}(i)$, we have that the $v_{\hat{d}}(i)$ must equal the immediate reward earned in a transition *plus* the value function of the state to which it jumps at the end of the transition. For instance, if you were measuring the total reward along a trajectory and you encounter two states i and j along the trajectory — i first and then j — then the reward earned from i to the end of the trajectory would be the sum of 1) the immediate reward earned in going from i to j and 2) the total discounted reward earned from j to the end of the trajectory. See also Figure 8.13. This is the basic idea underlying the Bellman equation.

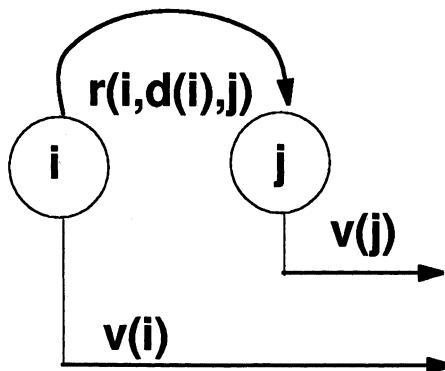


Figure 8.13. Total discounted reward calculation on an infinitely long trajectory

Let us next discuss what happens when the state transitions are probabilistic and there is a discounting factor λ . When the system is in a state i , it may jump to any one of the states in a Markov chain. Consider Figure 8.14.

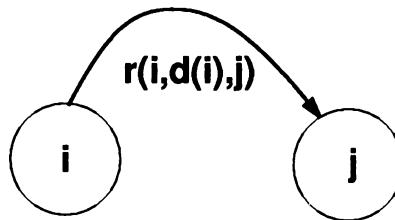


Figure 8.14. Immediate reward in one transition

So when the policy is \hat{d} , the expected total discounted reward earned from state i , which is $v_{\hat{d}}(i)$, can now be expressed as the sum of

- the immediate reward earned in going to state j and
- λ times the value function of the state j .

This is because the value function of state j denotes the expected total discounted reward from j onwards. But this quantity (that is the value function of state j) must be multiplied by λ , since it is earned one time step after leaving i . We assume that the immediate reward is earned *immediately* after state i is left. Hence the immediate reward is not multiplied by λ . However, j can actually be any one of the states in the Markov chain. Hence, in our calculation, we have to use an expectation over j . Thus, we have that

$$v_{\hat{d}}(i) = r(i, d(i), j) + \lambda \sum_{j=1}^{|S|} p(i, d(i), j) v_{\hat{d}}(j),$$

which turns out to be the Bellman equation for a policy (\hat{d}) for state i . We hope that this discussion has served as an intuitive proof of the Bellman equation. The Bellman *optimality* equation has a similar intuitive explanation.

We next discuss semi-Markov decision problems.

9. Semi-Markov decision problems

The **semi-Markov decision problem (SMDP)** is a more general version of the Markov decision problem. In other words, the Markov decision problem is a special case of the semi-Markov decision problem. In the SMDP, we assume that the time spent in any transition of the Markov chain is not necessarily unity. The time spent could be different for each transition; it could be a deterministic quantity or it could also be a random variable.

The SMDP has an **embedded** Markov chain underlying it. Hence, one may conceptualize an SMDP as a problem very like an MDP in which each transition can take an amount of time that is not equal to 1 unit. Our discussions on the SMDP will be based on this argument.

Since the transition time from one state to another depends on the state pair in question, one way to model this scenario is to store the transition times in the form of matrices. Let us first treat the case of deterministic transition times.

In the deterministic case, for a given state-state pair and for a given action, the transition time is fixed. The transition time of going from state i to state j under the influence of action a can then be denoted by $t(i, a, j)$ in a manner similar to the notation of the immediate reward ($r(i, a, j)$) and the transition probability ($p(i, a, j)$). Also, each $t(i, a, j)$ can then be stored in the form of a matrix — the transition time matrix (TTM). This matrix would be very like the transition reward matrix (TRM) or the transition probability matrix (TPM). An SMDP, in which the time spent in any transition is a deterministic quantity, will be referred to as a deterministic time Markov decision problem (DTMDP).

When the transition times are random variables from known distributions, we have what is called the most general version of an SMDP. The DTMDP is of course a special case of the SMDP. When the distribution is exponential, we have a so-called continuous time Markov decision problem (CTMDP). This is also a special case of the SMDP. A lot of beginners confuse the CTMDP with the SMDP and sometimes even with the MDP. So at the very outset, let us make a few things clear.

1. An SMDP has transition times that are **generally distributed** random variables.
2. When the transition times are exponentially distributed, one has a CTMDP. The CTMDP is a special case of the SMDP; as such, every SMDP is *not* a CTMDP.
3. The CTMDP, in its raw form, is **not** an MDP. The exponential distribution *does* have a memoryless property. This memoryless property is sometimes also called the Markov property, but it is quite distinct from the Markov assumption that we have discussed above in the context of stochastic processes. Hence the CTMDP cannot be regarded as an MDP in its raw form. However, via a process called **uniformization**, one can usually convert a CTMDP into an MDP. The “uniformized” CTMDP, which is an MDP in its own right, has transition probabilities that are **different** from those of the original CTMDP. Hence the CTMDP in its raw form is an SMDP and not an MDP.
4. Not every SMDP can be converted to an MDP through a perfect uniformization; unless the SMDP is a CTMDP, the uniformization may be approximate.

One way to handle the random transition time SMDP is to treat it as a DT-MDP by replacing the random transition time by its mean; the latter is a deterministic quantity. This paves the way for making $t(i, a, j)$ a deterministic quantity. Even when the transition times are random variables, it is possible to find the expected values of the $t(i, a, j)$ terms. In this book, we will use the DTMDP model for solving an SMDP. In the average reward case, the values of the individual $t(i, a, j)$ terms may not be needed. In any case, the DTMDP model makes it conceptually easy to understand the SMDP because one has a TTM, which is very analogous to the TPM and the TRM. Let us give some examples to make the DTMDP idea clearer.

Assume that the SMDP has two states numbered 1 and 2. The time spent in a transition from state 1 is uniformly distributed as $U(1, 2)$, while the same from state 2 is exponentially distributed with a mean of 3. Then, the DTMDP model would use the following values:

$$t(1, a, 1) = (1 + 2)/2 = 1.5,$$

$$t(1, a, 2) = (1 + 2)/2 = 1.5,$$

$$t(2, a, 1) = 3,$$

and

$$t(2, a, 2) = 3.$$

Let us consider one more example in which the time taken in a transition from state 1 to state 1 is uniformly distributed as $U(1, 9)$ and the same from state 1 to state 2 is uniformly distributed as $U(7, 9)$. Then:

$$t(1, a, 1) = (1 + 9)/2 = 5,$$

and

$$t(1, a, 2) = (7 + 9)/2 = 8.$$

It could also be that the time spent depends on the action. In any case, from the underlying distributions, it is possible to come up with the mean value for any $t(i, a, j)$.

9.1. The natural process and the decision-making process

In many MDPs and most SMDPs, we have two stochastic processes associated with the Markov chain. They are:

1. The natural process (NP).
2. The decision-making process (DMP).

Any stochastic process keeps track of the changes in the state of the associated system. The natural process keeps track of every state change that occurs. In

other words, whenever the state changes, it gets recorded in the natural process. Along the natural process, the system does not return to itself in one transition. This implies that the natural process remains in a state for a certain amount of time and then jumps to a different state.

The decision process has a different nature. It records only those states in which an action needs to be selected by the decision-maker. Thus, the decision process may come back to itself in one transition. A decision-making state is one in which the decision-maker makes a decision. All states in a Markov chain may not be decision-making states. There may be several states in which no decision is made. Thus typically a subset of the states in the Markov chain tends to be the set of decision-making states. Clearly, the decision-making process records only the decision-making states.

For example, consider a Markov chain with three states numbered 1, 2, 3, and 4. States 1 and 2 are decision-making states while 3 and 4 are not. Now consider the following trajectory:

1, 3, 4, 3, 2, 3, 2, 4, 2, 3, 4, 3, 4, 3, 4, 3, 4, 1.

In this trajectory, the NP will look identical to what we see above. The DMP however will be:

1, 2, 2, 2, 1.

This example also explains why the NP may change several times between one change of the DMP. It should also be clear that the DMP and NP coincide on the decision-making states (1 and 2).

We calculate the value functions of only the decision-making states. In our discussions on MDPs, when we said “state,” we meant a decision-making state. Technically, for the MDP, the non-decision-making states enter the analysis only when we calculate the immediate rewards earned in a transition from a decision-making state to another decision-making state. In the SMDP, calculation of the transition rewards and the transition times needs taking into account the non-decision-making states visited. This is because in this visit the system may have visited some non-decision-making states, which can dictate (1) the value of the immediate reward earned in the transition and (2) the transition time.

In simulation-based DP (reinforcement learning), the issue of determining which states are non-decision-making states becomes less critical because the simulator calculates the transition reward and transition time; as such we do not have to worry about the existence of non-decision-making states. However, if one wished to set up the Markov model, i.e., the TRM and the TPM, careful attention must be paid to this issue.

In SMDPs, the time spent in any transition of the Markov chain is not the same. In the MDP, the time is irrelevant, and as such in the calculation of the performance metric, we assume the time spent to be unity.

This means that the average reward per unit time for an MDP is actually average reward per **transition** (jump) of the Markov chain. For the SMDP, however, one must be careful in this respect. In the SMDP, average reward per unit time cannot be substituted by average reward per unit transition because the two quantities are different. In the SMDP, the average reward per transition will not mean the same thing.

Similarly, in the discounted reward case, the time consideration makes a difference. A dollar received t time units in the future has a present worth of $e^{-\lambda t}$ when the discounting factor per unit time is λ . Hence instead of using λ , the term $e^{-\lambda t}$ is used as a discounting factor in the SMDP.

In the next two subsections, we will discuss the average reward and the discounted reward SMDPs. Our discussions will be centered on algorithms that can be used to solve these problems.

9.2. Average reward SMDPs

We first need to define the average reward of an SMDP. The average reward of an SMDP is a quantity that we want to *maximize*. Recall that $t(i, a, j)$ is the transition time from state i to state j under the influence of action a . Now, the average reward, using a unit time basis, starting from a state i and following a policy $\hat{\mu}$, can be mathematically expressed as:

$$\rho_{\hat{\mu}}(i) \equiv \lim_{k \rightarrow \infty} \inf \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]}{E[\sum_{s=1}^k t(x_s, \mu(x_s), x_{s+1}) | x_1 = i]}$$

where x_s is the state from where the s th jump of the Markov chain occurs. The expectation is over the different trajectories that may be followed.

In the above, the notation “inf” denotes the infimum. An intuitive meaning of “inf” is minimum. Technically, the infimum is not equivalent to the minimum; however at this stage you can use the two interchangeably. The use of the infimum, here, implies that the average reward of a policy is the average reward earned in the *worst possible scenario*.

It can be shown that the average reward is not affected by the state from which the trajectory of the Markov chain starts. Therefore, one can get rid of i in the definition of average reward. The average reward on the other hand depends strongly on the policy used. Solving the SMDP means finding the policy that returns the highest average reward.

9.2.1 Exhaustive enumeration for average reward SMDPs

Like in the MDP case, for the SMDP, exhaustive enumeration is not an attractive proposition from the computational angle. However, from the conceptual angle, it is an important method. So we discuss it briefly, next.

Let us first define the **average** time spent in a transition from state i under the influence of action a as follows:

$$\bar{t}(i, a) = \sum_{j=1}^{|S|} p(i, a, j) t(i, a, j),$$

where $t(i, a, j)$ is the time spent in one transition from state i to state j under the influence of action a . Now, the average reward of an SMDP can also be defined as:

$$\rho_{\hat{\mu}} = \frac{\sum_{i=1}^{|S|} \pi_{\hat{\mu}}(i) \bar{r}(i, \mu(i))}{\sum_{i=1}^{|S|} \pi_{\hat{\mu}}(i) \bar{t}(i, \mu(i))} \quad (8.23)$$

where

- $\bar{r}(i, \mu(i))$ and $\bar{t}(i, \mu(i))$, respectively, denote the expected immediate reward earned and the expected time spent in a transition from state i under policy $\hat{\mu}$ and
- $\pi_{\hat{\mu}}(i)$ denotes the limiting probability of the underlying Markov chain for state i , when policy $\hat{\mu}$ is followed.

The numerator in the above denotes the expected immediate reward in any given transition, while the denominator denotes the expected time spent in any transition. The formulation of the above is based on the renewal reward theorem, which basically states that

$$\text{average reward} = \frac{\text{expected reward earned in a cycle}}{\text{expected time spent in a cycle}}.$$

For more on the use of renewal reward theorem in this definition, see Bertsekas [20].

In the next subsection, we will present a simple 2-state example SMDP. We will then show how to solve it using exhaustive enumeration. This example will be used in the rest of the book.

9.2.2 Example B

The example discussed next is similar to the Example A in Section 5.1 as far as the TPMs and the TRMs are considered. What is new here is the TTM (transition time matrix). There are two states numbered 1 and 2 in an MDP and two actions, also numbered 1 and 2, are allowed in each state. The TTM for action 1 is

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 5 \\ 120 & 60 \end{bmatrix},$$

and the same for action 2 is

$$\mathbf{T}_2 = \begin{bmatrix} 50 & 75 \\ 7 & 2 \end{bmatrix}.$$

The transition probability matrix (TPM) associated with action 1 is, hence,

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix},$$

and that for action 2 is

$$\mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

Similarly, the TRM for action 1 is

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix},$$

and the same for action 2 is

$$\mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

There are 4 possible policies that can be used to control the Markov chain:

$$\hat{\mu}_1 = (1, 1), \hat{\mu}_2 = (1, 2), \hat{\mu}_3 = (2, 1), \text{ and } \hat{\mu}_4 = (2, 2).$$

The TTMs of these policies are constructed from the individual TTMs of each action. The TTMs are:

$$\mathbf{T}_{\hat{\mu}_1} = \begin{bmatrix} 1 & 5 \\ 120 & 60 \end{bmatrix},$$

$$\mathbf{T}_{\hat{\mu}_2} = \begin{bmatrix} 1 & 5 \\ 7 & 2 \end{bmatrix},$$

$$\mathbf{T}_{\hat{\mu}_3} = \begin{bmatrix} 50 & 75 \\ 120 & 60 \end{bmatrix},$$

and

$$\mathbf{T}_{\hat{\mu}_4} = \begin{bmatrix} 50 & 75 \\ 7 & 2 \end{bmatrix}.$$

The TPMs and TRMs were calculated in Section 5.1. The value of each $\bar{t}(i, \mu(i))$ term can be calculated from the TTMs in a manner similar to that used for calculation of $\bar{r}(i, \mu(i))$. The values are:

$$\begin{aligned} \bar{t}(1, \mu_1(1)) &= p(1, \mu_1(1), 1)t(1, \mu_1(1), 1) + p(1, \mu_1(1), 2)t(1, \mu_1(1), 2) \\ &= 0.7(1) + 0.3(5) = 2.20. \end{aligned}$$

Similarly,

$$\bar{t}(2, \mu_1(2)) = 84,$$

$$\bar{t}(1, \mu_2(1)) = 2.20,$$

$$\bar{t}(2, \mu_2(2)) = 3.00,$$

$$\bar{t}(1, \mu_3(1)) = 52.5,$$

$$\bar{t}(2, \mu_3(2)) = 84.0,$$

$$\bar{t}(1, \mu_4(1)) = 52.5,$$

and

$$\bar{t}(2, \mu_4(2)) = 3.00.$$

The corresponding \bar{r} terms were calculated in Section 5.1. Then, using Equation (8.23), one obtains the average reward of each policy. The values are:

$$\rho_{\hat{\mu}_1} = 0.1564, \rho_{\hat{\mu}_2} = 2.1045, \rho_{\hat{\mu}_3} = 0.1796, \text{ and } \rho_{\hat{\mu}_4} = 0.2685.$$

Clearly, policy $\hat{\mu}_2$ is the optimal policy. The optimal policy is different from the optimal policy for the MDP (in Section 5.1), which had the same TPMs and TRMs. Although obvious, the fact to be noted is that the time spent in each state makes a difference to the average reward per unit time. So, it is the SMDP approach and not the MDP approach that will yield the optimal solution, when time is considered in measuring the average reward.

We will next discuss the more efficient DP algorithms for solving SMDPs.

9.2.3 Policy iteration for average reward SMDPs

Policy iteration of average reward MDPs has a clean and nice extension to SMDPs. The reason we say this is because value iteration of average SMDPs can get quite messy; policy iteration is definitely the method of choice for average reward SMDPs.

We will begin by presenting the Bellman equation for a given policy in the context of average reward SMDPs.

The Bellman equation for a given policy in the average reward context for SMDPs is:

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) - \rho_{\hat{\mu}} \bar{t}(i, \mu(i)) + \sum_{j=1}^{|S|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \quad (8.24)$$

The only difference between this equation and the MDP Bellman equation lies in the time term — $\bar{t}(i, \mu(i))$ term. The MDP equation can be obtained by setting the time term to 1 in the above. Like in the MDP case, the above is a system of linear equations; the number of equations is equal to the number of elements in the set \mathcal{S} . The unknowns in the equation are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$.

In Equation (8.24),

- $\mu(i)$ denotes the action selected in state i under policy $\hat{\mu}$,
- $\bar{r}(i, \mu(i))$ denotes the expected immediate reward in state i under policy $\hat{\mu}$,
- $\bar{t}(i, \mu(i))$ denotes the expected time spent in a transition from state i under policy $\hat{\mu}$,
- $p(i, \mu(i), j)$ denotes the one-step transition probability of jumping from state i to state j under policy $\hat{\mu}$, and
- $\rho_{\hat{\mu}}$ denotes the average reward per unit time associated with policy $\hat{\mu}$.

Steps. The steps needed for solving average reward SMDPs using policy iteration are given next.

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select an arbitrary policy. Let us denote the policy selected by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation:) Solve the following linear system of equations.

$$h^k(i) = \bar{r}(i, \mu_k(i)) - \rho^k \bar{t}(i, \mu_k(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu_k(i), j) h^k(j). \quad (8.25)$$

Here one linear equation is associated with each value of i . In this system, the unknowns are the h^k terms and ρ^k . Any one of the h^k terms should be set to 0.

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) - \rho^k \bar{t}(i, \mu_k(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j)].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i then stop and set $\mu^*(i) = \mu_k(i)$ for every i . Otherwise, increment k by 1 and go back to the second step.

Policy iteration on Example B. We will show the use of policy iteration on generic example from Section 9.2.2. The results are shown in Table 8.7. The optimal policy is $(2, 1)$.

9.2.4 Value iteration for average reward SMDPs

Value iteration for SMDPs in the average reward context gets us into inexact and approximate territory. We will pursue the topic however because value iteration is easy to code and holds a special place in simulation-based stochastic DP (reinforcement learning).

Recall that value iteration for MDPs in the average reward context also poses a few problems. For instance, the value of the average reward (ρ^*) of the optimal policy is seldom known beforehand, and hence the Bellman optimality equation cannot be used directly. Let us first analyze the Bellman optimality equation for average reward SMDPs.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) - \rho^* \bar{t}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^*(j)] \text{ for each } i \in \mathcal{S}. \quad (8.26)$$

The following remarks will explain the notation.

- $\mathcal{A}(i)$ is the set of actions allowed in state i .
- The J^* terms are the unknowns. They are the components of the optimal value function vector \vec{J}^* . The number of elements in the vector \vec{J}^* equals the number of states in the SMDP.
- The term $\bar{r}(i, a)$ denotes the expected immediate reward in state i when action a is selected in state i .
- The term $\bar{t}(i, a)$ denotes the expected time of transition from state i when action a is selected in state i .
- The term $p(i, a, j)$ denotes the one-step transition probability of jumping from state i to state j when action a is selected in state i . These terms can be obtained from the TPMs.

Table 8.7. Table showing calculations in policy iteration for average reward SMDPs (Example B)

Iteration (k)	Policy Selected ($\hat{\mu}^k$)	Values	ρ
1	(1, 1)	$h^1(1) = 0$ $h^1(2) = -7.852$	0.0156
2	(2, 2)	$h^2(1) = 0$ $h^2(2) = 33.972$	0.2685
3	(1, 2)	$h^3(1) = 0$ $h^3(2) = 6.432$	2.1044

- The term ρ^* denotes the average reward associated with the optimal policy.

Now in an MDP, although ρ^* is unknown, it is acceptable to replace ρ^* by 0 (regular value iteration), or to replace it by the value function associated with some state of the Markov chain. In the MDP case, this replacement does not affect the policy to which value iteration converges (although it does affect the actual values of the value function to which the algorithm converges). However, in the SMDP, such a replacement will violate the Bellman equation. This is the reason why we do not have a regular value iteration algorithm for the average reward SMDP. The following example serves as a *counterexample* (an exception) to the regular value iteration algorithm in SMDPs.

9.2.5 Counterexample for regular value iteration

We will now show, via an example, that regular value iteration cannot be used in average reward SMDPs. Furthermore, this will prove that *relative* value iteration of the kind prescribed for MDPs is also unusable for average reward SMDPs. Let us define $w(i, a)$ as follows:

$$w(i, a) \equiv \bar{r}(i, a) - \rho^* \bar{t}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)$$

for all (i, a) pairs, where $J^k(i)$ denotes the estimate of the value function element for the i th state in the k th iteration of the value iteration algorithm. Let us define $w'(i, a)$ as follows:

$$w'(i, a) \equiv \bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)$$

for all (i, a) pairs.

Now, consider an SMDP with two actions in each state, where $\bar{t}(i, 1) \neq \bar{t}(i, 2)$. Value iteration, using the Bellman equation, which is

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) - \rho^* \bar{t}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)],$$

can then be written as:

$$J^{k+1}(i) \leftarrow \max\{w(i, 1), w(i, 2)\}. \quad (8.27)$$

Regular value iteration, which is

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^k(j)],$$

can be written as:

$$J^{k+1}(i) \leftarrow \max\{w'(i, 1), w'(i, 2)\}. \quad (8.28)$$

Now, since $\bar{t}(i, 1) \neq \bar{t}(i, 2)$, it is entirely possible that using (8.27), one obtains:

$$J^{k+1}(i) = w(i, 1),$$

while using (8.28), one obtains:

$$J^{k+1}(i) = w'(i, 2).$$

In other words, the maximizing action selected in Step 2 of the two forms of value iteration does not have to be the same. This can cause the regular value iteration algorithm to stray from the path that the Bellman equation value iteration algorithm would have followed.

Notice that in the MDP case, since each $\bar{t}(i, a)$ is 1, the same action will be selected in Step 2. So, in the MDP case, if the Bellman equation value iteration selects $w(i, u)$, then the regular value iteration algorithm would also select $w'(i, u)$. Also, notice that in the MDP case, $w(i, u)$ would differ from $w'(i, u)$ by a constant ρ^* , but both algorithms would keep selecting the same maximizing action in Step 2; and thus we can conclude that in the MDP case, both algorithms would proceed along the same path.

9.2.6 Uniformization for SMDPs

A number of ways can be used for working around this difficulty. However each of these approaches leads us into approximate territory. One recommended procedure (that we have mentioned above) is to “uniformize” the SMDP to an MDP. If the SMDP is not a CTMDP, the uniformization is not exact. We will discuss this procedure, next.

The uniformization seeks to transform the SMDP into an MDP by converting the immediate rewards to rewards measured on a unit time basis. The “uniformized” Bellman optimality equation becomes:

$$J^*(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}^t(i, a) + \sum_{j=1}^{|\mathcal{S}|} p^t(i, a, j) J^*(j)], \text{ for all } i \in \mathcal{S},$$

where the replacements for $\bar{r}(i, a)$ and $p(i, a, j)$ are denoted by $\bar{r}^t(i, a)$ and $p^t(i, a, j)$, respectively, and are defined as:

$$\bar{r}^t(i, a) = \bar{r}(i, a)/y(i, a),$$

and

$$p^t(i, a, j) = \eta p(i, a, j)/y(i, a), \text{ if } i \neq j,$$

or

$$p^t(i, a, j) = 1 + \eta[p(i, a, j) - 1]/y(i, a), \text{ if } i = j.$$

In the above, the following must hold:

$$0 \leq \eta \leq y(i, a)/(1 - p(i, a, j)) \text{ for all } a, i \text{ and } j.$$

Now it is possible to use value iteration using the transformed Bellman equation to solve the SMDP. It should be clear that since the value of η is not unique, we cannot expect a unique solution. One may require some trial and error to find a good value for η that will generate a good solution.

If the iterates get unbounded, we can use a relative value iteration algorithm. We can use the relative value iteration algorithm because via uniformization we enter the MDP territory — and then all the regular MDP algorithms can be used for solution purposes.

It is to be noted that the uniformization changes the transition probabilities of the Markov chain.

9.2.7 Value iteration based on the Bellman equation

The other approach (for the SMDP) is to use the Bellman equation, as it is, in a value iteration scheme. One would have to use a “guess” of the average reward. This is possible if one can do some best-case analysis of the system and come up with an upper bound on the average reward. Then the guessed value of the best average reward is used in place of ρ^* in the Bellman optimality equation to formulate a value iteration algorithm. In simulation-based stochastic DP (reinforcement learning), using a sensible guess often works. We will discuss this in the next chapter. If the convergence to the optimal solution is of top priority, value iteration may not be a very attractive proposition.

9.2.8 Extension to random time SMDPs

In the average reward case, one may not need to view each $t(i, a, j)$ as a deterministic quantity, since in the Bellman equation, one needs deterministic values for $\bar{t}(i, a)$ — which, by definition, is a deterministic quantity. For any random time SMDP, hence, it is possible to determine the exact value of each of the $\bar{t}(i, a)$ terms. In other words, the algorithms that we have described should work for random time SMDPs as well.

9.3. Discounted reward SMDPs

We must begin with the definition of an appropriate discounting factor for the SMDP.

A dollar gained t time units in the future has a present worth of $e^{-\gamma t}$ if the discounting factor per unit time is γ . Therefore λ in the MDP context will be replaced by $e^{-\gamma t}$ to obtain the Bellman equation for the SMDP.

Let us first discuss the Bellman equation for a policy for the discounted SMDP (DTMDP to be more precise). The equation is given by:

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) + \sum_{j=1}^{|S|} e^{-\gamma t(i, \mu(i), j)} p(i, \mu(i), j) h_{\hat{\mu}}(j)$$

for each $i \in \mathcal{S}$.

The role of the discounting factor is played by $e^{-t(i, \mu(i), j)}$.

9.3.1 Policy iteration for discounted SMDPs

In what follows, we present a discussion on the use of policy iteration in solving discounted reward SMDPs. The policy iteration algorithm is very similar to that used for MDPs. The discounting factor accommodates the time element.

Steps.

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be denoted by \mathcal{S} . Select an arbitrary policy. Let us denote the policy selected by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation:) Solve the following linear system of equations. For $i = 1, 2, \dots, |\mathcal{S}|$,

$$h^k(i) = \bar{r}(i, \mu_k(i)) + \sum_{j=1}^{|S|} e^{-\gamma t(i, a, j)} p(i, \mu(i), j) h^k(j). \quad (8.29)$$

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) = \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|S|} e^{-\gamma t(i, a, j)} p(i, a, j) h^k(j)].$$

If possible one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i , then stop, and set $\mu^*(i) = \mu_k(i)$ for every i . Otherwise, increment k by 1, and go back to the second step.

We will next examine the value iteration algorithm in the context of discounted reward SMDPs.

9.3.2 Value iteration for discounted reward SMDPs

Value iteration can be carried out in an *exact* sense on discounted reward SMDPs. This is in contrast to the semi-exact uniformization approach for the average reward case.

Steps.

Step 1: Set $k = 1$ and select an arbitrary vector \vec{J}^1 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} e^{-\gamma t(i, a, j)} p(i, a, j) J^k(j)].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1, and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_j e^{-\gamma t(i, a, j)} p(i, a, j) J^k(j)],$$

and stop.

Remark: Step 3 is somewhat approximate because there is no one fixed value for the discounting factor. This means that when the algorithm terminates, the value function is not necessarily ϵ close to the optimal solution.

9.3.3 Extension to random time SMDPs

The algorithms discussed in the previous two subsections are actually designed for DTMDPs. If the SMDP has deterministic values for $t(i, a, j)$, then clearly the SMDP is a DTMDP, and then the algorithms discussed above are sufficient.

However if each of the $t(i, a, j)$ terms is random with its own distribution (that is, when the problem is a random time SMDP), then one way to handle the problem, as discussed previously, is to find the mean value of each $t(i, a, j)$ term and replace each $t(i, a, j)$ by its mean. This would make each $t(i, a, j)$ a deterministic quantity, and one would have a DTMDP.

9.3.4 Uniformization

Uniformization, which we have discussed above, can be used to convert a CTMDP to an MDP. This is usually the best approach to solve a CTMDP.

Of course, the motivation for this is that the (uniformized) MDP is like any other MDP, and as such can be solved by regular MDP methods. We, next, briefly discuss how an MDP can be generated from a CTMDP.

In an MDP, the system jumps after unit time. In the raw (unaltered) CTMDP this is not the case. Hence a jump of that nature needs to be introduced

— artificially if necessary — into the CTMDP to convert it to an MDP. Uniformization introduces such transitions, and in the process alters the contents of the transition probability and the transition reward matrices. The MDP can then be solved with regular methods used for solving it.

We would like to remind the reader that the exponential distribution of the CTMDP has several properties that enable us to simplify its analysis. In fact, introducing fictitious transformations can be done quite easily with the exponential distributions that govern CTMDPs. However, obviously the exponential distribution is not a universal distribution, and it is the case that those random time SMDPs that are not governed by exponential distributions cannot be uniformized in the exact sense.

We will not deal with the topic of uniformization in any more detail because the focus of this book is on simulation-based DP, where the transition probabilities are usually not available.

10. Modified policy iteration

Thus far, our discussion on solving MDPs and SMDPs has been limited to some of the fundamental algorithms of DP — regular value and policy iteration and their variants. Another approach goes by the name **modified policy iteration**. This approach has resulted from the work of several researchers. Action elimination is yet another approach that can lead to considerable computational advantages. Action elimination combined with modified policy iteration, which is a mixture of value and policy iteration, has also been suggested in the literature. Section 6.5 and section 6.7 of Puterman's book [140] present an insightful and detailed account on these methods. We will discuss, in some detail, the modified policy iteration algorithm.

The modified policy iteration algorithm combines advantages of value iteration and policy iteration and in some sense is devoid of the disadvantages of both. At this point, a discussion on the relative merits and demerits of both approaches is in order.

Policy iteration converges in less iterations but forces us to solve a system of linear equations in every iteration. Value iteration is slower than policy iteration but does not require the solution of any linear system of equations.

Solving linear equations is not difficult if one has a small system of linear equations. But in this book, we want to deal with models which can work without transition probabilities, and as such formulating a linear system will be something we will try to avoid. The real use of this algorithm will therefore be seen in simulation-based approaches to DP.

Modified policy iteration uses value iteration in the policy evaluation stage of policy iteration; thereby it avoids having to solve linear equations. (Instead of using Gauss elimination or some other linear equation solver, the algorithm uses value iteration to solve the system). However, it uses the scheme of policy

iteration of searching in the policy space. In other words, it goes from one policy to another in search of the best.

An interesting aspect of the modified policy iteration algorithm is that the policy evaluation stage can actually be an *incomplete* value iteration. Let us first discuss modified policy iteration for discounted reward MDPs.

10.1. Steps for discounted reward MDPs

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be denoted by \mathcal{S} . Assign arbitrary values to a vector \vec{h}^k . Choose a sequence $\{m^k\}_{k=0}^{\infty}$ where the elements of this sequence take arbitrary non-negative integer values. Examples are $\{5, 6, 7, 8, \dots\}$ and $\{6, 6, 6, 6, \dots\}$.

Step 2. (Policy Improvement) Choose a policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} \bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j).$$

If possible one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$ when $k > 0$.

Step 3. (Partial Policy Evaluation:)

Step 3a. Set $q = 0$ and for each $i \in \mathcal{S}$, compute:

$$W^q(i) \leftarrow \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) J^k(j)].$$

Step 3b. If

$$\|(\vec{W}^q - \vec{J}^k)\| < \epsilon(1 - \lambda)/2\lambda,$$

go to Step 4. Otherwise go to Step 3c.

Step 3c. If $q = m^k$, go to Step 3e. Otherwise, for each $i \in \mathcal{S}$, compute:

$$W^{q+1}(i) \leftarrow [\bar{r}(i, \mu_{k+1}(i)) + \lambda \sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) W^q(j)].$$

Step 3d. Increment q by 1, and return to Step 3c.

Step 3e. For each $i \in \mathcal{S}$, set:

$$J^{k+1}(i) \leftarrow W^{m^k}(i),$$

increment k by 1 and return to Step 2.

Step 4. The policy $\hat{\mu}_{k+1}$ is the ϵ -optimal policy.

Several interesting remarks are in order.

- A Gauss-Seidel version of the value iteration performed in Step 3a and Step 3c exists. This can further enhance the rate of convergence.
- The algorithm generates an ϵ -optimal policy unlike regular policy iteration.
- The policy evaluation step is actually an incomplete (partial) policy evaluation.
- The algorithm can be shown to converge for any values of the sequence $\{m^k\}_{k=0}^\infty$. However, the values of the elements can affect the convergence rate.
- The choice of the elements in the sequence $\{m^k\}_{k=0}^\infty$ naturally affects how complete the policy evaluation is. An increasing sequence, such as $\{5, 6, 7, \dots\}$, is usually a smart choice since then the evaluation becomes more and more complete as we approach the optimal policy. In the first few iterations, an incomplete policy evaluation should not cause much trouble.
- One option is to select values for $\{m^k\}_{k=0}^\infty$ in an adaptive manner. For example, a rule such as $\|\tilde{W}^{q+1} - \tilde{W}^q\| < \xi$ would require that the policy evaluation step be performed to some desirable level.
- In Chapter 9, we will see the usefulness of this algorithm in the context of simulation-based DP. The advantage, as mentioned above, is that no linear equation solving is necessary in what is essentially a search in policy space.

We next present the algorithmic details of modified policy iteration in the average reward MDP.

10.2. Steps for average reward MDPs

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be denoted by \mathcal{S} . Assign arbitrary values to a vector \vec{h}^k . Choose a sequence $\{m^k\}_{k=0}^\infty$ where the elements of this sequence take arbitrary non-negative integer values. Examples are $\{5, 6, 7, 8, \dots\}$ and $\{6, 6, 6, 6, \dots\}$.

Step 2. (Policy Improvement) Choose a policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) = \arg \max_{a \in \mathcal{A}(i)} \bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j).$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$, when $k > 0$.

Step 3. (Partial Policy Evaluation:)

Step 3a. Set $q = 0$, and for each $i \in \mathcal{S}$, compute:

$$W^q(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j \in \mathcal{S}} p(i, a, j) J^k(j)].$$

Step 3b. If

$$sp(\vec{W}^q - \vec{J}^k) \leq \epsilon,$$

go to Step 4. Otherwise go to Step 3c.

Step 3c. If $q = m^k$, go to Step 3e. Otherwise, for each $i \in \mathcal{S}$, compute:

$$W^{q+1}(i) = [\bar{r}(i, \mu_{k+1}(i)) + \sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) W^q(j)].$$

Step 3d. Increment q by 1 and return to Step 3c.

Step 3e. For each $i \in \mathcal{S}$, set:

$$J^{k+1}(i) = W^{m^k}(i),$$

increment k by 1 and return to Step 2.

Step 4. The policy $\hat{\mu}_{k+1}$ is the ϵ -optimal policy.

Some element of the value function \vec{W} can become unbounded because value iteration for average reward can make the iterates unbounded. As a safeguard, one can always use relative value iteration in the policy evaluation step of modified policy iteration.

11. Miscellaneous topics related to MDPs and SMDPs

In this section, we intend to cover, briefly, some other topics related to Markov decision theory. We will first discuss a parametric optimization approach to solving MDPs via linear programming. Then we will present the MDP as a special case of a game-theoretic problem. Thereafter we will discuss the finite horizon MDPs. We conclude this section with a brief account on Sennott's approximating sequence method for denumerably infinite state-space MDPs.

11.1. A parametric-optimization approach to solving MDPs

It is possible to set up both the average reward and the discounted reward MDPs as linear programs. Linear programming (LP) happens to be a heavily researched topic of operations research, and efficient methods exist for solving relatively large linear programs. However, in the process of casting even a small MDP as a linear program, one usually ends up in creating very large linear programs. Let us first consider the average reward case.

Average reward MDPs: An LP formulation . Let us define $\mathcal{A}(i)$ to be the set of actions allowed in state i . The linear programming formulation is:

Minimize ρ

subject to

$$\rho + v(i) - \sum_{j=1}^{|S|} p(i, \mu(i), j)v(j) \geq \bar{r}(i, \mu(i)),$$

for $i = 1, 2, \dots, |S|$ and all $\mu(i) \in \mathcal{A}(i)$,

$v(j)$ is URS (unrestricted in sign) for $j = 1, 2, \dots, |S|$, and

ρ is also URS.

The decision variables are the v terms and ρ . Once we have values for the optimal vector \vec{v} , the optimal policy can be derived from the vector \vec{v} . (See the last step of any value iteration algorithm.) For an elaborate discussion on this topic, the reader is referred to any standard text on stochastic dynamic programming ([168, 140]. We reiterate that, as of now, this does not seem to be the most efficient method to solve the MDP. But innovative research that may reduce the computational burden of the linear program may happen in the future.

Let us next consider the discounted reward case.

Discounted reward MDPs: An LP formulation . It can be shown that if

$$v(i) \geq \bar{r}(i, \mu(i)) + \lambda \sum_{j=1}^{|S|} p(i, \mu(i), j)v(j)$$

for all policies $\hat{\mu}$, then $v(i)$ is an upper bound for the optimal value $v^*(i)$. This paves the way for a linear programming formulation of the MDP. Let us define $\mathcal{A}(i)$ to be the set of actions allowed in state i . The formulation is:

Minimize $\sum_{j=1}^{|S|} x(j)v(j)$

subject to

$$\sum_{j=1}^{|S|} x(j) = 1,$$

$$v(i) - \lambda \sum_{j=1}^{|S|} p(i, \mu(i), j)v(j) \geq \bar{r}(i, \mu(i)),$$

for $i = 1, 2, \dots, |S|$ and all $\mu(i) \in \mathcal{A}(i)$,

$x(j) > 0$ for $j = 1, 2, \dots, |S|$,

$v(j)$ is URS for $j = 1, 2, \dots, |S|$.

The decision variables are the x and the v terms. Again for an elaborate discussion on this topic, the reader is referred to [140].

11.2. The MDP as a special case of a stochastic game

The theory of games has developed from the seminal works of many mathematical geniuses such as von Neumann and Morgenstern [181], Nash [122], and Shapley [159]. Historically, work in game theory can be seen to have run parallel to that of DP.

A stochastic game is in fact a generalization of an MDP. An MDP has only one decision maker. A stochastic game has multiple decision makers who have independent goals. Each decision maker in a stochastic game is characterized by its own transition probability and reward matrices. It is the case thus that the Markov decision problem that we have studied in this chapter so far is only a special case of a game theory problem. The theorem of Shapley (1953) [159], which appeared in his seminal paper, can be used to derive results in the MDP case. Filar and Vrieze [44] observe in their book on page 85:

“Surprisingly, perhaps, stochastic games preceded Markov decision processes, and, in the early years, the two theories evolved independently and yet in a somewhat parallel fashion.”

Clearly, a stochastic game with its multiple decision-makers, where each decision-maker is armed with its own transition probability and transition reward structures, creates a more complex scenario. For solving a multi-decision-maker MDP (that is a stochastic game), one may have to use non-linear programming techniques such as the Newton’s method. One reason for the popularity of MDPs is that value iteration and policy iteration have evolved as nice and “easy” algorithms for solution purposes. However, obviously, the importance of the stochastic game in the context of control optimization, especially as it is a generalization of the MDP, cannot be ignored.

The discerning reader may now have some grievances with this author.

- Why the fact that the MDP is only a *special case* of the stochastic game was brought to light at the end of this chapter and
- why the more general game-theoretic framework was ignored in totality; it should have been the starting point.

One explanation that we can provide is that the theory of games has not yet evolved to the level of MDPs, and as such in some ways the theory of MDPs has its own highly developed features such as existence of optimal policies and numerical algorithms for obtaining them.

The book written by Filar and Vrieze [44] presents a delightful account of existence results and solution methods for stochastic games — in both average reward and discounted reward cases. Moreover, even in the context of MDPs, their book presents some interesting results. They adopt a mathematical programming (linear and non-linear programming) perspective and force one to take a fresh look at MDPs in the broader framework of stochastic games. In the MDP case, they have used the linear programming approach to show the existence of solutions, and in the general stochastic game, they have used non-linear programming approaches for the same purpose. It is very likely that future research in stochastic game theory will have an impact on the way we

treat and solve MDPs. The reader interested in pursuing this line of research should read this important book [44]. A great many unsolved problem areas have been identified and elaborated upon there; those who enjoy mathematical challenges will find it a heavenly place!

Thus far, we have considered MDPs with a finite state space and finite action space case for the *infinite* (time) horizon. We will next study the *finite* horizon problem for a finite state-and-action space.

11.3. Finite Horizon MDPs

The finite horizon MDP is also called the stochastic shortest path problem. In the finite horizon problem, the objective function is calculated over a finite time (or stage) horizon. As such we can think of two objective functions:

1. Total Expected Reward and
2. Total Expected Discounted Reward.

Every time the Markov chain jumps, we will assume that the time or the number of *stages* elapsed since the start increases by 1. In each stage, the system can be in any one of the states in the state space. As a result, the value function depends not only on the state but also on the stage. The number of stages in a finite horizon problem.

In the infinite horizon problem, when we visit a state, we are really not concerned about whether it is the first jump of the Markov chain or the 119th (this is any arbitrary number) jump. For instance in the infinite horizon problem, state 15 visited in stage 1 is the same thing as state 15 visited in stage 119. This is because the value function is associated with a state. In other words, in the infinite horizon problem, for every state, we associate a unique element of the value function.

In the finite horizon problem, we associate a unique element of the value function to a given state in a given stage. Thus state 15 visited in stage 1 and the same state 15 visited in stage 119 are in fact different entities with value functions that may have different values.

In the infinite horizon problem, we have a unique transition probability matrix and a unique transition reward matrix *for a given action*. In the finite horizon case, we have a unique transition probability matrix and a unique transition reward matrix *for a given stage-action pair*.

It should be clear then that the finite horizon problem can become more difficult than the infinite horizon problem with the same number of states, if there is a large number of stages. For a small number of stages, however, the finite horizon problem can be solved easily. The methods used for solving a finite horizon problem are called backward and forward recursion dynamic programming. As we will see below, they have the flavor of value iteration.

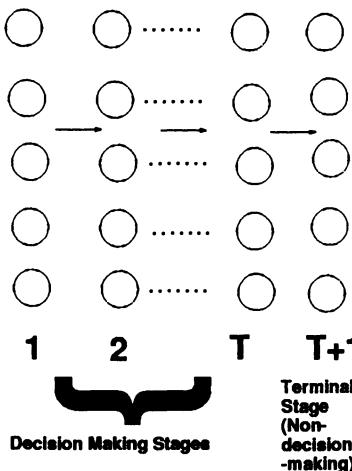


Figure 8.15. A Schematic for a Finite Horizon MDP

See Figure 8.15 to obtain a pictorial idea of a finite horizon MDP. In each of the stages, numbered 1 through T , the system can visit a subset of the states in the system. It is entirely possible that in each stage the set of states visited is a proper subset of the total state space; that is in each stage, *any* state in the system may be visited. Thus in general, we can construct a transition probability matrix and a transition reward matrix for each stage-action pair. In each stage, the system has a finite number of actions to choose from, and the system proceeds from one stage to the next till the horizon T is met. The goal is to maximize the total expected reward (or the total expected discounted reward) in the T jumps of the Markov chain.

Our approach is based on the value iteration idea. We will endeavor to compute the optimal value function for each state-stage pair. We will need to make the following assumption:

- Since there is no decision making involved in the states visited in the $(T + 1)$ th stage, (where the trajectory ends), the value function in every state in the $(T + 1)$ th stage will have the same value. For numerical convenience, the value will be 0.

The Bellman optimality equation for the finite horizon problem for expected total discounted reward is given by:

$$J^*(i, s) \leftarrow \max_{a \in \mathcal{A}(i, s)} [\bar{r}(i, s, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, s, a, j) J^*(j, s+1)] \quad (8.30)$$

for $i = 1, 2, \dots, |\mathcal{S}|$, and $s = 1, 2, \dots, T$ where

- $J^*(i, s)$ denotes the value function for the i th state when visited in the s th stage of the trajectory,
- a denotes an action,
- $\mathcal{A}(i, s)$ denotes the set of actions allowed in state i when visited in the s th stage of the trajectory,
- $\bar{r}(i, s, a)$ stands for the expected immediate reward earned when in state i , in the s th stage, action a is selected,
- λ denotes the discounting factor,
- \mathcal{S} denotes the set of states and $|\mathcal{S}|$ denotes the number of elements in \mathcal{S} , and
- $p(i, s, a, j)$ denotes the transition probability of going from state i in the s th stage to state j in the $(s+1)$ th stage when action a is selected in state i in the s th stage.

The optimality equation for *expected total reward* can be obtained by setting λ in Equation (8.30) to 1.

The algorithm that will be used to solve the finite horizon problem is not an iterative algorithm. It just makes one sweep through the stages and produces the optimal solution. We will next discuss the backward recursion technique for solving the problem.

The backward recursion technique starts with finding the values of the states in the T th stage (the final decision-making stage). For this, it uses (8.30). In the latter, one needs the values of the states in the next stage. We assume the values in the $(T+1)$ th stage to be known (they will all be 0 by our convention). Having determined the values in the T th stage, we will move one stage backwards, and then determine the values in the $(T-1)$ th stage.

The values in the $(T-1)$ th stage will be determined by using the values in the T th stage. In this way, we will proceed backward one stage at a time and find the values of all the stages. During the evaluation of the values, the optimal actions in each of the states will also be identified using the Bellman equation. We next present a step-by-step description of the backward recursion algorithm in the context of discounted reward. The expected *total reward* algorithm will use $\lambda = 1$ in the discounted reward algorithm.

A Backward Recursion Algorithm for Finite Horizon MDPs . Review notation definitions given for Equation (8.30).

Step 1. Let T (a positive integer) be the number of stages in the finite horizon problem. Decision-making will not be made in the $(T + 1)$ th stage of the finite horizon problem. The notation $u^*(i, s)$ will denote the optimal action in state i when the state is visited in the s th stage. The stages are numbered as $1, 2, \dots, T + 1$. For $i = 1, 2, \dots, |\mathcal{S}|$, set

$$J^*(i, T + 1) \leftarrow 0.$$

Step 2a. Set $s \leftarrow T$.

Step 2b. For $i = 1, 2, \dots, |\mathcal{S}|$,

$$u^*(i, s) \in \arg \max_{a \in \mathcal{A}(i, s)} [\bar{r}(i, s, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, s, a, j) J^*(j, s + 1)].$$

Step 2c. For $i = 1, 2, \dots, |\mathcal{S}|$, set

$$J^*(i, s) \leftarrow [\bar{r}(i, s, u^*(i, s)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, s, u^*(i, s), j) J^*(j, s + 1)].$$

Step 2d. If $s > 1$, decrement s by 1 and return to Step 2b; otherwise STOP.

For real-life examples of finite horizon MDPs in queuing, see Section 3.4 (page 51) of Sennott's book [157].

Our aim in the next subsection is to give some pointers to some recent path-breaking work in the case of the MDP with a **denumerably infinite** state space

11.4. The approximating sequence method

Sennott [157] discusses an approach to solving denumerably infinite MDPs (both average reward and discounted cases) via a method called the **approximating sequence method**.

An example of an MDP with a denumerably infinite state space is given by one in which the state is defined by the number of people in a queue with *infinite* capacity. Clearly, in such a situation, the number of states in the Markov chain is countably (denumerably) infinite.

The basic idea underlying this approach is to construct a sequence of MDPs with finite state spaces; the limit of this sequence should be the MDP to be solved. The state space of each member in the sequence would exceed that of the preceding member by 1. The idea is to come up with finite state-space approximation for the infinite state-space problem. In other words, the goal is to solve a finite state-space MDP that would closely approximate the actual MDP. Below, we provide a mathematical insight into the problem.

Let $\{b(n)\}_{n=1}^{\infty}$ be an increasing sequence of finite subsets of the state-space S such that $\cup_{n=1}^{\infty} b(n) = S$. Suppose the system is in state $i \in b(n)$ and action a is selected. For $j \notin b(n)$, the probability $p(i, a, j)$ is not 0.

Since each member of this sequence is a finite set Markov chain, we are leaving out an infinite number of states from the original Markov chain. Consequently, in the TPM of each member, the sum of the elements in each row will not be 1. In fact it will be less than 1. This problem is addressed by *increasing* the transition probabilities of the states that have been included. Thus for each $i, j \in b(n)$, the value of $p(i, a, j)$ is adjusted by a suitable amount so that the sums of the rows in the TPM equal 1. With a sufficiently large value for n , it can be shown that one can obtain a policy that is arbitrarily close to the optimal policy.

For an in-depth discussion, you are referred to Sennott [157]. According to her, it is possible that in practice it may be sufficient to compute the optimal policies of a few members of the sequence to generate a good solution.

12. Conclusions

This chapter discussed the fundamental ideas underlying MDPs and SMDPs. The focus was on problems with a finite state space and an infinite time horizon. The important methods of value and policy iteration were discussed and the two forms of the Bellman equation — the optimality equation and the policy equation, both for the average reward and the discounted reward problem, — were presented. The modified policy iteration algorithm was also discussed. The connection of MDPs with game theory was briefly discussed towards the end. The finite horizon problem was also discussed, briefly, towards the end. The chapter ended with pointers to some recent research in the denumerably infinite state-space problems and the approximating sequence method.

13. Bibliographic Remarks

Much of the material presented in this chapter is classical. It can be found in textbooks such as Puterman [140], Bertsekas [20], and Sennott [157], among other sources. For an in-depth discussion on stochastic processes, the reader is referred to Taylor and Karlin [169] and Ross [144]. For a highly intuitive and accessible account on MDPs and Markov chains, read Hillier and Lieberman [73] or Taha [168]. Filar and Vrieze [44] present the MDP as a special case of a game-theoretic problem thereby tying MDP theory to the much broader framework of stochastic games. They discuss several open research issues of related interest. Finally, for an in-depth discussion on the approximating sequence method for denumerably infinite state-space MDPs, see Sennott [157].

Much of the pioneering work in algorithms and DP theory was done by Bellman [14] (1954), Shapley [159] (value iteration, 1953), Howard [81] (policy

iteration, 1960), White [186] (relative value iteration, 1963), van Nunen [180] (modified policy iteration, 1976), and Sennott [156] (approximating sequence method, 1997), among others. Shapley [159] made important parallel contributions in game theory (which are applicable to MDPs) at a very early date (1953).

Further Reading. For a comprehensive account on the theory of MDPs, the structure of optimal policies, and performance measures outside of those discussed above, the reader is referred to our favorite text — Puterman [140]. Reading in control optimization cannot be complete without learning about stochastic optimal control theory in the continuous case (see the final chapter of Sethi and Thompson [158]).

14. Review Questions

1. A Markov chain has 2 states — numbered 1 and 2. In each state, one action can be selected from the set $\{1, 2\}$. Consider the following matrices.

$$\mathbf{A} = \begin{bmatrix} 0.2 & 0.8 \\ 0.7 & 0.3 \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} 0.4 & 0.6 \\ 0.1 & 0.9 \end{bmatrix},$$

$$\mathbf{A}' = \begin{bmatrix} 12 & 9 \\ -7 & -13 \end{bmatrix},$$

and

$$\mathbf{B}' = \begin{bmatrix} 12 & 4 \\ -6 & 9 \end{bmatrix}.$$

The matrices \mathbf{A} and \mathbf{A}' are associated with action 1 while \mathbf{B} and \mathbf{B}' are associated with action 2. \mathbf{A} and \mathbf{B} are the TPMs and the other matrices are the TRMs.

- a. Assuming a discounting factor of 0.4, find the value function associated with the policy $(2, 1)$.
- b. Determine if there is a value for the discounting factor which will make the policies $(2, 1)$ and $(1, 2)$ equally good.
- c. Perform two steps of regular value iteration with $\epsilon = 0.001$.
- d. Perform two steps of Gauss-Siedel value iteration with the same value for ϵ .
- e. Assuming that the problem information is given for an average reward MDP, do three steps of relative value iteration.
- f. Assuming that the problem information is given for an average reward MDP use policy iteration to find the optimal policy.

2. Why is DP preferred over exhaustive enumeration in solving MDPs? Is linear programming preferable to dynamic programming in solving MDPs?
3. Why is *relative* value iteration necessary for solving average reward MDPs—what are the drawbacks of regular value iteration? Does relative value iteration have any advantage over regular value iteration in solving discounted MDPs? Does Gauss-Siedel value iteration possess any advantage over regular value iteration in solving discounted reward MDPs?
4. Define the average reward of a given policy of an SMDP. Consider a policy, whose TPM, TTM, and TRM are defined below.

$$\mathbf{TPM} = \begin{bmatrix} 0.2 & 0.8 \\ 0.7 & 0.3 \end{bmatrix},$$

$$\mathbf{TRM} = \begin{bmatrix} 2 & 9 \\ 8 & -6 \end{bmatrix},$$

and

$$\mathbf{TTM} = \begin{bmatrix} 5 & 1 \\ 7 & 2 \end{bmatrix}.$$

Find the average reward of this policy.

5. Consider the TPM and the TRM given in the previous problem. Assume that these are the TPM and the TRM of policy $\hat{\mu}_1$ for a discounted reward MDP, whose discounting factor is 0.9. Now assume that there is another policy, $\hat{\mu}_2$, whose TPM and TRM are defined below.

$$\mathbf{TPM}' = \begin{bmatrix} 0.1 & 0.9 \\ 0.4 & 0.6 \end{bmatrix},$$

and

$$\mathbf{TRM}' = \begin{bmatrix} -3 & 6 \\ 25 & -6 \end{bmatrix}.$$

Which policy is better?

6. What is the difference between the Bellman optimality equation and the Bellman policy equation? Fill in the blanks using the following words: “policy” and “value.”

The Bellman optimality equation is used in — iteration and the Bellman policy equation is used in — iteration.

7. Discuss the difficulties associated with using value iteration for solving average reward SMDPs.
8. What is a DTMDP? How does it differ from a random time SMDP?
9. Consider a machine that is prone to failures. The time between failures is given by a random variable that is gamma distributed with a mean of 10 and a variance of 10. The time taken to produce a part is a constant equal to 1. Assume that time taken to repair or maintain the machine is negligible. It

costs \$5 to repair the machine and \$1 to maintain the machine. After repair or maintenance, the machine is assumed to be as good as new. Set up the problem of finding the optimal policy of maintenance as an MDP. Find the transition probabilities and the transition rewards associated with each action. Solve the problem using dynamic programming.

10. In a communication network, messages (packets) are generated for a server with an exponentially distributed inter-arrival time having a mean of 10. The time for processing is approximately 9 for any message. Since the buffer space (queue length) is usually limited, we assume that it costs $\$10k$ for a queue whose maximum permitted length is k . Any message that is rejected because the queue has already reached its capacity costs \$3. The problem is to find the maximum permissible capacity of the queue. Set this problem up as an MDP. Find the transition probabilities and the transition rewards associated with each action. Solve the problem using dynamic programming.

Chapter 9

CONTROL OPTIMIZATION WITH REINFORCEMENT LEARNING

*'Tis better to have loved and lost
Than never to have loved at all.*

— Alfred, Lord Tennyson (1809-1892)

1. Chapter Overview

This chapter focuses on a relatively new methodology called reinforcement learning. A prerequisite for this chapter is the previous chapter. Reinforcement learning (**RL**) is essentially a form of simulation-based dynamic programming and is primarily used to solve Markov and semi-Markov decision problems. It is natural to wonder why the word “learning” is a part of the name then. The answer is: pioneering work in this area was done by the artificial intelligence community, which views it as a machine “learning” method.

Other names have been suggested. Some examples are *neuro-dynamic programming* (NDP) and *dynamic programming stochastic approximation* (DPSA). The name neuro-dynamic programming was used in Bertsekas [20]. This is a suitable name, since it relates the topic to dynamic programming.

To be consistent with the operations research setting in this book, we will discuss RL through the viewpoint of dynamic programming. In our opinion, a suitable name for this science is **simulation-based dynamic programming**. In this book, however, we will stick to the name “reinforcement learning” in this book in honor of its pioneering contributors.

To follow this chapter, the reader should review the material presented in the previous chapter. In writing this chapter, we have followed an order that *differs somewhat* from the one followed in the previous chapter. We *begin* with discounted reward, and then discuss average reward.

2. The Need for Reinforcement Learning

It is good to remind the reader why we need RL in the first place. After all, **dynamic programming (DP)** is guaranteed to give optimal solutions to MDPs (Markov Decision Problems) and SMDPs (semi-Markov Decision Problems).

It is the case that obtaining the transition probabilities, the transition rewards, and the transition times (the transition times are needed for semi-Markov decision problems) is often a difficult and tedious process that involves complex mathematics. DP needs the values of all these quantities. RL does not, and despite that RL can generate near-optimal solutions.

The transition probabilities, the transition rewards, and the transition times together constitute what is known as the **theoretical model** of a system. Evaluating the transition probabilities often involves evaluating multiple integrals that contain the pdfs of random variables. A complex stochastic system with many random variables can make this a very challenging task. It is for this reason that DP is said to be plagued by the **curse of modeling**. In Chapter 14, we will discuss examples of such systems.

Solving a control-optimization problem, using an MDP model, yields optimal solutions. (Of course, every control-optimization problem cannot be set up as an MDP.) However, the MDP model is often avoided, and heuristic approaches are preferred in the real world. This is primarily because, as stated above, for complex systems, it is usually hard to construct the theoretical model required in an MDP formulation. Hence, if a method can solve an MDP, without having to construct the theoretical model, it surely is a very attractive proposition. *RL has the potential to do exactly this.*

Even for many *small* real-world systems with complex stochastic structures, computing the transition probabilities is a cumbersome process. RL is said to avoid the curse of modeling because it does not need transition probabilities.

For many large systems, classical DP, which we have discussed in the previous chapter, is simply ruled out. Why? Consider the following example: an MDP with a thousand states with two actions in each state. The TPM (transition probability matrix) of each action would contain $1000^2 = 10^6$ — that is, a million elements. It is thus clear that in the face of large-scale problems with huge solution spaces, DP may break down. It will be difficult to store such large matrices in our computers. This is called the **curse of dimensionality**. DP is thus doubly cursed — by the curse of dimensionality and by the curse of modeling.

RL can provide a way out of these difficulties in two ways.

1. The so-called model-free methods of RL do not need the transition probability matrices. *RL can thereby avoid the curse of modeling.*
2. RL *does* need the elements of the value function of DP. RL stores the value function in the form of the so-called *Q*-factors. When the MDP has millions

of states, RL does not store the Q -factors explicitly. RL then uses the so-called *function approximation* methods, such as neural networks, regression, and interpolation, which need only a small number of scalars to approximate Q -factors of millions of states. *Function approximation helps RL avoid the curse of dimensionality.*

Although the RL approach is approximate, it uses the Markov decision model which is so powerful that even near-optimal solutions to it — which RL is capable of generating — can often outperform “heuristic” approaches. Heuristic approaches are *inexact* methods that produce solutions in a reasonable amount of computer time — usually without extensive computation.

RL is rooted in the DP framework and inherits several of the latter’s attractive features — such as generating high-quality solutions. We will next discuss the tradeoffs in using heuristic methods and DP methods.

Consider a stochastic decision-making problem that can theoretically be cast as an MDP (or an SMDP) and can also be solved with heuristic methods. It may be hard to solve the problem using the MDP model, since the latter needs the setting up of the theoretical model. On the other hand, the heuristic approach to solve the same problem may require less effort technically. Let us compare the two approaches in this backdrop.

- A DP algorithm is iterative and requires several computations thereby making it computationally intensive. In addition, it needs the transition probabilities — computing which can get very technical and difficult. But when it produces a solution, the solution is of a high quality.
- A heuristic algorithm usually makes several modeling assumptions about the system. An example of a simplifying modeling assumption is replacing a random variable in a system by a deterministic quantity — usually the mean of the random variable. Usually, heuristics do not compute any transition probabilities, because they don’t need to, and thus come up with solutions relatively easily. It is another matter that the solutions may be far from optimal!

See Table 9.1 for a tabular comparison of RL, DP, and heuristics in terms of the level of modeling effort and the quality of solutions generated.

To give an example, the EMSR (expected marginal seat revenue) method [113] is a heuristic approach used to solve the seat-allocation problem in the airline industry. It is a simple model that quickly gives a good solution. However, it does not take into account several characteristics of the real-life airline reservation system, and as such its solution may be far from optimal. On the other hand, the MDP approach, which can be used to solve the same problem, yields superior solutions but forms a considerably harder route because one has to first estimate the transition probabilities and then use DP. In such cases, in

Table 9.1. A comparison of RL, DP, and heuristics. Note that both DP and RL use the MDP model.

Method	Level of modeling effort	Solution quality
DP	High	High
RL	Medium	High
Heuristics	Low	Low

practice, not surprisingly, the heuristic is often preferred over the exact MDP approach. So where is RL in all of this?

RL can generally outperform heuristics, and at the same time, the modeling effort in RL is less in comparison to that of DP. Furthermore, on large-scale MDPs, DP breaks down, but RL may not. In summary, in solving problems whose transition probabilities are hard to find, RL is an attractive approach that may outperform heuristics.

The main “tool” employed by RL is simulation. It uses simulation to avoid the computation of the transition probabilities. A simulator needs the distributions of the random variables that govern the system’s behavior. The transition rewards and the transition times are automatically calculated within a simulator. The avoidance of transition probabilities is not a miracle, but a fact backed by mathematics. We will discuss this issue in great detail, shortly.

In general, however, if one has access to the transition probabilities, the transition rewards, and the transition times, use of DP is guaranteed to generate the optimal solution, and then RL is not necessary.

3. Generating the TPM through straightforward counting

Since obtaining the TPM (transition probability matrix) may be a major obstacle in solving a problem, one way around it is to simulate the system and generate the transition probabilities from the simulator. After the transition probabilities and the transition rewards are obtained, we can then use the classical DP algorithms discussed in the previous chapter. We will next discuss how exactly this task may be performed. Let us state an important, although obvious, fact first.

To simulate a system that has a Markov chain underlying it, one does not need the transition probabilities of the system. To simulate a stochastic system, one needs the distributions of the random variables that govern the behavior of the system.

For example, to simulate a queuing system, which can be modeled as a Markov chain, one does not need the transition probabilities of the Markov

chain underlying it. One needs the distributions of the inter-arrival times and the service times.

To estimate transition probabilities via simulators, one can employ a straightforward counting procedure. Let us assume that we are interested in calculating the transition probability $p(i, a, j)$. Recall that this is the one-step transition probability of going from state i to state j when action a is selected in state i .

In the simulation program, we will need to keep two counters $V(i, a)$ and $W(i, a, j)$. Both will be initialized to 0. Whenever action a is selected in state i , the counter $V(i, a)$ will be incremented by 1 in the simulator. When, as a result of this action, the system goes to j from i in one step, $W(i, a, j)$ will be incremented by 1. From the definition of probabilities, (see Equation (3.1) on page 16),

$$p(i, a, j) \simeq \frac{W(i, a, j)}{V(i, a)}.$$

Of course, what we have above is an estimate that will *tend* to the correct value, when $V(i, a)$ tends to infinity. This means that each state-action pair must be tried infinitely often in the simulator — so that $V(i, a)$ tends to ∞ .

Using the mechanism shown above, one can estimate the transition probability matrices in the MDP. Similarly, transition reward matrices can also be obtained.

After generating estimates of the TPMs and the TRMs, one can solve the MDP using DP algorithms. A schematic showing the main ideas — underlying the approach suggested in this section — is Figure 9.1. This may not prove to be an efficient approach since the TPMs will not be perfect, and it may take a very long time to generate good estimates of the TPMs in the simulator. However, there is nothing inherently wrong in this approach, and the analyst should certainly try this for small systems (whose transition probabilities are difficult to find theoretically) instead of pursuing an RL approach.

A strong feature of this approach is that once the TPMs and the TRMs are generated, DP methods, which we know are guaranteed to generate good solutions, can be used.

For large systems with thousands of states, a naive application of this method runs into trouble because it becomes difficult to store each element of the TPMs and the TRMs. This method can then be coupled with a “state-aggregation” approach. In a state-aggregation approach, several states are combined to form a *smaller* number of “aggregate” states. This leads to an approximate model with a *manageable* number of states. This approach is a robust alternative to the RL methods that we will discuss in this chapter.

4. Reinforcement Learning: Fundamentals

At the outset, we would like to be clear about the following.

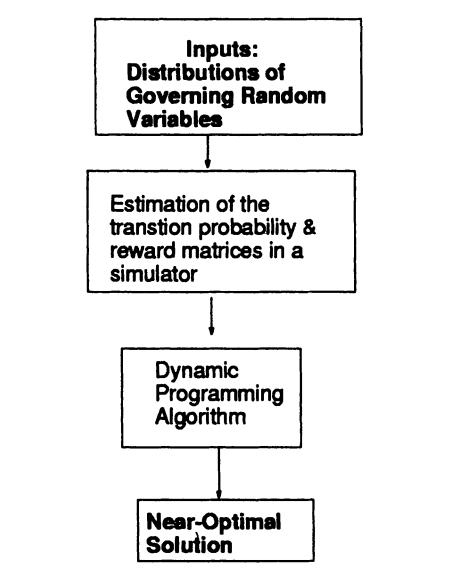


Figure 9.1. Solving an MDP using DP *after* estimating the TPMs and TRMs in a simulator

RL is an offshoot of DP; it is a way of doing dynamic programming (DP) within a simulator.

We have used this perspective throughout the book so that RL does not come across as some magical tool that can do wonders. Furthermore, we will strive to present every RL algorithm in a manner such that it comes across as being equivalent to some DP algorithm; the comforting thing about this is: we know that DP algorithms are guaranteed to generate optimal solutions. Therefore, in this chapter, we will attempt to show how an RL algorithm can be derived from a DP algorithm — although detailed “convergence” analysis of RL algorithms has been relegated to Chapter 13.

The main difference between the RL philosophy and the DP philosophy has been depicted in Figure 9.2. Like DP, RL needs the distributions of the random variables that govern the system’s behavior. However, after that, things work differently. In DP, the first step is to generate the transition probability and the transition reward matrices. The next step in DP is to use these matrices in a suitable DP algorithm to generate a solution. In RL, we do not estimate the transition probability or the reward matrices but instead simulate the system using the distributions of the governing random variables. Then within the simulator, a suitable RL algorithm is used to obtain a near-optimal solution.

We next discuss some fundamental concepts related to RL. A great deal of RL theory is based on:

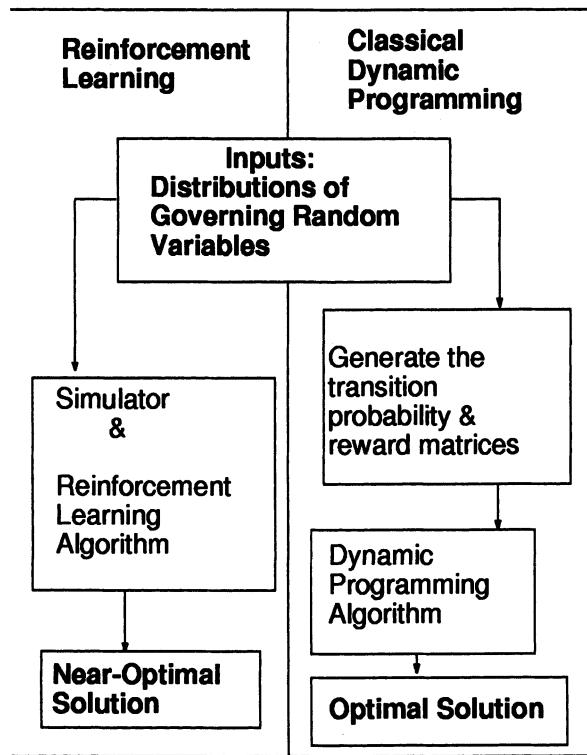


Figure 9.2. A schematic highlighting the differences in the methodologies of RL and DP

1. The Q -factor and
2. the Robbins-Monro algorithm [142].

Both topics will be discussed below.

We also need to define the following sets. We have defined them in the previous chapter, but let us redefine them here.

1. \mathcal{S} will denote the set of states in the system. These states are those in which decisions are made. In other words, \mathcal{S} will denote the set of decision-making states. Unless otherwise specified, a state will mean the same thing as a decision-making state.
2. $\mathcal{A}(i)$ will denote the set of actions allowed in state i .

The sets, \mathcal{S} and $\mathcal{A}(i)$ (for all i), will be assumed to be finite.

4.1. *Q*-factors

Most RL algorithms calculate the value function of DP — and this is the reason why RL has its roots in DP. The value function is stored in the form of the so-called *Q*-factors.

Let us recall the definition of the value function associated with the optimal policy for discounted reward MDPs. It should also be recalled that this value function is defined by the Bellman optimality equation, which we restate here.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J^*(j)] \quad (9.1)$$

for all $i \in \mathcal{S}$, where

- $J^*(i)$ denotes the i th element of the value function vector associated with the optimal policy,
- $\mathcal{A}(i)$ is the set of actions allowed in state i ,
- $p(i, a, j)$ denotes the one-step transition probability of going from state i to state j under the influence of action a ,
- $\bar{r}(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)r(i, a, j)$ denotes the expected immediate reward earned in state i when action a is selected in it; $r(i, a, j)$ here denotes the immediate reward earned when action a is selected in state i and the system goes to state j as a result,
- \mathcal{S} denotes the set of states in the Markov chain, and
- λ stands for the discounting factor.

In DP, we associate one element of the value function vector with a given state. In RL, we associate one element of the so-called *Q*-factor vector with a given *state-action* pair. To understand the idea, consider an MDP with three states and two actions allowed in each state. In DP, the value function vector \vec{J}^* would have three elements as shown below.

$$\vec{J}^* = \{J^*(1), J^*(2), J^*(3)\}.$$

In RL, we would have six *Q*-factors, since there are six state-action pairs. Thus if $Q(i, a)$ denotes the *Q*-factor associated with state i and action a , then

$$\vec{Q} = \{Q(1, 1), Q(1, 2), Q(2, 1), Q(2, 2), Q(3, 1), Q(3, 2)\}.$$

Now, we define a *Q*-factor . For a state-action pair (i, a) ,

$$Q(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)[r(i, a, j) + \lambda J^*(j)]. \quad (9.2)$$

Now, combining Equations (9.1) and (9.2), one has that

$$J^*(i) = \max_{a \in \mathcal{A}(i)} Q(i, a). \quad (9.3)$$

The above establishes the relationship between the value function of a state and the Q -factors associated with a state. Then it should be clear that, if the Q -factors are known, one can obtain the value function of a given state from (9.3). For instance for a state i with two actions if the two factors are $Q(i, 1) = 95$ and $Q(i, 2) = 100$, then

$$J^*(i) = \max\{95, 100\} = 100.$$

Using Equation (9.3), Equation (9.2) can be written as:

$$Q(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j, b)]. \quad (9.4)$$

for all (i, a) .

Equation (9.4) is a very important equation. It can be viewed as the Q -factor version of the Bellman optimality equation for discounted reward MDPs. In fact, it helps us develop a Q -factor version of value iteration. We present the step-by-step details of this algorithm next.

4.1.1 A Q -factor version of value iteration

Mind you, we are still very much in the DP arena; we have not made the transition to RL yet. In fact, the algorithm we are about to present is completely equivalent to the regular value iteration algorithm of the previous chapter.

Step 1: Set $k = 1$ and select an arbitrary vector \vec{Q}^0 . For example, set for all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$,

$$Q^0(i, a) = 0.$$

Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$Q^{k+1}(i) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j, b)].$$

Step 3: Calculate for each $i \in \mathcal{S}$

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} Q^{k+1}(i, a) \text{ and } J^k(i) = \max_{a \in \mathcal{A}(i)} Q^k(i, a).$$

Then if

$$||(J^{k+1} - J^k)|| < \epsilon(1 - \lambda)/2\lambda,$$

go to Step 4. Otherwise increase k by 1, and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg \max_{b \in \mathcal{A}(i)} Q(i, b),$$

where \hat{d} denotes the ϵ -optimal policy, and stop.

The equation in Step 2, it may be noted, is derived from the Equation (9.4). The equivalence of this algorithm to regular value iteration, which estimates the value function, is not difficult to see. Instead of estimating the value function, this algorithm estimates the Q -factors.

The conceptual significance of this algorithm needs to be highlighted because we will derive RL algorithms from it. In RL also, we will estimate Q -factors, but the updating equation will be slightly different than the one shown in Step 2.

Now, to make the transition to RL, we next need to understand the Robbins-Monro algorithm.

4.2. The Robbins-Monro algorithm

The Robbins-Monro algorithm is an old algorithm from the fifties [142] that helps us estimate the mean of a random variable from its samples. The idea is very simple. We know that the mean of a random variable can be estimated from the samples of the random variable by using a straightforward averaging process. Let us denote the i th independent sample of a random variable X by s^i and the expected value (mean) by $E(X)$. Then with probability 1, the estimate produced by

$$\frac{\sum_{i=1}^n s^i}{n}$$

tends to the real value of the mean as $n \rightarrow \infty$. (This follows from the strong law of large numbers (see Chapter 3).) In other words, with probability 1,

$$E[X] = \lim_{n \rightarrow \infty} \frac{\sum_{i=1}^n s^i}{n}$$

The samples, it should be understood, can be generated in a simulator.

Now, let us derive the Robbins-Monro algorithm from this straightforward averaging process. Let us denote the estimate of X in the n th iteration — that is, after n samples have been obtained — by X^n . Thus:

$$X^n = \frac{\sum_{i=1}^n s^i}{n}. \quad (9.5)$$

Now,

$$\begin{aligned}
 X^{n+1} &= \frac{\sum_{i=1}^{n+1} s^i}{n+1} \\
 &= \frac{\sum_{i=1}^n s^i + s^{n+1}}{n+1} \\
 &= \frac{X^n n + s^{n+1}}{n+1} \quad (\text{using Equation (9.5)}) \\
 &= \frac{X^n n + X^n - X^n + s^{n+1}}{n+1} \\
 &= \frac{X^n(n+1) - X^n + s^{n+1}}{n+1} \\
 &= \frac{X^n(n+1)}{n+1} - \frac{X^n}{n+1} + \frac{s^{n+1}}{n+1} \\
 &= X^n - \frac{X^n}{n+1} + \frac{s^{n+1}}{n+1} \\
 &= (1 - \alpha^{n+1})X^n + \alpha^{n+1}s^{n+1}
 \end{aligned} \tag{9.6}$$

if $\alpha^{n+1} = 1/(n+1)$.

Thus:

$$X^{n+1} = (1 - \alpha^{n+1})X^n + \alpha^{n+1}s^{n+1}. \tag{9.7}$$

This is called the Robbins-Monro algorithm. When $\alpha^{n+1} = 1/(n+1)$, the Robbins-Monro algorithm, it should be clear now, is equivalent to direct averaging. However, other definitions are also used for α — as long as they indirectly perform averaging.

We are going to use the Robbins-Monro algorithm (or scheme) heavily in RL. We may use different definitions for α — the step size or the learning rate — as long as convergence to the optimal solution is still guaranteed.

4.3. The Robbins-Monro algorithm and the estimation of Q -factors

The Robbins-Monro algorithm, it turns out, can be used for estimating Q -factors. Recall that value iteration is all about estimating the optimal value function. Similarly, from our discussion on the Q -factor version of value iteration, it should be obvious that the Q -factor version of value iteration is all about estimating the optimal Q -factors. So how does the Robbins-Monro scheme fit into the picture?

It can be shown that every Q -factor can be expressed as an average of a random variable. Let us consider the definition of the Q -factor in the Bellman

equation form (that is Equation (9.4)).

$$Q(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j, b)] \quad (9.8)$$

$$= E[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j, b)] \quad (9.9)$$

$$= E[\text{SAMPLE}], \quad (9.10)$$

where E is the expectation operator and the quantity in the square brackets of (9.9) is the random variable of which E is the expectation operator. Thus, if samples of the random variable can be generated within a simulator, we can then use the Robbins-Monro scheme to evaluate the Q -factor. Instead of using Equation (9.8) to estimate the Q -factor (as shown in the Q -factor version of value iteration), we could use the Robbins-Monro scheme in a simulator to estimate the same Q -factor. Using the Robbins-Monro algorithm (see Equation (9.7)), Equation (9.8) becomes:

$$Q^{n+1}(i, a) \leftarrow (1 - \alpha^{n+1})Q^n(i, a) + \alpha^{n+1}[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q^n(j, b)] \quad (9.11)$$

for each (i, a) pair.

What is the most exciting thing about this algorithm? Well, of course, we do not have the transition probabilities in it! *In other words, if this algorithm can generate the optimal Q -factors in a simulator, we do not need to know the transition probabilities of the underlying Markov chain.* All we need is a simulator of the system. The mechanism shown in (9.11) enables us to avoid transition probabilities in RL. An algorithm that does not use transition probabilities in its updating equations is called a **model-free** algorithm.

What we have derived above is the Q -learning algorithm for discounted MDPs. This was first discovered by Watkins [182]. The above discussion should make it clear that the algorithm can be derived from the Bellman optimality equation for discounted reward MDPs.

4.4. Simulators, asynchronous implementations, and step sizes

Using the Q -learning algorithm in a simulator creates a few difficulties. Since there is more than one state, it becomes necessary to simulate a trajectory of the Markov chain within a simulator. Now, in regular DP (or in the Q -factor version of value iteration), one proceeds in a synchronous manner in updating the value function (or the Q -factors). For instance, if there are three states, one would update the value function (or the Q -factors) associated with state 1, then that of state 2, then that of state 3, then come back to state 1 and so on. In a simulator, it is difficult to guarantee that the trajectory will be of a nice

$1, 2, 3, 1, 2, 3, \dots$ form. Instead the trajectory might be:

$$1, 3, 1, 3, 2, 3, 2, 2, 1, 3, 3, 1 \dots$$

This is just an example. Along this trajectory, updating would occur in the following way. After updating some Q -factor in state 1, we would go to state 3, update a Q -factor there and come back to state 1, then update a Q -factor in state 1, and so on. When a Q -factor of a state is updated, the updating equation usually needs a Q -factor from some other state, and the latest estimate of the Q -factor(s) from the other state will have to be used in this scenario. It is to be understood that in such a haphazard style of updating, at any given time, one Q -factor may be updated more frequently than another.

The point is that the updating will be heavily **asynchronous**. Now, asynchronism introduces an error. In RL, we have to live with this error and must find a way out of this difficulty. This means that a way must be found to ensure that the error is negligible.

To keep this error negligible, one must make sure that the step size is small. We will prove in Chapter 13 that by keeping the step size small enough, the error in asynchronism can be kept at a negligible level. Now, thus far, we have defined the step size as:

$$\alpha^n = 1/n$$

where n is the number of samples generated. Clearly, when n is small, α^n may not be small enough. For instance when $n = 1$, $1/n = 1$. As a result, a step size such as:

$$\alpha^n = A/n$$

where $A < 1$ is often suggested to circumvent this difficulty.

From our discussion on the derivation of the Robbins-Monro algorithm, it should be clear that the step size with $A \neq 1$ will not produce direct averaging.

A prime difficult with this rule is that A/n decays rapidly with the value of n , and in only a few iterations, the step size can become too small to do any updating. Not surprisingly, other step-size rules have been suggested in the literature. One such rule, given in Darken, Chang, and Moody [36], is:

$$\alpha^{n+1} = \frac{T_1}{1 + \frac{[n]^2}{1+T_2}}, \quad (9.12)$$

where T_1 is the starting value for α . Possible values for T_1 and T_2 are $.1$ and 10^6 respectively. Another simple rule, in which α can decay slowly, is:

$$\alpha^{n+1} = \alpha^n r,$$

where r is less than 1. The danger with this rule is that it violates a condition necessary for stochastic approximation.

5. Discounted reward Reinforcement Learning

In this section, we will discuss simulation-based RL algorithms for value and policy iteration in the discounted reward context. The first subsection will focus on value iteration and the second will focus on policy iteration.

5.1. Discounted reward RL based on value iteration

As discussed previously, value iteration based RL algorithms are centered on computation of the optimal Q -factors. The Q -factors in value iteration are based on the Bellman equation.

A quick overview of “how to incorporate an RL optimizer within a simulator” is in order at this point. We will do this with the help of an example.

Consider a system with two Markov states and two actions allowed in each state. This will mean that there are four Q -factors that need to be evaluated:

$$Q(1, 1), Q(1, 2), Q(2, 1), \text{ and } Q(2, 2),$$

where $Q(i, a)$ denotes the Q -factor associated with the state-action pair (i, a) . The bottom line here is that we want to estimate the values of these Q -factors — from running a simulation of the system. For the estimation to be perfect, we must obtain an infinite number of samples for each Q -factor. For this to happen, each state-action pair should be, theoretically, tried infinitely often. A safe strategy to attain this goal is to try each action in each state with equal probability and simulate the system in a fashion such that each state-action pair is tried a large number of times.

The job of the RL algorithm, which should be embedded within the simulator, is to update the values of the Q -factors using the equation given in (9.11). The simulator will take the system from one state to another selecting each action with equal probability in each state. *The simulator will not be concerned with what the RL algorithm is doing* — as long as each action is selected with equal probability. Let us discuss the scheme used in updating the Q -factors in a more technical manner.

Let us assume (consider Figure 9.3) that the simulator selects action a in state i and that the system goes to state j as a result of the action. During the time interval in which the simulator goes from state i to state j , the RL algorithm collects information from within the simulator; the information is $r(i, a, j)$, which is the immediate reward earned in going from state i to state j under the influence of action a .

When the simulator reaches state j , it uses $r(i, a, j)$ to generate a new sample of $Q(i, a)$. See Equation (9.9) to see what is meant by a sample of $Q(i, a)$ and why $r(i, a, j)$, among other quantities, is needed to find the value of the sample. Then $Q(i, a)$ is updated via Equation (9.11), with the help of the new sample. Thus, the updating for a state-action pair is done *after* the transition to the next

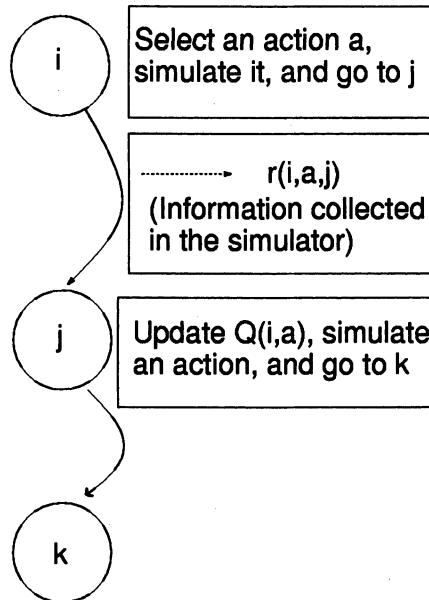


Figure 9.3. The updating of the Q -factors in a simulator. Each arrow denotes a state transition in the simulator. After going to state j , the Q -factor for the previous state i and the action a selected in i , that is, $Q(i, a)$ is updated.

state. This means that the simulator should be written in a style that permits updating of quantities after each state transition.

In what follows, we will present a step-by-step technical account of simulation-based value iteration, which is also called Q -learning. The number of times each state-action pair is tried has to be kept track of. This quantity is needed in the definition of the step size in the Robbins-Monro algorithm. Since the updating is asynchronous, this quantity can assume different values for each state-action pair. Hence, it is necessary to keep an individual counter for each state-action pair. We will call this quantity the visit factor. $V(i, a)$ will denote the visit factor for the state-action pair (i, a) .

5.1.1 Steps in Q -Learning

Step 1. Initialize the Q -factors to 0 and the visit factors also to 0. In other words: Set for all (l, u) where $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently

large number. Set A , the step size constant, to a positive number less than 1.

1. Start system simulation at any arbitrary state.

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried, by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j, b)].$$

Step 5. If $k < k_{max}$, set $i \leftarrow j$ and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

Several issues need to be discussed now.

What other step sizes can be used? The value of $\alpha = A/V(i, a)$ is motivated by the rule $\alpha^n = A/n$, where n denotes the number of generated samples of the variable, whose mean is to be estimated. Clearly, by its definition, $V(i, a)$ denotes the number of samples of the Q -factor in question. Please refer to other rules that were discussed earlier. In the rest of this book, we will use the rule: $\alpha = A/V(i, a)$, but in the light of its obvious drawbacks (discussed earlier), the reader is encouraged to try other rules as well.

Are visit factors needed? In a typical implementation of this algorithm, the visit factors are not measured. In other words, individual counters for $V(i, a)$ are not kept. Instead the step size, α , at any step, is made a function of k — the number of jumps of the Markov chain. This leads to a dramatic reduction in the storage space needed in the computer, since one does not need any of the visit factors. This kind of an implementation can only be justified, if all state-actions pairs are tried by the simulator, at least roughly, with the same frequency. Under this assumption, the value of the visit factor for each state-action pair would roughly be the same, which would justify using the same value for all state-action pairs. When the conditions in the simulator do not satisfy this assumption, using the same value for all $V(i, a)$ terms may not produce averaging, and the solution obtained from the algorithm may stray considerably from optimality.

What are look-up tables? The algorithm presented above is for the so-called **look-up table** implementation in which all Q -factors are stored explicitly. This implementation is possible only when we have a manageable number of state-action pairs — say up to a 10,000 state-action pairs. (10,000 state-action pairs will need 20,000 storage units; 10,000 Q -factors and 10,000 V -factors.) All the RL algorithms that appear in the following pages (unless specified otherwise) will be presented in a look-up table format. We will discuss function approximation, in which all the Q -factors are not stored explicitly, in Section 12.

Arg max notation: The term $\max_{b \in A(j)} Q(j, b)$ in the main updating equation for the Q -factors represents the maximum Q -factor in state j .

Termination Criteria: It must be understood that as k increases, so do all the $V(i, a)$ quantities. When these quantities start approaching very large numbers, the step size starts approaching 0. When the step size is very close to 0, we can stop the algorithm since the Q -factors will not change significantly any more. Thus instead of placing a limit on k (that is, assigning a finite value to k_{\max}), one may choose to terminate the algorithm by placing an upper limit on the value of the $V(i, a)$ term.

Performance evaluation: To determine the quality of the solution, one must run the simulation, again, this time using the policy learned in the final step of the algorithm. The idea — underlying such a re-run — is to estimate the value function vector associated with the policy learned in the simulator. To find the value function for state i , one must start the simulation in state i , and then sum the rewards earned over an infinitely long trajectory using a factor of λ . This must be averaged over several replications. The policy, to be followed in these trajectories, should be the policy learned in the final step of the algorithm. This has to be done for each state.

In *average reward* problems, in comparison to discounted reward problems, it turns out that testing the performance of a policy learned from running the RL algorithm is much easier since the performance metric (average reward) is a single scalar, which can be obtained from one simulation of the system (which may need multiple replications).

5.1.2 Reinforcement Learning: A “Learning” Perspective

RL was developed by researchers in the artificial intelligence (AI) community. In the AI community, RL was viewed as a machine “learning” technique. It is time to understand this viewpoint.

We can view the decision maker in an MDP as a *learning agent*. The “task” assigned to this agent is to obtain the optimal action in each state of the system. The operation of an RL algorithm can be described in the following manner. The algorithm starts with the same value for each Q -factor associated with a given state. All possible actions are simulated in every state. *The actions that*

produce good immediate rewards are rewarded and those that produce poor rewards are punished. The agent accomplishes this in the following manner. For any given state i , it *raises* the values of the Q -factors of the good actions and *diminishes* the values of the Q -factors of the bad actions.

We have viewed (earlier) the updating process in RL as the use of the Robbins-Monro algorithm for the Q -factor, and the definition of the Q -factor was obtained from the Bellman equation. Although the two viewpoints are essentially equivalent, it is considered instructive to understand this intuitive idea underlying RL; the idea is to reward the good actions, punish the bad actions, and in general **learn** from trial-and-error in the simulator. See Figure 9.4 for a schematic.

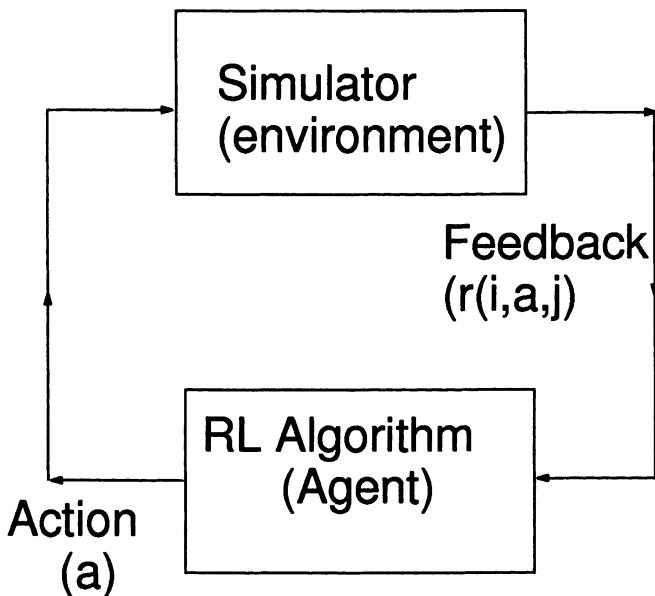


Figure 9.4. Trial and error mechanism of RL. The action selected by the RL agent (algorithm) is fed into the simulator. The simulator simulates the action, and the resultant feedback (immediate reward) obtained is fed back into the knowledge-base (Q -factors) of the agent. The agent uses the RL algorithm to update its knowledge-base, becomes smarter in the process, and then selects a better action.

The idea of trial-and-error is the fundamental notion underlying any *learning* algorithm. It is this notion that inspired us to quote the poet at the beginning of this chapter.

It would indeed be interesting to see whether the learning agent can learn (improve its behavior) within the simulator *during the run time of the simulation*. Since the discounted reward problem has a large number of performance

metrics, it is difficult to determine a trend when the simulation is running. In the average reward case, however, this can be done.

The performance of an algorithm is, however, best judged after the learning. To evaluate the performance, as mentioned above, we re-simulate the system using the policy learned. This, of course, is done after the learning phase is complete. This phase is referred to as the *frozen* phase. (The word “frozen” refers to the fact that the Q -factors do not change during this phase.)

5.1.3 On-line and Off-line

The word “learning” makes a great deal of sense in robotic problems. In these problems, the robot actually learns on a real-time basis. In the field of robotics, where RL is used, for various reasons, learning in simulators is not considered as sophisticated as learning in real time. In most of the problems that we will be considering in this book, it will be sufficient to learn in simulators. This is because we will assume that the distributions of the random variables that govern the system behavior are available, i.e., a good simulator of the system is available, and hence we do not really need to learn on a real-time basis.

In all the examples considered in this book, we will run the program on a simulator before implementing it in the real-life system. As a result, the word “learning” may not apply to problems for which good simulators are available. This means that the algorithms can be run off-line. In industrial problems (in the manufacturing and service industry) learning in real time can cause extraordinary losses, and in fact simulations are preferred because they do not disturb the actual system.

RL algorithms are often described as either on-line or off-line *algorithms* — causing a great deal of confusion to the beginner. Algorithms can be run off-line or on-line with respect to separate qualifiers, which are: 1) the implementation of the algorithm on an application and 2) the internal updating mechanism of the algorithm.

The implementation aspect: When we talk of an off-line *implementation*, we mean that the algorithm is run on the simulator before implementing it on the real system. On the other hand, in an on-line *implementation*, the algorithm does *not* use a simulator, but runs on a real-time basis in the real system. Now, all the model-free algorithms that we will discuss in this book can be implemented in either an off-line or an on-line sense.

The updating aspect: This is an internal issue of the algorithm. Most algorithms discussed in this book update the Q -factor of a state-action pair *immediately* after it is tried. Such an update is called an on-line *update*. Now, there are some algorithms that update Q -factors *after a finite number of state transitions*. (Some so-called “ $TD(\lambda)$ ” algorithms work in this fashion. An example is λ -SMART — see Gosavi, Bandla, and Das [64].) This is often referred to as an off-line *update* and has nothing to do with an off-line *implementation*.

An algorithm with an on-line updating mechanism may have an off-line implementation. All four combinations are possible.

5.1.4 Exploration

In general, any action selection strategy can be described as follows.
When in state i , select action u with a probability of $p^k(i)$, where k denotes the number of times the Markov chain has jumped so far.

Using $p^k(i) = 1/|\mathcal{A}(i)|$ reduces to the strategy that we have discussed all along.

We believe from our computational experience that this strategy of selecting each action with equal probability is a robust strategy. In the Robbins-Monro algorithm, for the averaging process in the Robbins-Monro algorithm to take place, each Q -factor must be tried infinitely often (theoretically), and one way to ensure this — in practice — is to select each action with equal probability. However, other successful strategies have been suggested in the literature. Let us discuss one such strategy.

A so-called **exploratory** strategy selects the action that has the highest Q -factor with a high probability and selects the other actions with a low non-zero probability. To describe an example, consider a scenario with two actions.

In the k th jump of the Markov chain, select action u in state i , where

$$u = \arg \max_{b \in \mathcal{A}(i)} Q(i, b),$$

with the probability

$$p^k = 1 - B/k,$$

and select the other action with probability B/k . An example value for B is 0.5.

It is clear that when such an exploratory strategy is pursued, in the beginning of the simulation, the learning agent will select the non-greedy (exploratory) action with some probability (B/k) . As k starts becoming large, the probability of selecting the non-greedy (exploratory) action diminishes. At the end of the simulation, when k is very large, the action selected will be the greedy action. As such, the action selected will also be the action prescribed by the policy learned by the algorithm. The value of p^k could be decreased with k via other schemes such as:

$$p^k = p^{k-1}r \text{ with } r < 1,$$

or using the scheme described in Equation (9.12) for decreasing the step size.

One benefit of using an action selection strategy of the exploratory nature is that the run time of the algorithm may be considerably reduced. The reason for this is: as k increases, the non-greedy Q -factors are evaluated with *decreasing* frequencies. Note, however, that this can lead to imperfect evaluation of many

Q -factors — which in turn can cause the solution to deviate from optimality. Furthermore, any action selection strategy should ensure that each Q -factor is updated infinitely often (theoretically). We reiterate that for the Robbins-Monro scheme to work (i.e., averaging to occur), theoretically, an infinitely large number of samples must be collected. This ensures that the estimates of the Q -factors converge to their mean values.

5.1.5 A worked-out example for Q -Learning

We next show some sample calculations performed in a simulator using Q -Learning. We will use Example A (see Section 5.1 of Chapter 8). We repeat the problem details below. The TPM associated with action 1 is

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix},$$

and that associated with action 2 is

$$\mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

The TRM for action 1 is

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix},$$

and that associated with action 2 is

$$\mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

The performance metric is discounted reward with a discounting factor of 0.8.

In what follows, we show step-by-step calculations of Q -Learning along one trajectory.

Let us assume that the system starts in state 1.

State 1. Set all the Q -factors to 0:

$$Q(1, 1) = Q(1, 2) = Q(2, 1) = Q(2, 2) = 0,$$

and all the V -factors (visit factors) to 0:

$$V(1, 1) = V(1, 2) = V(2, 1) = V(2, 2) = 0.$$

The set of actions allowed in state 1 is $\mathcal{A}(1) = \{1, 2\}$ and that allowed in state 2 is $\mathcal{A}(2) = \{1, 2\}$. Clearly $|\mathcal{A}(i)| = 2$ for $i = 1, 2$. Let the step size α be defined by A/n , where $A = 0.1$ and n denotes the number of visits to that particular state-action pair.

Select an action with probability $1/|\mathcal{A}(1)|$. Let the selected action be 1. Simulate action 1. Let the next state be 2.

State 2. The current state (j) is 2 and the old state (i) was 1. The action (a) selected in the old state was 1. So we now have to update $Q(1, 1)$.

Now:

$$V(1, 1) \leftarrow V(1, 1) + 1 = 0 + 1 = 1,$$

$$r(i, a, j) = r(1, 1, 2) = -5,$$

and

$$\max_b Q(j, b) = \max_b Q(2, b) = \max\{0, 0\} = 0.$$

Now $\alpha = 0.1/V(1, 1) = 0.1$.

Then

$$\begin{aligned} Q(1, 1) &\leftarrow (1 - \alpha)Q(1, 1) + \alpha[-5 + \lambda(0)] \\ &= -0.5. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 2. Simulate action 2. Let the next state be 2.

State 2 (again). The current state (j) is 2 and the old state (i) was also 2. The action (a) selected in the old state was 2. So we now have to update $Q(2, 2)$.

Now:

$$V(2, 2) \leftarrow V(2, 2) + 1 = 0 + 1 = 1,$$

$$r(i, a, j) = r(2, 2, 2) = 13,$$

and

$$\max_b Q(j, b) = \max_b Q(2, b) = \max\{0, 0\} = 0.$$

Now $\alpha = 0.1/V(2, 2) = 0.1$.

Then

$$\begin{aligned} Q(2, 2) &\leftarrow (1 - \alpha)Q(2, 2) + \alpha[13 + \lambda(0)] \\ &= 1.3. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 1. Simulate action 1. Let the next state be 1.

State 1 (again). The current state (j) is 1 and the old state (i) was 2. The action (a) selected in the old state was 1. So we now have to update $Q(2, 1)$.

Now:

$$V(2, 1) \leftarrow V(2, 1) + 1 = 0 + 1 = 1,$$

$$r(i, a, j) = r(2, 1, 1) = 7,$$

and

$$\max_b Q(j, b) = \max_b Q(1, b) = \max\{-0.5, 0\} = 0.$$

Now $\alpha = 0.1/V(2, 1) = 0.1$.

Then

$$\begin{aligned} Q(2, 1) &\leftarrow (1 - \alpha)Q(2, 1) + \alpha[7 + \lambda(0)] \\ &= 0.7. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(1)|$. Let the selected action be 2. Simulate action 2. Let the next state be 2.

State 2 (a third time). The current state (j) is 2 and the old state (i) was 1. The action (a) selected in the old state was 2. So we now have to update $Q(1, 2)$.

Now:

$$V(1, 2) \leftarrow V(1, 2) + 1 = 0 + 1 = 1,$$

$$r(i, a, j) = r(1, 2, 2) = 17,$$

and

$$\max_b Q(j, b) = \max_b Q(2, b) = \max\{0.7, 1.3\} = 1.3.$$

Now $\alpha = 0.1/V(2, 1) = 0.1$.

Then

$$\begin{aligned} Q(1, 2) &\leftarrow (1 - \alpha)Q(1, 2) + \alpha[17 + \lambda(1.3)] \\ &= 1.804. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 2. Simulate action 2. Let the next state be 1.

State 1 (a third time). The current state (j) is 1 and the old state (i) was 2. The action (a) selected in the old state was 2. So we now have to update $Q(2, 2)$.

Now:

$$V(2, 2) \leftarrow V(1, 2) + 1 = 1 + 1 = 2,$$

$$r(i, a, j) = r(2, 2, 1) = -14,$$

and

$$\max_b Q(j, b) = \max_b Q(1, b) = \max\{-0.5, 1.804\} = 1.804.$$

Now $\alpha = 0.1/V(2, 2) = 0.05$.

Then

$$\begin{aligned} Q(2, 2) &\leftarrow (1 - \alpha)Q(2, 2) + \alpha[-14 + \lambda(1.804)] \\ &= (1 - 0.05)(1.3) + 0.05[-14 + 0.8(1.804)]. \\ &= 0.60716. \end{aligned}$$

And so on. When we ran the algorithm within a simulator, we obtained the following results.

$Q(1, 1) = 1.6017$, $Q(1, 2) = 2.3544$, $Q(2, 1) = 1.4801$, and $Q(2, 2) = 1.0730$. The optimal policy is $(2, 1)$. Computer codes for Q -Learning can be found in Section 8.1 of Chapter 15.

5.2. Discounted reward RL based on policy iteration

In this section, we will pursue an RL approach, based on policy iteration, for solving discounted reward MDPs. Direct policy iteration is, of course, ruled out in the context of RL because in the policy evaluation phase, one would have to solve linear equations. And that would require the transition probabilities. As such, we will try to avoid having to solve linear equations in RL.

As a result, modified policy iteration (see Chapter 8) is clearly an idea worth pursuing because it does (partial) policy evaluation without solving any equations. In the policy evaluation phase of modified policy iteration, to determine the value function vector of the policy being evaluated, we perform value iteration of the given policy. Since value iteration can be done in a simulator using Q -factors, we can derive a policy iteration algorithm based on simulation.

The idea is simple. We start with an arbitrary policy. We evaluate the Q -factors associated with that policy in the simulator. Then we perform the policy improvement step, and select a new policy. Then we turn back to the simulator to evaluate the new policy. This will continue till no improvement is seen. We next discuss this idea in more detail.

5.2.1 Q -factor version of regular policy iteration

We need to define a Q -factor here *in terms of the policy*. For a state-action pair (i, a) , for a policy $\hat{\mu}$, where $a \in \mathcal{A}(i)$,

$$Q_{\hat{\mu}}(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) + \lambda J_{\hat{\mu}}(j)], \quad (9.13)$$

where $\vec{J}_{\hat{\mu}}$ is the value function vector associated with the policy $\hat{\mu}$. Notice the difference with the definition of the Q -factor in value iteration, which is:

$$Q(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) + \lambda J^*(j)], \quad (9.14)$$

where \vec{J}^* denotes the value function vector associated with the *optimal* policy.

Using the definition in Equation (9.13), we can come up with a version of policy iteration in terms of Q -factors.

Now, from the Bellman equation for a given policy $\hat{\mu}$, we have that:

$$J_{\hat{\mu}}(i) = \sum_{j=1}^{|S|} p(i, \mu(i), j)[r(i, \mu(i), j) + \lambda J_{\hat{\mu}}(j)], \quad \forall i. \quad (9.15)$$

From Equation (9.15) and Equation (9.13), one has that

$$J_{\hat{\mu}}(i) = Q_{\hat{\mu}}(i, \mu(i)), \quad \forall i. \quad (9.16)$$

Using Equation (9.16), Equation (9.13) can be written as:

$$Q_{\hat{\mu}}(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) + \lambda Q_{\hat{\mu}}(j, \mu(j))], \quad \forall i, a \in \mathcal{A}(i). \quad (9.17)$$

It is clear that Equation (9.17) is a Q -factor version of the equation used in the policy evaluation phase of policy iteration and is thus useful in devising a Q -factor version of policy iteration, which we present next.

5.2.2 Steps in the Q -factor version of regular policy iteration

Step 1: Set $k = 1$. Select a policy, $\hat{\mu}_k$, arbitrarily.

Step 2: For the policy, $\hat{\mu}_k$, obtain the values of $\vec{Q}_{\hat{\mu}_k}$ by solving the system of linear equations given below:

$$Q_{\hat{\mu}_k}(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) + \lambda Q_{\hat{\mu}_k}(j, \mu_k(j))] \quad$$

for $i = 1, 2, \dots, |\mathcal{S}|$.

Step 3: Generate a new policy $\hat{\mu}_{k+1}$, using the following relation:

$$\mu_{k+1}(i) \in \arg \max_{u \in \mathcal{A}(i)} Q_{\hat{\mu}_k}(i, u).$$

If possible, set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4: If the policy $\hat{\mu}_{k+1}$ is identical to policy $\hat{\mu}_k$, the algorithm terminates. Otherwise, set $k \leftarrow k + 1$, and go back to Step 2.

Instead of using linear algebra methods — such as Gauss-Jordan elimination — in Step 2 of the above algorithm, one can use the method of successive approximations (value iteration) to solve for the Q -factors. In a method of successive approximations, one starts with arbitrary values for the Q -factors and then uses an updating scheme derived from the Bellman equation repeatedly till the Q -factors converge. Clearly, the updating scheme, based on the equation in Step 2, would be:

$$Q_{\hat{\mu}_k}(i, a) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)[r(i, a, j) + \lambda Q_{\hat{\mu}_k}(j, \mu_k(j))]. \quad (9.18)$$

But the Q -factor can be expressed as an expectation as shown below:

$$\begin{aligned} Q_{\hat{\mu}_k}(i, a) &= E[r(i, a, j) + \lambda Q_{\hat{\mu}_k}(j, \mu_k(j))] \\ &= E[\text{SAMPLE}]. \end{aligned} \quad (9.19)$$

Then, using the Robbins-Monro scheme (see (9.7)), Equation (9.18) can be written, for all state-action pairs (i, a) , as:

$$\begin{aligned} Q_{\hat{\mu}_k}(i, a) &\leftarrow (1 - \alpha)Q_{\hat{\mu}_k}(i, a) + \alpha[\text{SAMPLE}] \\ &= (1 - \alpha)Q_{\hat{\mu}_k}(i, a) + \alpha[r(i, a, j) + \lambda Q_{\hat{\mu}_k}(j, \mu_k(j))], \end{aligned}$$

where the last relationship follows from Equation (9.19).

What we have derived above is a scheme to evaluate the Q -factors, associated with a policy, in a simulator. Let us next discuss how policy iteration may be carried out in a simulator in an iterative style.

We begin with an arbitrary policy. Then we use a simulator to estimate the Q -factors associated with the policy. This, of course, is the policy evaluation phase. We refer to each policy evaluation as an **episode**. When an episode ends, we carry out the *policy improvement* step. In this step, we copy the Q -factors into a new vector called P -factors, and then destroy the old Q -factors. Thereafter a new episode is started to determine the new Q -factors associated with the new policy. The P -factors define the new policy to be evaluated and as such are needed in evaluating the new Q -factors.

5.2.3 Steps in Q - P -Learning

We will next present the step-by-step details of Q - P -Learning, which is essentially an RL algorithm based on policy iteration.

Step 1. Initialize all the P -factors, $P(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to arbitrary values. Set all the visit-factors, $V(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to 0. Set E , the number of episodes, to 1. Initialize E_{\max} and k_{\max} to large numbers.

Step 2 (Policy Evaluation). Start fresh simulation. Set all the Q -factors, $Q(l, u)$, to 0. Let the current system state be i . Set k , the number of iterations within an episode, to 1.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 2b. Let the next state encountered in the simulator be j . Increment $V(i, a)$ by 1. Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j . Set $\alpha = 1/V(i, a)$. Then update $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \lambda Q(j, \arg \max_{b \in \mathcal{A}(j)} P(j, b))].$$

Step 2c. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$ and return to Step 2a. Otherwise go to Step 3.

Step 3 (Policy Improvement). Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$,

$$P(l, u) \leftarrow Q(l, u).$$

Then increment E by 1. If E equals E_{\max} , go to Step 4. Otherwise, go to Step 2.

Step 4. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

Several comments regarding the algorithm are in order here.

- It is to be understood that the estimation of the Q -factors of a policy will, necessarily, be imperfect. This is so since we use a finite number (k_{\max}) of iterations to estimate them. We will prove in Chapter 13 that it may not be necessary to obtain perfect estimates of the Q -factors.
- It makes sense to increase the value of k_{\max} with E . This will ensure that the policy-evaluation performance keeps improving as the algorithm approaches the optimal solution.

- Instead of choosing each action with equal probability in Step 2a, one can choose an *exploratory* strategy (see Section 5.1.4).
- The algorithm presented above is the Modified Q -Learning algorithm of Rummery and Niranjan [150]; the slight variation is in Step 2a where they used an exploratory strategy.

6. Average reward Reinforcement Learning

This section will discuss DP-based RL algorithms for average reward MDPs. Like in the discounted case, we will divide this section into two parts — one devoted to value iteration and the other to policy iteration. Before discussing the algorithms for average reward RL, we need to discuss whether special algorithms are needed for the average reward case.

6.1. Discounted RL for average reward MDPs

Before average reward RL algorithms were developed, the use of discounted reward algorithms to solve average reward MDPs was quite popular. This tendency was corrected probably after the publication of some papers that directly attacked this issue; see Mahadevan [108] and Mahadevan [109]. See also Schwartz [155]. It may be possible to justify the use of **discounted** RL algorithms on **average** reward MDPs by using a value very close to 1 (tending to 1) for the discounting factor. Now, if the discounted algorithms were sufficient, there would be no need for any average reward DP algorithm. Then, we could use a value for λ that is very close to 1, and use a discounted reward algorithm.

It has been established in the literature that *larger* the value of λ , the *longer* it takes for value iteration to converge. In fact, the number of iterations required, to obtain a given amount of accuracy, increases quite dramatically as the value of λ increases. See Section 6.3.2 of Puterman [140] (page 162) for a worked-out example. It is quite clear from his example that in practice, that is through the computational viewpoint, *this* strategy for solving average reward problems will be inefficient. As such, it is important that we develop the theory of average reward RL separately.

6.2. Average reward RL based on value iteration

As mentioned earlier in Chapter 8, *regular* value iteration is rarely prescribed for average reward MDPs because the iterates can become unbounded. Since RL algorithms are based on DP algorithms, this difficulty carries over into the RL arena. In other words, RL inherits difficulties from DP algorithms. As such, we will have to derive an RL algorithm from *relative* value iteration. We will present a general RL scheme based on relative value iteration, since it is known that relative value iteration keeps the iterates bounded.

The Q -Learning algorithm for average reward would simply be:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b)] \quad (9.20)$$

for all (i, a) pairs. This can cause the iterates to be unbounded. To keep this from happening, we can do the following:

- Select a state-action pair arbitrarily. Let us denote it by (i^*, a^*) . Then update each Q -factor using the following rule:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b) - Q(i^*, a^*)].$$

We will refer to this algorithm as the “Relative Q -Learning” algorithm.

We next present step-by-step details of Relative Q -Learning.

6.2.1 Steps in Relative Q -Learning

Step 1. Initialize all the Q -factors to 0 and all the visit factors to 0. In other words: Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set A , the step size constant, to a number less than 1. Start system simulation at any arbitrary state. Select any one state-action pair to be a distinguished state-action pair (i^*, a^*) .

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried, by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b) - Q(i^*, a^*)].$$

Step 5. If $k < k_{\max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

The following comment is in order.

- Relative value iteration can diverge asynchronously. That is, asynchronous relative value iteration may produce a solution different from that of synchronous relative value iteration. However, it can be shown that asynchronous Relative Q -Learning, which is the algorithm presented above, approximates the behavior of the synchronous algorithm as long as the step size is small enough.

Now, we must address an important issue. The performance of the algorithm can be easily tested by using the policy in a simulator. We can re-run the simulator using the policy, and calculate the average reward in the process. (This is called the frozen phase.) The following routine will help us determine the average reward of a given policy. The simulation must be run for a sufficiently long time (MAX_TIME) to obtain a good estimate.

6.2.2 Calculating the average reward of a policy in a simulator

Step 1. Set TOTAL_REWARD to 0 and TOTAL_TIME to 0. Set MAX_TIME to a large number.

Step 2. Let the current state be i . Select action a such that:

$$a \in \arg \max_{u \in \mathcal{A}(i)} Q(i, u).$$

In other words, choose the action associated with the maximum Q -factor for that state.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . Let $t(i, a, j)$ be the time spent in the same transition. The quantities $r(i, a, j)$ and $t(i, a, j)$ will be determined by the simulator. Then update TOTAL_REWARD and TOTAL_TIME as shown below:

$$\text{TOTAL_REWARD} \leftarrow \text{TOTAL_REWARD} + r(i, a, j)$$

and

$$\text{TOTAL_TIME} \leftarrow \text{TOTAL_TIME} + t(i, a, j).$$

Step 4. If $\text{TOTAL_TIME} < \text{MAX_TIME}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 5.

Step 5. Calculate the average reward of the policy as:

$$\rho = \text{TOTAL_REWARD}/\text{TOTAL_TIME};$$

and stop.

The above routine calculates the average reward using one *replication* of the simulation. Ideally, we should calculate the average reward from several replications and then declare the mean of the values obtained from all the replications to be the average reward. This mean represents a “good” estimate of the actual average reward. To start a fresh replication, one must change the seed (see Chapter 4) in the simulation program.

Relative Q -Learning and Example A. Results from Example A (see Section 5.1 of Chapter 8) with Relative Q -Learning are as follows. The Q -factors obtained are:

$$Q(1, 1) = 6.873, Q(1, 2) = 14.326, Q(2, 1) = 10.690, \text{ and } Q(2, 2) = 8.162.$$

As a result, the policy learned is $(2, 1)$. When this policy is run in a simulator, the average reward obtained is 10.56. Computer codes for Relative Q -Learning can be found in Section 8.2 of Chapter 15.

6.3. Other algorithms for average reward MDPs

There are some other algorithms that have been suggested in the literature for average reward MDPs. Examples are R -Learning, SMART (Semi-Markov Average Reward Technique), and Relaxed-SMART. The motivation for these algorithms is the Bellman equation. The Bellman optimality equation for average reward contains the term, ρ^* , which denotes the average reward of the *optimal* policy. Now, the fact is that since ρ^* is initially unknown (since the optimal policy is unknown), for average-reward value-iteration algorithms, we strive to use versions of the Bellman equation that do not contain the term ρ^* . This is in fact the motivation for regular value iteration and relative value iteration. In the literature on RL, one finds a number of value iteration algorithms partially based on the Bellman equation. These algorithms use *estimates* of ρ^* instead of the actual value of ρ^* . R -Learning, SMART, and Relaxed-SMART are examples of such algorithms.

We will discuss these algorithms next, beginning with R -Learning.

6.3.1 Steps in R -Learning

Step 1. Initialize all the Q -factors to 0 and all the visit factors to 0. In other words: Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently

large number. Set A , the step size constant, to a number less than 1. Start system simulation at any arbitrary state.

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$. Set $\theta \leftarrow 0$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b) - \theta].$$

Update θ using the following equation:

$$\theta \leftarrow (1 - \beta)\theta + \beta[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b) - \max_{c \in \mathcal{A}(i)} Q(i, c)],$$

where $\beta = 1/k$.

Step 5. If $k < k_{\max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

R-Learning was proposed by Schwartz [155]. The action-selection strategy in *R*-Learning as proposed in [155] is, however, one of exploration (see Section 5.1.4) and not the one described here.

SMART and Relaxed-SMART were designed for solving SMDPs. Below, we present an MDP version.

6.3.2 Steps in SMART for MDPs

Step 1. Initialize all the Q -factors to 0 and all the visit factors to 0. In other words: Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently

large number. Set A , the step size constant, to a number less than 1. Start system simulation at any arbitrary state. Set n to 0. Set ϕ to 1. Set $\theta \leftarrow 0$.

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

If

$$a \in \arg \max_{u \in \mathcal{A}(i)} Q(i, u),$$

set $\phi = 0$. Otherwise, set $\phi = 1$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried, by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b) - \theta].$$

If ϕ equals 0 (that is, the action selected in Step 2 was greedy),

- update θ using the following equation:

$$\theta \leftarrow (1 - \beta)\theta + \beta[r(i, a, j)],$$

where $\beta = B/(n + 1)$ with $B = 1$ and

- then increment n by 1.

Step 5. If $k < k_{max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

We must point out at this stage that θ is an estimate of the average reward. Indeed, θ should start with an estimate of the average reward of the optimal policy ρ^* . The idea is to use, right from the beginning, an approximation of the Bellman equation in the updating the Q -factors; the approximation gets better with iterations since θ hopefully tends to ρ^* .

In SMART, θ starts out as 0. In Relaxed-SMART, in Step 1, θ is initialized to a guessed estimate of θ . (We will discuss how to guess an estimate when we discuss Relaxed-SMART for SMDPs below.) The second difference between Relaxed-SMART and SMART is that B in Step 4 of Relaxed-SMART

is set to a value less than 1. These changes are necessary to show the convergence of the algorithm mathematically; however, in practice on most problems SMART works just as well as Relaxed-SMART. In fact, SMART may actually converge faster than Relaxed-SMART. For journal paper references of SMART and Relaxed-SMART, see [37] and [63], respectively.

6.4. An average reward RL algorithm based on policy iteration

An average reward RL algorithm based on policy iteration can be developed in a manner analogous to the derivation of Q - P -Learning for the discounted case. In the average reward problem, instead of using the Q -Learning algorithm in the policy evaluation phase, we will have to use the Relative Q -Learning algorithm. Since the ideas are very similar to those discussed in the discounted reward case, we will not discuss this topic any further. We supply the steps in the algorithm and an example, next.

6.4.1 Steps in Q - P -Learning for average reward

Step 1. Initialize all the P -factors, $P(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to arbitrary values. Set all the visit-factors, $V(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$ to 0. Set E , the number of episodes, to 1. Initialize E_{\max} and k_{\max} to large numbers. Select any one state-action pair to be a distinguished state-action pair (i^*, a^*) .

Step 2 (Policy Evaluation). Start fresh simulation. Set all the Q -factors, $Q(l, u)$, to 0. Let the current system state be i . Set k , the number of iterations within an episode, to 1.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 2b. Let the next state encountered in the simulator be j . Increment $V(i, a)$ by 1. Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j . Set $\alpha = 1/V(i, a)$. Then update $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1-\alpha)Q(i, a) + \alpha[r(i, a, j) - Q(i^*, a^*) + Q(j, \arg \max_{b \in \mathcal{A}(j)} P(j, b))]$$

Step 2c. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$, and return to Step 2a. Otherwise go to Step 3.

Step 3 (Policy Improvement). Set for all (l, u) pairs,

$$P(l, u) \leftarrow Q(l, u).$$

Then increment E by 1. If E equals E_{\max} , go to Step 4. Otherwise, go to Step 2.

Step 4. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

After each policy improvement step (that is Step 3), one can find the average reward of the policy using the routine discussed in Section 6.2.2. This way, one gets an idea of where the algorithm is headed — that is, whether the average reward is improving. The SMDP version of Q - P -Learning can also be used for the MDP setting the transition time to 1.

With this, we have come to an end to our discussion on RL methods for MDPs. We next discuss RL methods for SMDPs.

7. Semi-Markov decision problems and RL

In this section, we will first discuss the discounted reward case for solving semi-Markov decision problems (SMDPs), using RL, and then discuss the average reward case.

7.1. Discounted Reward

The discounted reward SMDP can be solved using either a value iteration based approach or else a policy iteration based approach. We will first discuss the value iteration based approach.

The value iteration based algorithm for SMDPs is a simple extension of Q -Learning for MDPs. Please refer to value iteration for SMDPs and also review definitions of DTMDPs, random time SMDPs, and CTMDPs (see Chapter 8).

The algorithm we present will work for DTMDPs (Deterministic Time Markov Decision Problems). Like in the previous chapter, we will assume that the same algorithm will work for random time SMDPs and CTMDPs. When there is a continuous reward rate, the algorithms we present might run into trouble, since the discounting may have to take care of the time at which the immediate reward was received. In what follows, we present the RL analogue of discounted reward value iteration for DTMDPs (see Chapter 8).

7.1.1 Steps in Q -Learning for discounted reward DTMDPs

Step 1. Initialize the Q -factors to 0 and the visit factors also to 0. In other words: Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently

large number. Set A , the step size constant, to a positive number less than 1. Let γ denote the discounting factor per unit time. Start system simulation at any arbitrary state.

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . Let $t(i, a, j)$ be the time spent in the same transition. The quantities $r(i, a, j)$ and $t(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried, by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + e^{-\gamma t(i, a, j)} \max_{b \in \mathcal{A}(j)} Q(j, b)].$$

Step 5. If $k < k_{max}$, set $i \leftarrow j$ and then go to Step 2. Else go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

The following section is devoted to *Q-P-Learning* for discounted reward DTMDPs.

7.1.2 Steps in *Q-P-Learning* for discounted reward DTMDPs

Step 1. Initialize the *P*-factors, $P(l, u)$, for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to arbitrary values. Set the visit-factors, $V(l, u)$, for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to 0. Set E , the number of episodes, to 1. Initialize E_{max} and k_{max} to large numbers.

Step 2 (Policy Evaluation). Start fresh simulation. Set all the *Q*-factors, $Q(l, u)$ to 0. Let the system state be i . Set k , the number of iterations within an episode, to 1. Let γ denote the discounting factor per unit time.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 2b. Let the next state encountered in the simulator be j . Increment $V(i, a)$ by 1. Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j . Set $\alpha = 1/V(i, a)$. Then update $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + e^{-\gamma t(i, a, j)} \max_{b \in \mathcal{A}(j)} P(j, b)].$$

Step 2c. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$ and return to Step 2a.
Otherwise go to Step 3.

Step 3 (Policy Improvement). Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$P(l, u) \leftarrow Q(l, u).$$

Then increment E by 1. If E equals E_{\max} , go to Step 4. Otherwise, go to Step 2.

Step 4. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

7.2. Average reward

The problem of solving average reward SMDPs using RL is perhaps harder than all the other problems discussed above. Recall from the previous chapter that value iteration for average reward SMDPs is inexact. The value iteration approach for SMDPs requires a so-called “uniformization,” which is 1) inexact and 2) can be used only when one has access to the transition probabilities. Because of the latter, we avoid uniformization in RL. But RL depends on value iteration (remember that even policy iteration based RL approaches need value iteration in the policy evaluation phase). As such ways must be devised to work around this difficulty and solve the problem without resorting to uniformization.

SMART and Relaxed-SMART, which we have discussed in the context of MDPs, attempt to solve the average reward SMDP without uniformizing the SMDPs. They have gained encouraging empirical success, and their convergence properties have been studied in [63].

It must be kept in mind that 1) a great majority of real-world problems tend to be SMDPs (and it is not very straightforward to study them as MDPs) and 2) average reward tends to be a more popular performance measure in the industry perhaps because the discounting factor is not easy to measure. As a result, despite the inherent difficulties of solving this problem, it happens to be important from the commercial viewpoint.

We will next discuss SMART. SMART, as we have said earlier, is partly based on the Bellman equation. Recall that in average reward MDPs, we do not use the Bellman equation *directly* to formulate the updating equation for value iteration. The reason is that it contains an unknown term — ρ^* , the average reward of the optimal policy.

In MDPs, the unknown term, ρ^* , can be avoided in the algorithm. See the material presented for value iteration for average reward MDPs in Chapter 8.

In SMDPs, we cannot eliminate the ρ^* term from the value iteration updating equation, unless we uniformize the SMDP. For an explanation of this, see Example in Section 9.2.5 in Chapter 8.

A Q -factor version of the Bellman optimality equation for average reward SMDPs can be derived in a manner similar to the derivation of the Q -factor version of the same equation for discounted reward. The equation is:

$$Q(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) - \rho^* t(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b)]$$

for all (i, a) . This suggests a value iteration algorithm of the form :

$$Q(i, a) \leftarrow \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) - \rho^* t(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b)] \quad \forall(i, a).$$

The difficulty with this updating equation is that the value of ρ^* is unknown. So instead one can use:

$$Q(i, a) \leftarrow \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) - \tilde{\rho}^* t(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b)], \quad \forall(i, a)$$

where $\tilde{\rho}^*$ is a guessed estimate of ρ^* . This updating equation is the basis for the algorithm SMART that we will present next. In the algorithm, we will not only update the Q -factors, but also update the value of ρ . The idea is to ensure that ρ tends to ρ^* .

The updating of ρ is done using a step size based updating equation. The equation is updated only when the action selected in the simulator is a greedy action.

7.2.1 Steps in SMART for SMDPs

Step 1. Initialize the Q -factors to 0 and the visit factors also to 0. In other words: Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$

$$Q(l, u) \leftarrow 0 \text{ and } V(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set A , the step size constant, to a number less than 1. Start system simulation at any arbitrary state. Set n to 0. Set each of TOTAL_REWARD and TOTAL_TIME to 0. Set ϕ to 1. Set $\rho \leftarrow 0$.

Step 2. Let the state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$. If

$$a \in \arg \max_{u \in \mathcal{A}(i)} Q(i, u),$$

set $\phi = 0$. Otherwise set $\phi = 1$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . Let $t(i, a, j)$ denote the time spent in the same transition. The quantities $r(i, a, j)$ and $t(i, a, j)$ will be determined by the simulator. Increment $V(i, a)$, which denotes the number of times the state-action pair (i, a) has been tried by 1. Increment k by 1. Then calculate $\alpha = A/V(i, a)$.

Step 4a. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) - \rho t(i, a, j) + \max_{b \in \mathcal{A}(j)} Q(j, b)].$$

Step 4b. If ϕ equals 0 (that is, a is a greedy action), update TOTAL_REWARD and TOTAL_TIME using the following.

$$\text{TOTAL_REWARD} \leftarrow \text{TOTAL_REWARD} + r(i, a, j)$$

and

$$\text{TOTAL_TIME} \leftarrow \text{TOTAL_TIME} + t(i, a, j).$$

Step 4c. Update ρ using the following equation:

$$\rho \leftarrow [\text{TOTAL_REWARD}/\text{TOTAL_TIME}].$$

Step 5. If $k < k_{max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

Relaxed-SMART has two differences with SMART. They are:

- In Relaxed-SMART, in Step 1, we set $\rho \leftarrow \tilde{\rho}^*$, where $\tilde{\rho}^*$ is a guessed estimate of ρ^* , the average reward of the optimal policy. There are two techniques to “guess” the value of ρ^* .
 - In many real-world problems, *heuristic algorithms* are available to solve the problem. One can use the average reward of the heuristic as the guessed value for $\tilde{\rho}^*$.
 - When no other heuristic is available, one may use Q -Learning with $\lambda = 0.99$ (or some value close to 1.) Q -Learning, in practice, has been

shown to produce extremely good solutions on many average problems and serves as an excellent heuristic for average reward problems. The average reward of the solution produced by Q -Learning can then be used as $\tilde{\rho}^*$.

2. The updating of ρ (Step 4c) in Relaxed-SMART involves a step size and is shown below. (Steps 4a and 4b in Relaxed-SMART are identical to those of SMART).

Step 4c. Update ρ using the following equation:

$$\rho \leftarrow (1 - \beta)\rho + \beta[\text{TOTAL_REWARD}/\text{TOTAL_TIME}],$$

where $\beta = B/(n + 1)$ with $B < 1$ and then increment n by 1.

Relaxed-SMART and Example B. When Relaxed-SMART is used on Example B (see Section 9.2.2 of Chapter 8), we obtain the following results.

$$Q(1, 1) = -575.98, Q(1, 2) = -2786.013, Q(2, 1) = -1986.860, \text{ and}$$

$$Q(2, 2) = -661.33.$$

It follows that the policy learned is $(1, 2)$ and the average reward of this policy (which can be found by simulation) is 2.0981. Computer codes for Relaxed-SMART can be found in Section 8.3 of Chapter 15.

We next discuss RL approach — based on policy iteration — for solving SMDPs. A motivation for using policy iteration based algorithms for solving average reward SMDPs is that value iteration is *inexact*. Policy iteration for average reward SMDPs is, however, *exact!* We next present a Q - P -Learning algorithm for solving average reward SMDPs.

7.2.2 Steps in Q - P -Learning for SMDPs

The Q - P -Learning algorithm has a policy evaluation phase, which is performed via value iteration. For discounted reward MDPs and SMDPs, regular value iteration (derived from the Bellman policy equation) keeps the iterates bounded and also converges. For average reward MDPs, one uses relative value iteration (derived from the Bellman policy equation). For average reward SMDPs, which is the problem under consideration in this section, we cannot use relative value iteration for the same reason discussed in the previous section. But it turns out, fortunately, that one can use the Bellman equation directly for value iteration. The Bellman equation that is needed in this case is the Bellman policy equation, which can be expressed in terms of Q -factors as follows.

$$Q(i, a) = \sum_{j=1}^{|S|} p(i, a, j)[r(i, a, j) - \rho \mu t(i, a, j) + Q(j, \mu(j))]$$

The above is the Bellman policy equation for the policy $\hat{\mu}$. Now, the question is how does one obtain the value of $\rho_{\hat{\mu}}$? The answer is: before updating the Q -factors using this equation, one can estimate the average reward of the policy $\hat{\mu}$ in a simulator. Read the following description carefully. In Step 2, we estimate the average reward of the policy, whose Q -factors are evaluated later in Step 3.

Step 1: Initialize the vector $P(l, u)$ for all states $l \in S$ and all $u \in \mathcal{A}(l)$ to random values. Set $E = 1$ (E denotes the number of episodes) and initialize E_{\max} and k_{\max} to large numbers.

Step 2: Set $Q(l, u) = 0$ for all $l \in S$ and all $u \in \mathcal{A}(l)$. Simulate the system for a sufficiently long time using in state m , the action given by $\arg \max_{b \in \mathcal{A}(m)} P(m, b)$. At the end of the simulation, divide the total of immediate rewards by the total of immediate times to obtain an estimate of the average reward ρ . Use averaging with several replications to obtain a good estimate of ρ . Set k , the number of iterations within an episode, to 1.

Step 3 (Policy evaluation) Start fresh simulation. Let the current system state be $i \in S$.

Step 3a: Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 3b: Let the next decision-making state encountered in the simulator be j . Also, let $t(i, a, j)$ be the transition time (from state i to state j) and let $r(i, a, j)$ be the immediate reward.

Step 3c: Calculate β using m . Then change $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1-\beta)Q(i, a) + \beta[r(i, a, j) - \rho t(i, a, j) + Q(j, \arg \max_{b \in \mathcal{A}(j)} P(j, b))].$$

Step 3 d: Increment k by 1. Calculate k_{\max} using E . If $k < k_{\max}$ set current state i to new state j and then go to Step 3a; else go to Step 4.

Step 4: (*Q to P conversion - policy improvement*) Set $P(l, u) \leftarrow Q(l, u)$ for all l and $u \in \mathcal{A}(l)$. Set $E \leftarrow E + 1$. If E equals E_{\max} go to Step 5; else go back to step 2.

Step 5. For each $l \in S$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

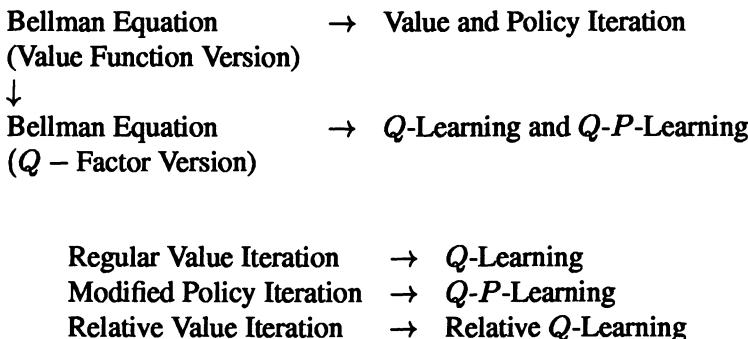
The policy (solution) generated by the algorithm is \hat{d} . Stop.

Some important remarks are in order.

- It has been shown that with a sufficiently large value for k_{\max} the policy evaluation phase can be carried out to near-perfection. See Chapter 13 for a convergence analysis of this algorithm.
- In Step 2, one calculates an **estimate** of the average reward of the policy, which is subsequently evaluated. It is necessary to ensure that the estimate is good; however, it can be shown that even if the estimate is imperfect, the algorithm can proceed along the right track under certain conditions. Again see Chapter 13 for an analysis of this issue.
- The above algorithm can be used for solving MDPs by setting $t(i, a, j)$ to 1. This may cause the Q -factors to diverge. If that happens, it is best to use the relative value iteration version given in Q - P -Learning for average reward MDPs.

8. RL Algorithms and their DP counterparts

At this point, it will be a good idea to summarize the relationship between RL algorithms and their DP counterparts. The two main algorithms of DP: value and policy iteration are based on the Bellman equation, which contains the elements of the **value function** as the unknowns. One can, as discussed above, derive a Q -factor version of the Bellman equation. Most RL algorithms, like Q -Learning and Q - P -Learning, are based on the Q -factor version of the Bellman equation. The following diagrams help summarize the relationship between DP and RL. In the diagrams, the entity at the tip of the arrow will denote one that can be derived from the entity at the tail.



9. Actor-Critic Algorithms

The so-called actor-critic algorithms have been around for a longer time than the RL algorithms that we have discussed above. Actor-critic algorithms were the precursors to modern RL. They follow policy iteration in spirit, and in spite of this differ in many ways from the regular policy iteration algorithm that is based on the Bellman equation.

The general idea underlying the working mechanism of actor-critic algorithms is to start with a “stochastic policy” in the simulator. A stochastic policy is one in which actions are selected in probabilistic manner. In other words, associated with each state-action pair (i, a) is a probability, $P(i, a)$, that satisfies

$$\sum_{a \in \mathcal{A}(i)} P(i, a) = 1,$$

and in state i , action a is *selected* with the probability $P(i, a)$.

In actor-critic algorithms, the Bellman equation is used to update the probabilities in the simulator. The goal is to ensure that in the long run, the probability of selecting the optimal action in each state tend to 1. If there are m optimal actions in a given state, then the probability of each of the m actions should tend to $1/m$.

To show how the Bellman equation is used, consider the following update rule for the discounted MDP case:

$$P(i, a) \leftarrow P(i, a) + \alpha \delta$$

where

$$\delta = r(i, a, j) + \lambda J(j) - J(i).$$

This update rule can be used within a simulator. It is performed after the state transition from i to j .

10. Model-building algorithms

Scattered in the literature on RL, one finds some papers that discuss a class of Algorithms called model-based algorithms. We prefer to call them model-building algorithms because they actually *build* the transition probability model *during the run time of the algorithm*.

It is to be understood clearly that model-building algorithms do not require the transition probability model; like the model-free algorithms of the previous sections, all they need is a *simulation model* of the system.

We reserve the word model-based for those algorithms that already have the transition probability model. All DP algorithms are model-based.

The first issue that must be addressed here is: how does one build a model within a simulator? The mechanism of model building is, in fact, conceptually simple. We have already discussed how to construct the transition probability model — in a simulator — by straightforward counting (see Section 3). We will employ the same principle here to generate the model. Model-building algorithms do not wait for the model to be built. They start DP within the simulator *even as the model is being built!* It is precisely for this reason that the name model-building seems more appropriate.

Most of the model-building algorithms proposed in the literature are based on the idea of computing the value function rather than computing Q -factors. However, since the updating within a simulator is asynchronous, we believe that step-size-based, Q -factor versions are perhaps more appropriate.

We will first present model-building algorithms that compute the *value function* and not the Q -factors. These algorithms will be referred to as H -Learning algorithms, since they strive to find the value function vector — \vec{h} . The name H -Learning was first used by Tadepalli and Ok [167]. In the third and fourth subsections of this section, we will present a model-building Q -learning (discounted reward) algorithm and a model-building Relative Q -learning (average reward) algorithm, respectively. These algorithms, as their name suggests, will strive to evaluate Q -factors and are step-size based. The following algorithm is based on the work in [11].

10.1. H -Learning for discounted reward

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$h(l) \leftarrow 0, V(l, u) \leftarrow 0, W(l, u, m) \leftarrow 0, \text{ and } \bar{r}(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Start system simulation at any arbitrary state.

Step 2. Let the state be i . Select an action in state i using some action selection strategy (see Comment 1 below for more on this topic). Let the selected action be a .

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator.

Step 4. Increment both $V(i, a)$ and $W(i, a, j)$ by 1.

Step 5. Update $\bar{r}(i, a)$ as follows:

$$\bar{r}(i, a) \leftarrow \bar{r}(i, a) + [r(i, a, j) - \bar{r}(i, a)]/V(i, a).$$

Step 6. Calculate for all $l \in \mathcal{S}$,

$$p(i, a, l) \leftarrow W(i, a, l)/V(i, a).$$

Step 7. Update $h(i)$ using the following equation:

$$h(i) \leftarrow \max_{u \in \mathcal{A}(i)} [\bar{r}(i, u) + \lambda \sum_{l \in \mathcal{S}} p(i, u, l)h(l)].$$

Step 8. Increment k by 1. If $k < k_{\max}$, set

$$i \leftarrow j,$$

and go to Step 2. Otherwise go to Step 9.

Step 9. For each $m \in \mathcal{S}$, select

$$d(m) \in \arg \max_{u \in \mathcal{A}(m)} [\bar{r}(m, u) + \lambda \sum_{l \in \mathcal{S}} p(m, u, l) h(l)].$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

Some comments are necessary here.

- The action selection strategy can be one of exploration (see Section 5.1.4). One may also select each action with equal probability.
- The values of the $V(., .)$ and the $W(., ., .)$ terms contain the transition probability model. Steps 4, 5 and 6 are related to model-building.
- Step 5 can, in fact, be written in the Robbins-Monro form as shown below:

$$\bar{r}(i, a) \leftarrow [1 - \frac{1}{V(i, a)}] \bar{r}(i, a) + \frac{1}{V(i, a)} r(i, a, j).$$

- Notice that in comparison to model-free algorithms, the computation required per iteration in model-building algorithms is more intensive. This is because Steps 5 and 6 do not exist in model-free algorithms. Furthermore, the computational burden in Step 7 — in comparison to say Step 4 in Q -Learning — is heavier; the additional burden is due to the sum in the right hand side of the equation in Step 7.

10.2. H -Learning for average reward

The H -Learning algorithm that we are about to present is for average reward MDPs. It has similarities with the algorithm presented in the previous section and with R -Learning. Remarkable empirical success has been reported with the use of this algorithm [167].

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$h(l) \leftarrow 0, V(l, u) \leftarrow 0, W(l, u, m) \leftarrow 0, \text{ and } \bar{r}(l, u) \leftarrow 0.$$

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently

large number. Set α , the step size, to a small value. Start system simulation at any arbitrary state. Set ρ , the average reward estimate, to 0.

Step 2. Let the current state be i . Select an action in state i using some action selection strategy (see Comment 1 below for more on this topic). Let the selected action be a .

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1.

Step 4. Increment both $V(i, a)$ and $W(i, a, j)$ by 1.

Step 5. Update $\bar{r}(i, a)$ as follows:

$$\bar{r}(i, a) \leftarrow \bar{r}(i, a) + [r(i, a, j) - \bar{r}(i, a)]/V(i, a).$$

Step 6. Calculate for all $l \in \mathcal{S}$,

$$p(i, a, l) \leftarrow W(i, a, l)/V(i, a).$$

Step 7. Define the set $\mathcal{G}(i)$ to be the set of all actions u that maximize:

$$\bar{r}(i, u) + \sum_{l \in \mathcal{S}} p(i, u, l)h(l).$$

If $a \in \mathcal{G}(i)$, then

- set

$$\rho \leftarrow (1 - \alpha)\rho + \alpha[\bar{r}(i, a) - h(i) + h(j)].$$

- set

$$\alpha \leftarrow \alpha/(1 + \alpha).$$

Step 8. Update $h(i)$ using the following equation:

$$h(i) \leftarrow \max_{u \in \mathcal{A}(i)} [\bar{r}(i, u) + \sum_{l \in \mathcal{S}} p(i, u, l)h(l)] - \rho.$$

Step 9. Increment k by 1. If $k < k_{\max}$, set

$$i \leftarrow j$$

and go to Step 2. Otherwise go to Step 10.

Step 10. For each $m \in \mathcal{S}$, select

$$d(m) \in \arg \max_{u \in \mathcal{A}(m)} [\bar{r}(m, u) + \sum_{l \in \mathcal{S}} p(m, u, l)h(l)].$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

Some comments are in order.

- The action selection strategy suggested in Tadepalli and Ok is one of exploration (see Section 5.1.4)..
- For more details on exploratory strategies that seem to have worked in practice for the H -Learning algorithm presented above, see [167].
- The initial value of α should preferably be a small number. Large values (such as 1) for step sizes can cause instability especially if there is a large amount of variance in the immediate rewards.

10.3. Model-building Q -Learning

The algorithm presented in this section is for discounted reward MDPs. It has two notable differences with the model-building algorithms presented earlier. 1) It computes Q -factors — as opposed to computing the value function vector (\vec{h}) and 2) it uses a step size.

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$Q(l, u) \leftarrow 0, V(l, u) \leftarrow 0 \text{ and } \bar{r}(l, u) \leftarrow 0, \text{ and } W(l, u, m) \leftarrow 0.$$

Set the number of jumps of the Markov chain k to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set α , the step size, to a small value. Start system simulation at any arbitrary state.

Step 2. Let the current state be i . Select action $u \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$. Let the selected action be a .

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1.

Step 4. Increment both $V(i, a)$ and $W(i, a, j)$ by 1.

Step 5. Update $\bar{r}(i, a)$ as follows:

$$\bar{r}(i, a) \leftarrow \bar{r}(i, a) + [r(i, a, j) - \bar{r}(i, a)]/V(i, a).$$

Step 6. Calculate for all $l \in \mathcal{S}$,

$$p(i, a, l) \leftarrow W(i, a, l)/V(i, a).$$

Step 7. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[\bar{r}(i, a) + \lambda \sum_{l \in \mathcal{S}} p(i, a, l) \max_{b \in \mathcal{A}(l)} Q(l, b)].$$

Step 8. Increment k by 1. If $k < k_{\max}$, set

$$i \leftarrow j$$

and go to Step 2. Otherwise go to Step 9.

Step 9. For each $m \in \mathcal{S}$, select

$$d(m) \in \arg \max_{b \in \mathcal{A}(m)} Q(m, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

The step size, α , can be of a diminishing form. It plays the role of smoothing the transformation, which of course is necessary to ensure convergence in the asynchronous conditions of the simulator.

10.4. Model-building relative Q -Learning

The algorithm that we will present next is for average reward. It is very similar to the algorithm of the previous section excepting for the fact that this is based on *relative* value iteration.

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$Q(l, u) \leftarrow 0, V(l, u) \leftarrow 0 \text{ and } \bar{r}(l, u) \leftarrow 0, \text{ and } W(l, u, m) \leftarrow 0.$$

Set the number of jumps of the Markov chain k to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set α the step size to a small value. Select some state-action pair (i^*, a^*) to be a distinguished state-action pair. Start system simulation at any arbitrary state.

Step 2. Let the state be i . Select action $u \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$. Let the selected action be a .

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence

of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1.

Step 4. Increment both $V(i, a)$ and $W(i, a, j)$ by 1.

Step 5. Update $\bar{r}(i, a)$ as follows:

$$\bar{r}(i, a) \leftarrow \bar{r}(i, a) + [r(i, a, j) - \bar{r}(i, a)]/V(i, a).$$

Step 6. Calculate for all $l \in \mathcal{S}$,

$$p(i, a, l) \leftarrow W(i, a, l)/V(i, a).$$

Step 7. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1-\alpha)Q(i, a) + \alpha[\bar{r}(i, a) + \sum_{l \in \mathcal{S}} p(i, a, l) \max_{b \in \mathcal{A}(l)} Q(l, b) - Q(i^*, a^*)].$$

Step 8. Increment k by 1. If $k < k_{\max}$, set

$$i \leftarrow j$$

and go to Step 2. Otherwise go to Step 9.

Step 9. For each $m \in \mathcal{S}$, select

$$d(m) \in \arg \max_{b \in \mathcal{A}(m)} Q(m, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

It is clear that model-building algorithms require more storage space in comparison to their model-free counterparts. The additional space is of course in the form of the V terms, the W terms, and the \bar{r} terms.

11. Finite Horizon Problems

Finite horizon problems are also called stochastic path problems. Finite horizon problems can be solved using the methods discussed for infinite horizon problems. One would have to associate a Q -factor with the following triple: i, a, s , where i denotes the state, a the action and s the stage. For total reward, one could use value iteration or relative value iteration based schemes. For total discounted reward, one could use Q -Learning. Since the problem solution methodology will be similar in every other manner to what was discussed in the infinite horizon case, we do not pursue this topic any further.

With this, we end our discussion on RL algorithms. In the next section, we will discuss some function approximation strategies that can be used to solve problems with a large number of state-action pairs.

12. Function approximation

A major difficulty associated with the Bellman equation based approaches is the curse of dimensionality. To illustrate it, consider a problem with a million state-action pairs. Using model-free algorithms, one can avoid having to store the huge transition probability matrices. This, however, does not solve the problem completely. One must also find some method to store the one million Q -factors. We will next discuss some strategies to work around this difficulty. We will focus on strategies that seem to have worked in practice. Function approximation can be done in a number of ways, some of which are:

1. State Aggregation,
2. Function Fitting (Neural Networks and Regression), and
3. Function Interpolation.

We will also discuss a number of other related topics in this section. We will begin this section with a discussion on state aggregation.

12.1. Function approximation with state aggregation

State aggregation means lumping several states together. With lumping, one can often obtain a small number of state-action pairs. This means that then one may be able to store each Q -factor individually. When each Q -factor is stored individually, the approach is said to be *look-up table* based. All the algorithms discussed so far in this book were presented in a look-up table format. Thus, after a suitable aggregation of states, one may be able to use the algorithms discussed previously.

When states are lumped together, we usually lose the Markov property, the DP solution is no longer guaranteed to be optimal, and consequently the procedure becomes heuristic. However, empirical evidence tells us that in spite of this, frequently, a DP approach on a lumped state space, despite being a heuristic itself, can outperform other heuristic procedures.

Under some situations, it is possible to combine states without losing the Markov property (see the *lumpability theorem* in Kemeny and Snell [87]), but it cannot be guaranteed that the solution obtained from the lumped (reduced) state space would be identical to the solution obtained without lumping. In other words, even if the Markov property is retained, optimality cannot be guaranteed. And in spite of this, state aggregation, in our opinion, is a robust approach to avoiding the curse of dimensionality. Instead of worrying about technical correctness (such as issues related to the lumpability theorem), here, we will discuss practical issues that seem to have worked for us. In other words, we suggest that you try lumping states in innovative ways, even if you are not meeting all the technical considerations such as those in the lumpability theorem.

A general rule for combining states is to combine states with "similar" characteristics together. For instance, consider the next example.

Example 1. The state is defined by (x, y, z) , where x, y and z are integers in the range $(1, 50)$. Without aggregation, one would have 50^3 states. Let us assume that the attributes x and y are more important than z . Then one way to aggregate is to treat all values of z ranging from 1 to 25 as one value and values of z ranging from 25 to 50 as another. This will lead to $(50)(50)(2)$ states. Of course, doing this makes a lot of sense, if z is really an extraneous descriptor, but not otherwise.

We said above that states with "similar" features may be combined. Well, the natural question is: similar in what respect? In the above example, similar meant similar values (or at least values in a given range) of a state parameter (z in this particular example).

However, "similar" could also mean similar in terms of transition rewards (and may be even probabilities). In our opinion, similarity with respect to transition rewards should be given more importance because our objective is to maximize some reward related objective function. Hence an important point to remember is that states with very *dissimilar* transition rewards should *not* be combined. Since the TRM (and TPM) is unavailable, deciding which states have similar transition rewards may not be an easy task. However, from the nature of the problem, one can often identify the "good" states, which if entered would result in high profits, and the "bad" states, which if entered would result in losses. If it is possible to make such a classification, then aggregation should be done in a manner such that the good and bad states are not mixed together.

Furthermore, sometimes, it is possible to devise a scheme to generate a spectrum over the state space — one end of the spectrum is "good" and the other is "bad." For instance, consider the following example.

Example 2. The state space is defined as in Example 1. Also,

$$s = 2x + 4y + 8z,$$

where large values of s mean good states and small values mean bad states. Then it is perhaps better to use the state space s instead of (x, y, z) . For a real-life example of a similar nature (drawn from the airline industry), see Gosavi [61].

A scheme such as the one shown above would also aggregate states, but not necessarily in a *geometric* sense as suggested in Example 1. In a geometric lumping, we typically lump neighboring states, which may or may not be a good idea because geometric lumping may lump states with dissimilar transition rewards together.

Many so-called encoding schemes that are suggested in the neural network literature actually cause several states to be lumped together. In these cases, the analyst should also see how a lumped state space (using the encoding scheme to lump states together) with a look-up table approach fares in comparison to the neural network approach. We next discuss neural network approaches.

12.2. Function approximation with function fitting

The other popular approach to deal with huge state spaces is to use function-fitting methods such as regression and neural networks. The idea of function fitting has been discussed in Chapter 6.

Conceptually, the idea is store the Q -factors for a given action as a function of the state index. Thus, for instance, imagine that for an MDP with 2 actions in each state, for state $i \in \mathcal{S}$,

$$Q(i, 1) = f(i)$$

and

$$Q(i, 2) = g(i).$$

An example for $f(i)$, assuming i is a scalar, is $f(i) = A + B i + C i^2$. Then instead of storing each Q -factor for action 1 individually, one would need to store only the following values: A , B , and C . So, during the run time of the algorithm, to access the value of a given Q -factor for state i , it will suffice to plug in the value of i in the respective function (f or g). Clearly, this will lead to a considerable reduction in the storage space needed, and technically this would make it possible to store Q -factors for a large state space. It is important to remember that i could be an n -tuple.

12.2.1 Difficulties

What we have stated above is easier said than done. There are four major difficulties associated with this approach.

1. The Q -factors keep changing with every iteration in the look-up table based approach that we have discussed so far. This implies that the scalars (such as A , B , and C) that define the function must also change. Thus, it is not as though, if we somehow obtain the values of these scalars, we can then use them throughout the run time of the algorithm. In fact, we must keep changing them *appropriately* during the run time of the algorithm. In RL, whenever a Q -factor is updated, one more data piece related to the value function (Q -factor function) becomes available. Recall from Chapter 6 that to approximate a function, whose data becomes available piece by piece, one can use incremental regression or incremental neural network methods. However, it is the case that for these methods to work, all the data pieces must be related to the same function. Unfortunately, in RL, the (value)

function (that is the Q -factor function) keeps changing (getting updated) and the data pieces that we get are not really from the same function.

One way around this difficulty is to use small step sizes in updating the Q -factors so that the value function changes *slowly*.

2. When a Q -factor changes in a look-up-table approach, the change is made only to that particular Q -factor. When one uses a function approximation approach, however, change in one Q -factor impacts the values of several Q -factors. Often, it may be difficult to ensure that a change in one Q -factor does not *adversely* affect the values of *other* Q -factors. We refer to this as the “spill-over” effect.

To minimize the damage done by the spill-over effect, one can divide the state space into compartments, with an independent function approximation scheme in each compartment. Then when a change occurs in a Q -factor, the spill-over will be limited to the states *within the compartment*.

3. The third difficulty is that the shape of the value function is never known beforehand. Value functions that are linear can be approximated nicely by neurons. However, when they are non-linear, which is often the case, neurons do not work. It is then that backpropagation (non-linear neural network) is often recommended. This is because the backpropagation method (non-linear neural network) does not need the metamodel of the function. However, backpropagation may get trapped in local optima.

A more robust, although approximate, approach is to assume that the function is piecewise linear and use a linear neuron in each piece. Neurons are not subject to getting trapped in local optima, and many a non-linear function can indeed be approximated by a piece-wise linear function.

4. A fourth issue has to do with the fact that when one uses a function approximation method, it is difficult to show the convergence of the algorithm mathematically. However, see [19], where some in-depth analysis of function-approximation coupled RL algorithms has been done. Although, this issue is often ignored in practice, the fact remains that a great deal still remains to be understood about RL behavior when it is coupled with function approximation. It is important to realize that with a function approximation scheme, a given Q -factor must remain at roughly the same value that the look-up table would have produced. Otherwise, the solution can deviate from the solution generated by a look-up table. The solution of the look-up table, remember, is an optimal or a near-optimal (to be more precise) solution.

12.2.2 Steps in *Q*-Learning coupled with neural networks

Let us next understand the mechanism used to couple a neural network with an RL algorithm. We will use the instance of the *Q*-Learning algorithm (for discounted MDPs) to demonstrate the coupling process. The coupling process will be perfectly analogous for all other algorithms that use *Q*-factors.

In case, the state is defined as an n -tuple, the neural network would need to have n inputs and one additional input for the bias. In the following discussion, although we will refer to the state as i or j , it is important to remember that the state could be defined as an n -tuple.

It is preferable in practice to use a separate neural network *for each action*. Also, usually, an incremental style of updating is used since the information for the *Q*-factor becomes available in a gradual manner — that is, one data piece at a time. The reader should review the material related to neural networks from Chapter 6 at this stage.

Step 1. Initialize the weights of the neural network for any given action to small random numbers. Initialize the corresponding weights of all the other neural networks to identical numbers. (This ensures that all the *Q*-factors for a given state have the same value initially.)

Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set A , the step size constant, to a positive number less than 1. Start system simulation at any arbitrary state.

Step 2. Let the state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1. Calculate α using k as discussed previously.

Step 4. Determine the output of the neural network for i and a . Let the output be q . Also find, using state j as the input, the outputs of all the neural networks of the actions allowed in state j . Call the maximum of those outputs — q_{next} .

Step 4a. Update q as follows.

$$q \leftarrow (1 - \alpha)q + \alpha[r(i, a, j) + \lambda q_{next}].$$

Step 4b. Then use q to update the neural network (that is, update the weights of the neural network) associated with action a . One must use an incremental style neural network updating rule. The new data piece would have q as the function value and i as the input.

Step 5. If $k < k_{max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. The policy learned is stored in the weights of the neural network. To determine the action associated with a state, find the outputs of the neural networks associated with the actions that are allowed in that state. The action(s) with the maximum output is the action dictated by the policy learned. Stop.

The neural network associated with action a is identified in Step 4b, and then the weights of the associated network are updated in an incremental style. Sometimes in practice, just one (or two) iteration(s) of training is sufficient in the updating process of the neural network. In other words, when a Q -factor is to be updated, only one iteration of updating is done within the neural network. Usually doing too many iterations can lead to “over-fitting.” Over-fitting implies that the Q -factor values in parts of the state space other than those being trained are incorrectly updated. It is important that training be localized and limited to the state in question. Some amount of spill-over to neighboring states cannot be avoided; however it should be minimized.

Function approximation using neural networks or regression has not always worked well in a number of well-publicized experiments in RL (see Boyan and Moore [30]). This remains an area that needs a great deal of more research. Some of the main difficulties were outlined in the Section 12.2.1.

12.3. Function approximation with interpolation methods

Interpolation is a widely-studied subject in numerical mathematics and in the rapidly-developing area called *data mining*. Interpolation is known to be a robust method of approximating a function. Interpolation methods can be used effectively for function approximation purposes in RL.

The idea is to store a few *representative* Q -factors; all other Q -factors are determined using interpolation techniques — interpolation of all or a subset of the representative Q -factors. The choice of subset used in interpolation may be based on the location of the “query,” that is, the point at which the updating is to be performed. We next discuss two methods for interpolation.

1. k -nearest-neighbors: The subset of Q -factors used in this scheme contains k nearest neighbors of the query point — that is, the first k representatives, if all representatives are arranged in the descending order of their distances from the query points. If the state is an n -tuple, the distance can be calculated in either a Euclidean or a Manhattan (rectilinear) style. The Euclidean and Manhattan distance metrics between two points (x_1, y_1) and (x_2, y_2) are defined,

respectively, as:

$$d_{\text{Euclidean}} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

and

$$d_{\text{Manhattan}} = |x_1 - x_2| + |y_1 - y_2|.$$

Note that here (x_i, y_i) denotes the state i . The value of k depends on the level of interpolation desired. It could range from 1 to $N - 1$, where N denotes the number of representatives.

The values of the k nearest neighbors can then be *averaged*, or else one can perform *regression* (linear or non-linear) to obtain the value of the Q -factor at the query point.

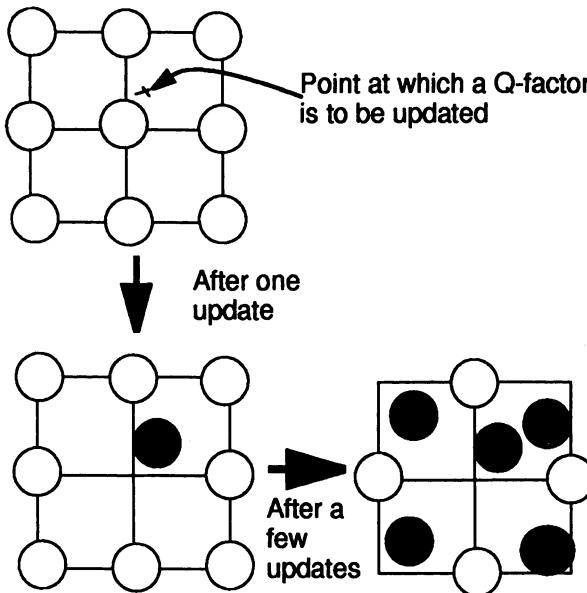


Figure 9.5. Function approximation using interpolation: The circles denote the representative Q -factors. Notice how the locations of the representatives change with updating. The darker circles indicate the *updated* Q -factors. When a Q -factor at a location, where no Q -factor exists, has to be updated, its nearest neighbor is *eliminated*. It is not necessary to start out with uniformly-spaced Q -factors as shown in this figure.

2. Kernel methods: Kernel methods apply a weight to the value of each representative, and then compute a *weighted average* at the query point. Then if x_0 denotes the query point, and Q_{x_i} denotes the Q -factor value for state x_i , then the Q -factor value at the query point is:

$$Q_{x_0} = \frac{\sum_{i=1}^N w_{x_i, x_0} Q_{x_i}}{\sum_{i=1}^N w_{x_i, x_0}}, \quad (9.21)$$

where w_{x_i, x_0} denotes the weight associated with the i th representative. One way to define the weight is:

$$w_{x_i, x_0} = D\left(\frac{|x_0 - x_i|}{\omega}\right),$$

where ω is the so-called *smoothing* parameter, and

$$D(s) = 3/4(1 - s^2) \quad \text{if } |s| \leq 1;$$

and

$$D(s) = 0 \quad \text{otherwise.}$$

The above rule for computing the weights is the so-called Epanechnikov quadratic kernel [69]. This ensures that the fitted function is *continuous*. More importantly, the rule ensures that weights decay (die off) with increasing distance from the query point and are 0 beyond a certain radius of the query point. The smoothing parameter, ω , can be set to a value such as 0.1; but do note that its value in some sense dictates the radius beyond which weights are 0, and should be selected carefully. More sophisticated kernel methods, that actually do regression with the weighted data, are discussed in [69].

Both kernel methods and nearest-neighbor methods require substantial memory, and also a great deal of computation. Memory is needed to store the representatives. In the nearest-neighbor approach the computation comes in the form of having to *find* the nearest neighbors. In the kernel method, one has to compute sums over large number of quantities — see Equation (9.21).

So now, we know how to find the value of the Q -factor to be updated. Then with the RL scheme, we can update the Q -factor.

The question, then, is: what is to be done with the updated Q -factor? Do we just *add* the updated Q -factor to the data structure that holds (stores) all the representatives?

The answer is: some other Q -factor — usually, its *nearest* representative (neighbor) — is *replaced* by the updated Q -factor. (See Figure 9.5 for a pictorial demonstration of this process.) Unless replacement of this nature is performed, the number of representatives can grow without bound, while, as stated above, the computer's memory is limited. A program that requires very large memory tends to run slowly.

The issue discussed above is critical in the context of interpolation methods. Remember in function-fitting methods, this difficulty does not arise.

“Binary trees” are data structures that help store large amount of data in a compact manner. More importantly, under certain conditions, data can be efficiently stored and retrieved from a binary tree. The latter can be used to store the representative Q -factors in interpolation methods. Although arrays too can be used for the purpose of storage, identifying the nearest neighbors

can be quite difficult with an array. If an array is used, one must use clever search methods to identify the nearest neighbors. A naive use of arrays can slow down the program considerably.

The conditions under which storage and retrieval of data with a binary tree become efficient are met when the states are *not* visited sequentially by the algorithm. Fortunately, in simulators, we do not visit states sequentially. If states are visited sequentially, however, function interpolation can become very time consuming.

Sometimes, only a small subset of the state space is actually visited within the simulator. Consider this: a state space is defined by (x, y, z) where each of x, y , and z is an integer ranging from (49, 250). Theoretically, the state space is 200^3 . However, in practice, let us assume that the number of states visited is much less — say 5000. A look-up table should be sufficient for this problem. And yet defining an array such as $Q[x][y][z]$ will not work here since the array will be too large. A trick that can be used in this scenario is to store the Q -factors in binary trees. In a binary tree the memory used by the computer increases dynamically during the run time of the program. (A binary tree uses linked lists [66]).

We next present the steps to be followed in an RL algorithm that uses function interpolation. The algorithm in these steps is the Q -Learning algorithm for discounted reward. For any other algorithm, the steps are perfectly analogous.

Step 1. Randomly generate a large number — N — of Q -factors in the state space. The value of N depends on the memory that can be used without slowing down the program. Set k , the number of jumps of the Markov chain, to 0. We will run the algorithm for k_{\max} iterations, where k_{\max} is chosen to be a sufficiently large number. Set A , the step size constant, to a positive number less than 1. Start system simulation at any arbitrary state.

Step 2. Let the current state be i . Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a . Let the next state be j . Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a . The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1. Calculate α using k .

Step 4. Search for the Q -factor of i, a from the binary tree. This may require finding k of its nearest neighbors. Then by averaging or regression, determine the value of the Q -factor of i, a . Let us denote this value by q . Also search for and determine the values of all Q -factors for state j . Denote the maximum of all the Q -factors for state j by q_{next} .

Step 4a. Update q as follows.

$$q \leftarrow (1 - \alpha)q + \alpha[r(i, a, j) + \lambda q_{\text{next}}].$$

Step 4b. Then use q to update the binary tree as follows: Replace the *nearest neighbor* of the updated Q -factor by the updated Q -factor.

Step 5. If $k < k_{max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. To determine the action associated with a state, find the Q -factors associated with the actions that are allowed in that state. The action(s) with the maximum Q -factor is the action dictated by the policy learned. Stop.

12.4. Linear and non-linear functions

If the value function is linear or approximately linear, it may be approximated either by:

1. incremental linear regression or
2. a neuron.

Approximating a non-linear value function (that is, the Q -factor function in RL) with function approximation can be done in three ways:

1. Using *one* neural network over the entire state space. Since the value function is non-linear over the state space, *backpropagation* can be used.
2. Using a piecewise linear function to approximate the state space. In each linear piece, one can use a *neuron*.
3. Using a piecewise linear function to approximate the state space. In each linear piece, one can use *incremental linear regression*.

12.5. A robust strategy

From our limited experience, we have found the following strategy to be robust.

Step 1. First, the important elements of the n -tuple, of which the state is made up, should be identified. For instance, the state may be a 3-tuple such as (x, y, t) . It is entirely possible that the state may be adequately described by a subset of all these elements. An example (subset) is (x, t) . Also, sometimes, it is possible to convert all these elements into one element using a scheme of the following type:

$$o = f(x, y, t).$$

Identifying the important elements and/or a suitable **encoding** function f requires some experimentation with various schemes.

Step 2. Divide the (encoded) state space into compartments, and use a neuron in each compartment. (See Figure 9.6.) This reduces the damage done by the spill-over effect, and at the same time this serves as a non-linear function approximator. It is the case that the linear neuron, unlike backpropagation, is immune to the disease of getting trapped in local optima, and is hence preferable.

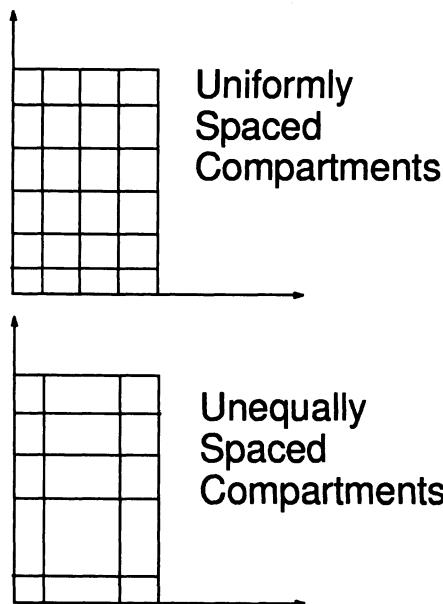


Figure 9.6. Schematics showing a 2-dimensional state space. The top schematic shows equal sized compartments and the bottom one shows an unequal sized compartments. Within each compartment, a neuron may be placed.

12.6. Function approximation: Model-building algorithms

Bayesian networks can sometimes be used to build the model in a compact form. Read Tadepalli and Ok [167] for more on this. When state aggregation is done, using model-building algorithms is a feasible approach. Model-building algorithms, usually, require a large amount of computation per iteration (due to the fact in every update one has to compute sums over the entire state space), but may need fewer iterations. Building compact representations of the model is an important and attractive area of research that needs to be pursued.

13. Conclusions

This chapter was meant to serve as an introduction to the fundamental ideas related to RL. Many RL algorithms based on Q -factors were discussed. Their

DP roots were exposed and step-by-step details of some algorithms were presented. Some methods of function approximation, for the value function, were discussed. Brief accounts of actor-critic algorithms, model-building algorithms, and finite horizon problems were also presented.

14. Bibliographic Remarks

In what follows, we have attempted to cite some important references related to research in RL. In spite of our best efforts, we fear that this survey is incomplete in many respects, and we would like to apologize to the researchers whom we have not been able acknowledge below.

14.1. Early works

Watkins [182], Rummery and Niranjan [150], and Sutton [165, 166, 163] are some of the initial works in RL. However, the history can be traced to work before these. The idea of learning can be traced to the work of Samuel [151] (1959) and Klopff [94] (1972). Holland [77] is also a work related to temporal differences. Some other related research can be found in Holland [78] and Booker [23].

14.2. Neuro-Dynamic Programming

Bertsekas and Tsitsiklis, via an outstanding book [19], strengthened the connection between RL and DP. For advanced topics in RL, this book is strongly recommended. The book not only discusses a treasure of algorithmic concepts that are likely to stimulate further research in RL in the coming years, but also presents a detailed convergence analysis of many RL algorithms. They have used the name Neuro-Dynamic Programming to refer to RL.

14.3. RL algorithms based on Q -factors

The Q -Learning algorithm for discounted reward MDPs is due to Watkins [182] (1989). This work appears to have established the link between DP and RL for the first time. However, Werbös [184] seems to have argued for this in an interesting earlier work in 1987.

The Q - P -Learning algorithm for discounted reward is due to Rummery and Niranjan [150]. They called it modified Q -learning — perhaps because it is based on modified policy iteration. Their algorithm is also popularly known as SARSA [164]. The Relative Q -Learning algorithm for average reward can be found in Abounadi [1], and in Abounadi, Bertsekas, and Borkar [2], among other places. The Q - P -Learning algorithm for average reward MDPs, which uses relative value iteration in the policy evaluation phase, is new material. The same algorithm for SMDPs is from Gosavi [61]. The notion of visit factors is

also new material. The Q -Learning algorithm for discounted reward SMDPs is due to Bradtko and Duff [31].

SMART has appeared in Das, Gosavi, Mahadevan and Marchalleck [37]; Relaxed-SMART is described in Gosavi [63] (also see Gosavi [59]). R -Learning is due to Schwartz [155]. For a nice summary of early average reward RL algorithms, the reader should read Mahadevan [109]. A comprehensive overview of early RL algorithms can be found in Kaelbling, Littman, and Moore [85].

14.4. Actor-critic Algorithms

Actor-critic algorithms were studied in [189], and [12]. These algorithms were rigorously analyzed — for the first time — by Konda and Borkar [96]. They have discussed some discounted reward (asynchronous) and some average reward (synchronous) algorithms in their paper.

Actor-critic algorithms, in spite of their dramatic name, have received less research attention in comparison to other RL algorithms. Recent analytical work of Konda and Borkar should pave the way for a clearer understanding of these algorithms. In our opinion, the actor-critic algorithms have the flavor of learning automata algorithms (that we will discuss in the next chapter) and hence also of game theory — a connection that also needs to be studied in greater depth and perhaps exploited to develop more powerful actor-critic algorithms.

14.5. Model-building algorithms

A model-building (model-based) algorithm for discounted reward and average reward can be found in [11] and Tadepalli and Ok [167], respectively. The algorithm in the latter reference is called H -Learning. In this book, algorithms that estimate the value function, directly, within a simulator were referred to as H -Learning algorithms. The model-building algorithms of the Q -Learning variety (which learn the Q -factors and not the value function) that were presented in this book are new material.

At this point, it is not clear what the future holds for model-building algorithms. Model-free algorithms seem to be more popular, especially on large-scale implementations, but research in future may change all that. In our opinion, compared to their model-free counterparts, model-building algorithms make *more* use of the information available in a simulator (since they also build the model). Hence coupled either with robust state-aggregation schemes or compact methods to represent large models, they have the potential to outperform model-free algorithms. On those MDPs in which the effect of an action is felt in a great distance in the trajectory may be candidates for applying model-building algorithms. Much research remains to be done in this area.

14.6. Function Approximation

For an excellent discussion on function approximation schemes, including nearest neighbor methods, kernel methods, and neural networks, read the books of Hastie, Tibshirani and Friedman [69] and Devroye, Gyorfi and Lugosi [41]. In the context of reinforcement learning, function approximation based on neural networks requires encoding. Some of the pioneering references in this area are: Hinton [74] (coarse coding), Albus [4] (CMAC coding), and Kanerva [86] (kanerva coding). For function approximation via regression and interpolation see: Boyan and Moore [30] and Davies [40].

A number of papers related to function approximation with classical DP have been written. Some of them are: Bellman and Dreyfus [16], Turgeon [177], Johnson *et al.* [84], and Philbrick and Kitanidis [132].

14.7. Some other references

Some other works related to RL in general are Kumar [98], Kumar and Varaiya [99], and of course all the literature related to learning automata (see Narendra and Thathachar [120] for textbook treatment and the next chapter). Another important work is that of Jalali and Ferguson [83]. We should also mention research related to some *multi-step* extensions of value iteration algorithms: Watkins [182], Peng and Williams [128] (both discuss extensions of *Q*-Learning) and Gosavi, Bandla, and Das [64] (extension of SMART). Finally, we must mention the pioneering early work of Robbins and Monroe [142] whose algorithm forms the foundation stone of RL. Its use in DP related areas was to come much later though.

14.8. Further reading

For further reading on RL, we would like to give some friendly advice. It is very easy to get lost in reading this literature because of the plethora of terms and acronyms used, and hence one should constantly remind oneself of the inviolability of the fundamentals of DP. It will stand the reader in good stead to remember that almost every RL algorithm has (or at least should have) roots in classical DP — regardless of the name of the algorithm.

15. Review Questions

1. The following values were determined for the *Q*-factors of the three state-action pairs in an MDP — using a look-up-table-based RL algorithm. $Q(i, a)$ denotes the *Q*-factor for state *i* and action *a*.

$$Q(1, 1) = 34.9, Q(1, 2) = 56.8, Q(2, 1) = 12.98, Q(2, 2) = -68.4,$$

$$Q(3, 1) = 23.78, \text{ and } Q(3, 2) = 20.5.$$

Determine the policy defined by these Q -factors. Also determine the value function (for each state) associated with this policy. When a crude function-approximation scheme was used on the same MPD, the Q -factors were determined to be:

$$Q(1, 1) = 30.9, Q(1, 2) = 57.8, Q(2, 1) = 14.78, Q(2, 2) = -59.5,$$

$$Q(3, 1) = 19.87, \text{ and } Q(3, 2) = 21.5.$$

What is the policy generated when the algorithm is used with the function-approximation scheme? Is this policy identical to that generated by the look-up table?

2. Consider Problem 9 from Chapter 8. Use a Q -learning algorithm on this problem, and show a few steps in the calculation. The objective function is average cost.
3. Consider Problem 10 from Chapter 8. Use a Q -learning algorithm on this problem, and show a few steps in the calculation. Use discounted cost or average cost as the performance metric.
4. Consider a robot that has been assigned the task of moving from a *starting point* to a *destination*. See Figure 9.7 (page 275). Squares which have obstacles in them are costly. The cost of moving into any given square is provided in the figure. (Blank squares have a zero cost.) The robot can move *east*, *west*, *north*, or *south*. In other words, it has four actions in most states. Assume that there is a wall surrounding the grid shown in the figure. The robot is not allowed to bump into a wall by its vision sensors. Assume that when the robot selects an action, it succeeds with a probability of 0.8 and with the remaining probability it goes in the opposite direction. In one step, it can jump only one square. Set this up as an MDP in which the state is determined by the current position of the robot and the objective is to reach the destination minimizing the average cost per step. Show the first few steps in a Q -learning algorithm to solve the problem.
5. Compute the mean of the following numbers using the Robbins-Monro algorithms.

$$12, 11.9, 14, 12.6, 13.0, 11.9, 13.2.$$

- a) Use $\alpha = 1/n$ and $b)\alpha = 0.1/n$. Compare the estimates obtained in part a and part b. Also compare these values to the actual sample mean.
6. Identify 3 nearest neighbors for $(5, 7, 11)$, form the following points, using Euclidean distance.

$$(1, 8, 11), (2, 6, 14), (12, 6, 15), (3, 12, 10), (1, 7, 15), (2, 14, 11), (3, 4, 12).$$

Do the same using Manhattan distance. The latter, between two points (x, y) and (a, b) is defined as:

$$|x - a| + |y - b|.$$

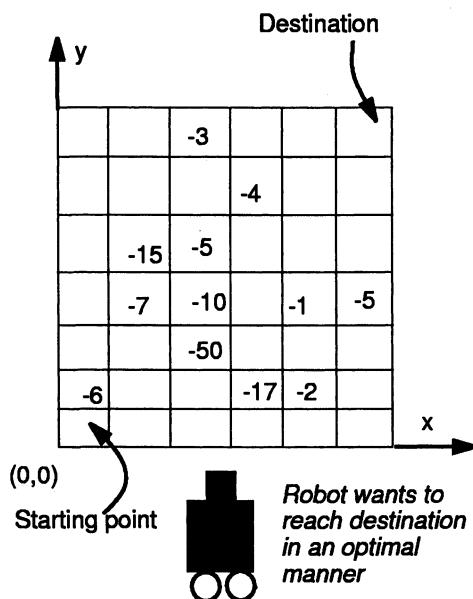


Figure 9.7. A Grid World

Chapter 10

CONTROL OPTIMIZATION WITH LEARNING AUTOMATA

If a man does not keep pace with his companion, perhaps it is because he hears a different drummer. Let him step to the music which he hears, however measured and far away.

— H.D. Thoreau (1817-1862)

1. Chapter Overview

In this chapter, we will discuss an alternative to Reinforcement Learning for solving Markov decision problems (MDPs) and Semi-Markov decision problems (SMDPs). The methodology that we will discuss in this chapter is generally referred to as Learning Automata. We have already discussed the theory of learning automata in the context of parametric optimization. It turns out that in control optimization too, in particular for solving problems modeled with Markov chains, learning automata methods can be useful.

We will refer to the theory of learning automata in the context of Markov chain related control optimization as **Markov Chain Automata Theory (MCAT)**. At the very outset, we must mention that much of the work related to MCAT has stemmed from the pioneering paper of Wheeler and Narendra [185]. MCAT is different from the learning automata theory discussed in Chapter 7, although they have some common roots.

For a description of MDPs and SMDPs, the reader should read Chapter 8. This chapter will discuss an MCAT algorithm for the SMDP. The MDP will be treated as a special case of the SMDP.

2. The MCAT framework

The MCAT framework for solving MDPs is very distinct from that of dynamic programming. It is rooted in the principles of game theory. We will not explore these roots here; we will stick to an intuitive explanation.

Like RL (see Chapter 9), the MCAT framework can be used in a simulator. The origins of the MCAT framework are in the literature on “adaptive learning.” Some references on the latter are: Jalali and Ferguson [83] and Borkar and Varaiya [24]. We must note that the literature on adaptive learning was indeed meant for on-line learning. We will use the MCAT framework in a simulation environment, where there is no real “learning.” In other words, an MCAT algorithm can be viewed as any other optimization algorithm that just happens to run in a simulator. (Note that we have advocated this viewpoint in the chapter on Reinforcement Learning — Chapter 9.)

2.1. The working mechanism of MCAT

The reader is advised to read Chapter 8 and should have a clear notion of what an MDP or an SMDP constitutes. As mentioned above, MCAT provides a simulation-based methodology used to solve MDPs and SMDPs; furthermore, it does not use the dynamic programming framework, unlike reinforcement learning.

The MCAT algorithm that we will focus on, like any other learning automata algorithm, uses a set of probabilities. A probability is associated with each state-action pair (i, a) . This is the probability of selecting action a in state i in the simulator. Initially, this probability is same for every action. For instance, if 5 actions are allowed in a given state, the probability of selecting any of those actions in that state will initially be $1/5$. The MCAT algorithm, through its interactions with the simulator, updates these probabilities till the optimal action(s) has the highest probability. This has to be true for each state.

We will need some notation to be able to discuss the MCAT framework. Let $\mathcal{A}(i)$ denote the set of actions available at state i . Hence the union of $\mathcal{A}(i)$ over i gives the action space. With every state-action pair, we associate a probability $p(i, a)$ of selecting action a in state i . The set of states is denoted by \mathcal{S} . Obviously,

$$\sum_{a \in \mathcal{A}(i)} p(i, a) = 1 \quad i \in \mathcal{S}.$$

As mentioned above, in the beginning of the learning process, each action is equally likely. Hence, for each $i \in \mathcal{S}$, and each $a \in \mathcal{A}(i)$, in the beginning,

$$p(i, a) = \frac{1}{|\mathcal{A}(i)|},$$

where $|\mathcal{A}(i)|$ denotes the number of elements in the set $\mathcal{A}(i)$, that is, the number of actions allowed in state i .

The updating process employs the following scheme, which is intuitively appealing. The algorithm is run within the simulator. The simulator simulates a trajectory of states. In each of those states, actions are selected based on the probabilities. If the performance of an action is considered to be *good* (based on the costs or rewards generated), the probability of that particular action is *increased*. For the same reason, if the performance is considered to be *weak*, the probability is *reduced*. Of course, this “updating” scheme must always ensure that the sum of the probabilities of all the actions in a given state is 1. Eventually, the probability of the optimal action converges to 1, while that of each of the other actions converges to 0. If more than one action is optimal — say there are k optimal actions — the probability of each of the optimal actions converges to $1/k$.

We next describe the feedback mechanism in some detail. In the simulator, every state is visited repeatedly. Assume that a state i in the system has been revisited. We now measure the average reward earned in the system since its last visit to state i . This average reward is calculated by the total reward earned since the last visit to that state divided by the number of state transitions since the last visit to that state. This average reward is called the **response** of the system to the action selected in the last visit to state i . Thus, if state i is under consideration, the response $s(i)$ in the MDP is

$$s(i) = \frac{R(i)}{N(i)},$$

where $R(i)$ denotes the **total** reward earned since the last visit to state i and $N(i)$ denotes the number of state transitions that have taken place since the last visit to i . A transition, here, refers to a transition from one *decision-making* state to another. For the SMDP, $N(i)$ is replaced by $T(i)$. The latter denotes the total time spent in the simulator since the last visit to i . Thus, the expression for response in the SMDP becomes:

$$s(i) = \frac{R(i)}{T(i)}.$$

The response is then **normalized** to ensure that it lies between 0 and 1. The normalized response is called **feedback**. The normalization is performed via the following relationship:

$$\beta(i) = \frac{s(i) - s_{\min}}{s_{\max} - s_{\min}},$$

where s_{\min} is the *minimum* response possible in the system and s_{\max} is the *maximum* response possible in the system. The response, here, should be un-

derstood as the average reward of a policy. (Average reward and policy are defined in Chapter 8.)

The feedback is used to update $p(i, a)$, that is, the probability of selecting action a in state i . Unless normalization is done, the probabilities can exceed 1 or become negative. The conversion of the response to feedback is a research topic by itself; more on this can be found in Narendra and Thathachar [120].

Many schemes have been suggested in the literature to update these probabilities. All schemes are designed to punish the bad actions and reward the good ones. A popular scheme, known as the Reward-Inaction scheme, is described next in the context of two actions. The multiple-action scenario is discussed in Section 2.4. We discuss the Reward-Inaction scheme in the next paragraph.

Reward-Inaction Scheme . Assume that the system visits a new state i . Then, the probabilities of the actions allowed in state i are first updated. To this end, the feedback $\beta(i)$ has to be computed. The formula for $\beta(i)$ was given above. Let $D(i)$ denote the action taken in state i in its last visit and η denote the learning rate . Then, according to the Reward-Inaction scheme, the probability $p(i, a)$ will have to be updated via the rule given below:

$$p(i, a) \leftarrow p(i, a) + \eta\beta(i)I[D(i) = a] - \eta\beta(i)p(i, a), \quad (10.1)$$

where $I[.]$ equals 1 if the condition inside the brackets is satisfied and $I[.] = 0$ otherwise.

The reason why the reward-inaction scheme is so named is: a good action automatically has a high value for β , and therefore the scheme increases the probability of that action (carefully read line (10.1)). An action that produces a poor reward has a low value for β , and therefore the probabilities are not changed significantly (again carefully read line (10.1)). Thus a reward is “acted upon,” while a weak response is disregarded (read inaction). That the scheme will eventually converge to an optimal solution has been established in [185].

Next, we supply step-by-step details of an MCAT algorithm. In Section 2.3, we will illustrate the use of this algorithm on a simple 3-state SMDP.

2.2. Step-by-step details of an MCAT algorithm

Before presenting the steps, we would like to make the following comments.

- Running an MCAT algorithm requires the knowledge of s_{\max} and s_{\min} , which can be often obtained from the values of the cost elements in the problem. Sometimes, the cost structure in the problem permits an extreme cost analysis thereby allowing upper and lower bounds to be constructed on the average reward. On many problems, heuristics are available. The average reward of the heuristic can then be used to deduce values for these quantities. For instance, k times the average reward of the heuristic could be set as s_{\max} , where $k > 1$.

- The description given below pertains to SMDPs. The MDP version can be obtained by setting time spent in each transition to 1.
- Terminating the algorithm is an important issue. You can run the algorithm till all probabilities have converged to 0 or 1. This can take a very long time. The second approach is to run the algorithm for a large number of iterations. The third and the more practical approach is to run the algorithm till one of the probabilities in each state exceeds a pre-fixed value such as 0.9. This action is then declared to be the optimal action for that state.
- In MCAT, the learning rate is fixed and need not be changed with iterations.
- The algorithm description presented below assumes that there are 2 possible actions in each state. We will discuss a generalization to multiple actions later.

Steps. Set iteration count $m = 0$. Initialize action probabilities $p(i, a) = \frac{1}{|\mathcal{A}(i)|}$ for all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$. Set the cumulative reward $C_r(i) = 0$ and the cumulative time $C_t(i) = 0$ for all $i \in \mathcal{S}$. Also, initialize to 0 the following quantities: the total reward earned in system from the start, R_{current} , and the simulation clock, T_{current} . Initialize s_{\min} and s_{\max} as described above. Set, for every $i \in \mathcal{S}$, $u(i)$ to any action that is allowed in state i . Start system simulation. Let the starting state be denoted by i .

1. If this is the first visit to state i , go to Step 3. Otherwise, set in the order given below:

$$R(i) \leftarrow R_{\text{current}} - C_r(i),$$

$$T(i) \leftarrow T_{\text{current}} - C_t(i),$$

$$s(i) \leftarrow \frac{R(i)}{T(i)},$$

and finally

$$\beta(i) \leftarrow \frac{s(i) - s_{\min}}{s_{\max} - s_{\min}}.$$

2. Let $D(i)$ denote the action taken in the last visit to i . Update $p(i, u(i))$ using:

$$p(i, u(i)) \leftarrow p(i, u(i)) + \eta \beta(i) I[D(i) = u(i)] - \eta \beta(i) p(i, u(i)),$$

where $I[.] = 1$ if the condition within brackets is satisfied and 0 otherwise. Then, update the other action so that sum of the probabilities of actions for state i is 1.

3. With probability $p(i, a)$, select an action a from the set $\mathcal{A}(i)$.
4. Set $D(i) \leftarrow a$, $C_r(i) \leftarrow R_{\text{current}}$, and $C_t(i) \leftarrow T_{\text{current}}$.

Simulate action a . Let the system state at the next decision epoch be j . Also let $t(i, a, j)$ denote the transition time, and $g(i, a, j)$ denote the immediate reward earned in the transition resulting from selecting action a in state i .

5. Set

$$R_{\text{current}} \leftarrow R_{\text{current}} + g(i, a, j),$$

and

$$T_{\text{current}} \leftarrow T_{\text{current}} + t(i, a, j).$$

6. Set $i \leftarrow j$ and $m \leftarrow m + 1$. If $m < \text{MAX_STEPS}$, return to Step 1; otherwise STOP.

2.3. An illustrative 3-state example

In this section, we will show how the MCAT algorithm, discussed above, works on a simple 3-state SMDP. The example comes from Gosavi *et al.* [65]. For the sake of simplicity, we will first assume that each state has two possible actions.

Obviously, we will not require the values of the transition probabilities of the underlying Markov chain; a simulator of the system will be sufficient. Using the simulator, a trajectory of states is generated, and updating is performed for a state, when the simulator visits it. Let the first few states in the trajectory be defined as:

$$2 \rightarrow 1 \rightarrow 3 \rightarrow 1 \rightarrow 2.$$

We will show how updating is done for these states. The general scheme of updating is summarized in the next paragraph.

In the beginning, all the action probabilities are supposed to be equal. Since there are two actions in each state, we set

$$p(1, 1) = p(1, 2) = p(2, 1) = p(2, 2) = p(3, 1) = p(3, 2) = 0.5.$$

With a probability of $p(i, m)$, the m th action is chosen in the current state, i , where $m \in \{1, 2\}$. The updating required by the algorithm is performed (within the simulator), and the action chosen is simulated. The next state is determined by the random variables that govern the behavior of this system. In other words, the next state is determined by the simulator. Again, an action is chosen in the next state, some quantities are updated, and we move on to the next state. This continues till the termination criterion for the algorithm is met.

We begin by setting $C_r(i) = 0$ and $C_t(i) = 0$, $\forall i$. We also set R_{current} and T_{current} to 0. Also, we will assume that $u(i) = 1$, $\forall i$, $s_{\max} = 10$, $s_{\min} = -5$, and $\eta = 0.1$. The updating calculations performed in each state are listed below.

- **State 2:** This is the first visit to state 2. Hence the action probabilities are not updated. Let the action selected, a , be 1. Then, following Step 4, we set $D(2) = 1$, $C_r(2) \leftarrow R_{current}(= 0)$ and $C_t(2) \leftarrow T_{current}(= 0)$. The next state is 1. Here $g(i, a, j) = 4.5$ and $t(i, a, j) = 2.34$ for $i = 2, j = 1$. (Notice that both values: 4.5 and 2.34 are generated by the simulator.) Next, following Step 6, we update $R_{current}$ and $T_{current}$. New values for these variables are: $R_{current} = 4.5$ and $T_{current} = 2.34$. We are now in state 1.
- **State 1:** This is the first visit to state 1 and the calculations performed will be similar to those shown above. Let the action selected be 2. So, following Step 4, we set $D(1) = 2$, $C_r(1) \leftarrow R_{current}(= 4.5)$ and $C_t(1) \leftarrow T_{current}(= 2.34)$. The next state is 3. From the simulator, $g(i, a, j) = 3.5$, and $t(i, a, j) = 0.11$ for $i = 1, j = 3$. Following Step 6, next we must update $R_{current}$ and $T_{current}$. The new values for these variables are: $R_{current} = 4.5 + 3.5 = 8$ and $T_{current} = 2.34 + 0.11 = 2.45$. We are now in state 3.
- **State 3:** This is the first visit to state 3 and once again the calculations performed will be similar to those shown above. Let the action selected be 2. Again,, following Step 4, we set $D(3) = 2$, $C_r(3) \leftarrow R_{current}(= 8)$ and $C_t(3) \leftarrow T_{current}(= 2.45)$. The next state is 1. From the simulator, $g(i, a, j) = -1$, and $t(i, a, j) = 1.55$ for $i = 3, j = 1$. We next update $R_{current}$ and $T_{current}$ following Step 6. The new values for these variables are: $R_{current} = 8 - 1 = 7$ and $T_{current} = 2.45 + 1.55 = 4$. We are now in state 1.
- **State 1 (again):** This is a re-visit (second visit) to state 1. Therefore, we first need to execute Step 1. We set

$$R(1) = R_{current} - C_r(1) = 7 - 4.5 = 3.5,$$

$$T(1) = T_{current} - C_t(1) = 4 - 2.34 = 1.64,$$

$$s(1) = R(1)/T(1) = 3.5/1.64 = 2.134,$$

and

$$\beta(1) = \frac{s(1) - s_{\min}}{s_{\max} - s_{\min}} = \frac{2.134 + 5}{10 + 5} = 0.4756.$$

Now, $D(1) = 2$ and so $D(1) \neq u(1) = 1$. Hence we update $p(1, 1)$ as follows:

$$p(1, 1) \leftarrow p(1, 1) - \eta \beta(1) p(1, 1) = 0.5 - 0.1(0.4756)(0.5) = 0.47622$$

Let the action selected in this state be 1. Therefore, $D(1) = 1$, $C_r(1) = 7$, and $C_t(1) = 4$. From the simulator, $g(i, a, j) = 2.4$, and $t(i, a, j) = 1.23$,

for $i = 3, j = 1$. We next update R_{current} and T_{current} following Step 6. The new values for these variables are: $R_{\text{current}} = 7 + 2.4 = 9.4$ and $T_{\text{current}} = 4 + 1.23 = 5.23$. We are now in state 2.

- **State 2 (again):** This is a re-visit (second visit) to state 2. Hence we first execute Step 1. We set

$$R(2) = R_{\text{current}} - C_r(2) = 9.4 - 0 = 9.4,$$

$$T(2) = T_{\text{current}} - C_t(2) = 5.23 - 0 = 5.23,$$

$$s(2) = R(2)/T(2) = 9.4/5.23 = 1.7973,$$

and

$$\beta(2) = \frac{s(2) - s_{\min}}{s_{\max} - s_{\min}} = \frac{1.7973 + 5}{10 + 5} = 0.4531.$$

Now, $D(2) = 1$ and so $D(2) = u(2) = 1$. Hence we update $p(2, 1)$ as follows:

$$p(2, 1) \leftarrow p(2, 1) + \eta \beta(2)(1 - p(2, 1)) = 0.5 + 0.1 \cdot 0.4531(1 - 0.5) = 0.5226.$$

Let the action selected in this state be 2. Then, $A(2) = 2$, $C_r(2) = 9.4$, and $C_t(2) = 5.23$. Let the next state be 3. From the simulator, $g(i, a, j) = -1.9$, and $t(i, a, j) = 4.8$, where $i = 2, j = 3$. We next update R_{current} and T_{current} following Step 6. The new values for these variables are: $R_{\text{current}} = 9.4 - 1.9 = 7.5$ and $T_{\text{current}} = 5.23 + 4.8 = 10.03$.

Updating will continue in this fashion for a large number of iterations. The most likely action (the action with the largest probability) in each state will be considered to be the best action for that state.

2.4. What if there are more than two actions?

When more than two actions are allowed in each state, the updating scheme for actions other than $u(i)$ will have to be specified. Note that so far we have assumed that only two actions are allowed in each state.

A number of ways have been suggested in the literature to address this issue. We will describe one approach below.

If state i is the current state, update $p(i, u(i))$ as described earlier for the case with 2 actions. Now, let us denote the change in the probability, $p(i, u(i))$, by Δ , which implies that:

$$p(i, u(i)) \leftarrow p(i, u(i)) + \Delta.$$

Clearly, Δ may be a positive or a negative quantity. Now, if the total number of actions in state i is k , then update all probabilities other than $p(i, u(i))$ using the following rule:

$$p(i, a) \leftarrow p(i, a) - \frac{\Delta}{k - 1},$$

where $a \neq u(i)$. Of course, the objective here is to ensure that the sum of the probabilities equals 1. Unfortunately, this penalizes or rewards several actions with the *same quantity*. This is a limitation of this approach. More intelligent schemes have also been suggested in the literature (see [120]).

3. Concluding Remarks

This chapter presented an alternative to DP/RL based simulation-based methods to solve MDPs and SMDPs for average reward. There are several important areas in which research needs to be done to make MCAT a powerful alternative to the DP-based RL. It is not clear at this stage how function approximation methods may be combined with MCAT to enable it to handle problems with millions of states. The MCAT algorithm, as presented above, needs the storage of a look-up table for the probabilities of each state-action pair and several other quantities such as the C_r and C_t terms for each state. Clearly, this is a potential area for further research.

4. Bibliographic Remarks

Much of the research related to MCAT has stemmed from the pioneering work of Wheeler and Narendra [185]. Narendra and Thathachar [120] discuss these topics in considerable detail. Several other references provided in [120]. Our discussion in this chapter, especially that in the context of SMDPs, follows Gosavi *et al.* [65].

5. Review Questions

1. In what respects does the MCAT framework differ from the RL framework? In what ways are the two frameworks similar?
2. Does the MCAT framework need the transition probabilities of the Markov chain?
3. Read about schemes other than the Reward-Inaction scheme from the literature.
4. What are some of the limitations of this framework in the context of problems with several million states? Can they be overcome?

Chapter 11

CONVERGENCE: BACKGROUND MATERIAL

Mathematics is the language of engineering.

1. Chapter Overview

This chapter introduces some fundamental mathematical notions that will be useful in understanding the analysis presented in the subsequent chapters. The aim of this chapter is to introduce elements of the mathematical framework needed for analyzing the convergence of algorithms discussed in this book. Much of the material presented in this chapter is related to mathematical analysis and as such a reader with a good grasp of mathematical analysis may skip this chapter. To follow Chapter 12, the reader should read all material up to and including Theorem 11.2 in this chapter. All the ideas developed in this chapter will be needed in Chapter 13.

So far in the book we have restricted ourselves to an intuitive understanding of why algorithms generate optimal solutions. In this chapter, our aim is to make a transition from the nebulous world of intuition to the rock solid world of mathematical analysis. We have made every attempt to make this transition as gentle as possible.

It is important that this transition be made. Apart from the obvious fact that an algorithm's usefulness is doubted unless mathematical arguments show it, there are at least two other reasons for this.

1. The use of rigorous mathematical arguments to prove that an algorithm converges helps one identify the conditions under which an algorithm can generate an optimal (or near-optimal) solution. Every problem scenario tends to produce unique conditions, and as such knowing the limitations of an algo-

rithm is extremely useful when it comes to applying the algorithm to real-world problems.

2. Analyzing the convergence of an algorithm helps one gain insight into how an algorithm actually works. This knowledge can help one derive new and better algorithms for the same problem. This, of course, is a crucial part of theoretical operations research that keeps the subject alive and makes it so exciting, challenging, and beautiful.

In the first reading, even if you do not follow every detail in a proof, it is important that you strive for a clear understanding of the definitions and statements of theorems. Read ahead, and then come back to this chapter as and when the proof of a result needs to be understood.

We will begin this chapter with a discussion on vectors and vector spaces.

2. Vectors and Vector Spaces

We assume that you have a sound idea of what is meant by a **vector**. We will now attempt to tie this concept to the dependable framework of **vector spaces**. We will begin with simple concepts that you are familiar with.

A set is a collection of objects; we refer to these objects as elements. A set may contain a finite number of elements, or it may contain an infinite number of elements. We will use the calligraphic letter (e.g., \mathcal{A} , \mathcal{R}) to denote a set. A finite set can be described explicitly — that is, via a definition of each element. Consider for example:

$$\mathcal{A} = \{1, 4, 1969\}.$$

Here \mathcal{A} denotes a finite set because it contains a finite number of elements; note each of the elements: 1, 4 and 1969 is clearly defined in this definition of the set.

An infinite set is often described using a *conditional* notation which can take one of the two equivalent forms:

$$\mathcal{B} = \{x : 1 \leq x \leq 2\}$$

or

$$\mathcal{B} = \{x | 1 \leq x \leq 2\}.$$

In this notation, whatever follows “ $:$ ” or “ $|$ ” is the condition; the implication here is that \mathcal{B} is an infinite set that contains all real numbers between 1 and 2, including 1 and 2. In other words, $\mathcal{B} = [1, 2]$ is a *closed* interval.

The set \mathcal{R} is the set of real numbers — geometrically, it is the real number line. Algebraically, the idea lifts easily to n dimensions. We will see how, next.

Consider the following example.

Example 1. A set \mathcal{X} which is defined as: $\mathcal{X} = \{1, 2, 7\}$. There are 3 members in this set and each member is actually drawn from the real line — \mathcal{R} . Thus \mathcal{X} is a subset of the set \mathcal{R} .

Now consider the next example.

Example 2. A set \mathcal{Y} which is defined as :

$$\mathcal{Y} = \{(1.8, 3), (9.5, 1969), (7.2, 777)\}.$$

Each member of this set is actually a pair of two numbers. If each number in the pair is drawn from the real line, each pair is said to be a member of \mathcal{R}^2 . The pair is also called a 2-tuple. Thus each member of the set \mathcal{Y} is a two-tuple.

Now consider the following example.

Example 3. Each member of a set \mathcal{Z} assumes the following form:

$$(x(1), x(2), \dots, x(n)).$$

Then each of these members is said to belong to the set \mathcal{R}^n . Each member in this case is referred to as an n -tuple. An n -tuple has n elements.

The n -tuple is said to be an n -dimensional vector, if the n -tuple satisfies certain properties. We use the notation \rightarrow above a letter to denote a vector and the notation \wedge to denote the n -tuple. For instance, \vec{x} will denote a vector, but \hat{y} will denote an n -tuple that may or may not be a vector.

The properties that an n -tuple should possess to be called a vector are related to its addition and scalar multiplication. We describe these properties, next.

Addition Property. The addition property requires that the sum of the two n -tuples $\hat{x} = (x_1, x_2, \dots, x_n)$ and $\hat{y} = (y_1, y_2, \dots, y_n)$ be defined by

$$\hat{x} + \hat{y} = (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n).$$

Scalar Multiplication Property. The scalar multiplication property implies that a real number c times the n -tuple \hat{x} — denoted by $c\hat{x}$ — be defined by

$$c\hat{x} = (cx_1, cx_2, \dots, cx_n).$$

These definitions qualify the n -tuples to be called vectors, and many other properties follow from these two key properties. (See any standard text on linear algebra for more.)

Let us show some examples to demonstrate these properties. The sum of $\hat{a} = (1, 2)$ and $\hat{b} = (4, -7)$ has to be:

$$((1 + 4), (2 - 7)) = (5, -5),$$

if \hat{a} and \hat{b} are vectors.

So also the multiplication of \hat{a} by a scalar such as 10 should be given by:

$$10(1, 2) = (10, 20).$$

When endowed with these two key properties (addition and scalar multiplication), we refer to the set \mathcal{R}^n as a **vector space**. As mentioned above, the framework of vector spaces will be very useful in studying the convergence properties of the algorithms that we have seen in this book.

Some examples of Vector Spaces. \mathcal{R}^2 , \mathcal{R}^{132} , and \mathcal{R}^{1969} .

The next section deals with the important notion of vector norms.

3. Norms

A **norm**, in a given vector space V , is a scalar quantity associated with a given vector in that space. To give a rough geometric interpretation, it denotes the **length** of a vector. There are many ways to define a norm, and we will discuss some standard norms and their properties.

The so-called **max norm**, which is also called **infinity norm** or **sup norm**, is defined as follows.

$$\|\vec{x}\|_\infty = \max_i |x(i)|.$$

In the above definition, $\|\vec{x}\|_\infty$ denotes the max norm of the vector \vec{x} , and $x(i)$ denotes the i th element of the vector \vec{x} . The following example will illustrate the idea.

Example. $\vec{a} = (12, -13, 9)$ and $\vec{b} = (10, 12, 14)$ are two vectors in \mathcal{R}^3 . Their max norms are:

$$\|\vec{a}\|_\infty = \max\{|12|, |-13|, |9|\} = 13 \text{ and}$$

$$\|\vec{b}\|_\infty = \max\{|10|, |12|, |14|\} = 14.$$

There are other ways to define a norm. The Euclidean norm, for instance, is defined as follows:

$$\|\vec{x}\|_E = \sqrt{\sum_i (x(i))^2}.$$

It denotes the Euclidean length of a vector. The Euclidean norm of vectors \vec{a} and \vec{b} are:

$$\|\vec{a}\|_E = \sqrt{(12)^2 + (-13)^2 + (9)^2} = 19.84943$$

and

$$\|\vec{b}\|_E = \sqrt{(10)^2 + (12)^2 + (14)^2} = 20.9761.$$

Some important properties of norms are discussed next.

3.1. Properties of Norms

A scalar must satisfy the following properties in order to be called a norm. The notation $\|\vec{x}\|$ is a norm of a vector \vec{x} belonging to the vector space V if

1. $\|\vec{x}\| \geq 0$ for all \vec{x} in V , where $\|\vec{x}\| = 0$ if and only if $\vec{x} = \vec{0}$.
2. $\|a\vec{x}\| = |a| \|\vec{x}\|$ for all \vec{x} in V and all a .
3. $\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$ for all \vec{x}, \vec{y} in V .

The last property is called the triangle inequality. It is not hard to show that all the three conditions are true for both max norms and Euclidean norms.

4. Normed Vector Spaces

A vector space equipped with a norm is called a **normed vector space**. Study the following examples.

Example 1. The vector space defined by the set \mathcal{R}^2 along with the Euclidean norm.

The norm for this space is:

$$\|\vec{x}\|_E = \sqrt{\sum_{i=1}^2 [x(i)]^2}$$

if the vector \vec{x} is in \mathcal{R}^2 .

Example 2. The vector space defined by the set \mathcal{R}^{69} equipped with the max norm.

The norm for this space is:

$$\|\vec{x}\|_\infty = \max_{1 \leq i \leq 69} |x(i)|$$

where the vector \vec{x} is in \mathcal{R}^{69} .

We will return to the topic of vector spaces after discussing sequences.

5. Functions and Mappings

It is assumed that you are familiar with the concept of a function from high school calculus courses. In this section, we will present some definitions and some examples of functions.

5.1. Domain and Range of a function

A simple example of a function is:

$$y = x + 5$$

where x can take on any real value.

We often denote this as:

$$f(x) = x + 5.$$

The rule $f(x) = x + 5$ leaves us in no doubt about what the value of y will be when the value of x is known. A function is thus a **rule**. You are familiar with this idea. Now let us interpret the function as a set of pairs; one value in the pair will be a legal value of x , and the other value will be the corresponding value of y . For instance, the function considered above can be defined as a set of (x, y) pairs; some examples of these pairs are $(1, 6)$, $(1.2, 6.2)$, and so on.

Notice that the values of x actually come from \mathcal{R} — the set of real numbers. And the values of y , as a result of how we defined our function, also come from the same set \mathcal{R} . The set from which the values of x come is called the **domain** of the function, and the set from which the values of y come is called the **range** of the function. The domain and range of a function can be denoted using the following notation.

$$f : \mathcal{A} \rightarrow \mathcal{B}$$

where \mathcal{A} is the domain and \mathcal{B} is the range. This is read as: “a function f from \mathcal{A} to \mathcal{B} .”

The example given above is the simplest form of a function. Let us consider some more complicated functions. Consider the following function.

$$y = 4 + 5x_1 + 3x_2 \quad (11.1)$$

in which each of x_1 and x_2 takes on values from the set \mathcal{R} . Now, a general notation to express this function is

$$y = f(x_1, x_2).$$

This function (given in Equation 11.1) clearly picks up a **vector** such as $(1, 2)$ and assigns a value of 15 to y . Thus, the function (11.1) can be represented as a set of ordered triples (x_1, x_2, y) — examples of which are: $(1, 2, 15)$ and $(0.1, 1.2, 8.1)$, and so on. This makes it possible to view this function as an operation, whose input is a vector of the form (x_1, x_2) and whose output is a scalar. In other words, the domain of the function is \mathcal{R}^2 and its range is \mathcal{R} .

Functions that deal with vectors are also called **mappings** or **maps** or **transformations**. It is not hard to see that we can define a function from \mathcal{R}^2 to \mathcal{R}^2 . An example of such a function is defined by the following two equations:

$$\begin{aligned} x'_1 &= 4x_1 + 5x_2 + 9, \text{ and} \\ x'_2 &= 5x_1 + 7x_2 + 7. \end{aligned}$$

In the context of the above function, consider the vector with $x_1 = 1$ and $x_2 = 3$. It is a member of the set \mathcal{R}^2 . When the function or transformation is

applied on it (that is, used as an input in the right hand sides of the equations above), what we get is another vector belonging to the set \mathcal{R}^2 . In particular, we get the vector $x'_1 = 28$, and $x'_2 = 33$. One of the reasons why a vector function is also called a transformation is: a vector function generates a new vector from the one that is supplied to it.

The function shown above is actually defined by two linear equations. It is convenient to represent such a function with vector notation. The following illustrates the idea.

$$\vec{x}' = \mathbf{A}\vec{x} + \mathbf{B}. \quad (11.2)$$

Here

$$\vec{x} = (x_1, x_2)^T$$

and

$$\vec{x}' = (x'_1, x'_2)^T$$

are the two-dimensional vectors in question and \mathbf{A} and \mathbf{B} are the matrices. (\vec{x}^T denotes the transpose of the vector \vec{x} .) For the example under consideration, the matrices are:

$$\mathbf{A} = \begin{bmatrix} 4 & 5 \\ 5 & 7 \end{bmatrix},$$

and

$$\mathbf{B} = \begin{bmatrix} 9 \\ 7 \end{bmatrix}.$$

In general, functions or transformations can be defined from \mathcal{R}^{n_1} to \mathcal{R}^{n_2} , where n_1 and n_2 are positive integers that may or may not be equal to each other.

5.2. The notation for transformations

In general, we will use the following compact notation for transformations on vectors:

$$F\vec{a} \equiv F(\vec{a}).$$

Here F denotes a transformation (or a mapping or a function) that transforms the vector \vec{a} while $F\vec{a}$ denotes the **transformed vector**. This implies that $F\vec{a}$ is a vector that may be different from \vec{a} — the vector that was transformed by F .

We will be using operators of the form F^k frequently in the remainder of this book. The meaning of this operator needs to be explained. Let us examine some examples, next.

F^1 will mean the same thing as F . Now, $F^2(\vec{a})$, in words, is the vector that is obtained after applying transformation F to the vector $F(\vec{a})$. In other

words, $F^2(\vec{a})$ denotes the vector obtained after applying the transformation F two times on the vector \vec{a} . Mathematically:

$$F^2(\vec{a}) \equiv F(F(\vec{a})).$$

In general, F^k means the following:

$$F^k \vec{a} \equiv F(F^{k-1}(\vec{a})).$$

We will, next, discuss the important principle of mathematical induction.

6. Mathematical Induction

The principle of mathematical induction will be used heavily in this book. As such, it is important that you understand it clearly. Before we present it, let us define \mathcal{J} to be the set of positive integers—that is,

$$\mathcal{J} = \{1, 2, 3, 4, \dots\}.$$

The basis for mathematical induction is the well-ordering principle that we state below without proof.

Well-Ordering Principle. Every non-empty subset of \mathcal{J} has a member that can be called its smallest member.

To understand this principle, consider some non-empty subsets of \mathcal{J} .

$$\mathcal{A} = \{1, 3, 5, 7, \dots\},$$

$$\mathcal{B} = \{3, 4, 5, 6\},$$

and

$$\mathcal{C} = \{34, 1969, 4, 11\}.$$

Clearly, each of these sets is a subset of \mathcal{J} and has a smallest member. The smallest members of \mathcal{A} , \mathcal{B} , and \mathcal{C} are respectively 1, 3, and 4.

THEOREM 11.1 (Principle of Mathematical Induction)

If $R(n)$ is a statement containing the integer n such that

- (i) $R(1)$ is true, and
- (ii) after assuming that $R(k)$ is true, $R(k + 1)$ can be shown to be true for every k in \mathcal{J} ,

then $R(n)$ is true for all n in \mathcal{J} .

This theorem implies that the relation holds for $R(2)$ from the fact that $R(1)$ is true, and from the truth of $R(2)$ one can show the same for $R(3)$. In this way, it is true for all n in \mathcal{J} . All these are intuitive arguments. We now present a rigorous proof.

Proof We will use the so-called contradiction logic. It is likely that you have used this logic in some of your first experiences in mathematics in solving geometry riders. In this kind of a proof, we will begin by **assuming** that the result we are trying to prove is **false**. Using the falseness of the result, we will go on to show that something else that we assumed to be true cannot be true. Hence our hypothesis—that the “result is false”—cannot be true. As such, the result is true.

- Let us assume that:
- $R(1)$ is true, and
 - if $R(k)$ is true, $R(k + 1)$ is true for every k in \mathcal{J} .

In addition, let us assume that

- $R(n)$ is not true for **some** values of n .

Let us define a set \mathcal{S} to be a set that contains **all** values of n for which $R(n)$ is not true. Then from assumption (iii), \mathcal{S} is a non-empty subset of \mathcal{J} . From the well-ordering principle, one has that \mathcal{S} must have an element that can be described as its smallest element. Let us say that the value of n for this smallest element is p . Now $R(1)$ is true and so p must be greater than 1. This implies that $(p - 1) > 0$, which means that $(p - 1)$ belongs to \mathcal{J} .

Since p is the smallest element of \mathcal{S} , the relation $R(n)$ must be true for $(p - 1)$ since $(p - 1)$ does not belong to \mathcal{S} —the set of all the elements for which $R(n)$ is false. Thus we have that $R(p - 1)$ is true but $R(p)$ is not. Now this cannot be right since in (ii) we had assumed that if $R(k)$ is true then $R(k + 1)$ must be true. Thus setting $k = p - 1$, we have a contradiction, and hence our initial hypothesis must be wrong. In other words, the result must be true for **all** values of n in \mathcal{J} . ■

Some general remarks are in order here. We have started our journey towards establishing that the algorithms discussed in previous chapters, indeed, produce optimal or near-optimal solutions. To establish these facts, we will have to come up with proofs written in the style used above.

The proof presented above is our first “real” proof in this book. Any “mathematically rigorous” proof has to be worked out to its last detail, and if any detail is missing, the proof is not acceptable.

Let us see, next, how the theorem of mathematical induction is used in practice. Consider the following problem.

Prove that

$$1 + 3 + 5 + \cdots + (2p - 1) = (p)^2.$$

(We will use *LHS* to denote the left hand side of the relation and *RHS* the right hand side. Notice that it is easy to use the formulation for the arithmetic progression series to prove the above, but our intent here is to demonstrate how induction proofs work.)

Now, when $p = 1$, $LHS = 1$ and the $RHS = 1^2 = 1$, and thus the relation (the equation in this case) is true when $p = 1$.

Next let us assume that it is true when $p = k$, and hence

$$1 + 3 + 5 + \cdots + (2k - 1) = k^2.$$

Now when $p = (k + 1)$ we have that:

$$\begin{aligned} LHS &= 1 + 3 + 5 + \cdots + (2k - 1) + (2(k + 1) - 1) \\ &= k^2 + 2(k + 1) - 1 \\ &= k^2 + 2k + 2 - 1 \\ &= (k + 1)^2 \\ &= RHS \text{ when } p = k + 1. \end{aligned}$$

Thus we have proved that the relation is true when $n = k + 1$ if the relation is true when $p = k$. Then, using the theorem of mathematical induction, one has that the relation is true.

As you have probably realized that to prove a relation using induction, one must **guess** what the relation should be. But then the obvious question that arises is: how should one guess? For instance, in the example above, how may one guess that the RHS should be p^2 ? We will address this question in a moment. But keep in mind that once we have a guess, we can use induction to prove that the relation is true (that is if the guess is right in the first place).

Let us explore the *technology* underlying the making of good guesses. Needless to say, guessing randomly rarely works. Often, knowledge of the behavior of a relationship for **given** values of p is **known**, and from that it may be possible to see a general pattern. Once a pattern is recognized, one can generalize to obtain a plausible expression. All this is part of **rough work** that we do not show in proofs when we present them. But it forms an important part of establishing new results. We will illustrate these ideas using the result discussed above. (We imagine for the sake of this exercise that we do not know anything about arithmetic progressions.)

The RHS denotes the sum of the first p terms, which we will denote by S^p . So let us see what values S^p takes on for small values of p . The values are:

$$S^1 = 1,$$

$$S^2 = 4,$$

$$S^3 = 9,$$

$$S^4 = 16,$$

$$S^5 = 25,$$

and so on.

You have probably noticed that there is a relationship between

$$1, 4, 9, 16, 25, \dots$$

Yes, you've guessed right! They are the squares of

$$1, 2, 3, 4, 5 \dots$$

And hence a logical guess for S^p is

$$S^p = p^2.$$

We can then, as we have done above, use induction to show that it is indeed true. The example that we considered above is very simple but the idea was to show you that mathematical results are never the consequence of "divine inspiration" or "superior intuition" (to quote Gaughan [49]) but invariably the consequence of such rough work that we do not like to show (or rather don't need to show) once we have a fascinating result.

7. Sequences

It is assumed that you are familiar with the notion of a sequence. A familiar example of a sequence is:

$$a, ar, (a)^2, a(r)^3, \dots$$

We associate this with the geometric progression in which a is the first term in the sequence and r is the common ratio.

A sequence, by its definition, has an infinite number of terms. We will be dealing with sequences heavily in this book. Our notation for a sequence is:

$$\{x^p\}_{p=1}^{\infty}.$$

Here x^p denotes the p th term of the sequence. For the geometric sequence shown above:

$$x^p = (a)^{p-1}.$$

A sequence can be viewed as a function whose *domain* is the set of positive integers $(1, 2, \dots)$ and the *range* of which is the set that includes all possible values that the terms of the sequence can take on.

We are often interested in finding the value of the sum of the first m terms of a sequence. Let us consider the geometric sequence given above. The sum of the first m terms of this sequence, it can proved, is given by:

$$S^m = \frac{a(1 - (r)^m)}{(1 - r)}. \quad (11.3)$$

The sum itself forms the sequence

$$\{S^1, S^2, S^3, \dots, \}$$

We can denote this sequence by $\{S^m\}_{m=1}^{\infty}$. We will prove (11.3) using mathematical induction.

Proof Since the first term is a , S^1 should be a . Plugging in 1 for m in the (11.3), we can show that. So clearly the relation is true for $m = 1$.

Now let us assume that it is true for $m = k$; that is,

$$S^k = \frac{a(1 - (r)^k)}{(1 - r)}.$$

The $(k + 1)$ th term is (from the way the sequence is defined) ar^k . Then the sum of the first $(k + 1)$ terms is the sum of the first k terms and the $(k + 1)$ th term. Thus:

$$\begin{aligned} S^{k+1} &= \frac{a(1 - (r)^k)}{(1 - r)} + a(r)^k \\ &= \frac{a(1 - (r)^k) + a(r)^k(1 - r)}{1 - r} \\ &= \frac{a - a(r)^k + a(r)^k - a(r)^{k+1}}{1 - r} \\ &= \frac{a(1 - (r)^{k+1})}{1 - r} \end{aligned}$$

This proves that the relation is true when $m = k + 1$. Then, as a result of the theorem of mathematical induction, the relation given in (11.3) is true. ■

We will next discuss sequences that share an interesting property—namely that of convergence.

7.1. Convergent Sequences

Let us illustrate the idea of convergence of a sequence using familiar ideas. As you probably know, the sum of the GP series **converges** if the common ratio r lies in the interval $[0, 1)$. Convergence of a sequence, $\{x^p\}_{p=1}^{\infty}$, usually, means that as p starts becoming large, the terms of the sequence start approaching a **finite quantity**. The reason why the GP series converges is that the limit of the sum when p tends to infinity is a finite quantity. This is shown next.

$$\lim_{p \rightarrow \infty} S^p = \lim_{p \rightarrow \infty} a(1 - (r)^p)/(1 - r) = a(1 - 0)/(1 - r) = a/(1 - r).$$

The above uses the fact that

$$\lim_{p \rightarrow \infty} (r)^p = 0$$

when $0 \leq r < 1$.

Another simple example of a convergent sequence is one whose p th term is defined by $1/p$. The sequence can be written as:

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \dots$$

It is intuitively obvious that as p increases, the term $1/p$ keeps getting smaller. Also, we know from our knowledge of high school calculus that:

$$\lim_{p \rightarrow \infty} \frac{1}{p} = 0.$$

From this stage, we will gradually introduce rigor into our discussion on sequences.

In particular, we need to define some concepts such as convergence and Cauchy sequences. We will use the abbreviation **iff** to mean “if and only if.” The implication of ‘iff’ needs to be understood. When we say that the condition A holds iff B is true, the following is implied: A is true if B is true and B is true if A is true.

DEFINITION 11.1 A Convergent Sequence: A sequence $\{a^p\}_{p=1}^{\infty}$ is said to converge to a real number A iff for any $\epsilon > 0$, there exists a positive integer N such that for all $p \geq N$, we have that

$$|a^p - A| < \epsilon.$$

Let us explain this definition with an example. Consider the sequence whose p th term is given by $\frac{1}{p}$. It is not hard to see why this sequence converges to 0. If you supply us with a value for ϵ , and if the sequence indeed converges to 0, then we should be able to come up with a value for N such that for any integer, p , which is greater than N , $|a^p - 0|$ must be less than ϵ . For instance, let us assume that $\epsilon = 0.01$. Then,

$$|a^p - 0| = \left| \frac{1}{p} - 0 \right| = \frac{1}{p} < \epsilon.$$

This implies that

$$p > \frac{1}{\epsilon} = \frac{1}{0.01} = 100.$$

This means that for $p \geq 101$, the condition $|a^p - 0| < \epsilon$ must be satisfied. It is not hard to see from this discussion that the value of N depends on the value of ϵ , but if the sequence converges, then for any value of ϵ , one can come up with a suitable finite value for N .

We next define some special properties of sequences.

7.2. Increasing and decreasing sequences

DEFINITION 11.2 A sequence $\{a^p\}_{p=1}^{\infty}$ is said to be **decreasing**, if for all p ,

$$a^{p+1} \leq a^p.$$

For the same reason, it is called an **increasing sequence**, if for all p ,

$$a^{p+1} \geq a^p.$$

7.3. Boundedness

We next define the concepts of “bounded above” and “bounded below,” in the context of a sequence. A sequence $\{a^p\}_{p=1}^{\infty}$ is said to be **bounded below**, if there exists a finite value L such that:

$$a^p \geq L$$

for all values of p . L is then called a **lower bound** for the sequence.

Similarly, a sequence is said to be **bounded above**, if there exists a finite value U such that:

$$a^p \leq U$$

for all values of p . U is then called an **upper bound** for the sequence.

The sequence:

$$\{1, 2, 3, \dots\}$$

is bounded below, but not above. Notice that 1 is a *lower bound* for this sequence and so are 0 and -1 and -1.5 . But 1 is the highest lower bound (often called the infimum).

In the sequence:

$$\{1, 1/2, 1/3, \dots\},$$

1 is an upper bound and so are 2 and 3 etc. Here 1 is the lowest of the upper bounds (also called the supremum).

A sequence that is bounded both above and below is said to be a **bounded sequence**.

We will next examine a useful result related to decreasing (increasing) sequences that are bounded below (above). The result states that a decreasing (increasing) sequence that is bounded below (above) converges.

Intuitively, it should be clear that a decreasing sequence, that is, a sequence in which each term is less than or equal to the previous term, should converge to a finite value because the values of the terms cannot go below a finite value M . So the terms keep decreasing and once they reach a point below which they cannot go, they stop decreasing; so the sequence should converge. Now, let us prove this idea using precise mathematics. One should remember that to prove convergence, one must show that the sequence satisfies Definition 11.1.

THEOREM 11.2 *A decreasing (increasing) sequence converges if it is bounded below (above).*

Proof We will work out the proof for the decreasing sequence case. For the increasing sequence case, the proof can be worked out in a similar fashion. Let us denote the sequence by $\{a^p\}_{p=1}^{\infty}$. Let L be the highest of the lower bounds on the sequence. Then, for any p , since L is a lower bound,

$$a^p \geq L. \quad (11.4)$$

Choose a strictly positive value for the variable ϵ . Then $\epsilon > 0$. Then $L + \epsilon$, which is greater than L , is not a lower bound. (Note that L is the highest lower bound.) Then, it follows that there exists an N , such that

$$a^N < L + \epsilon. \quad (11.5)$$

Then, for $p \geq N$, since it is a decreasing sequence,

$$a^p \leq a^N.$$

Combining the above, with Inequalities (11.4) and (11.5), we have that for $p \geq N$:

$$L \leq a^p \leq a^N < L + \epsilon.$$

The above implies that for $p \geq N$:

$$L \leq a^p < L + \epsilon.$$

Using the above and the fact that $L - \epsilon < L$, we have that for $p \geq N$:

$$L - \epsilon < a^p < L + \epsilon.$$

The above means that for $p \geq N$:

$$|a^p - L| < \epsilon$$

where $\epsilon > 0$. From Definition 11.1, this means that $\{a^p\}_{p=1}^{\infty}$ is convergent. ■

The definition of convergence (Definition 11.1) requires the knowledge of the limit (A) of the sequence. We, next, discuss a concept that will lead us to a method that can determine whether a sequence is convergent without attempting to identify its limit.

DEFINITION 11.3 A Cauchy sequence: *A sequence $\{a^p\}_{p=1}^{\infty}$ is called a Cauchy sequence iff for each $\epsilon > 0$, there exists a positive integer N such*

that if one chooses any m greater than or equal to N and any k also greater than or equal to N , that is, $k, m \geq N$, then

$$|a^k - a^m| < \epsilon.$$

This is an important definition that needs to be understood clearly. What it says is that a sequence is Cauchy, if for any given value of ϵ , there exists a finite integer N such that the difference in any two terms of the sequence beyond and including the N th term is less than ϵ . This means that beyond and including that point N in the sequence, any two terms are ϵ -close —that is, the absolute value of their difference is less than ϵ . Let us illustrate this idea with an example.

Let us assume that $\{1/p\}_{p=1}^{\infty}$ is a Cauchy sequence. (In fact every convergent sequence is Cauchy, and so this assumption can be justified.) Select $\epsilon = 0.0125$, pick *any* two terms that are beyond the 79th term ($N = 80 = 1/0.0125$ for this situation, and clearly depends on the value of ϵ), and calculate the absolute value of their difference. The value will be less than 0.025. The reason is at the 80th term, the value will be equal to $1/80 = 0.0125$, and hence the difference is bound to be less than 0.0125. Note that:

$$\frac{1}{80} - \frac{1}{p} < 0.0125 = \epsilon$$

for a strictly positive value of p .

In summary, the value of N will depend on ϵ , but the definition says that for a Cauchy sequence, one can always find a value for N beyond which terms are ϵ -close.

It can be proved that every convergent sequence is Cauchy. Although this is an important result, its converse is even more important. Hence we omit its proof and turn our attention to its converse.

The converse states that every Cauchy sequence is convergent (Theorem 11.5, page 305). This will prove to be a very useful result as it will provide us with a mechanism to establish the convergence of a sequence. The reason is: by showing that a given sequence is Cauchy, we will be able to establish that the sequence is convergent.

To prove this important result, we need to discuss a few important topics, which are: the boundedness of Cauchy sequences, accumulation points, neighborhoods, and the Bolzano-Weierstrass Theorem. The next few paragraphs will be devoted to discussing these topics.

For a bounded sequence, each of its terms is a finite quantity. As a result, the sequence $\{a^p\}_{p=1}^{\infty}$ is bounded if there is a positive number M such that $|a^p| \leq M$ for all p . This condition implies that:

$$-M \leq a^p \leq M.$$

THEOREM 11.3 *Every Cauchy sequence is bounded.*

This is equivalent to saying that every term of a Cauchy sequence is a finite quantity. The proof follows.

Proof Assume that $\{a^p\}_{p=1}^{\infty}$ is a Cauchy sequence. Let us set $\epsilon = 1$. By the definition of a Cauchy sequence, there exists an N such that, for all $k, m \geq N$, we have $|a^k - a^m| < 1$. Let $m = N$. Then for $k \geq N$,

$$\begin{aligned}|a^k| &= |a^k - a^N + a^N| \\ &\leq |a^k - a^N| + |a^N| < 1 + |a^N|.\end{aligned}$$

Since N is finite, a^N must be finite. Hence $|a^N|$ must be a positive finite number. Then it follows from the above inequality that $|a^k|$ is less than a positive finite quantity when $k \geq N$. In other words, a^k is bounded for $k \geq N$. Now, since N is finite, all values of a^k from $k = 1, 2, \dots, N$ are also finite; that is a^k is finite for $k \leq N$. Thus we have shown that a^k is finite for all values of k . The proof is complete. ■

In the proof above, if you want to express the boundedness condition in the form presented just before the statement of this theorem, define M as

$$M = \max\{a^1, a^2, \dots, a^{N-1}, a^N, a^N + 1\}.$$

Then $|a^k| \leq M$ for all values of k . Thus $|a^k|$ is bounded for all values of k .

DEFINITION 11.4 *A neighborhood of a real number x with a positive radius r is the open interval $(x - r, x + r)$.*

An open interval (a, b) is a set that contains infinitely many points from a to b but the points a and b themselves are not included in it. Examples are: $(2, 4)$ (neighborhood of 3) and $(1.4, 79.2)$. In the scalar world, a neighborhood of x is an open interval centered at x .

DEFINITION 11.5 *A point x is called the accumulation point of a set S iff every neighborhood of x contains infinitely many points of S .*

Although this is a very elementary concept, one must have a very clear idea of what constitutes and what does not constitute an accumulation point of a given set.

Example 1. Let a set S be defined by the open interval $(1, 2.4)$. Then every point in S is an accumulation point. Consider any point in the interval—say 2. Whichever neighborhood of 2 is chosen, you will find that one of the following is true:

- The neighborhood is a subset of \mathcal{S} (for example: if the neighborhood is the interval $(1.8, 2.2)$)
- The neighborhood contains a subset of \mathcal{S} (for example: the neighborhood is the interval $(1.2, 2.8)$)
- The set \mathcal{S} is a subset of the neighborhood (for example: the neighborhood is the interval $(0.5, 3.5)$).

In any case, the neighborhood will contain infinitely many points of the set \mathcal{S} .

Example 2. We next consider an example of a point that cannot be an accumulation point of the interval $(1, 2.4)$. Consider the point 5. Now, the interval $(2, 8)$ is a neighborhood of 5. This neighborhood contains infinitely many points of the interval $(1, 2.4)$. Yet, 5 is not an accumulation point of the interval $(1, 2.4)$, since one can always construct a neighborhood of 5 that does not contain even a single point from the interval $(1, 2.4)$. An example of such a neighborhood is the interval $(4, 6)$.

Notice that the definition of an accumulation point of a set says that every neighborhood of the set must contain infinitely many points of the set.

Example 3. Consider the sequence $\{\frac{1}{p}\}_{p=1}^{\infty}$. Recall that the range of a sequence is the set of all the different values that the sequence can assume. The range of this sequence is an infinite set since one can keep increasing the value of p to get positive terms that keep getting smaller. The terms of course approach 0, but we can never find a finite value for p for which $a(p)$ will equal 0. Hence 0 is not a point in the range of this sequence. And yet 0 is an accumulation point of the range. This is because any neighborhood of 0 contains infinitely many points of the range of the sequence.

We are now at a point to discuss the famous Bolzano-Weierstrass theorem. We will not present its proof, so as to not distract the reader from our major theme which is the convergence of a Cauchy sequence. The proof can be found in any undergraduate text on mathematical analysis such as Gaughan [49] or Douglass [42]. This theorem will be needed in proving that Cauchy sequences are convergent.

THEOREM 11.4 (Bolzano-Weierstrass Theorem:) *Every bounded infinite set of real numbers has at least one accumulation point.*

The theorem describes a very important property of bounded infinite sets of real numbers. It says that a bounded infinite set of real numbers has at least one accumulation point. We will illustrate the statement of the theorem using an example.

Consider the interval $(1, 2)$. It is bounded (by 1 below and 2 above) and has infinitely many points. Hence it must have at least one accumulation point. We have actually discussed above how *any* point in this set is an accumulation point.

A finite set does not have any accumulation point because, by definition, it has a finite number of points, and hence none of its neighborhoods can contain an infinite number of points.

The next result is a key result that will be used in the convergence analysis of reinforcement learning algorithms.

THEOREM 11.5 *A Cauchy sequence is convergent.*

Proof Let $\{a^p\}_{p=1}^\infty$ be a Cauchy sequence. The range of the sequence can be an infinite or a finite set. Let us handle the finite case first, as it is easier.

Let $\{s_1, s_2, \dots, s_r\}$ consisting of r terms denote a finite set representing the range of the sequence $\{a^p\}_{p=1}^\infty$. Now if we choose

$$\epsilon = \min\{|s_i - s_j| : i \neq j; i, j = 1, 2, \dots, r\},$$

then there is a positive integer N such that for any $k, m \geq N$ we have that $|a^k - a^m| < \epsilon$. Now it is the case that $a^k = s_c$ and $a^m = s_d$ for some c and some d belonging to the set $\{1, 2, \dots, r\}$. Thus:

$$|s_c - s_d| = |a^k - a^m| < \epsilon = \min\{|s_i - s_j| : i \neq j; i, j = 1, 2, \dots, r\}.$$

From the above, it is clear that the absolute value of the difference between s_c and s_d is strictly less than the minimum of the absolute value of the differences between the terms. As a result,

$$|a^k - a^m| = 0,$$

that is

$$a^k = a^m$$

for $k, m \geq N$. This implies that from some point (N) onwards, the sequence values are constant. This means that at this point the sequence converges to a finite quantity. Thus the convergence of the Cauchy sequence with a finite range is established.

Next, let us assume that the set we refer to as the range of the sequence is infinite. Let us call the range S . The range of a Cauchy sequence is bounded by Theorem 11.3. Since S is infinite and bounded, by the Bolzano-Weierstrass theorem (Theorem 11.4), S must have an accumulation point; let us call it x . We will prove that the sequence converges to x .

Choose an $\epsilon > 0$. Since x is an accumulation point of S , by its definition, the interval $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$ is a neighborhood of x that contains infinitely many

points of \mathcal{S} . Now, since $\{a^p\}_{p=1}^\infty$ is a Cauchy sequence, there is a positive integer N such that for $k, m \geq N$, $|a^k - a^m| < \epsilon/2$.

Since $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$ contains infinitely many points of \mathcal{S} , and hence infinitely many terms of the sequence $\{a^p\}_{p=1}^\infty$, there exists a t such that $t \geq N$ and that a^t is a point in the interval $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$. The last statement implies that:

$$|a^t - x| < \epsilon/2.$$

Now, since $t > N$ and by the selection of N above, we have from the definition of a Cauchy sequence that

$$|a^p - a^t| < \epsilon/2.$$

Then, we have that

$$|a^p - x| \leq |a^p - a^t| + |a^t - x| = \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.$$

This implies that $\{a^p\}_{p=1}^\infty$ converges to x . ■

8. Sequences in \mathcal{R}^n

Thus far, our discussions have been limited to **scalar** sequences. A scalar sequence is one whose terms are scalar quantities. Scalar sequences are also called sequences in \mathcal{R} . The reason for this is that scalar quantities are members of the set \mathcal{R} .

A sequence in \mathcal{R}^n is a sequence whose terms are vectors. We will refer to this sequence as a **vector sequence**. For example, consider a sequence $\{\vec{a}^p\}_{p=1}^\infty$ whose p th term is defined as:

$$\vec{a}^p = \left(\frac{1}{p}, p^2 + 1 \right).$$

This sequence, starting at $p = 1$, will take on the following values:

$$\{(1, 2), (\frac{1}{2}, 5), (\frac{1}{3}, 10), \dots\}.$$

The above is an example of a sequence in \mathcal{R}^2 . This concept nicely extends to any dimension. Each of the individual scalar sequences in such a sequence is called a **coordinate sequence**. Thus, in the example given above, $\{\frac{1}{p}\}_{p=1}^\infty$ and $\{p^2 + 1\}_{p=1}^\infty$ are the coordinate sequences of $\{\vec{a}^p\}_{p=1}^\infty$.

Remark. We will use the notation $a^p(i)$ to denote the p th term of the i th coordinate sequence of the vector sequence $\{\vec{a}^p\}_{p=1}^\infty$. For instance, in the example given above,

$$\vec{a}^p = \{a^p(1), a^p(2)\},$$

where

$$a^p(1) = \frac{1}{p}$$

and

$$a^p(2) = p^2 + 1.$$

We will next define what is meant by a Cauchy sequence in \mathcal{R}^n .

9. Cauchy sequences in \mathcal{R}^n

The concept of Cauchiness also extends elegantly from sequences in \mathcal{R} (where we have seen it) to sequences in \mathcal{R}^n . The Cauchy condition in \mathcal{R}^n is defined next.

DEFINITION 11.6 A sequence in \mathcal{R}^n , denoted by $\{\vec{a}^p\}_{p=1}^\infty$, is said to be a Cauchy sequence in \mathcal{R}^n , if for any given $\epsilon > 0$, there exists an N such that for any $m, k \geq N$, $\|\vec{a}^k - \vec{a}^m\| < \epsilon$.

Here $\|\cdot\|$ denotes any norm.

You've probably noticed that the definition of Cauchiness in \mathcal{R}^n uses a norm whereas the same in \mathcal{R} uses the absolute value of the difference between two points. The definition of Cauchiness for \mathcal{R} is of course a special case of Definition 11.6. We next state a result that relates the Cauchiness of the individual coordinate sequences within a vector sequence to the Cauchiness of the parent (vector) sequence.

LEMMA 11.6 If a vector sequence $\{\vec{a}^p\}_{p=1}^\infty$ in \mathcal{R}^n is Cauchy, then each coordinate sequence in the vector sequence is a Cauchy sequence in \mathcal{R} . In other words, if the vector sequence is Cauchy, then for any given $\epsilon > 0$, there exists an N such that for all $k, m \geq N$,

$$|a^m(i) - a^k(i)| < \epsilon$$

for $i = 1, 2, \dots, n$.

Proof We will prove the result for the max norm. The result can be proved for any norm. Since $\{\vec{a}^p\}_{p=1}^\infty$ is Cauchy, we have that for a given value of $\epsilon > 0$, there exists an N such that for any $m, k \geq N$,

$$\|\vec{a}^k - \vec{a}^m\| < \epsilon.$$

From the definition of the max norm, we have that for any $k, m \geq N$,

$$\|\vec{a}^k - \vec{a}^m\| = \max_i |a^k(i) - a^m(i)|.$$

Combining the information in the preceding equation and the inequation before it, one has that for any $k, m \geq N$,

$$\max_i |a^k(i) - a^m(i)| < \epsilon.$$

Now in this equation, the $<$ relation holds for \max_i in the left hand side. Hence the result must be true for all i . Thus, for any $k, m \geq N$,

$$|a^k(i) - a^m(i)| < \epsilon$$

is true for all i . This implies that each coordinate sequence is Cauchy in \mathcal{R} .

■

We next need to understand an important concept that plays a central role in the convergence of discounted reinforcement learning algorithms. It is important that you digest the main idea around which the next section revolves.

10. Contraction mappings in \mathcal{R}^n

This section is about a special type of mapping (or transformation) — a so-called **contraction** mapping or contractive mapping. Let us begin with some geometric insight into this idea. To this end, consider two distinct vectors, \vec{a} and \vec{b} , in \mathcal{R}^2 space. Let us define \vec{c} as follows:

$$\vec{c} = \vec{a} - \vec{b}.$$

Then \vec{c} denotes a difference of the two vectors. Now apply a mapping F to both \vec{a} and \vec{b} . We will be applying this transformation repeatedly.

Let us define the notation to be used when the mapping F is applied repeatedly. We will use \vec{x}^k to denote the vector obtained after k applications of F to the vector \vec{x} . As such \vec{a}^0 will stand for \vec{a} , \vec{a}^1 will stand for $F\vec{a}^0$, \vec{a}^2 will stand for $F^2\vec{a}^0$, and so on. The difference between the transformed vectors \vec{a}^k and \vec{b}^k will be denoted by \vec{c}^k . Thus:

$$\vec{c}^1 \equiv F(\vec{a}) - F(\vec{b}),$$

and

$$\vec{c}^2 \equiv F^2(\vec{a}) - F^2(\vec{b}),$$

and so on.

If the length (norm) of the vector \vec{c}^1 is strictly smaller than that of \vec{c}^0 , then the difference vector can be said to have become smaller. In other words, the difference vector can be said to have “shrunk.” If this is true of every occasion the mapping is applied, then we go on to declare that the mapping is **contractive** in that space. What is more interesting is that the vectors \vec{a} and

\vec{b} keep approaching each other with every application of such a mapping. This happens because the difference between the two vectors keeps getting smaller and smaller. Eventually, the vectors will become identical. This vector — that will be obtained ultimately — is called a **fixed point** of the mapping. Essentially, this implies that given any vector (\vec{a} or \vec{b} or any other vector), if one keeps applying the transformation F , one eventually obtains the same vector. (The final vector obtained in the limit is usually called the fixed point.) We next provide a technical definition of a fixed point.

DEFINITION 11.7 *A \vec{x} is said to be a fixed point of the transformation F if:*

$$F\vec{x} = \vec{x}.$$

Note that the definition says nothing about contractions, and in fact, F may have a fixed point even if it is not contractive.

Let us illustrate the idea of generating a fixed point with a contractive mapping using an example. Consider the following vectors.

$$\vec{a}^0 = (4, 2), \text{ and } \vec{b}^0 = (2, 2).$$

We will apply the transformation G in which $x^k(i)$ will denote the i th component of the vector \vec{x}^k . Let us assume that G is contractive (this can be proved) and is defined as:

$$\begin{aligned} x^{k+1}(1) &= 5 + 0.2x^k(1) + 0.1x^k(2), \\ x^{k+1}(2) &= 10 + 0.1x^k(1) + 0.2x^k(2)). \end{aligned}$$

Table 11.1 shows the results of applying transformation G repeatedly. A careful observation of the table will reveal the following. With every iteration, the difference vector \vec{c}^k becomes smaller. Note that when $k = 12$, the vectors \vec{a}^k and \vec{b}^k have become one, if one ignores anything beyond the fifth place after the decimal point. The two vectors will become one, generally speaking, when $k = \infty$. In summary, one starts with two different vectors but eventually goes to a **fixed point** which, in this case, seems to be very close to (7.936507, 13.492062).

What we have demonstrated above is an important property of a contractive mapping. One may start with any vector, but successive applications of the mapping transforms the vector into a fixed vector.

See Figures 11.1, 11.2, and 11.3 to get a geometric feel for how a contraction mapping keeps “shrinking vectors” in \mathbb{R}^2 space, and carries any given vector to a unique fixed point. The figures are related to the data given in Table 11.1. The contraction mapping is very much like a dog that carries every bone (read vector) that it gets to its own hidey hole (read a unique fixed point), regardless of the size of the bone or where the bone has come from.

Table 11.1. Table showing the change in values of the vectors \vec{a} and \vec{b} after repeated applications of G

k	$a^k(1)$	$a^k(2)$	$b^k(1)$	$b^k(2)$	$c^k(1)$	$c^k(2)$
0	4.000000	2.000000	2.000000	2.000000	2.000000	0.000000
1	6.000000	10.800000	5.600000	10.600000	0.400000	0.200000
2	7.280000	12.760000	7.180000	12.680000	0.100000	0.080000
3	7.732000	13.280000	7.704000	13.254000	0.028000	0.026000
4	7.874400	13.429200	7.866200	13.421200	0.008200	0.008000
5	7.917800	13.473280	7.915360	13.470860	0.002440	0.002420
6	7.930888	13.486436	7.930158	13.485708	0.000730	0.000728
7	7.934821	13.490376	7.934602	13.490157	0.000219	0.000219
8	7.936002	13.491557	7.935936	13.491492	0.000066	0.000066
9	7.936356	13.491912	7.936336	13.491892	0.000020	0.000020
10	7.936462	13.492018	7.936456	13.492012	0.000006	0.000006
11	7.936494	13.492050	7.936492	13.492048	0.000002	0.000002
12	7.936504	13.492059	7.936503	13.492059	0.000001	0.000001
13	7.936507	13.492062	7.936507	13.492062	0.000000	0.000000

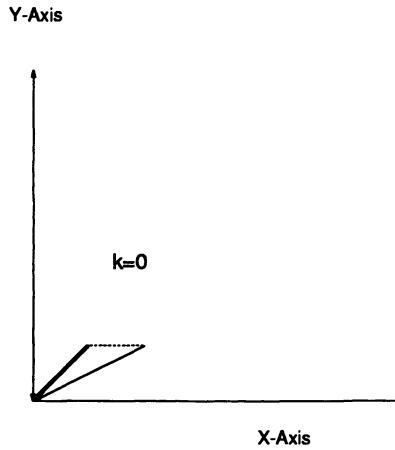


Figure 11.1. The thin line represents vector \vec{a} , the dark line represents the vector \vec{b} , and the dotted line the vector \vec{c} . This is before applying G .

Let us now examine a more mathematically-precise definition of a contraction mapping.

DEFINITION 11.8 *A mapping (or transformation) F is said to be a contraction mapping in \mathbb{R}^n , if there exists a λ , such that $0 \leq \lambda < 1$ and*

$$\|F\vec{v} - F\vec{u}\| \leq \lambda \|\vec{v} - \vec{u}\|$$

for all \vec{v}, \vec{u} in \mathbb{R}^n .

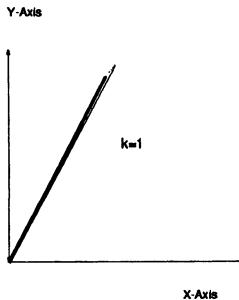


Figure 11.2. The thin line represents vector \vec{a} , the dark line represents the vector \vec{b} , and the dotted line the vector \vec{c} . This is the picture after one application of G . Notice that the vectors have come closer.

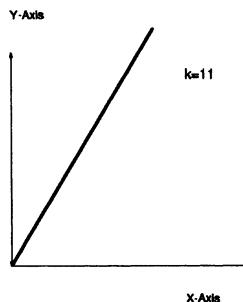


Figure 11.3. This is the picture after 11 applications of F . By now the vectors are almost on the top of each other and it is difficult to distinguish between them.

As is clear from the definition, the norm represents what was referred to as ‘length’ in our informal discussion on contraction mappings.

By applying F repeatedly, one obtains a sequence of vectors. Consider a vector \vec{a} on which the transformation F is applied repeatedly. This will form a sequence of vectors, which we will denote by $\{\vec{a}^k\}_{k=0}^{\infty}$. Here \vec{a}^k denotes the k th term of the sequence. It must be noted that each term is itself a vector. The relationship between the terms is given by

$$\vec{v}^{k+1} = F\vec{v}^k.$$

Thus the sequence can also be denoted as:

$$\{\vec{v}^0, \vec{v}^1, \vec{v}^2, \dots\}.$$

It is clear that

$$\vec{v}^1 = F\vec{v}^0,$$

$$\vec{v}^2 = F\vec{v}^1,$$

and so on. Now if

$$\|F\vec{v}^0 - F\vec{u}^0\| \leq \lambda \|\vec{v}^0 - \vec{u}^0\|$$

for all vectors in \mathcal{R}^n , then

$$\begin{aligned}\|F^2\vec{v}^0 - F^2\vec{u}^0\| &\leq \lambda \|F\vec{v}^0 - F\vec{u}^0\| \\ &\leq \lambda^2 \|\vec{v}^0 - \vec{u}^0\|.\end{aligned}$$

Similarly,

$$\begin{aligned}\|F^3\vec{v}^0 - F^3\vec{u}^0\| &\leq \lambda \|F^2\vec{v}^0 - F^2\vec{u}^0\| \\ &\leq \lambda^2 \|F\vec{v}^0 - F\vec{u}^0\| \\ &\leq \lambda^3 \|\vec{v}^0 - \vec{u}^0\|.\end{aligned}$$

In general,

$$\|F^m\vec{v}^0 - F^m\vec{u}^0\| \leq \lambda^m \|\vec{v}^0 - \vec{u}^0\|. \quad (11.6)$$

Equation (11.6) can be proved, from Definition 11.8, using induction. This is left as an exercise.

Remark. Consider F to be a contraction mapping in \mathcal{R}^n . As such for any two vectors \vec{x} and \vec{y} in \mathcal{R}^n , one has that:

$$\|F\vec{x} - F\vec{y}\| \leq \lambda \|\vec{x} - \vec{y}\|. \quad (11.7)$$

Now consider a vector sequence $\{\vec{a}^k\}_{k=0}^\infty$ in which

$$\vec{a}^1 = F\vec{a}^0, \quad \vec{a}^2 = F\vec{a}^1,$$

and so on, where \vec{a}^k is a member of \mathcal{R}^n . Then one has that

$$\begin{aligned}\|\vec{a}^1 - \vec{a}^2\| &= \|F\vec{a}^0 - F\vec{a}^1\| \\ &\leq \lambda \|\vec{a}^0 - \vec{a}^1\|.\end{aligned} \quad (11.8)$$

Equation (11.8) follows from Equation (11.7).

We are now ready to prove a major theorem that will prove very useful in the convergence analysis of some dynamic programming algorithms.

THEOREM 11.7 (Fixed Point Theorem) Suppose F is a contraction mapping in \mathcal{R}^n . Then

- there exists a unique vector — let us call it \vec{v}_* — in \mathcal{R}^n such that $F\vec{v}_* = \vec{v}_*$ and

- for any \vec{v}^0 in \mathcal{R}^n , the sequence $\{\vec{v}^k\}_{k=0}^\infty$ defined by

$$\vec{v}^{k+1} = F\vec{v}^k = F^k \vec{v}^0 \quad (11.9)$$

converges to \vec{v}_* .

Here \vec{v}_* denotes the fixed point of F .

Proof For any $m' > m$,

$$\|\vec{v}^m - \vec{v}^{m'}\| = \|F^m \vec{v}^0 - F^{m'} \vec{v}^0\| \quad (11.10)$$

$$\leq \lambda^m \|\vec{v}^0 - \vec{v}^{m'-m}\| \quad (11.11)$$

$$= \lambda^m \|\vec{v}^0 - \vec{v}^1 + \vec{v}^1 - \vec{v}^2 + \dots\| \quad (11.12)$$

$$+ \dots + \vec{v}^{m'-m-1} - \vec{v}^{m'-m}\| \quad (11.12)$$

$$\leq \lambda^m (\|\vec{v}^0 - \vec{v}^1\| + \|\vec{v}^1 - \vec{v}^2\| + \dots + \dots + \|\vec{v}^{m'-m-1} - \vec{v}^{m'-m}\|) \quad (11.12)$$

$$= \lambda^m \|\vec{v}^0 - \vec{v}^1\| \cdot [1 + \lambda + \lambda^2 + \dots + \lambda^{m'-m-1}] \quad (11.13)$$

$$= \lambda^m \|\vec{v}^0 - \vec{v}^1\| \left[\frac{1 - \lambda^{m'-m}}{1 - \lambda} \right] \quad (11.14)$$

$$< \lambda^m \|\vec{v}^0 - \vec{v}^1\| \left[\frac{1}{1 - \lambda} \right] \quad (11.15)$$

In the above:

- Line (11.10) follows from the fact that F is contractive (see Equation 11.9).
- Line (11.11) follows from Inequation (11.6) by setting $\vec{u}^0 = \vec{v}^{m'-m}$, which can be shown to imply that:

$$F^m \vec{u}^0 = F^{m'} \vec{v}^0.$$

- Line (11.12) follows from Property 3 of norms given in Section 3.
- Line (11.13) follows from the Remark related to Inequation (11.8).
- Line (11.14) follows from the sum of a finite GP series given in Equation (11.3).
- Line (11.15) follows from the fact that

$$\lambda^{m'-m} > 0$$

since $\lambda > 0$ and $m' > m$.

From (11.15), one can state that by selecting a large enough value for m , $\|\vec{v}(m) - \vec{v}(m')\|$ can be made as small as needed. In other words, for a given value of ϵ , which satisfies:

$$\epsilon > 0,$$

one can come up with a finite value for m such that

$$\|\vec{v}^m - \vec{v}^{m'}\| < \epsilon.$$

Since $m' > m$, the above ensures that the vector sequence $\{\vec{v}^k\}_{k=0}^\infty$ is Cauchy. From Lemma 11.6 (on page 307), it follows that if the vector sequence satisfies the Cauchy condition (see Definition 11.6 on page 307), then each coordinate sequence is also Cauchy. From Theorem 11.5 (page 305), a Cauchy sequence converges and thus each coordinate sequence converges to a finite number. Consequently, the vector sequence converges to a finite valued vector. Let us denote the limit by \vec{v}_* . Hence we have that

$$\lim_{k \rightarrow \infty} \|\vec{v}^k - \vec{v}_*\| = 0. \quad (11.16)$$

Now we need to show that $F\vec{v}_* = \vec{v}_*$.

From the properties of norms (see section 3), it follows that

$$\begin{aligned} 0 &\leq \|F\vec{v}_* - \vec{v}_*\| \\ &= \|F\vec{v}_* - \vec{v}^k + \vec{v}^k - \vec{v}_*\| \\ &\leq \|F\vec{v}_* - \vec{v}^k\| + \|\vec{v}^k - \vec{v}_*\| \\ &= \|F\vec{v}_* - F\vec{v}^{k-1}\| + \|\vec{v}^k - \vec{v}_*\| \\ &\leq \lambda \|\vec{v}_* - \vec{v}^{k-1}\| + \|\vec{v}^k - \vec{v}_*\| \end{aligned} \quad (11.17)$$

From (11.16), both terms of the right hand side of (11.17) can be made arbitrarily small by choosing a large enough k . Hence

$$\|F\vec{v}_* - \vec{v}_*\| = 0.$$

This implies that $F\vec{v}_* = \vec{v}_*$, and so \vec{v}_* is a fixed point of F .

What remains to be shown is that the fixed point \vec{v}_* is unique. To prove this, let us assume that there are two vectors \vec{a} and \vec{b} that satisfy $\vec{x} = F\vec{x}$. Hence

$$\vec{a} = F\vec{a},$$

and

$$\vec{b} = F\vec{b}.$$

Then using the contraction property one has that:

$$\begin{aligned} \lambda \|\vec{b} - \vec{a}\| &\geq \|F\vec{b} - F\vec{a}\| \\ &= \|\vec{b} - \vec{a}\| \end{aligned}$$

which implies that:

$$\lambda \|\vec{b} - \vec{a}\| \geq \|\vec{b} - \vec{a}\|.$$

Since a norm is always non-negative, and since $\lambda > 1$, the above must imply that:

$$\|\vec{b} - \vec{a}\| = 0.$$

As a result, $\vec{a} = \vec{b}$, and uniqueness follows. ■

11. Bibliographic Remarks

The material in this chapter is classical and is easily more than a hundred years old. Consequently, all of it can be found in any standard text on mathematical analysis. The results that we presented will be needed in subsequent chapters.

Gaughan [49] and Douglass [42] cover most of the topics dealt with here. The fixed point theorem can be found in Rudin [147].

12. Review Questions

- Consider the following vectors: $\vec{a} = (16, 19, 13)$, $\vec{b} = (11, 14, 9)$, and $\vec{c} = (10, 15, 12)$. Find the max norm and the Euclidean norm of each of the following vectors:

$$\vec{a}, \vec{a} - \vec{b}, \text{ and } \vec{a} + \vec{b} - \vec{c}.$$

- If a sequence is convergent, it is bounded. Is the converse generally true?
- Give an example of an increasing sequence that is bounded above.
- Define the accumulation point of a set.
- Prove that 5 is an accumulation point of $(4, 8)$ and not an accumulation point of $(1, 4)$.
- Define the convergence of a sequence to a finite point. Prove that the sequence:

$$\left\{ \frac{1}{p} + 5 \right\}_{p=1}^{\infty}$$

converges to 5, identifying the value of ϵ that will satisfy the definition.

- Consider the sequence:

$$\{1, 2, 1, 2, 1, 2, \dots\}.$$

What is the range of this sequence? Is it convergent?

- The following transformation is known to be contractive. Identify its fixed point.

$$x^{k+1} = 5 + 0.3x^k + 0.1y^k.$$

$$y^{k+1} = 12 + 0.2x^k + 0.4y^k.$$

- Prove Equation (11.6) from Definition 11.8, using induction.

Chapter 12

CONVERGENCE ANALYSIS OF PARAMETRIC OPTIMIZATION METHODS

It is truth very certain that, when it is not in our power to determine what is true, we ought to follow what is most probable.

— René Descartes (1596-1650)

1. Chapter Overview

This chapter deals with some simple convergence results related to the parametric optimization methods discussed in Chapter 7. The main idea underlying “convergence analysis” of a method is to identify (mathematically) the solution to which the method converges. Hence to prove that an algorithm works, one must show that the algorithm converges to the optimal solution. In this chapter, this is precisely what we will attempt to do with some algorithms of Chapter 7.

The convergence of simulated annealing requires some understanding of Markov chains and transition probabilities. To this end, it is sufficient to read all sections of Chapter 8 up to and including Section 3.3. It is also necessary to read about convergent sequences. For this purpose, it is sufficient to read all the material in Chapter 11 up to and including Theorem 11.2. Otherwise, all that is needed to read this chapter is a basic familiarity with the material of Chapter 7.

Our discussion on the analysis of the gradient-descent rule begins in Section 3. Before plunging into the mathematical details of the convergence analysis of gradient descent, we need definitions of some elementary ideas from calculus and a simple theorem.

2. Some Definitions and a result

In this section, we will define the following concepts.

1. Continuous functions
2. Partial derivatives
3. Continuously differentiable functions
4. Stationary points, local optima, and global optima
5. Taylor series.

2.1. Continuous Functions

DEFINITION 12.1 A function $f(\vec{x})$ defined as $f : \mathcal{D} \rightarrow \mathcal{R}$ is said to be continuous on the set \mathcal{D} if:

$$\lim_{\vec{x} \rightarrow \vec{c}} f(\vec{x}) = f(\vec{c})$$

for every $\vec{c} \in \mathcal{D}$.

Let us illustrate this idea with the following example function from \mathcal{R} to \mathcal{R} :

$$f(x) = 63x^2 + 5x.$$

Note that for any $c \in \mathcal{R}$, we have that:

$$\lim_{x \rightarrow c} f(x) = \lim_{x \rightarrow c} (63x^2 + 5x) = 63c^2 + 5c = f(c),$$

and hence the definition of continuity is satisfied.

Now consider the following example. A function $f : \mathcal{R} \rightarrow \mathcal{R}$ is defined as:

$$f(x) = \frac{x^2 - 5x + 6}{x^2 - 6x + 8} \text{ when } x \neq 2$$

and

$$f(x) = 90 \text{ when } x = 2.$$

Now, at $x \neq 2$, we have:

$$\lim_{x \rightarrow 2} f(x) = \lim_{x \rightarrow 2} \frac{x^2 - 5x + 6}{x^2 - 6x + 8} = \lim_{x \rightarrow 2} \frac{(x-2)(x-3)}{(x-2)(x-4)} = \lim_{x \rightarrow 2} \frac{x-3}{x-4} = \frac{1}{2} \neq 90.$$

This implies, from the definition above, that the function is not continuous at $x = 2$, and hence the function is not continuous on \mathcal{R} .

2.2. Partial derivatives

DEFINITION 12.2 *The partial derivative of a function of multiple variables $(x(1), x(2), \dots, x(k))$ with respect to the i th variable, $x(i)$ is defined as*

$$\frac{\partial f(x(1), x(2), \dots, x(k))}{\partial x(i)} \equiv$$

$$\lim_{h \rightarrow 0} \frac{f(x(1), x(2), \dots, x(i) + h, \dots + x(k)) - f(x(1), x(2), \dots, x(k))}{h}.$$

The partial derivative defined here is actually the *first* partial derivative. It is likely that you are familiar with this concept from a preliminary college calculus course.

2.3. A continuously differentiable function

DEFINITION 12.3 *A function $f : \mathcal{D} \rightarrow \mathcal{R}$ is said to be continuously differentiable if each of its partial derivatives is a continuous function on the domain (\mathcal{D}) of the function f .*

Example: Consider the function

$$f(x, y) = 5x^2 + 4xy + y^3.$$

Then,

$$\frac{\partial f(x, y)}{\partial x} = 10x + 4y, \text{ and}$$

$$\frac{\partial f(x, y)}{\partial y} = 4x + 3y^2.$$

It is not hard to show that both partial derivatives are continuous functions. Hence, the function $f(x, y)$ must be continuously differentiable. It is by no means easy to prove that this condition holds for objective functions in simulation optimization — where the closed form of the objective function is not available.

2.4. Stationary points, local optima, and global optima

DEFINITION 12.4 *A stationary point of a function is the point (i.e., vector) at which the first partial derivative has a value of 0.*

Hence if the function is denoted by $f(x(1), x(2), \dots, x(k))$, then the stationary point can be determined by solving the following system of linear equations (composed of k equations):

$$\frac{\partial f(x(1), x(2), \dots, x(k))}{\partial x(j)} = 0,$$

for $j = 1, 2, \dots, k$.

DEFINITION 12.5 A local minimum of a function $f : \mathcal{R}^k \rightarrow \mathcal{R}$ is a point (i.e., vector), \vec{x}^* , which is no worse than its neighbors; that is, there exists an $\epsilon > 0$ such that:

$$f(\vec{x}^*) \leq f(\vec{x}), \quad (12.1)$$

for all $\vec{x} \in \mathcal{R}^k$ satisfying the following property:

$$\|\vec{x} - \vec{x}^*\| < \epsilon, \text{ where } \|\cdot\| \text{ denotes any norm.}$$

Note that the definition does not hold for any ϵ , but says that there exists an ϵ that satisfies the condition above.

DEFINITION 12.6 If the condition in (12.1) of the previous definition is satisfied with a strict inequality, that is, if:

$$f(\vec{x}^*) < f(\vec{x}),$$

we have what is called a strict local minimum of the function f . When the definition is satisfied with an equality, what we have is still a local minimum, but this one is not a strict local minimum. If for every neighborhood of \vec{x} that does not include \vec{x} ,

$$f(\vec{x}^*) - f(\vec{x}),$$

assumes both negative and positive values, \vec{x} is called a saddle point of the function f .

DEFINITION 12.7 A global minimum for a function $f : \mathcal{R}^k \rightarrow \mathcal{R}$ is a point (i.e., vector) which is no worse than all other points in the domain of the function; that is,

$$f(\vec{x}^*) \leq f(\vec{x}),$$

for all $\vec{x} \in \mathcal{R}^k$.

This means that the global minimum is the minimum of all the local minima.

The definitions for local and global *maxima* can be similarly formulated. See Figure 12.1 to get a geometric feel for strict local optima, and saddle points. See Figure 12.2 for understanding the difference between local and global optima.

2.5. Taylor's theorem

The Taylor's theorem is a well-known result (see Kreyszig [97] or any other elementary mathematics text). It shows that a function can be expressed as an infinite series involving derivatives if derivatives exist and if some other conditions hold. We present it next, without proof.

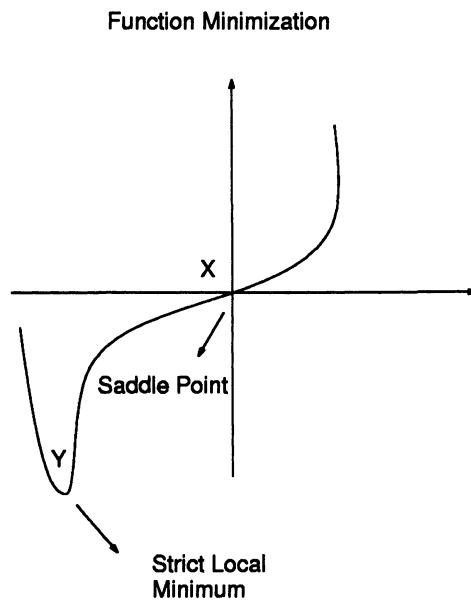
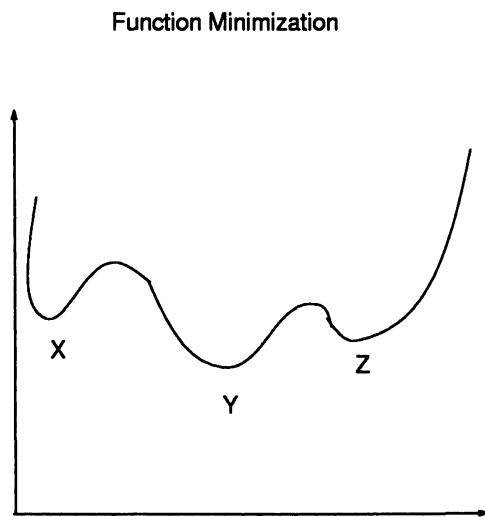


Figure 12.1. Strict local optima and saddle points



X, Y, and Z are local optima. Y is the global optimum

Figure 12.2. Local optima and global optima

THEOREM 12.1 (Taylor Series with one variable) A function $f(x + h)$ can be expressed as follows:

$$f(x + h) = f(x) + h \frac{df(x)}{dx} + \frac{(h)^2}{2!} \frac{d^2 f(x)}{dx^2} + \dots + \frac{(h)^n}{n!} \frac{d^n f(x)}{dx^n} + \dots \quad (12.2)$$

We will now prove that a local minimum is a stationary point in the single variable case. The result can be easily extended to multiple variables, and can also be shown for the local maximum case.

THEOREM 12.2 A local minimum x^* of a function $f(x)$ is a stationary point — that is:

$$\frac{df(x)}{dx}|_{x=x^*} = 0.$$

Proof If $|h|$ in the Taylor's series is a small quantity, one can ignore terms with h raised to 2 and higher values. Then setting $x = x^*$ in the Taylor's series, and selecting a sufficiently small value for $|h|$, one has that:

$$f(x^* + h) = f(x^*) + h \frac{df(x)}{dx}|_{x=x^*}. \quad (12.3)$$

We will use contradiction logic. Let us assume that x^* is *not* a stationary point. Then,

$$\frac{df(x)}{dx}|_{x=x^*} \neq 0,$$

that is,

$$\text{either } \frac{df(x)}{dx}|_{x=x^*} > 0 \text{ or } \frac{df(x)}{dx}|_{x=x^*} < 0.$$

In either case, by selecting a suitable sign for h , one can always have that:

$$h \frac{df(x)}{dx}|_{x=x^*} < 0.$$

Using the above in Equation (12.3) one has that

$$f(x^* + h) < f(x^*). \quad (12.4)$$

From the definition of a local minimum, we have that there exists an ϵ such that

$$f(x^* \pm \epsilon) \geq f(x^*).$$

If the value of $|h|$ that was selected above satisfies: $|h| < \epsilon$, we have that:

$$f(x^* \pm |h|) \geq f(x^*).$$

This implies that:

$$f(\vec{x}^* + h) \geq f(\vec{x}^*),$$

which contradicts Inequation (12.4). As a result, the local minimum must be stationary point. ■

To prove that a stationary point is indeed a local optimum, one has to establish some conditions related to the second derivatives of the function — which we do not discuss. These conditions are not very useful in simulation-based optimization.

It needs to be understood that a *stationary* point may be local optimum or it may be a global optimum. In other words, just because a point is a local optimum, we have no guarantee that the point is a global optimum. To ascertain whether a local optimum is also a global optimum, one needs to establish the so-called convexity properties for the function. Most of these conditions — including convexity — are hard to verify in simulation-based optimization because the closed form of the function is unknown. As such, we will content ourselves with analysis related to the first partial derivative. We will be also assuming — somewhat arbitrarily — that the first partial derivatives exist.

In summary, our analysis of gradient (derivative) methods will be restricted to showing that our algorithms can reach *stationary* points in the function. We will have to hope that by searching a large number of these points, the algorithm is able to identify the *global* optimum.

It is however very important to understand that if the function does *not* possess the first partial derivatives, the algorithm may not converge, and even if the first partial derivatives do exist, we cannot be sure that the local optimum obtained is a global optimum without convexity arguments.

3. Convergence of gradient-descent approaches

In this book, the principle of gradient descent was discussed in the contexts of neural networks and simultaneous perturbation. We will now prove that the gradient-descent rule with a *constant* step size converges to a stationary point under certain conditions. We need some notation to introduce this convergence result.

- The main transformation in gradient descent is:

$$\vec{x}^{m+1}(i) = \vec{x}^m(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)} \Big|_{\vec{x}=\vec{x}^m} \text{ for } i = 1, 2, \dots, k \quad (12.5)$$

where k is the number of decision variables (parameters) and μ is the value of the step size.

- We will need the following column vector in the proof.

$$\nabla f(\vec{x}) = \begin{bmatrix} \frac{\partial f(\vec{x})}{\partial x(1)} \\ \frac{\partial f(\vec{x})}{\partial x(2)} \\ \vdots \\ \frac{\partial f(\vec{x})}{\partial x(k)} \end{bmatrix} \quad (12.6)$$

We will be using the following notation, more frequently, to save space.

$$\nabla f(\vec{x}) = \left[\frac{\partial f(\vec{x})}{\partial x(1)} \quad \frac{\partial f(\vec{x})}{\partial x(2)} \quad \dots \quad \frac{\partial f(\vec{x})}{\partial x(k)} \right]^T.$$

The following result analyzes the convergence of gradient descent for constant step sizes.

THEOREM 12.3 *Let \vec{x}^m denote the vector of values (of the parameters) in the m th iteration of the gradient-descent approach defined in Equation (12.5).*

Let us assume that the function $f(.)$ is continuously differentiable and that it satisfies the following condition:

$$\|\nabla f(\vec{a}_1) - \nabla f(\vec{a}_2)\| \leq L\|\vec{a}_1 - \vec{a}_2\|, \quad \forall \vec{a}_1, \vec{a}_2 \in \mathcal{R}^k, \quad (12.7)$$

for some finite $L > 0$. This is called the Lipschitz condition.

Now, if $f(.)$ is bounded below, then for a sufficiently small value for μ in Equation (12.5),

$$\lim_{m \rightarrow \infty} \nabla f(\vec{x}^m) = \vec{0}.$$

The above result basically says that in the limit, the algorithm will converge to a point in which the gradient is 0, that is, a stationary point.

Proof In the proof, we will use the Euclidean norm. So $\|\cdot\|$ will mean $\|\cdot\|_E$. Consider two vectors \vec{x} and \vec{z} , and a scalar quantity ζ . Let

$$g(\zeta) = f(\vec{x} + \zeta\vec{z}).$$

Then, from the chain rule, one has that:

$$\frac{dg(\zeta)}{d\zeta} = [\vec{z}]^T \nabla f(\vec{x} + \zeta\vec{z}). \quad (12.8)$$

We will use this below. We have

$$\begin{aligned}
 f(\vec{x} + \vec{z}) - f(\vec{x}) &= g(1) - g(0) \text{ (follows from the definition of } g(\zeta)). \\
 &= \int_0^1 dg(\zeta) \\
 &= \int_0^1 \frac{dg(\zeta)}{d\zeta} d\zeta \\
 &= \int_0^1 [\vec{z}]^T \nabla f(\vec{x} + \zeta \vec{z}) d\zeta \text{ from (12.8)} \\
 &\leq \int_0^1 [\vec{z}]^T \nabla f(\vec{x}) d\zeta + \left| \int_0^1 \vec{z}^T (\nabla f(\vec{x} + \zeta \vec{z}) - \nabla f(\vec{x})) d\zeta \right| \\
 &\leq \int_0^1 [\vec{z}]^T \nabla f(\vec{x}) d\zeta + \int_0^1 \|\vec{z}\| \cdot \|\nabla f(\vec{x} + \zeta \vec{z}) - \nabla f(\vec{x})\| d\zeta \\
 &\leq [\vec{z}]^T \nabla f(\vec{x}) \int_0^1 d\zeta + \|\vec{z}\| \int_0^1 L\zeta \|\vec{z}\| d\zeta \text{ from (12.7)} \\
 &= [\vec{z}]^T \nabla f(\vec{x}) \int_0^1 d\zeta + L\|\vec{z}\|^2 \int_0^1 \zeta d\zeta \\
 &= [\vec{z}]^T \nabla f(\vec{x}) \cdot 1 + L\|\vec{z}\|^2 \cdot \frac{1}{2}
 \end{aligned}$$

In the above, setting $\vec{x} = \vec{x}^m$ and $\vec{z} = -\mu \nabla f(\vec{x}^m)$, we obtain:

$$f(\vec{x}^m - \mu \nabla f(\vec{x}^m)) - f(\vec{x}^m) \leq -\mu [\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) + \frac{1}{2} \mu^2 L \|\nabla f(\vec{x}^m)\|^2.$$

The above can be written as:

$$\begin{aligned}
 f(\vec{x}^m) - f(\vec{x}^m - \mu \nabla f(\vec{x}^m)) &\geq \mu [\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) - \frac{1}{2} \mu^2 L \|\nabla f(\vec{x}^m)\|^2 \\
 &= \mu \|\nabla f(\vec{x}^m)\|^2 - \frac{1}{2} \mu^2 L \|\nabla f(\vec{x}^m)\|^2 \quad (12.9)
 \end{aligned}$$

$$= \frac{\mu L}{2} \left(\frac{2}{L} - \mu \right) \|\nabla f(\vec{x}^m)\|^2 \quad (12.10)$$

$$\geq 0 \text{ if } \mu < 2/L. \quad (12.11)$$

In the above, (12.9) follows from the fact that

$$[\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) = \|\nabla f(\vec{x}^m)\|^2.$$

(Remember, we are using the Euclidean norm here.)

From Inequation (12.11), it follows that if μ is sufficiently small (that is less than $2/L$),

$$f(\vec{x}^m) - f(\vec{x}^{m+1}) \geq 0$$

for any m . In other words, the values of the objective function $f(\vec{x}^m)$ for $m = 1, 2, 3, \dots$ form a decreasing sequence. A decreasing sequence that is bounded below converges (see Theorem 11.2 from Chapter 11) to a finite number. Hence:

$$\lim_{m \rightarrow \infty} [f(\vec{x}^m) - f(\vec{x}^{m+1})] = 0.$$

From the above and (12.10), we have that

$$\lim_{m \rightarrow \infty} \frac{\mu L}{2} \left(\frac{2}{L} - \mu \right) \|\nabla f(\vec{x}^m)\|^2 \leq 0.$$

In the above, all the quantities in the left hand side are ≥ 0 . Consequently,

$$\lim_{m \rightarrow \infty} \nabla f(\vec{x}^m) = \vec{0}. \quad \blacksquare$$

With some more work, this proof can be extended to the decreasing step size case. We refer you to Chapter 3 of [19] for the details. We need to make some important comments on the material presented above.

- By **gradient descent**, we refer to what is often known as **steepest descent** in the literature. However, many authors use the term “gradient descent” in a *general* sense to mean any method that uses some form of the function gradient and not just the steepest-descent method.
- The gradient-descent rule analyzed above is used in deriving neural network rules such as the Widrow-Hoff rule and the backpropagation rule that was discussed in Chapter 6 in the context of neural networks.
- Simulation optimization is usually used on those problems whose closed form is unknown. Unfortunately, the Lipschitz condition (12.7) is difficult to verify when the closed form of the objective function is unknown.
- When an error is introduced into the gradient value — by simulation noise or otherwise — it can be shown that the algorithm *oscillates* in the vicinity of the stationary point. How close it goes to the stationary point depends on the magnitude of the error. Bertsekas [18] discusses these issues in some detail.

Standard step size conditions. A number of gradient-descent algorithms require a standard set of conditions that we will present below. These conditions are often useful when noise is present in the function estimate or the derivative estimate. If μ^m denotes the step size in the m th iteration of the algorithm, then the conditions are:

$$\sum_{m=1}^{\infty} \mu^m = \infty, \quad \sum_{m=1}^{\infty} [\mu^m]^2 < \infty. \quad (12.12)$$

A step size that follows the following law:

$$\mu^m = \frac{a}{m} \text{ for a strictly positive } a$$

satisfies these conditions.

4. Perturbation Estimates

In this section, we first present an analysis of two types of finite difference perturbation estimates of derivatives. Thereafter, some analysis of the simultaneous perturbation estimate is presented.

4.1. Finite Difference Estimates

We next present a result that shows why the central difference formula (Equation 7.1) yields better results than the forward difference formula (Equation 7.2).

THEOREM 12.4 *The forward difference formula (7.2) ignores terms of the order of h^2 and higher order terms while the central difference formula (7.1) ignores terms of the order of h^3 and higher order terms of h , but not the terms of the order of h^2 .*

Comment: The result implies that the central difference estimates are more accurate.

Proof The forward difference formula assumes that all terms of the order of h^2 and higher orders of h are negligible. If h is small, this is a reasonable assumption, but it produces an error nevertheless.

The central difference formula on the other hand does not neglect terms of the order of h^2 . It neglects the terms of the order of h^3 and higher orders of h .

From the Taylor Series, ignoring terms with h^2 and higher orders of h , we have that:

$$f(x+h) = f(x) + h \frac{df(x)}{dx}.$$

This, after re-arrangement of terms, yields:

$$\frac{df(x)}{dh} = \frac{f(x+h) - f(x)}{h},$$

which of course is the forward difference formula given in Equation (7.2).

Now, from the Taylor series,

$$f(x-h) = f(x) - h \frac{df(x)}{dx} + \frac{(h)^2}{2!} \frac{d^2 f(x)}{dx^2} - \dots + (-1)^n \frac{(h)^n}{n!} \frac{d^n f(x)}{dx^n} + \dots \quad (12.13)$$

Now if we ignore terms of the order of h^3 and higher in both Equation (12.2) and Equation (12.13), then subtracting equation (12.2) from equation (12.13), we have that:

$$f(x + h) - f(x - h) = 2h \frac{df(x)}{dx},$$

which after re-arrangement yields

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x - h)}{2h}.$$

The above is the central differences formula of Equation (7.1). It should be thus clear that the latter was obtained without ignoring terms of the order of h^2 .

■

4.2. Notation

We need to define some of the notation that we will be using in this section. The shorthand notation $f(\vec{x}^m + \vec{h}^m)$ will be used to denote:

$$f(x^m(1) + h^m(1), x^m(2) + h^m(2), \dots, x^m(k) + h^m(k)).$$

$D^m(i)$ will denote the actual partial derivative value of the function under consideration with respect to the i th variable at the m th iteration of the algorithm; the derivative is calculated at:

$$\vec{x} = \vec{x}^m.$$

Thus mathematically:

$$D^m(i) \equiv \frac{\partial f(\vec{x})}{\partial x(i)}|_{\vec{x}=\vec{x}^m}. \quad (12.14)$$

$D_s^m(i)$ will be used to denote the *simultaneous perturbation* estimate of the derivative in the m th iteration of the algorithm; the mathematical definition is:

$$D_s^m(i) \equiv \frac{f(\vec{x}^m + \vec{h}^m) - f(\vec{x}^m - \vec{h}^m)}{2h^m(i)}. \quad (12.15)$$

The assumption in the above is that **exact** values of the function are available.

4.3. Simultaneous Perturbation Estimates

In this subsection, we present a convergence analysis of simultaneous perturbation via Theorem 12.8. We begin by stating a powerful result related to gradient descent. This result will be used in the convergence analysis that will follow.

THEOREM 12.5 *Let us assume that a function $f : \mathcal{R}^k \rightarrow \mathcal{R}$ satisfies the following conditions:*

- $f(\vec{x}) \geq 0$ everywhere, and
- the function $f(\vec{x})$ satisfies the Lipschitz continuous (see Equation 12.7) and is continuously differentiable.

Next, consider the gradient-descent algorithm defined by:

$$\vec{x}^{m+1}(i) = \vec{x}^m(i) - \mu^m \left[\frac{\partial f(\vec{x})}{\partial \vec{x}(i)}|_{\vec{x}=\vec{x}^m} + w^m(i) \right] \text{ for } i = 1, 2, \dots, k$$

where k is the number of decision variables, μ^m is the value of the step size in the m th iteration, and $w^m(i)$ is a noise term. Let the step size satisfy the step size conditions defined in (12.12).

Let us define the history of the algorithm up to and including the m th iteration by the set:

$$\mathcal{F}^m = \{\vec{x}^0, \vec{x}^1, \dots, \vec{x}^m, \vec{D}_s^0, \vec{D}_s^1, \dots, \vec{D}_s^m, \mu^0, \mu^1, \dots, \mu^m\}.$$

If

$$E[w^m(i)|\mathcal{F}^m] = 0 \text{ for every } i \quad (12.16)$$

and

$$E[\|\vec{w}^m\|^2|\mathcal{F}^m] = A + B\|\nabla f(\vec{x}^m)\|^2 \quad (12.17)$$

for some constants A and B , then, with probability 1,

R1. the sequence $\{f(\vec{x}^m)\}_{m=1}^\infty$ converges, and

R2.

$$\lim_{m \rightarrow \infty} \nabla f(\vec{x}^m) = 0.$$

The condition **R2** implies that the algorithm will converge with probability 1 to a stationary point at which the gradient will be 0.

In the above theorem, result **R1** was shown by Poljack and Tsyplkin [136] and result **R2** can be found in Bertsekas and Tsitsiklis [19]. A more general version of this theorem can be found in Proposition 4.1 on page 141 of [19]. The proof of this result requires the martingale and the supermartingale convergence theorems. Since these topics have not been introduced here, we have omitted the proof of Theorem 12.5.

We now need to show that simultaneous perturbation satisfies the conditions of Theorem 12.5. This will be achieved in Theorem 12.7. For the latter, the following elementary result will be needed.

THEOREM 12.6 (Taylor Series for a function of two variables: $x(1)$ and $x(2)$) A function $f(x(1) + h(1), x(2) + h(2))$ can be expressed as follows:

$$\begin{aligned} f(x(1) + h(1), x(2) + h(2)) &= f(x(1), x(2)) + h(1) \frac{\partial f(\vec{x})}{\partial x(1)} + h(2) \frac{\partial f(\vec{x})}{\partial x(2)} \\ &+ \frac{1}{2!} \left[[h(1)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(1)} + 2h(1)h(2) \frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)} + [h(2)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(2)} \right] + \dots \end{aligned} \quad (12.18)$$

where $\vec{x} = (x(1), x(2))$.

The proof of the above can be found in any standard calculus text. We are now at a position to state and prove Theorem 12.7. The proof is from Gosavi [60].

THEOREM 12.7 *The simultaneous perturbation algorithm described in Chapter 7 is of the form described in Theorem 12.5.*

Proof We will first analyze how the simultaneous perturbation estimate differs from the actual derivative. We will assume for the time being that $k = 2$.

From the Taylor series result, that is, Equation (12.18), ignoring terms with h^3 and higher orders of h and suppressing the superscript m , we have that:

$$\begin{aligned} f(x(1) + h(1), x(2) + h(2)) &= f(x(1), x(2)) + h(1) \frac{\partial f(\vec{x})}{\partial x(1)} + h(2) \frac{\partial f(\vec{x})}{\partial x(2)} + \\ &\frac{1}{2!} \left[[h(1)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(1)} + 2h(1)h(2) \frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)} + [h(2)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(2)} \right] \end{aligned} \quad (12.19)$$

From the same Taylor series, we also have that:

$$\begin{aligned} f(x(1) - h(1), x(2) - h(2)) &= f(x(1), x(2)) - h(1) \frac{\partial f(\vec{x})}{\partial x(1)} - h(2) \frac{\partial f(\vec{x})}{\partial x(2)} + \\ &+ \frac{1}{2!} \left[[h(1)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(1)} + 2h(1)h(2) \frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)} + [h(2)]^2 \frac{\partial^2 f(\vec{x})}{\partial x^2(2)} \right]. \end{aligned} \quad (12.20)$$

Subtracting Equation (12.20) from Equation (12.19), we have

$$\begin{aligned} f(x(1) + h(1), x(2) + h(2)) - f(x(1) - h(1), x(2) - h(2)) &= \\ 2h(1) \frac{\partial f}{\partial x(1)} + 2h(2) \frac{\partial f}{\partial x(2)}. \end{aligned}$$

From the above, by re-arranging terms, we have:

$$\frac{f(x(1) + h(1), x(2) + h(2)) - f(x(1) - h(1), x(2) - h(2))}{2h(1)} = \frac{\partial f(\vec{x})}{\partial x(1)} + \frac{h(2)}{h(1)} \frac{\partial f(\vec{x})}{\partial x(2)}.$$

Noting the fact that we had suppressed m in the superscript and from the definitions of $D_s^m(i)$ and $D^m(i)$, the above can be written as

$$D_s^m(1) = D^m(1) + \frac{h^m(2)}{h^m(1)} \frac{\partial f(\vec{x})}{\partial x(2)}. \quad (12.21)$$

Equation (12.21), using the k -variable version of Equation (12.18), can be generalized to:

$$D_s^m(i) = D^m(i) + \sum_{j \neq i; j=1}^k \frac{h^m(j)}{h^m(i)} \frac{\partial f(\vec{x})}{\partial x(j)} \text{ for every } i. \quad (12.22)$$

Let us define the noise term (this is not simulation-induced noise but noise generated by an imperfect estimate of the derivative) as follows:

$$w^m(i) = D_s^m(i) - D^m(i) \text{ for every } i,$$

which means that:

$$w^m(i) = \sum_{j \neq i; j=1}^k \frac{h^m(j)}{h^m(i)} \frac{\partial f(\vec{x})}{\partial x(j)} \text{ for every } i. \quad (12.23)$$

Let us also define the set of the history of the algorithm up to and including the m th iteration by:

$$\mathcal{F}^m = \{\vec{x}^0, \vec{x}^1, \dots, \vec{x}^m, \vec{D}_s^0, \vec{D}_s^1, \dots, \vec{D}_s^m, \mu^0, \mu^1, \dots, \mu^m\}.$$

Now, if the history of the algorithm is known, from the Bernoulli distribution used for \vec{h} (see algorithm description in Chapter 7), it follows that for every j ,

$$E \left[\frac{h^m(j)}{h^m(i)} \frac{\partial f(\vec{x})}{\partial x(j)} | \mathcal{F}^m \right] = [(0.5)(-1) + (0.5)(1)] E \left[\frac{\partial f(\vec{x})}{\partial x(j)} | \mathcal{F}^m \right] = 0.$$

From the above equation and from Equation (12.23), it follows that for every i ,

$$E[w^m(i) | \mathcal{F}^m] = \sum_{j \neq i; j=1}^k 0 = 0,$$

thereby proving condition (12.16) in Theorem 12.5. Condition (12.17) in the same theorem remains to be shown.

We will use the Euclidean norm below. So $\|\cdot\|$ will mean $\|\cdot\|_E$. Remember from the algorithm description in Chapter 7 that for any i, j ,

$$\frac{h(j)}{h(i)} = 1 \text{ or } -1.$$

Hence for any j ,

$$\left[\frac{h(j)}{h(i)} \right]^2 = 1. \quad (12.24)$$

Then it can be shown that:

$$\begin{aligned} \|\vec{w}^m\|^2 &= [w^m(1)]^2 + [w^m(2)]^2 + \cdots + [w^m(k)]^2 \\ &\leq \sum_{j=1}^k \left[\frac{\partial f(\vec{x})}{\partial x(j)} \right]^2 + \sum_{j=1}^k \left[\frac{\partial f(\vec{x})}{\partial x(j)} \right]^2 + \cdots + \\ &\quad \cdots + \sum_{j=1}^k \left[\frac{\partial f(\vec{x})}{\partial x(j)} \right]^2 \end{aligned} \quad (12.25)$$

$$= k \sum_{j=1}^k \left[\frac{\partial f(\vec{x})}{\partial x(j)} \right]^2$$

$$= k \|\nabla f(\vec{x}^m)\|^2 \quad (12.26)$$

$$= 0 + k \|\nabla f(\vec{x}^m)\|^2$$

In the above, line (12.25) uses Equations (12.23) and (12.24), while line (12.26) follows from the notation definition in Equation (12.6). The above implies that Condition (12.17) in Theorem 12.5 is satisfied. ■

In the proof above, let us reiterate, the noise term did *not* represent the noise induced by simulation. It denoted the noise in the derivative due to Spall's formula. Remember that Spall's formula does not give the exact derivative. And now, finally, we come to the main result of this subsection.

THEOREM 12.8 *The simultaneous perturbation algorithm, as described in Chapter 7, converges to a stationary point of the objective function, $f(\vec{x})$, if (i) exact function values are used in the algorithm, (ii) the objective function satisfies the Lipschitz condition (12.7), (iii) the objective function is continuously differentiable, (iv) $f(\vec{x}) \geq 0$ everywhere, and (v) the step size is made to satisfy the conditions defined in (12.12).*

Proof This result follows from a combination of Theorems 12.5 and 12.7. ■

The above result is shown under the following conditions 1) exact (noise-free) function values are available, and 2) the objective function satisfies the Lipschitz condition. Both conditions are somewhat unrealistic *in the context of simulation optimization*. This is because the first condition rarely applies in this context and the second condition is difficult to verify. For a more general result, the reader is referred to Spall [162]. Understanding Spall's proof requires the martingale convergence theorem, which is beyond the scope of our treatment, and is hence omitted.

The cleverness of Spall's algorithm lies in that it essentially exploits the fact that gradient descent *can* be performed even when the derivative is not calculated *perfectly*. This paves the way for a very efficient mechanism to do gradient descent in which only two function evaluations are needed per iteration, regardless of the number of parameters in the problem. Our analysis demonstrates this under some simplifying assumptions. As stated above, for a more thorough analysis, the reader should read the proof supplied in Spall [162].

5. Convergence of Simulated Annealing

The convergence of simulated annealing was first studied by Lundy and Mees [106] in a paper that has considerable historical significance because almost all convergence related papers on simulated annealing that followed have basically used the idea used in their paper; the idea is to model the optimization process as a Markov process. Mathematically, the convergence proof of the algorithm is as attractive as the algorithm itself.

At the very outset, it is perhaps prudent to state that in optimization problems having a finite solution space, a proof that guarantees convergence when the number of iterations tends to infinity does not necessarily prove that the time taken by the algorithm to find the optimal solution is less than the time that would be taken by an algorithm that does *exhaustive search*. (An exhaustive search algorithm searches every point in the solution space.) It is clear that without this assurance, the algorithm may not be useful at all, since one may be better off with an exhaustive search.

One way to determine whether an algorithm is superior to exhaustive search (complete enumeration) is via a *theoretical* study of the *rate* of convergence. The other method is empirical — it calculates the ratio of the number of iterations needed and the total number of solutions in the solution space. If this ratio is small, empirically, the algorithm can be said to have done "well" on that particular problem. For many meta-heuristics, such studies have been carried out on specific problems. Of course, it is wrong to generalize from empirical studies, and hence an empirical method — such as this — enjoys limited significance. This issue is very important; however, a discussion on convergence rates is beyond the scope of this text.

The scheme used in showing that simulated annealing can converge to an optimal solution can be described as follows. Assume that each solution in the solution space is a state in a Markov chain. Then the optimization process of moving from one solution to another can be viewed as a *transition* in the Markov chain. The goal is to show that the limiting probability of the optimal state — that is, the optimal solution (at least one of the optimal solutions when multiple optimal solutions exist) is 1. Now, we will see below that associated with every temperature, one has a unique Markov chain. In other words, when the temperature changes, one has a new Markov chain. Hence our goal in the convergence analysis is to show that the Markov chains at low values of the temperature have values approaching 1 for the limiting probability of the optimal state.

For the convergence proof, we need a simple result, which is often called the time reversibility result of a Markov chain.

LEMMA 12.9 *Consider a regular Markov chain. Let the transition probability of jumping from state i to state j in the chain be denoted by $P(i, j)$. If a vector \vec{y} satisfies the following relations:*

$$\sum_i y(i) = 1$$

and

$$y(i)P(i, j) = y(j)P(j, i) \quad (12.27)$$

for all i, j , then \vec{y} is the limiting probability vector of the Markov chain.

Proof Equation (12.27), when summed over i , yields

$$\begin{aligned} \sum_i y(i)P(i, j) &= y(j) \sum_i P(j, i) \\ &= y(j)1, \end{aligned}$$

for any j , which implies that

$$\sum_i y(i)P(i, j) = y(j), \quad \sum_i y(i) = 1$$

for every j .

From Markov chain theory, it is known that the unique solution of the above is the limiting probability vector of the Markov chain. ■

Before stating the main result of this section, we need to introduce the concept of symmetric neighborhood generation schemes.

Symmetric neighborhood generation schemes. We assume that the neighborhood generation scheme is symmetric if the following is true: if a point \vec{y} is a neighbor of a point \vec{x} , then \vec{x} must be an *equally likely* neighbor of \vec{y} . This implies that if the algorithm generates \vec{y} as a neighbor of \vec{x} with a probability of q , then \vec{x} is generated as a neighbor of \vec{y} with an **equal** probability (q).

We now state the main result related to the convergence of simulated annealing.

PROPOSITION 12.10 *Let $\vec{x}^{(m,T)}$ denote the simulated annealing iterate vector in the m th iteration of the phase in which the temperature is T . The algorithm is used to minimize the value of a function of \vec{x} .*

Now, if the neighborhood generation scheme is symmetric and if \vec{x}_ denotes the optimal solution, then:*

$$\lim_{m \rightarrow \infty; T \rightarrow 0} \vec{x}^{(m,T)} = \vec{x}_*.$$

The proposition implies that the algorithm will find the optimal solution in the limit.

Proof In the proof, at any given temperature, the optimization process of moving from one solution to another is modeled as a regular Markov chain. When the algorithm is at a given point, its next point is independent of where it has been in the past. As a result, this Markov modeling is justified. The transition probability of going from point i to point j will be denoted by $P(i, j)$.

Let \mathcal{X} denote the set of states (solutions). We assume the number of states (solution points) to be finite and that the states are numbered as follows:

$$1, 2, 3, \dots, |\mathcal{X}|$$

such that

$$i < j \text{ if } f(\vec{x}_i) < f(\vec{x}_j).$$

Using this ordering,

$$f(\vec{x}_1) = f(\vec{x}_*).$$

In solution \vec{x}_i , the algorithm generates a neighbor \vec{x}_j randomly. We denote the probability of generating this neighbor by $g(i, j)$. This probability depends on the neighborhood selection strategy and its exact value is not needed in the proof. However, from the assumption of symmetric neighborhoods in the statement of the result,

$$g(i, j) = g(j, i) \text{ for all } i, j.$$

Now let us denote the probability of accepting a neighbor \vec{x}_j , when in solution \vec{x}_i , by $a(i, j)$. Then

$$a(i, j) = 1 \text{ for all } i \geq j.$$

This is so because the algorithm accepts a better solution with probability 1. Also,

$$\text{if } i < j, \quad 0 < a(i, j) < 1.$$

From the above definitions, it follows that:

$$P(i, j) = g(i, j)a(i, j) \text{ for all } i, j. \quad (12.28)$$

The above follows from the elementary product rule of probabilities, which says that

$$P(a \cap g) = P(g)P(a|g).$$

We need to use a slightly enhanced notation for $a(i, j)$, the notation being: $a_T(i, a)$, where T is the temperature on which $a(i, j)$ depends. In the algorithm, this probability is defined in the following manner:

$$a_T(i, j) = \begin{cases} \exp\left(\frac{f(\vec{x}_j) - f(\vec{x}_i)}{T}\right) & \text{if } f(\vec{x}_j) > f(\vec{x}_i), \\ 1 & \text{if } f(\vec{x}_j) \leq f(\vec{x}_i). \end{cases} \quad (12.29)$$

From the above, it is not hard to show that if $p < j < i$

$$\begin{aligned} a_T(p, j)a_T(j, i) &= \exp\left(\frac{f(\vec{x}_j) - f(\vec{x}_p)}{T}\right) \exp\left(\frac{f(\vec{x}_i) - f(\vec{x}_j)}{T}\right) \\ &= \exp\left(\frac{f(\vec{x}_i) - f(\vec{x}_p)}{T}\right) \\ &= a_T(p, i). \end{aligned}$$

Thus, from the above, we can conclude that when $p < j < i$,

$$a_T(p, j)a_T(j, i) = a_T(p, i). \quad (12.30)$$

We now define a vector in the following manner:

$$\vec{\pi}_T = (l, l a_T(1, 2), l a_T(1, 3), \dots, l a_T(1, |\mathcal{X}|)), \quad (12.31)$$

and then claim that this vector is the limiting probability vector of the Markov chain, whose transition probability from state i to state j was defined as $P(i, j)$. To prove this claim, some work is needed, which we present below. The subscript T will be dropped from the notation for the sake of clarity. When $1 < j < i$,

$$\begin{aligned} (\pi(i))(P(i, j)) &= (l a(1, i))(a(i, j)g(i, j)) \\ &\quad (\text{from definitions in Equations (12.31), (12.28)}) \end{aligned}$$

$$\begin{aligned}
&= l a(1, i) g(i, j) \text{ (since } a(i, j) = 1 \text{ when } j < i) \\
&= l a(1, j) a(j, i) g(i, j) \\
&\quad \text{(by setting } p = 1 \text{ in Equation (12.30))} \\
&= (l a(1, j))(a(j, i) g(j, i)) \\
&\quad \text{(from the assumption of symmetric neighborhoods)} \\
&= \pi(j) P(j, i) \\
&\quad \text{(from definitions in Equations (12.31), (12.28)).}
\end{aligned}$$

Thus, from the above, one can conclude that when $1 < j < i$,

$$\pi(i) P(i, j) = \pi(j) P(j, i).$$

In a similar manner, the same result can be shown for the cases

$$1 < i < j \text{ and } i = j.$$

After reintroducing the dropped subscript in the notation, from the above, we have that

$$\pi_T(i) P(i, j) = \pi_T(j) P(j, i) \text{ is true for all values of } i, j.$$

Then from Lemma 12.9, we have that the vector $\vec{\pi}_T$ is the limiting probability vector of the Markov chain.

Now it follows from the definition of $a_T(i, j)$ in Equation (12.29) that

$$\lim_{T \rightarrow 0} a_T(1, j) = 0 \text{ for } j \neq 1. \quad (12.32)$$

Now from Equation (12.31)

$$\begin{aligned}
\lim_{T \rightarrow 0} \vec{\pi}_T &= (l, l \lim_{T \rightarrow 0} a_T(1, 2), l \lim_{T \rightarrow 0} a_T(1, 3), \dots, l \lim_{T \rightarrow 0} a_T(1, |\mathcal{X}|)) \\
&= (l, 0, 0, \dots, 0) \text{ (using Equation (12.32))} \\
&= (1, 0, 0, \dots, 0) \text{ (since limiting probabilities sum to 1).}
\end{aligned}$$

Thus, the Markov chain ends up in the limiting state (solution) (that is when $m \rightarrow \infty$) numbered 1 in the limit (that is $T \rightarrow 0$). From our numbering scheme, this is the **optimal** solution. Thus

$$\lim_{m \rightarrow \infty; T \rightarrow 0} \vec{x}^{(m, T)} = \vec{x}_*. \quad \blacksquare$$

An important observation is that the proof ensures the algorithm will drive the system into the optimal solution only as long as a *large* number of iterations are performed at *small* values of T . The reason for this observation is that

the optimum is obtained when $T \rightarrow 0$ (small values of T) and the Markov chain enters its limiting state (large number of iterations in a phase). Thus, it makes sense to perform an increasing number of iterations with every phase as the temperature falls with every phase. Some other important observations are made below.

1. The assumption of symmetric neighborhoods can be relaxed to obtain a slightly different result (see [106].).
2. Our analysis does not touch several important and difficult issues related to *convergence rates*, *temperature reduction schedules*, and *the need* for reduction of temperature. Calculating convergence rates, as stated previously, can often determine whether an algorithm takes less time than exhaustive search. Needless to say, this is an important topic. The bibliographic remarks at the end of this chapter contain references which deal with these issues.
3. The convergence we studied here is often referred to as convergence with **homogeneous** or **stationary** Markov chains. This is because we assume that at any temperature, we have a stationary Markov chain whose transition probabilities do not change with time and that at each temperature a sufficient number of iterations are allowed to occur so that the chances of finding the global optimum are increased. However, several researchers have advocated algorithms in which the number of iterations in one phase is 1. When this happens, one can model the system with a Markov chain whose transition probabilities change with every iteration. This algorithm can be called a non-homogeneous (or non-stationary) algorithm. The analysis gets more complicated as one must now turn to the theory of non-stationary Markov chains. We must point out that there is a rich literature (see bibliographic remarks) that deals with the use of non-stationary Markov chains in proving convergence of such algorithms.

The effect of simulation-induced noise . The simulation-induced noise produces an undeniable effect on the simulated annealing algorithm and as such it is important to study it. We will investigate whether the effect of noise can be brought down to an extent where it makes no difference in the algorithm's progress. To this end, we present the following result from Gosavi [60].

PROPOSITION 12.11 *With probability 1, the version of simulated annealing algorithm that uses Simulation-based estimates of the function can be made to mimic the version that uses exact function values with probability 1.*

Proof To prove this result, one needs to take a critical look at Step 3 of the algorithm. It is here that things can go wrong, that is, the noise can cause the

algorithm to stray from its designed path. Two possible things can happen in Step 3 (see algorithm steps given in Chapter 7), which are as follows: Case 1 in which $\Delta \leq 0$ and Case 2 in which $\Delta > 0$.

Let us consider Case 1 first.

Case 1: Let us denote the simulation estimate of the function at \vec{x} by $\tilde{f}(\vec{x})$ and the exact value by $f(\vec{x})$. Thus:

$$\tilde{f}(\vec{x}) = f(\vec{x}) + \eta$$

where η denotes the noise that can be positive or negative. Then we can write:

$$\tilde{f}(\vec{x}_{\text{current}}) = f(\vec{x}_{\text{current}}) + \eta_{\text{current}}$$

and

$$\tilde{f}(\vec{x}_{\text{new}}) = f(\vec{x}_{\text{new}}) + \eta_{\text{new}}.$$

In Step 3, for Case 1, we are okay as long as

$$\tilde{f}(\vec{x}_{\text{new}}) - \tilde{f}(\vec{x}_{\text{current}}) \leq 0. \quad (12.33)$$

Now, if

$$\eta_1 = |\eta_{\text{new}}| \text{ and } \eta_2 = |\eta_{\text{current}}|$$

then we have four scenarios:

Scenario 1.

$$\tilde{f}(\vec{x}_{\text{new}}) = f(\vec{x}_{\text{new}}) + \eta_1 \text{ and } \tilde{f}(\vec{x}_{\text{current}}) = f(\vec{x}_{\text{current}}) + \eta_2$$

Scenario 2.

$$\tilde{f}(\vec{x}_{\text{new}}) = f(\vec{x}_{\text{new}}) + \eta_1 \text{ and } \tilde{f}(\vec{x}_{\text{current}}) = f(\vec{x}_{\text{current}}) - \eta_2$$

Scenario 3.

$$\tilde{f}(\vec{x}_{\text{new}}) = f(\vec{x}_{\text{new}}) - \eta_1 \text{ and } \tilde{f}(\vec{x}_{\text{current}}) = f(\vec{x}_{\text{current}}) + \eta_2$$

Scenario 4.

$$\tilde{f}(\vec{x}_{\text{new}}) = f(\vec{x}_{\text{new}}) - \eta_1 \text{ and } \tilde{f}(\vec{x}_{\text{current}}) = f(\vec{x}_{\text{current}}) - \eta_2$$

Now let us assume that

$$\eta_1 < -\frac{\Delta}{2} \text{ and } \eta_2 < -\frac{\Delta}{2}. \quad (12.34)$$

We will shortly identify conditions that can enforce this assumption.

To prove that our result holds for Case 1, we need to show that Inequation (12.33) is satisfied. This is what we set out to do next.

Let us consider Scenario 1. What we are about to show can be shown for each of the other scenarios.

$$\begin{aligned}
 \tilde{f}(\vec{x}_{\text{new}}) - \tilde{f}(\vec{x}_{\text{current}}) &= f(\vec{x}_{\text{new}}) - f(\vec{x}_{\text{current}}) + \eta_1 - \eta_2 \\
 &\quad (\text{from scenario 1}) \\
 &= \Delta + \eta_1 - \eta_2 \\
 &\leq \Delta - \frac{\Delta}{2} - \eta_2 \quad (\text{from Inequation (12.34)}) \\
 &= \frac{\Delta}{2} - \eta_2 \\
 &\leq 0 \quad (\text{from Inequation (12.34)})
 \end{aligned}$$

From the above, our claim in Inequation (12.33) holds for Scenario 1. The claim, as stated above, can be proved for other scenarios similarly.

What remains to be shown is how Inequation (12.34) can be satisfied. From the strong law of large numbers (see Theorem 3.1), η_1 and η_2 can be made arbitrarily small — that is, for a given value of $\epsilon > 0$, a sufficiently large number of replications (samples) can be selected such that with probability 1,

$$\eta_1 < \epsilon, \quad \eta_2 < \epsilon.$$

By choosing

$$\epsilon = -\frac{\Delta}{2}$$

we have that the claim in Inequation (12.34) is true. Thus the result follows for Case 1.

Case 2:

Using arguments very similar to those used above, we can show that by selecting a suitable number of replications, we can ensure that:

$$\tilde{f}(\vec{x}_{\text{new}}) - \tilde{f}(\vec{x}_{\text{current}}) > 0 \quad (12.35)$$

when $\Delta > 0$. What remains to be seen is how the probability of selecting a worse neighbor is affected and what happens to it in the limit. Remember, it must converge to 0 as $m \rightarrow \infty$ in the noise-free version. In the noisy version, we have to show the same behavior.

The probability expression (U) corrupted by noise can be

$$\exp\left(\frac{\tilde{f}(\vec{x}_{\text{new}}) - \tilde{f}(\vec{x}_{\text{current}})}{T}\right). \quad (12.36)$$

From Inequation (12.35), the numerator in the power of the exponential term will always be strictly positive. As a result,

$$\lim_{T \rightarrow 0} \exp\left(\frac{\tilde{f}(\vec{x}_{\text{new}}) - \tilde{f}(\vec{x}_{\text{current}})}{T}\right) = 0.$$

The above proves that the limiting behavior of U in the noisy version will be identical to that in the noise-free version. ■

6. Concluding Remarks

The principle of using numerical methods of optimization in combination with *simulation-based* function values, rather than using the exact function values, is based on the idea expressed in the opening quote of this chapter. The convergence analysis for noisy function estimates employs the same principle.

We presented some analysis related to the convergence of Gradient-descent methods and a convergence proof related to simulated annealing, in this chapter. The discussion was at an elementary level. The bibliographic remarks mention several references to *more sophisticated* analyses of these methods. The literature contains papers that deal with the convergence analysis of tabu search, the genetic algorithm, and the learning automata search technique.

7. Bibliographic Remarks

Classical gradient-descent theory can be found in Bertsekas and Tsitsiklis [19] (Chapter 3) and Bertsekas [18]. Our account follows [19].

The convergence of simultaneous perturbation in the presence of noise has been established in Spall [162]. Our analysis was under stronger assumptions (Theorems 12.7 and 12.8) and follows Gosavi [60].

Simulated annealing was first analyzed in Lundy and Mees [106]. Subsequently, a large number of papers that deal with its convergence have appeared. Homem-de-Mello [79], Fox and Heine [46], Gelfand and Mitter [50], Alrefaei and Andradóttir [5], and Gosavi [60] have studied the effects of noise in simulated annealing. For the use of non-stationary Markov chains in convergence of simulated annealing, please see Fielding [43], Cohn and Fielding [34], and Tsitsiklis [174]. Our account follows Lundy and Mees [106] and Gosavi [60].

Tabu search (see Glover and Hanafi [52]), the genetic algorithm (see Rudolph [148]), and the Learning Automata Search Technique (see Thathachar and Sastry [173]) have also been treated for convergence analysis in the literature.

8. Review Questions

1. Prove that the gradient-descent algorithm gets trapped in a stationary point. (Hint: The main transformation in the algorithm itself yields the proof.)
2. In what algorithms in this book have we used gradient descent?
3. Consider the function:

$$f(x, y) = 5x^3 - 2x^2y^2 + 6y^3 - x - y + 9.$$

Find the partial derivatives with respect to x and y at $(x = 5, y = 10)$ using calculus. Then compare them to the numerical estimates found via the finite

differences approach using (a) the central differences approach and (b) forward differences approach.

4. A parametric optimization problem has 10 parameters. In one iteration of gradient descent, how many function evaluations will be needed using (a) finite difference gradient estimates, and (b) using simultaneous perturbation estimates.

Chapter 13

CONVERGENCE ANALYSIS OF CONTROL OPTIMIZATION METHODS

Nothing is ever settled until it is settled right.

— Rudyard Kipling (1865-1936)

1. Chapter Overview

This chapter will discuss the proofs of optimality of most of the algorithms discussed in the context of control optimization. Since all the related algorithms are iterative and generate **sequences** of values, we will be dealing with **convergence** of these sequences to optimal solutions.

First, we will establish that the Bellman equation can indeed be used to generate an optimal solution. Then we will prove that the classical versions of value and policy iteration can be used to generate optimal solutions. It has already been discussed that the classical value function based algorithms have Q -factor equivalents. Our second step in convergence analysis will show that step-size based versions of these algorithms will converge to near-optimal solutions as long as updating is *synchronous*. The final step in convergence analysis will establish that the asynchronous versions, under certain conditions, can closely approximate the behavior of their synchronous counterparts.

The convergence proofs for the discounted reward case and the average reward case will be treated separately. The convergence analysis of the discounted case is much easier. As such, we treat this case first and the average reward case second. The reader will need material from Chapter 11 and should go back to review material from there, as and when it is required.

We begin this chapter with some definitions and notation related to discounted and average reward Markov decision problems.

2. Dynamic programming transformations

We have seen the Bellman equation in various forms so far. For the sake of convergence analysis, we need to express the Bellman equation, which is actually a set of linear equations, in the form of a transformation. The value function can then be viewed as a vector that gets transformed every time the Bellman transformation is applied on it.

We will first define a couple of transformations related to the discounted problem, and then define the corresponding transformations for the average reward problem.

The transformation T , you will recognize, is the one that we use in the value iteration algorithm for discounted reward (of course, it is derived from the Bellman optimality equation). T is defined as:

$$T(J(i)) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|S|} p(i, a, j) J(j)] \quad (13.1)$$

for all $i \in \mathcal{S}$. Although all the terms used here have been defined in previous chapters, we repeat the definitions for the sake of your convenience.

- i and j stand for the states of the Markov chain and are members of \mathcal{S} — the set of states. The notation $|\mathcal{S}|$ denotes the number of elements in this set.
- a denotes the action and $\mathcal{A}(i)$ denotes the set of actions allowed in state i .
- λ stands for the discounting factor.
- $p(i, a, j)$ is the probability of transition (of the Markov chain) in one step from state i to state j when action a is selected in state i .
- $\bar{r}(i, a)$ denotes the **expected** immediate reward earned in a one-step transition (of the Markov chain) when action a is selected in state i . The term $\bar{r}(i, a)$ is defined as shown below:

$$\bar{r}(i, a) = \sum_{j=1}^{|S|} p(i, a, j) r(i, a, j), \quad (13.2)$$

where $r(i, a, j)$ is the immediate reward earned in a one-step transition (of the Markov chain) when action a is selected in state i and the next state happens to be j .

- $J(i)$ is the i th component of the vector \vec{J} — the vector that is transformed by T .

The summation notation used in (13.2) will also be sometimes denoted by

$$\sum_{j \in \mathcal{S}}$$

or simply by

$$\sum_j.$$

The next important transformation that we define is the one associated with the Bellman equation for a given policy. It is denoted by $T_{\hat{\mu}}$ and defined as:

$$T_{\hat{\mu}} J(i) = \bar{r}(i, \mu(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) J(j) \text{ for all } i \in \mathcal{S}. \quad (13.3)$$

Here $\mu(i)$ denotes the action to be taken in state i when policy $\hat{\mu}$ is followed.

By setting $\lambda = 1$ in T , one obtains L . Similarly, by setting $\lambda = 1$ in $T_{\hat{\mu}}$ one obtains $L_{\hat{\mu}}$. $L_{\hat{\mu}}$ and L are the corresponding average reward operators for $T_{\hat{\mu}}$ and T respectively. We note however, that in the average reward algorithms, we often use modifications of L and $L_{\hat{\mu}}$. For the sake of completeness, we next define the transformations L and $L_{\hat{\mu}}$.

For all $i \in \mathcal{S}$,

$$L(J(i)) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) J(j)]$$

and

$$L_{\hat{\mu}}(J(i)) = \bar{r}(i, \mu(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) J(j).$$

All the relevant terms have been defined in the context of defining T and $T_{\hat{\mu}}$. Some more discussion on notation and definitions is needed, which follows.

3. Some definitions

Let F denote a dynamic programming operator (such as T or $L_{\hat{\mu}}$). Then the notation F^k has a special meaning, which has been explained in the previous chapter. We quickly explain it again here.

$T_{\hat{\mu}}^2 \vec{J}$ will denote the mapping $T_{\hat{\mu}}$ (see Equation 13.3) applied to the vector $T_{\hat{\mu}}(\vec{J})$. In other words, the definition is:

$$T_{\hat{\mu}}^2 J(i) = \bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) T_{\hat{\mu}} J(j)$$

for any i .

In general,

$$T_{\hat{\mu}}^k(\vec{J}) = T(T_{\hat{\mu}}^{k-1}\vec{J}).$$

Similarly, T^2 will denote the following:

$$T^2 J(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) T J(j)]$$

for all i . Now, T has been defined in (13.1).

The meaning of the notations L^k and $L_{\hat{\mu}}^k$ follows in an analogous manner.

4. Monotonicity of T , $T_{\hat{\mu}}$, L , and $L_{\hat{\mu}}$

The monotonicity of a transformation F implies that given two vectors \vec{J} and \vec{J}' , which satisfy the relationship:

$$J(i) \leq J'(i)$$

for all values of i , the following is true for every positive value of k :

$$F^k(J(i)) \leq F^k(J'(i)).$$

We will establish this monotonicity result for T and $T_{\hat{\mu}}$. The monotonicity result can be established for L and $L_{\hat{\mu}}$ by setting $\lambda = 1$ in the respective results for T and $T_{\hat{\mu}}$.

Let us consider the result for T . We will use an induction argument. $\mathcal{A}(i)$ will denote the set of actions allowable in state i . Now for all $i, j \in \mathcal{S}$

$$\begin{aligned} TJ(i) &= \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) J(j)] \\ &\leq \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) J'(j)] \\ &= TJ'(i). \end{aligned}$$

Thus, the relation is true when $k = 1$. Next, we assume that the result holds when $k = m$. Thus if

$$J(i) \leq J'(i)$$

for all values of i then

$$T^m(J(i)) \leq T^m(J'(i)).$$

Now, for all $i, j \in \mathcal{S}$

$$\begin{aligned} T^{m+1} J(i) &= \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) T^m J(j)] \\ &\leq \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) T^m J'(j)] \\ &= T^{m+1} J'(i). \end{aligned}$$

This completes the induction proof.

Next, we will show that the result holds for $T_{\hat{\mu}}$, that is, for policy $\hat{\mu}$. The proof is similar to that presented above.

Since for all $i, j \in \mathcal{S}$

$$\begin{aligned} T_{\hat{\mu}} J(i) &= [\bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) J(j)] \\ &\leq [\bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) J'(j)] \\ &= T_{\hat{\mu}} J'(i), \end{aligned}$$

the result is true when $k = 1$. Now assuming that the result is true when $k = m$, then

$$J(i) \leq J'(i)$$

for all values of i will imply that

$$T_{\hat{\mu}}^m(J(i)) \leq T_{\hat{\mu}}^m(J'(i)).$$

Then for all $i, j \in \mathcal{S}$

$$\begin{aligned} T_{\hat{\mu}}^{m+1} J(i) &= [\bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) T_{\hat{\mu}}^m J(j)] \\ &\leq [\bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) T_{\hat{\mu}}^m J'(j)] \\ &= T_{\hat{\mu}}^{m+1} J'(i). \end{aligned}$$

This completes the induction proof.

5. Some results for average & discounted MDPs

Next, we will present without proof, some useful results for discounted and average reward. These results will be used in proving the optimality of the Bellman equation and can be found in [20].

Lemma 13.1 is related to discounted reward and Lemma 13.2 is related to average reward.

LEMMA 13.1 *Given a vector \vec{h} of dimension $|\mathcal{S}|$, if $r(x_s, a, x_{s+1})$ denotes the immediate reward earned in the s th jump of the Markov chain under the influence of action a , and $\hat{\mu}$ denotes the policy used to control the Markov chain, then*

$$T_{\hat{\mu}}^k h(i) = E[\lambda^k h(x_{s+1}) + \sum_{s=1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i],$$

for all values of $i \in \mathcal{S}$.

In the above, x_s denotes the state from which the s th jump of the Markov chain occurs.

This can be proved via induction on k . The proof is simple, but the notation used to express it can become quite complicated. Since we do not want to begin this chapter with complicated notation, we skip the proof. The result can be found in Bertsekas [20]. Let us however verify the result's truth for some values of k .

When $k = 1$ for all $i \in \mathcal{S}$,

$$\begin{aligned} T_{\hat{\mu}} h(i) &= \sum_j p(i, \mu(i), j)[r(i, \mu(i), j) + \lambda h(j)] \\ &= E[\lambda^1 h(x_{s+1}) + \sum_{s=1}^1 \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]. \end{aligned}$$

When $k = 2$, for all $i \in \mathcal{S}$,

$$\begin{aligned} T_{\hat{\mu}}^2 h(i) &= \sum_j p(i, \mu(i), j)[r(i, \mu(i), j) + \lambda Th(j)] \\ &= \sum_j p(i, \mu(i), j)[r(i, \mu(i), j) \\ &\quad + \lambda \sum_l p(j, \mu(j), l)\{r(j, \mu(j), l) + \lambda h(l)\}] \\ &= \sum_j p(i, \mu(i), j)[r(i, \mu(i), j)] \\ &\quad + \sum_j p(i, \mu(i), j)[\lambda \sum_l p(j, \mu(j), l)r(j, \mu(j), l)] \\ &\quad + \sum_j p(i, \mu(i), j) \sum_l p(j, \mu(j), l)[\lambda^2 h(l)] \\ &= E[\lambda^2 h(x_{s+1}) + \sum_{s=1}^2 \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]. \end{aligned}$$

The following lemma can be obtained by setting $\lambda = 1$ in the previous lemma.

LEMMA 13.2 *Given a vector \vec{h} of dimension $|\mathcal{S}|$, if $r(x_s, a, x_{s+1})$ denotes the immediate reward earned in the s th jump of the Markov chain under the influence of action a , and $\hat{\mu}$ denotes the policy used to control the Markov chain, then*

$$L_{\hat{\mu}}^k h(i) = E[h(x_{s+1}) + \sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i],$$

for all values of $i \in \mathcal{S}$.

The implication of Lemma 13.1 (Lemma 13.2) is that if one selects any $|\mathcal{S}|$ -dimensional vector and applies the mapping $T_{\hat{\mu}}$ ($L_{\hat{\mu}}$) upon it k times, one obtains the expected *total discounted reward* (*total reward*) earned in a finite trajectory of k jumps (of the Markov chain) starting at state i and using the policy $\hat{\mu}$.

6. Discounted reward and classical dynamic programming

This section will deal with the analysis of the convergence of algorithms used for the (total) discounted reward criterion. This section will only discuss some classical dynamic programming methods.

6.1. Bellman Equation for Discounted Reward

The Bellman equation for discounted reward was motivated in a heuristic sense in previous chapters. It is now time to establish its optimality — that is, to show that a solution of the Bellman equation identifies the optimal solution for a discounted reward MDP. To prove the optimality of the Bellman equation, we need a definition (Definition 13.1) and some results (Lemma 13.3, Proposition 13.4, and Lemma 13.6).

DEFINITION 13.1 *The value function vector associated with a given policy $\hat{\mu}$ for the discounted reward case is defined as:*

$$J_{\hat{\mu}}(i) = \lim_{k \rightarrow \infty} E\left[\sum_{s=1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \quad \text{for all } i. \quad (13.4)$$

In this definition, x_s is the state from where the s th transition occurs in the Markov chain. The definition implies that $J_{\hat{\mu}}(i)$ is the total discounted reward earned along an infinitely long trajectory (sequence) of states by starting at state i and following policy $\hat{\mu}$ at every state encountered. So in other words, we control the Markov chain with a policy $\hat{\mu}$, and observe the rewards over an infinitely large number of jumps (transitions). Of course, future rewards are discounted with a factor of λ . The expectation operator is used in the definition above because x_{s+1} is a stochastic element. (Remember there is no fixed x_{s+1} associated with a given x_s because of the stochastic nature of the transitions). The expectation is over all the trajectories.

The next result will help define the optimal value function.

LEMMA 13.3 *The value function vector, which is defined as:*

$$J^*(i) = \max_{\hat{\mu}} J_{\hat{\mu}}(i) \quad \text{for each } i, \quad (13.5)$$

is the optimal value function vector for the discounted reward MDP in question.

The result says that the vector \vec{J}^* is the vector associated with the optimal policy, and each component of the vector takes on the maximum possible value.

Proof From Lemma 13.1 and Definition 13.1, one has that $J_{\hat{\mu}}(i)$ is the **expected total discounted reward earned over an infinitely long trajectory** by using policy $\hat{\mu}$ and starting at state i . This is the performance metric for state i . Then from our definition of $J^*(i)$ above, it is clear that $J^*(i)$ denotes the best possible discounted reward earned by starting at state i . This is true for each i . This means that \vec{J}^* must be the optimal value function vector. ■

We will next prove an important result that will be needed to prove the optimality of the Bellman equation.

PROPOSITION 13.4 *For any bounded function $h : S \rightarrow \mathcal{R}$, the value function vector (defined in Equation (13.5)) satisfies the following equation:*

$$J^*(i) = \lim_{k \rightarrow \infty} T^k(h(i)) \quad \text{for all } i. \quad (13.6)$$

Furthermore, this value function vector is the optimal value function vector.

The proposition implies that if any vector (say \vec{h}) with finite values for all its components is selected and the transformation T is applied infinitely many times on it, one obtains the **optimal discounted reward value function vector** (that is, \vec{J}^*).

Proof From Lemma 13.3, we know this value function vector to be the optimal value function vector. Now, what remains to be shown is: theoretically the optimal value function vector can be obtained by applying T on any vector infinitely many times.

We will first use the definition given in Equation (13.4). The reward earned over an infinite time horizon can be broken down into two parts; one part constitutes of the reward earned in the first P steps and the second constitutes of the same earned over the remainder of the trajectory. Thus:

$$\begin{aligned} J_{\hat{\mu}}(i) &= \lim_{k \rightarrow \infty} E\left[\sum_{s=1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \\ &= \lim_{k \rightarrow \infty} E\left[\sum_{s=1}^P \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \\ &\quad + \lim_{k \rightarrow \infty} E\left[\sum_{s=P+1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \end{aligned}$$

$$\begin{aligned}
&= E\left[\sum_{s=1}^P \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \\
&\quad + \lim_{k \rightarrow \infty} E\left[\sum_{s=P+1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \quad (13.7)
\end{aligned}$$

Throughout this book, we have assumed that the immediate rewards are always finite quantities. Therefore:

$$|r(x_s, \mu(x_s), x_{s+1})| \leq M \quad \text{for all } s$$

for some positive value of M .

Hence

$$\begin{aligned}
&\lim_{k \rightarrow \infty} \left| E\left[\sum_{s=P+1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right] \right| \\
&\leq \lim_{k \rightarrow \infty} \left[\sum_{s=P+1}^k \lambda^{s-1} M \right] \\
&= M \sum_{s=P+1}^{\infty} \lambda^{s-1} \\
&= M \frac{\lambda^P}{1 - \lambda} \quad \text{using the result on page 298.}
\end{aligned}$$

Denoting

$$\lim_{k \rightarrow \infty} E\left[\sum_{s=P+1}^k \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right]$$

by A , we have from the above that,

$$|A| < \frac{M\lambda^P}{1 - \lambda}$$

which implies that:

$$-\frac{M\lambda^P}{1 - \lambda} < A < \frac{M\lambda^P}{1 - \lambda}.$$

Multiplying the above inequalities by -1 , we have that

$$\frac{M\lambda^P}{1 - \lambda} > -A > -\frac{M\lambda^P}{1 - \lambda}.$$

Adding $J_{\hat{\mu}}(i)$ to each side, we have:

$$J_{\hat{\mu}}(i) + \frac{M\lambda^P}{1 - \lambda} > J_{\hat{\mu}}(i) - A > J_{\hat{\mu}}(i) - \frac{M\lambda^P}{1 - \lambda}.$$

Using (13.7), the above can be written as:

$$\begin{aligned} J_{\hat{\mu}}(\mathbf{i}) + \frac{M\lambda^P}{1-\lambda} &> E\left[\sum_{s=1}^P \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = \mathbf{i}\right] \\ &> J_{\hat{\mu}}(\mathbf{i}) - \frac{M\lambda^P}{1-\lambda}. \end{aligned} \quad (13.8)$$

Now, since

$$\max_i |h(i)| \geq h(i) \quad \text{for all } i, \text{ we can write}$$

$$-\max_i |h(i)| \leq E[h(i)] \leq \max_i |h(i)| \quad \text{for all } i.$$

Here $E[h(i)]$ is an abbreviation for $\sum_i p(i)h(i)$ in which $\sum_i p(i) = 1$.

Then, since $\lambda > 0$, one has that:

$$\max_i |h(i)|\lambda^P \geq E[h(i)]\lambda^P \geq \max_i |h(i)|\lambda^P \quad \text{for all } i. \quad (13.9)$$

Adding (13.8) and (13.9), it follows that

$$\begin{aligned} J_{\hat{\mu}}(\mathbf{i}) + \frac{M\lambda^P}{1-\lambda} + \max_i |h(i)|\lambda^P &> \\ E\left[\sum_{s=1}^P \lambda^{k-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = \mathbf{i}\right] + E[h(i)]\lambda^P &> \\ J_{\hat{\mu}}(\mathbf{i}) - \frac{M\lambda^P}{1-\lambda} - \max_i |h(i)|\lambda^P. \end{aligned}$$

Using Lemma 13.1 (see page 347) the above becomes:

$$J_{\hat{\mu}}(\mathbf{i}) + \frac{M\lambda^P}{1-\lambda} + \max_i |h(i)|\lambda^P \geq T_{\hat{\mu}}^P h(\mathbf{i}) \geq J_{\hat{\mu}}(\mathbf{i}) - \frac{M\lambda^P}{1-\lambda} - \max_i |h(i)|\lambda^P.$$

The above is true for any policy. Selecting a policy, which maximizes the terms above, the above becomes

$$\begin{aligned} J^*(\mathbf{i}) + \frac{M\lambda^P}{1-\lambda} + \max_i |h(i)|\lambda^P &\geq T^P h(\mathbf{i}) \\ &\geq J^*(\mathbf{i}) - \frac{M\lambda^P}{1-\lambda} - \max_i |h(i)|\lambda^P. \end{aligned} \quad (13.10)$$

Taking the limit with $P \rightarrow \infty$, from the above, one obtains the following relationship:

$$J^*(i) \geq \lim_{P \rightarrow \infty} T^P h(i) \geq J^*(i). \quad (13.11)$$

(The above uses the fact that $\lim_{P \rightarrow \infty} \lambda^P = 0$ which is true if $0 \leq \lambda < 1$.) Clearly (13.11) implies that:

$$\lim_{P \rightarrow \infty} T^P h(i) = J^*(i)$$

for all $i \in \mathcal{S}$. ■

The next result follows directly from Proposition 13.4.

COROLLARY 13.5 *For any bounded function $h : \mathcal{S} \rightarrow \mathcal{R}$, the value function vector associated with a policy (defined in Equation 13.4) satisfies the following equation.*

$$J_{\hat{\mu}}(i) = \lim_{k \rightarrow \infty} T_{\hat{\mu}}^k(h(i)) \text{ for all } i. \quad (13.12)$$

The proof of this result is very similar to that of Proposition 13.4. The result implies that associated with any policy $\hat{\mu}$, there is a value function vector denoted by $\vec{J}_{\hat{\mu}}$ that can be obtained by applying the transformation $T_{\hat{\mu}}$ on any given vector infinitely many times. Also note the following:

- The i th element of this vector denotes the expected total discounted reward earned over an infinite time horizon, if one starts at state i and follows policy $\hat{\mu}$.
- In contrast, the i th element of the vector \vec{J}^* denotes the expected total discounted reward earned over an infinite time horizon, if one starts at state i and follows the optimal policy.

We need to prove one more simple result before working with the Bellman equation.

LEMMA 13.6 *If T denotes the Bellman operator for discounted reward, \vec{J} denotes a vector, and c denotes a scalar, then*

$$T(J(i) + c) = T(J(i)) + \lambda c.$$

Proof By its definition for all i :

$$T(h(i)) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j)h(j)].$$

Then, setting $h(i) = J(i) + c$ for all i ,

$$\begin{aligned} T(J(i) + c) &= \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j)(J(j) + c)] \\ &= \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j)J(j)] + \lambda c \\ &= T(J(i)) + \lambda c. \quad \blacksquare \end{aligned}$$

Now we are ready to prove the optimality of the Bellman equation.

PROPOSITION 13.7 *Consider the system of linear equations defined by:*

$$h(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j)h(j)] \text{ for all } i. \quad (13.13)$$

Using the shorthand notation introduced previously, the equation can be expressed as:

$$h(i) = T(h(i)) \text{ for all } i.$$

The vector \vec{J}^ (which by Proposition 13.4 can be obtained theoretically by taking any finite-valued vector and performing the T operation infinitely many times on it) is a solution of this equation. Furthermore, \vec{J}^* is the unique solution of this equation. Also, this vector is the optimal value function vector.*

Note that in the linear system of equations presented above, the h terms are the unknowns. Finding the solution to this equation holds the key to finding the optimal solution to the MDP. This is because we proved in Lemma 13.3 that the vector \vec{J}^* is the optimal value function vector.

Proof Note that from Proposition 13.4, we know \vec{J}^* to be the optimal value function vector. What remains to be shown is that this vector is the unique solution to Equation (13.13). Note that Equation (13.13) is the so-called Bellman *optimality* equation for discounted reward MDPs.

Using Inequation (13.10) from Proposition 13.4 and choosing a vector \vec{v} such that $v(i) = 0$ for all i , we have that

$$J^*(i) - \frac{\lambda^P M}{1-\lambda} \leq T^P v(i) \leq J^*(i) + \frac{\lambda^P M}{1-\lambda} \text{ for all } i.$$

Since

$$J^*(i) - \frac{\lambda^P M}{1-\lambda} \leq T^P v(i) \text{ for all } i,$$

using the operator T on both sides of the above and using the monotonicity results from Section 4, one can write that for all i ,

$$T \left(J^*(i) - \frac{\lambda^P M}{1-\lambda} \right) \leq T^{P+1} v(i). \quad (13.14)$$

Since

$$T^P v(i) \leq J^*(i) + \frac{\lambda^P M}{1-\lambda} \quad \text{for all } i$$

once again using the monotonicity results one has that

$$T^{P+1} v(i) \leq T \left(J^*(i) + \frac{\lambda^P M}{1-\lambda} \right) \quad \text{for all } i. \quad (13.15)$$

Combining (13.14) and (13.15), one can write:

$$T \left(J^*(i) - \frac{\lambda^P M}{1-\lambda} \right) \leq T^{P+1} v(i) \leq T \left(J^*(i) + \frac{\lambda^P M}{1-\lambda} \right)$$

for all i .

Using Lemma 13.6, the above can be written as:

$$T J^*(i) - \lambda \left(\frac{\lambda^P M}{1-\lambda} \right) \leq T^{P+1} v(i) \leq T J^*(i) + \lambda \left(\frac{\lambda^P M}{1-\lambda} \right)$$

for all i .

Taking the limit with $P \rightarrow \infty$, one obtains that for all i :

$$T J^*(i) \leq \lim_{P \rightarrow \infty} T^{P+1} v(i) \leq T J^*(i),$$

noting that

$$\lim_{P \rightarrow \infty} \lambda^P = 0 \quad \text{since } 0 \leq \lambda < 1.$$

But $\lim_{P \rightarrow \infty} T^{P+1} v(i) = J^*(i)$ from Proposition 13.4 for any finite-valued vector \vec{v} . Hence we have that

$$T J^*(i) \leq J^*(i) \leq T J^*(i)$$

for all i , which implies that

$$T J^*(i) = J^*(i) \quad \text{for all } i.$$

It is hereby proved that the vector \vec{J}^* satisfies $\vec{h} = T \vec{h}$ and is hence a solution of the Bellman optimality equation — that is Equation (13.13). What remains to be shown is that \vec{J}^* is the unique solution.

Let us assume that \vec{J}' is another solution of the Bellman equation. Then

$$T \vec{J}' = \vec{J}'.$$

Using induction on k , it is very easy to prove that

$$\vec{J}' = T^k \vec{J}', \quad \text{for any } k.$$

Hence we have that as $k \rightarrow \infty$

$$\vec{J}' = \lim_{k \rightarrow \infty} T^k \vec{J}'.$$

But from Proposition 13.4,

$$\lim_{k \rightarrow \infty} T^k \vec{J}' = \vec{J}^*.$$

Hence

$$\vec{J}' = \vec{J}^*. \quad \blacksquare$$

The result given above implies that a policy $\hat{\mu}$, which satisfies

$$T_{\hat{\mu}} \vec{h} = \vec{h},$$

for any given vector \vec{h} , is an optimal policy.

The proposition presented above is related to the Bellman **optimality equation**. Via replacement of the T operator by $T_{\hat{\mu}}$, it is possible to obtain the following result from the preceding proposition.

COROLLARY 13.8 *The Bellman equation for discounted reward for a given policy, $\hat{\mu}$, is a system of linear equations defined by:*

$$h(i) = [\bar{r}(i, \mu(i)) + \lambda \sum_j p(i, \mu(i), j) h(j)] \text{ for all } i.$$

Using the shorthand notation introduced previously,

$$h(i) = T_{\hat{\mu}}(h(i)) \text{ for all } i.$$

The vector $\vec{J}_{\hat{\mu}}$ (which by Corollary 13.5 can be obtained by taking any finite-valued vector and performing the $T_{\hat{\mu}}$ operation infinitely many times on it) is a solution of the Bellman equation given above. Furthermore, $\vec{J}_{\hat{\mu}}$ is the unique solution.

Note that, as usual, in the Bellman equation, the h terms are the unknowns. The above result is related to a given policy. Its proof is very similar to that of Proposition 13.7, and so we do not present its details.

6.2. Policy Iteration

In this subsection, we will prove that policy iteration for discounted reward generates the optimal solution to the MDP.

Steps in Policy Iteration for Discounted Reward MDPs. For the sake of the reader's convenience, we repeat the step-by-step details of policy iteration below.

Step 1. Set $k = 0$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select an arbitrary policy. Let us denote the policy selected by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. Policy Evaluation: Solve the following linear system of equations.

$$v^k(i) = \bar{r}(i, \mu_k(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu_k(i), j) v^k(j). \quad (13.16)$$

Here one linear equation is associated with each value of i . In this system, the unknowns are the v^k terms.

Step 3. Policy Improvement: Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) v^k(j)].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for all i then stop and set $\mu^*(i) = \mu_k(i)$ for all i . Otherwise, increment k by 1 and go back to the second step.

We now present a result that states the convergence of policy iteration for discounted reward MDPs.

PROPOSITION 13.9 *The policy iteration algorithm described above generates an optimal solution in a finite number of iterations.*

Proof (Convergence of Policy Iteration)

The equation in Step 2 of the algorithm can be written as:

$$v^k(i) = T_{\hat{\mu}_k} v^k(i) \quad (13.17)$$

for all i . A careful examination of Step 3 will reveal that

$$T_{\hat{\mu}_{k+1}} v^k(i) = T v^k(i) \quad (13.18)$$

for all i . Thus for every i ,

$$\begin{aligned} v^k(i) &= T_{\hat{\mu}_k} v^k(i) \quad (\text{from Equation (13.17)}) \\ &\leq T v^k(i) \quad (\text{follows from the fact that } T \text{ is a max operator}) \\ &= T_{\hat{\mu}_{k+1}} v^k(i) \quad (\text{from Equation (13.18)}). \end{aligned}$$

Hence for every i

$$v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i).$$

Then for every i , using the monotonicity result from Section 4, it follows that

$$T_{\hat{\mu}_{k+1}} v^k(i) \leq T_{\hat{\mu}_{k+1}}^2 v^k(i)$$

which implies that:

$$v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i) \leq T_{\hat{\mu}_{k+1}}^2 v^k(i).$$

Repeatedly applying $T_{\hat{\mu}_{k+1}}$, one has that for all i ,

$$v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i) \leq T_{\hat{\mu}_{k+1}}^2 v^k(i) \leq \dots \leq T_{\hat{\mu}_{k+1}}^P v^k(i).$$

Since the above is also true when $P \rightarrow \infty$, one has that for all i ,

$$v^k(i) \leq \lim_{P \rightarrow \infty} T_{\hat{\mu}_{k+1}}^P v^k(i).$$

From Corollary 13.5, we know that $\lim_{P \rightarrow \infty} T_{\hat{\mu}_{k+1}}^P v^k(i)$ exists. Let us denote the limit by $J_{\hat{\mu}_{k+1}}(i)$. Hence

$$v^k(i) \leq J_{\hat{\mu}_{k+1}}(i) \text{ for all } i. \quad (13.19)$$

Now, by Corollary 13.8, $J_{\hat{\mu}_{k+1}}(i)$ satisfies

$$J_{\hat{\mu}_{k+1}}(i) = T_{\hat{\mu}_{k+1}} J_{\hat{\mu}_{k+1}}(i) \quad (13.20)$$

for all i . But from Step 2 of the algorithm it follows that:

$$v^{k+1}(i) = T_{\hat{\mu}_{k+1}} v^{k+1}(i) \quad (13.21)$$

for all i .

From Equations (13.20) and (13.21), it is clear that both vectors \vec{v}^{k+1} and $\vec{J}_{\hat{\mu}_{k+1}}$ satisfy

$$\vec{h} = T_{\hat{\mu}} \vec{h}.$$

But the solution of this equation is unique by Corollary 13.8. Hence:

$$\vec{v}^{k+1} = \vec{J}_{\hat{\mu}_{k+1}}.$$

Therefore, from Equation (13.19), one can now write that:

$$v^k(i) \leq v^{k+1}(i) \quad (13.22)$$

for all i .

This means that in each iteration (k) the value of the vector (\vec{v}^k) either increases or does not change. This iterative process cannot go on infinitely as the total number of policies is finite (since the number of states and the number of actions are finite). In other words, the process must terminate in a finite number of steps. When the policy repeats — that is, when $\mu_k(i) = \mu_{k+1}(i)$ for all i , it is the case that one has obtained the optimal solution. Here is why:

$$v^k(i) = T_{\vec{\mu}_k} v^k(i) = T_{\vec{\mu}_{k+1}} v^k(i) = T v^k(i)$$

for all i . The first equality sign follows from (13.17) and the last equality sign follows from (13.18). Thus:

$$v^k(i) = T v^k(i)$$

for all i . This means that the Bellman optimality equation has been solved! By Proposition 13.7, we have that $\vec{\mu}_k$ is the optimal policy. ■

This proof can be found in Bertsekas [20] and is as clever as the algorithm itself. Convergence of policy iteration has been established in many different ways in the literature. The proof presented above is easy to understand, although it may not qualify to be called the briefest of proofs. A comparatively brief proof, which is presented in Puterman [140] (page 175 — Proposition 6.4.1), needs the knowledge of some advanced properties of matrices such as spectral radii and series expansions of inverses. The reader is urged to read the proof in [140] as well. A different mechanism for proving the same result is invariably a mathematical treasure to be cherished and appreciated.

6.3. Value iteration for discounted reward MDPs

Value iteration, the way we have described it, unlike policy iteration, cannot produce an optimal policy in a finite number of steps. In other words, it takes an infinite number of iterations for value iteration to converge. Fortunately, there is a way to work around this. For all practical purposes, it can generate a policy that is very close to optimal. In this context, we will define a so-called ϵ -optimal policy, which is the best we can do with value iteration.

A straightforward argument that can be used to prove the convergence of value iteration stems from Proposition 13.4. It implies that if you can take any finite-valued vector and use the mapping T on it an infinite number of times, you have the optimal value function vector. Once we have the “optimal” value function vector, since we know it satisfies the Bellman optimality equation, we can find the maximizing action in each state by using the Bellman optimality equation; the latter is the last step in value iteration. Thus one way to prove the convergence of value iteration is to use Proposition 13.4. However, the fact is that we can run our algorithm in practice for a finite number of times only, and as such it is a good idea to delve deeper into its convergence issues.

We will prove that value iteration can produce a policy that is as close to the optimal policy as one wants. One way to determine how close is close enough is to use the norm of the difference of the value function vectors generated in successive iterations of the algorithm. The norm of the difference vector is an estimate of how close the two vectors are. When we have performed an infinitely large number of iterations, we have that the vectors generated in two successive iterations are **identical**. (Because $\vec{J}^* = T\vec{J}^*$.)

To prove the ϵ -convergence of value iteration, we need a result that was presented as Proposition 13.10. Proposition 13.10 will be an important result from many standpoints. It will prove Proposition 13.4 partially. Proposition 13.4 shows that

$$\lim_{P \rightarrow \infty} T^P \vec{h}$$

exists, and moreover it also proves that the limit is the **optimal** reward function vector.

Proposition 13.10 will also show that the limit exists. But it will serve a different purpose than Proposition 13.4 cannot serve. In particular, Proposition 13.10 will establish that the mapping T is **contractive** — a property that will help us prove the ϵ -convergence of value iteration.

To proceed any further, one must be thoroughly familiar with the notion of contraction mappings and the Fixed Point Theorem (Theorem 11.7 on page 312)— both topics are discussed in detail in Chapter 11.

PROPOSITION 13.10 *The mapping T , that is, the Bellman operator for discounted reward, is contractive with respect to the max norm. This means that given two vectors: \vec{J} and \vec{J}' ,*

$$\|T\vec{J} - T\vec{J}'\| \leq \zeta \|\vec{J} - \vec{J}'\|,$$

where $\|\cdot\|$ is the max-norm and $\zeta \in (0, 1)$. Also ζ is equal to λ , the discounting factor.

Proof Let us first assume that $TJ(i) \geq TJ'(i)$ for all i . And let

$$a(i) \in \arg \max_{u \in A(i)} [\bar{r}(i, u) + \lambda \sum_j p(i, u, j) J(j)]$$

for every i . In other words, let $a(i)$ denote one of the actions in the i th state that will maximize the quantity in the square brackets above. It implies that:

$$TJ(i) = [\bar{r}(i, a(i)) + \lambda \sum_j p(i, a(i), j) J(j)]$$

for every i .

Similarly, let

$$b(i) \in \arg \max_{u \in \mathcal{A}(i)} [\bar{r}(i, u) + \lambda \sum_j p(i, u, j) J'(j)]$$

for all i , which will imply that:

$$TJ'(i) = [\bar{r}(i, b(i)) + \lambda \sum_j p(i, b(i), j) J'(j)]$$

for every i . Since action $b(i)$ maximizes the quantity in the square brackets above,

$$TJ'(i) \geq [\bar{r}(i, a(i)) + \lambda \sum_j p(i, a(i), j) J'(j)]$$

for every i . Therefore

$$-TJ'(i) \leq -[\bar{r}(i, a(i)) + \lambda \sum_j p(i, a(i), j) J'(j)]$$

Combining this with the definition of $TJ(i)$ and the fact that $TJ(i) \geq TJ'(i)$ for all i , we have that for every i :

$$\begin{aligned} 0 &\leq TJ(i) - TJ'(i) \\ &\leq [\bar{r}(i, a(i)) + \lambda \sum_j p(i, a(i), j) J(i)] - \\ &\quad [\bar{r}(i, a(i)) + \lambda \sum_j p(i, a(i), j) J'(i)] \\ &= \lambda \sum_j p(i, a(i), j) [J(i) - J'(j)] \\ &\leq \lambda \sum_j p(i, a(i), j) \max_j |J(j) - J'(j)| \\ &= \lambda \max_j |J(j) - J'(j)| \left(\sum_j p(i, a(i), j) \right) \\ &= \lambda \max_j |J(j) - J'(j)|(1) \\ &= \lambda \|\vec{J} - \vec{J}'\|. \end{aligned}$$

Thus, we can write:

$$TJ(i) - TJ'(i) \leq \lambda \|\vec{J} - \vec{J}'\| \tag{13.23}$$

for all i .

Now let us assume that $TJ'(i) \geq TJ(i)$ for all i . Using logic similar to that used above, one can show that

$$TJ'(i) - TJ(i) \leq \lambda \|\vec{J}' - \vec{J}\|$$

for all i . Since $\|\vec{J}' - \vec{J}\| = \|\vec{J} - \vec{J}'\|$, one can write that:

$$TJ'(i) - TJ(i) \leq \lambda \|\vec{J} - \vec{J}'\| \quad (13.24)$$

for all i .

Together (13.23) and (13.24) imply that:

$$|TJ(i) - TJ'(i)| \leq \lambda \|\vec{J} - \vec{J}'\| \quad (13.25)$$

for all i . This follows from the fact that the LHS of each of the Inequations (13.23) and (13.24) has to be positive. Hence when the absolute value of the LHS is selected, both (13.23) and (13.24) will imply (13.25).

Since Inequation (13.25) holds for any i , it also holds for the value of i that maximizes $|TJ(i) - TJ'(i)|$. Thus:

$$\max_i |TJ(i) - TJ'(i)| \leq \lambda \|\vec{J} - \vec{J}'\|.$$

In other words:

$$\|T\vec{J} - T\vec{J}'\| \leq \lambda \|\vec{J} - \vec{J}'\|. \quad \blacksquare$$

We next state a corollary of this result. It states that the property also holds for the mapping associated with a given policy. One can independently prove this result using arguments similar to those used above.

COROLLARY 13.11 *The mapping $T_{\hat{\mu}}$; that is the dynamic programming operator associated with a policy $\hat{\mu}$ for discounted reward is contractive. In particular we will prove that given two vectors \vec{J} and \vec{J}' ,*

$$\|T_{\hat{\mu}}\vec{J} - T_{\hat{\mu}}\vec{J}'\| \leq \lambda \|\vec{J} - \vec{J}'\|,$$

where $\|\cdot\|$ is the max-norm and $\lambda \in (0, 1)$ is the discounting factor.

The contraction properties established above will help us prove that the policy generated by value iteration is ϵ -optimal.

Steps in the value iteration algorithm for discounted reward. Let us review value iteration, next.

Step 1: Set $k = 0$ and select an arbitrary vector \vec{J}^0 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) J^k(j)].$$

Step 3: If

$$\|\vec{J}^{k+1} - \vec{J}^k\| \leq \epsilon(1 - \lambda)/2\lambda,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \lambda \sum_j p(i, a, j) J^k(j)]$$

and stop.

Basically, the algorithm, for a given value of ϵ , returns a policy. The value function vector of the returned policy is such that the norm of the difference between this vector and the optimal value function vector is less than ϵ . Proving this is the main idea underlying the next Proposition.

PROPOSITION 13.12 (Convergence of Value Iteration:)

The value iteration algorithm generates an ϵ -optimal policy; that is, if $\vec{J}_{\hat{d}}$ denotes the value function vector associated with the policy (solution) \hat{d} , and \vec{J}^ denotes the optimal reward vector, then:*

$$\|\vec{J}_{\hat{d}} - \vec{J}^*\| < \epsilon.$$

Proof $\vec{J}_{\hat{d}}$ denotes the vector associated with the policy \hat{d} . Note that this vector is never really calculated in the value iteration algorithm. But it is the reward vector associated with the solution policy returned by the algorithm. From Corollary 13.5, we know that $\lim_{P \rightarrow \infty} T_{\hat{d}} \vec{h}$ converges for any \vec{h} and that the limit is $\vec{J}_{\hat{d}}$. From Corollary 13.8, we know that

$$T_{\hat{d}} \vec{J}_{\hat{d}} = \vec{J}_{\hat{d}}. \quad (13.26)$$

Now from Step 4, from the way the policy \hat{d} is selected, it follows that for any vector \vec{h}

$$T_{\hat{d}} \vec{h} = T \vec{h}. \quad (13.27)$$

We will use this fact below.

It follows from the properties of a norm (see Section 3) that:

$$\|\vec{J}_{\hat{d}} - \vec{J}^*\| \leq \|\vec{J}_{\hat{d}} - \vec{J}^{k+1}\| + \|\vec{J}^{k+1} - \vec{J}^*\|. \quad (13.28)$$

Now,

$$\begin{aligned} \|\vec{J}_{\hat{d}} - \vec{J}^{k+1}\| &= \|T_{\hat{d}} \vec{J}_{\hat{d}} - \vec{J}^{k+1}\| \text{ using (13.26)} \\ &= \|T_{\hat{d}} \vec{J}_{\hat{d}} - T \vec{J}^{k+1} + T \vec{J}^{k+1} - \vec{J}^{k+1}\| \end{aligned}$$

$$\begin{aligned}
&\leq \|T_{\hat{d}}\vec{J}_{\hat{d}} - T\vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - \vec{J}^{k+1}\| \\
&\quad \text{using a property of a norm} \\
&\leq \|T_{\hat{d}}\vec{J}_{\hat{d}} - T\vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - T\vec{J}^k\| \\
&= \|T\vec{J}_{\hat{d}} - T\vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - T\vec{J}^k\| \text{ using (13.27)} \\
&\leq \lambda\|\vec{J}_{\hat{d}} - \vec{J}^{k+1}\| + \lambda\|\vec{J}^{k+1} - \vec{J}^k\| \\
&\quad \text{using the contraction property of } T \text{ and } T_{\hat{d}}.
\end{aligned}$$

Rearranging terms, one has that

$$\|\vec{J}_{\hat{d}} - \vec{J}^{k+1}\| \leq \frac{\lambda}{1-\lambda}\|\vec{J}^{k+1} - \vec{J}^k\|. \quad (13.29)$$

Similarly,

$$\begin{aligned}
\|\vec{J}^{k+1} - \vec{J}^*\| &\leq \|\vec{J}^{k+1} - T\vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - \vec{J}^*\| \\
&\quad \text{using a standard norm property} \\
&= \|T\vec{J}^k - T\vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - \vec{J}^*\| \\
&\leq \lambda\|\vec{J}^k - \vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - \vec{J}^*\| \\
&\leq \lambda\|\vec{J}^k - \vec{J}^{k+1}\| + \|T\vec{J}^{k+1} - T\vec{J}^*\| \text{ since } T\vec{J}^* = \vec{J}^* \\
&\leq \lambda\|\vec{J}^k - \vec{J}^{k+1}\| + \lambda\|\vec{J}^{k+1} - \vec{J}^*\|.
\end{aligned}$$

Rearranging terms,

$$\|\vec{J}^{k+1} - \vec{J}^*\| \leq \frac{\lambda}{1-\lambda}\|\vec{J}^{k+1} - \vec{J}^k\|. \quad (13.30)$$

Using (13.29) and (13.30) in (13.28), one has that

$$\|\vec{J}_{\hat{d}} - \vec{J}^*\| \leq 2\frac{\lambda}{1-\lambda}\|\vec{J}^{k+1} - \vec{J}^k\| < \epsilon.$$

The last inequation in the above stems from step 3 of the algorithm. Thus:

$$\|\vec{J}_{\hat{d}} - \vec{J}^*\| < \epsilon. \quad \blacksquare$$

7. Average reward and classical dynamic programming

In this section, we will establish a number of results related to average reward in the context of classical dynamic programming. Our discussion will be more or less consistent with the approach used in the discounted reward sections. We will begin with the optimality of the Bellman equation and then go on to discuss the convergence of value and policy iteration methods.

7.1. Bellman equation for average reward

The Bellman equation for average reward was used without proof in previous chapters. It is now time to establish that the equation is indeed useful in generating an optimal solution. Our first result will introduce the Bellman equation and establish its optimality for MDPs with average reward.

PROPOSITION 13.13 *If a scalar ρ and a $|\mathcal{S}|$ -dimensional vector \vec{h} , where $|\mathcal{S}|$ denotes the number of elements in the set of states in the Markov chain — \mathcal{S} , satisfy*

$$\rho + h(i) = \bar{r}(i, \mu(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j)h(j), \quad i = 1, 2, \dots, |\mathcal{S}|, \quad (13.31)$$

then ρ is the average reward associated with the policy $\hat{\mu}$.

Furthermore if a scalar ρ^ and a $|\mathcal{S}|$ -dimensional vector $J(i)$ satisfy*

$$\rho^* + J(i) = \max_{u \in \mathcal{A}(i)} [\bar{r}(i, u) + \sum_{j=1}^{|\mathcal{S}|} p(i, u, j)J(j)], \quad i = 1, 2, \dots, |\mathcal{S}|, \quad (13.32)$$

where $\mathcal{A}(i)$ is the set of allowable actions in state i , then ρ^ is the average reward associated with the policy $\hat{\mu}^*$ that attains the max in the RHS of equation (13.32).*

The policy $\hat{\mu}^$ is the optimal policy.*

Proof Equation (13.31) can be written in vector form as:

$$\rho \vec{e} + \vec{h} = L_{\hat{\mu}} \vec{h}, \quad (13.33)$$

where $L_{\hat{\mu}}$ is a mapping associated with policy $\hat{\mu}$ and \vec{e} is an $|\mathcal{S}|$ -dimensional vector of ones; that is each component of \vec{e} is 1. In other words, \vec{e} is

$$\begin{bmatrix} 1 \\ 1 \\ \vdots \\ \vdots \\ 1 \end{bmatrix}.$$

We will first prove that

$$L_{\hat{\mu}}^k h(i) = k\rho + h(i) \quad (13.34)$$

for $i = 1, 2, \dots, |\mathcal{S}|$.

The above can be written in the vector form as:

$$L_{\hat{\mu}}^k \vec{h} = k\rho \vec{e} + \vec{h}. \quad (13.35)$$

We will use an induction argument for the proof. From Equation (13.33), the above is true when $k = 1$. Let us assume that the above is true when $k = m$. Then we have that

$$L_{\hat{\mu}}^m \vec{h} = m\rho \vec{e} + \vec{h}.$$

Using the transformation $L_{\hat{\mu}}$ on both sides of this equation, we have:

$$\begin{aligned} L_{\hat{\mu}}(L_{\hat{\mu}}^m \vec{h}) &= L_{\hat{\mu}}(m\rho \vec{e} + \vec{h}) \\ &= m\rho \vec{e} + L_{\hat{\mu}} \vec{h} \\ &= m\rho \vec{e} + \rho \vec{e} + \vec{h} \text{ (using Equation (13.33))} \\ &= (m+1)\rho \vec{e} + \vec{h}. \end{aligned}$$

Thus Equation (13.35) is established using induction on k .

Using Lemma 13.2 given on page 348, we have for all i :

$$L_{\hat{\mu}}^k h(i) = E[A + \sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i],$$

where x_s is the state from where the s th jump of the Markov chain occurs and A is a finite quantity.

Using the above and Equation (13.34), we have that:

$$E[A + \sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i] = k\rho + h(i).$$

Therefore,

$$\frac{E[A]}{k} + \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]}{k} = \rho + \frac{h(i)}{k}.$$

Taking limits as $k \rightarrow \infty$, we have:

$$\lim_{k \rightarrow \infty} \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i]}{k} = \rho.$$

(The above follows from the fact that

$$\lim_{k \rightarrow \infty} a/k = 0, \text{ if } a \text{ is finite.}$$

In words, this means from the definition of average reward that the average reward of the policy $\hat{\mu}$ is indeed ρ , and the first part of the proposition is thus established.

Now for the second part. Using the first part of the proposition, one can show that a policy, let us call it $\hat{\mu}^*$, which attains the max in the RHS of Equation (13.32), produces an average reward of ρ^* .

We will now show that any policy that deviates from $\hat{\mu}^*$ will produce an average reward that is lower than or equal to ρ^* . This will establish that the policy $\hat{\mu}^*$ generates the maximum possible average reward and is therefore an optimal policy. Thus all we need to show is that a policy, $\hat{\mu}$, which does not necessarily attain the max in Equation (13.32), produces an average reward that is less than or equal to ρ^* .

Equation (13.32) can be written in vector form as:

$$\rho \vec{e} + \vec{J} = L(\vec{J}). \quad (13.36)$$

We will first prove that:

$$L_{\hat{\mu}}^k \vec{J} \leq k \rho^* \vec{e} + \vec{J}. \quad (13.37)$$

As before, we use an induction argument. Now from Equation (13.36)

$$L(\vec{J}) = \rho \vec{e} + \vec{J}.$$

But we know that

$$L_{\hat{\mu}}(\vec{J}) \leq L(\vec{J}),$$

which follows from the fact that any given policy may not attain the max in Equation (13.32). Thus:

$$L_{\hat{\mu}} \vec{J} \leq \rho \vec{e} + \vec{J}.$$

This proves that Equation (13.37) holds when $k = 1$. Assuming that it holds when $k = m$, we have that:

$$L_{\hat{\mu}}^m \vec{J} \leq m \rho^* \vec{e} + \vec{J}.$$

Using the fact that $L_{\hat{\mu}}$ is monotonic from the results presented in Section 4, it follows that

$$\begin{aligned} L_{\hat{\mu}}(L_{\hat{\mu}}^m) \vec{J} &\leq L_{\hat{\mu}}(m \rho^* \vec{e} + \vec{J}) \\ &= m \rho^* \vec{e} + L_{\hat{\mu}} \vec{J} \\ &\leq m \rho^* \vec{e} + \rho \vec{e} + \vec{J} \text{ (using Equation (13.36))} \\ &= (m+1) \rho^* \vec{e} + \vec{J}. \end{aligned} \quad (13.38)$$

This establishes Equation (13.37).

The following bears similarity to the proof of the first part of this proposition.

Using Lemma 13.2, which is given on page 348, we have for all i :

$$L_{\hat{\mu}}^k J(i) = E[A + \sum_{s=1}^k r(x_s, \mu(x_s), x_{s+1}) | x_1 = i],$$

where x_s is the state from which the s th jump of the Markov chain occurs and A is a finite quantity.

Using this and Equation (13.37), we have that:

$$E[A + \sum_{s=1}^k r(x_s, \mu(x_s)), x_{s+1} | x_1 = i] \leq k\rho^* + J(i).$$

Therefore,

$$\frac{E[A]}{k} + \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s)), x_{s+1} | x_1 = i]}{k} \leq \rho^* + \frac{J(i)}{k}.$$

Taking the limits with $k \rightarrow \infty$, we have:

$$\lim_{k \rightarrow \infty} \frac{E[\sum_{s=1}^k r(x_s, \mu(x_s)), x_{s+1} | x_1 = i]}{k} \leq \rho^*.$$

(As before, the above follows from the fact that

$$\lim_{k \rightarrow \infty} \frac{a}{k} = 0, \text{ if } a \text{ is finite.})$$

In words, this means that the average reward of the policy $\hat{\mu}$ is less than or equal to ρ^* . This implies that the policy that attains the max in the RHS of Equation (13.32) is indeed the optimal policy since no other policy can beat it.

7.2. Policy iteration for average reward MDPs

In this subsection, we will prove that policy iteration can generate an optimal solution to the average reward problem. We will assume that all states are recurrent under every allowable policy. This is not a very restrictive assumption. For a proof that relaxes this assumption, the reader is referred to Puterman (Chapter 8 of [140], page 380).

To prove the convergence of policy iteration under the condition of recurrence, we need Lemma 13.14 and Lemma 13.15.

LEMMA 13.14 *Let $\Pi_{\hat{\mu}}(i)$ denote the limiting probability of the i th state in a Markov chain run by the policy $\hat{\mu}$. If $p(i, \mu(i), j)$ denotes the element in the i th row and j th column of the transition probability matrix of the Markov chain run by the policy $\hat{\mu}$ then for $i, j \in \mathcal{S}$ (the set of states in the Markov chain)*

$$\sum_{i=1}^{|\mathcal{S}|} \Pi_{\hat{\mu}}(i) \left[\sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h(j) - h(i) \right] = 0,$$

where \vec{h} is any finite valued $|\mathcal{S}|$ -dimensional vector.

Proof In the following proof,

$$\sum_i \equiv \sum_{i=1}^{|S|}.$$

From Theorem 8.2 on page 144, $\vec{\Pi}\mathbf{P} = \vec{\Pi}$ one can write that for all $j \in \mathcal{S}$:

$$\sum_i \Pi_{\hat{\mu}}(i)p(i, \mu(i), j) = \Pi_{\hat{\mu}}(j),$$

which can be written as:

$$\sum_i \Pi_{\hat{\mu}}(i)p(i, \mu(i), j) - \Pi_{\hat{\mu}}(j) = 0.$$

Hence:

$$\sum_i \Pi_{\hat{\mu}}(i)p(i, \mu(i), j)h(i) - \Pi_{\hat{\mu}}(j)h(i) = 0 \quad \text{for all } i, j \in \mathcal{S}.$$

Then summing the LHS of the above equation over all j , one obtains:

$$\sum_j [\sum_i \Pi_{\hat{\mu}}(i)p(i, \mu(i), j)h(i) - \Pi_{\hat{\mu}}(j)h(i)] = 0. \quad (13.39)$$

Now Equation (13.39), by suitable rearrangement of terms, can be written as:

$$\sum_i \Pi_{\hat{\mu}}(i) \sum_j p(i, \mu(i), j)h(j) - \sum_i \Pi_{\hat{\mu}}(i)h(i) = 0.$$

This establishes Lemma 13.14. If the last step is not clear, see the Remark below. ■

Remark. The last step in the lemma above can be *verified* for a two-state Markov chain. From $\vec{\Pi}\mathbf{P} = \vec{\Pi}$, if \mathbf{P} denotes the transition probability matrix for policy $\hat{\mu}$, we have that:

$$\Pi(1)P(1, 1) + \Pi(2)P(2, 1) = \Pi(1)$$

and

$$\Pi(1)P(1, 2) + \Pi(2)P(2, 2) = \Pi(2).$$

(Note that in the above, $P(x, y) \equiv p(x, \mu(x), y)$.) Multiplying both sides of the first equation by $h(1)$ and those of the second by $h(2)$ and then adding the resulting equations one has that:

$$\Pi(1)P(1, 1)h(1) + \Pi(2)P(2, 1)h(1) + \Pi(1)P(1, 2)h(2) + \Pi(2)P(2, 2)h(2)$$

$$= \Pi(1)h(1) + \Pi(2)h(2).$$

This can be written, after rearranging the terms, as:

$$\begin{aligned} & \Pi(1)P(1,1)h(1) + \Pi(1)P(1,2)h(2) + \Pi(2)P(2,1)h(1) + \Pi(2)P(2,2)h(2) \\ & - \Pi(1)h(1) - \Pi(2)h(2) = 0, \end{aligned}$$

which can be written as:

$$\sum_{i=1}^2 \Pi(i) \sum_{j=1}^2 p(i, \mu(i), j)h(j) - \sum_{i=1}^2 \Pi(i)h(i) = 0.$$

This should give some insight into the last step of the proof of Lemma 13.14.

We now review the policy iteration algorithm.

Steps in policy iteration for average reward MDPs..

Step 1. Set $k = 0$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select an arbitrary policy. Let us denote the policy selected in the k th iteration by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation:) Solve the following linear system of equations.

$$h^k(i) = \bar{r}(i, \mu_k(i)) - \rho^k + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j)h^k(j).$$

Here one linear equation is associated with each value of i . In this system, the unknowns are the h^k terms and ρ^k .

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg \max_{a \in \mathcal{A}(i)} \bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)h^k(j).$$

If possible one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i then stop and set $\mu^*(i) = \mu_k(i)$ for every i . Otherwise, increment k by 1 and go back to the second step.

We next present Lemma 13.15 which will be used directly in proving the convergence of policy iteration. Lemma 13.15 will use Lemma 13.14.

LEMMA 13.15 *If ρ^k denotes the average reward (determined using the policy evaluation step) in the k th iteration of the policy iteration algorithm, then*

$$\rho^{k+1} \geq \rho^k.$$

Proof We know from our discussions in Chapter 9 that the **average reward of a policy $\hat{\mu}$** can be written as:

$$\sum_i \Pi_{\hat{\mu}}(i) \bar{r}(i, \mu(i))$$

where $\vec{\Pi}_{\hat{\mu}}$ denotes the limiting probability vector of the Markov chain associated with the policy $\hat{\mu}$.

From Proposition 13.13, we know that the term ρ in the Bellman equation for a policy $\hat{\mu}$ denotes the average reward associated with the policy. Hence we can write an expression for the average reward in the $(k+1)$ th iteration of policy iteration as:

$$\begin{aligned} \rho^{k+1} &= \sum_i \Pi_{\hat{\mu}_{k+1}}(i) \bar{r}(i, \mu_{k+1}(i)) \\ &= \rho^k - \rho^k + \sum_i \Pi_{\hat{\mu}_{k+1}}(i) \bar{r}(i, \mu_{k+1}(i)) \\ &= \rho^k + \sum_i \Pi_{\hat{\mu}_{k+1}}(i) [\bar{r}(i, \mu_{k+1}(i)) - \rho^k] \\ &= \rho^k + \sum_i \Pi_{\hat{\mu}_{k+1}}(i) [\bar{r}(i, \mu_{k+1}(i)) - \rho^k] \\ &\quad + \sum_i \Pi_{\hat{\mu}_{k+1}}(i) [\sum_j p(i, \mu_{k+1}(i), j) h^k(j) - h^k(i)] \\ &\quad \text{(using Lemma 13.14)} \\ &= \rho^k + \sum_i \Pi_{\hat{\mu}_{k+1}}(i) [\bar{r}(i, \mu_{k+1}(i)) - \\ &\quad \rho^k + \sum_j p(i, \mu_{k+1}(i), j) h^k(j) - h^k(i)]. \end{aligned} \tag{13.40}$$

Now, from the policy improvement step, it is clear that μ_{k+1} is chosen in a way such that :

$$\bar{r}(i, \mu_{k+1}(i)) + \sum_j p(i, \mu_{k+1}, j) h^k(j) \geq \bar{r}(i, \mu_k(i)) + \sum_j p(i, \mu_k, j) h^k(j)$$

which implies that for each i :

$$\begin{aligned} \bar{r}(i, \mu_{k+1}(i)) + \sum_j p(i, \mu_{k+1}, j) h^k(j) - \rho^k - h^k(i) &\geq \\ \bar{r}(i, \mu_k(i)) + \sum_j p(i, \mu_k, j) h^k(j) - \rho^k - h^k(i). \end{aligned}$$

But the policy evaluation stage of the policy iteration algorithm implies that the RHS of the above inequation equals 0. Thus for each i :

$$\bar{r}(i, \mu_{k+1}(i)) + \sum_j p(i, \mu_{k+1}, j) h^k(j) - \rho^k - h^k(i) \geq 0. \tag{13.41}$$

Now, (13.41) combined with the last equation of (13.40) implies the following:

$$\rho^{k+1} \geq \rho^k. \quad \blacksquare$$

The next proposition will establish the convergence of policy iteration under the assumption of recurrence.

PROPOSITION 13.16 *The policy $\hat{\mu}^*$ generated by the policy iteration algorithm for average reward MDPs is an optimal policy if all states are recurrent.*

Proof From Lemma 13.15, the sequence of average rewards is an increasing sequence till a policy repeats. Since the number of states and actions is finite, there is a finite number of policies, and the convergence criterion $\hat{\mu}_{k+1} = \hat{\mu}_k$ must be satisfied at a finite value of k . When the policy repeats, we have that $\hat{\mu}_{k+1} = \hat{\mu}_k$ which means from the policy improvement and the policy evaluation steps that the policy $\hat{\mu}_k$ satisfies the Bellman equation. From Proposition 13.13, this implies that $\hat{\mu}_k$ is the optimal policy. \blacksquare

7.3. Value Iteration for average reward MDPs

Convergence analysis of value iteration for average reward gets a little more complicated than the same for discounted reward. The Bellman optimality equation cannot be used directly because the average reward of the optimal policy (ρ^*) is not known beforehand (since the optimal policy is not known beforehand). A way to work around this problem is to set ρ^* to 0 or some constant. It is okay to do this since the values that the modified Bellman optimality equation (with a replaced ρ^*) generates differ from those generated by the actual Bellman optimality equation by a constant value. This does not affect the policy selected in any stage of the value iteration algorithm.

The major difficulty with value iteration, as discussed in previous chapters, is that due to the lack of a discounting factor the value iteration mapping is not contractive and as such the max norm cannot be used for ensuring convergence. Hence the chief difficulty is to find a mechanism for terminating the algorithm.

In addition, some of the iterates (or values) become unbounded, that is, they converge to negative or positive infinity. One way around this is to subtract a pre-determined element of the value vector in Step 2 of value iteration. This ensures that the values themselves remain bounded. This is called relative value iteration.

The **span seminorm** can be very useful in identifying the optimal policy. In the literature, the convergence analysis of average reward value iteration is under restrictive assumptions, and a very general proof is yet to be devised. We will present a proof for recurrent Markov chains.

Convergence of **regular** value iteration using a span semi-norm has been shown in Puterman [140]. The convergence of **relative** value iteration has been established using a different approach in Bertsekas [20].

We present next a proof for the convergence of regular value iteration under the assumption that all states in the Markov chain are recurrent under any allowable policy.

The central idea underlying the convergence proof of value iteration for *discounted* reward MDPs was that of convergence in the norm; that is with successive iterations of the algorithm, the norm of the “difference vector” became smaller and smaller. In value iteration for average reward, we will not be able to show that. Instead we will show convergence in the span seminorm. Let us quickly review the definition of the span seminorm (or span).

The span of the vector \vec{x} , denoted by $sp(\vec{x})$, is defined as:

$$sp(\vec{x}) = \max_i x(i) - \min_i x(i).$$

Example: $\vec{a} = (9, 2)$ and $\vec{b} = (-3, 8, 16)$. Then $sp(\vec{a}) = 7$ and $sp(\vec{b}) = 19$.

In order to show that the span of the “difference vector” in value iteration converges to 0, we need the following theorem.

THEOREM 13.17 *Suppose F is an M -stage span contraction mapping; that is, for any two vectors \vec{u} and \vec{w} in a given vector space, for some positive, finite, and integral value of M ,*

$$sp(F^M \vec{u} - F^M \vec{w}) \leq \alpha sp(\vec{u} - \vec{w}) \quad (13.42)$$

for $0 \leq \alpha < 1$.

Further suppose that there is a sequence $\{\vec{v}^k\}_{k=1}^\infty$ defined by:

$$\vec{v}^{k+1} = F\vec{v}^k = F^{k+1}\vec{v}^0$$

then, given an $\epsilon > 0$, there exists an N such that

$$sp(\vec{v}^{kM+1} - \vec{v}^{kM}) < \epsilon$$

for all $k \geq N$.

We will not state the proof of this theorem. For more along these lines, and for a more rigorous treatment, the reader is referred to Puterman [140]. Basically, the theorem revolves around the fact that the span keeps decreasing with k , if one can show the condition in (13.42).

In order to prove that value iteration converges, one needs to show that the span of the difference between vectors generated in successive iterations keeps decreasing with every iteration. This is going to be the central idea underlying the convergence proof of value iteration. Let us first review the steps in value iteration for average reward MDPs.

Steps in value iteration for average reward. The algorithm description comes from Chapter 8.

Step 1: Set $k = 0$ and select an arbitrary vector \vec{J}^0 . Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_j p(i, a, j) J^k(j)].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} [\bar{r}(i, a) + \sum_j p(i, a, j) J^k(j)]$$

and stop. The ϵ -optimal policy is \hat{d} .

To prove the convergence of this algorithm, we need to prove two results, which will be presented in the form of lemmas, next.

LEMMA 13.18 *Let \mathbf{Q} be a matrix with $|\mathcal{S}|$ columns, the elements of each row in which are non-negative and sum to 1.*

Let us define $b(i, j, l)$ as

$$b(i, j, l) = \min_{l \in \mathcal{S}} \{Q(i, l), Q(j, l)\}.$$

In the above, both i and j are indices for the row in the matrix \mathbf{Q} and l is the index for the column.

Let us then define $B(i, j)$ as:

$$B(i, j) = \sum_{l \in \mathcal{S}} b(i, j, l),$$

and α as:

$$\alpha = 1 - \min_{i, j} B(i, j).$$

Let \vec{x} be any arbitrary column vector with $|\mathcal{S}|$ components. Now, if we denote:

$$\mathbf{Q}' \equiv \mathbf{Q}\vec{x},$$

we claim that

$$sp(\mathbf{Q}') \leq \alpha sp(\vec{x}).$$

Proof In the above, \mathbf{Q} may not be a square matrix. However, the number of columns in this matrix is $|\mathcal{S}|$. Since \vec{x} is a column vector with $|\mathcal{S}|$ rows, it is clear that $\mathbf{Q}\vec{x}$ is a column vector with as many rows as \mathbf{Q} .

Then

$$\begin{aligned}
 & Q'(i) - Q'(j) \\
 &= \sum_{l \in S} Q(i, l)x(l) - \sum_{l \in S} Q(j, l)x(l) \\
 &= \sum_{l \in S} [Q(i, l) - b(i, j, l)]x(l) - \sum_{l \in S} [Q(j, l) - b(i, j, l)]x(l) \\
 &\leq \sum_{l \in S} [Q(i, l) - b(i, j, l)] \max_{l \in S} x(l) - \sum_{l \in S} [Q(j, l) - b(i, j, l)] \min_{l \in S} x(l) \\
 &= [1 - \sum_{l \in S} b(i, j, l)] \left[\max_{l \in S} x(l) - \min_{l \in S} x(l) \right] \\
 &= [1 - \sum_{l \in S} b(i, j, l)] sp(\vec{x}). \\
 &\leq [1 - \min_{i,j} \sum_{l \in S} b(i, j, l)] sp(\vec{x}) \\
 &= \alpha sp(\vec{x}).
 \end{aligned}$$

From the above, one can summarize that:

$$Q'(i) - Q'(j) \leq \alpha sp(\vec{x}).$$

But

$$\max_{i \in S} Q'(i) - \min_{i \in S} Q'(i) \leq Q'(i) - Q'(j).$$

This implies that:

$$\max_{i \in S} Q'(i) - \min_{i \in S} Q'(i) \leq \alpha sp(\vec{x}),$$

which, from the definition of span, can be written as:

$$sp(\vec{Q'}) \leq \alpha sp(\vec{x}). \quad \blacksquare$$

Note that the matrix \mathbf{Q} in the preceding result could be composed of two matrices stacked over each other. For example, consider the following example:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{bmatrix},$$

where \mathbf{P}_1 and \mathbf{P}_2 are two matrices. Let us further imagine that these two matrices are the one-step transition probability matrices associated with two different policies but raised to the power M . It is not difficult to see then that if

all the elements of both the matrices were to be non-zero, then it is quite likely that α will be strictly less than 1. For instance consider these cases:

$$\mathbf{P}_1 = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.4 & 0.3 \\ 0.8 & 0.1 & 0.1 \end{bmatrix}$$

and

$$\mathbf{P}_2 = \begin{bmatrix} 0.5 & 0.4 & 0.1 \\ 0.2 & 0.3 & 0.5 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}$$

Then:

$$\mathbf{Q} = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.4 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.5 & 0.4 & 0.1 \\ 0.2 & 0.3 & 0.5 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}.$$

For \mathbf{Q} ,

$$b(1, 2, 1) = \min\{0.2, 0.3\} = 0.2,$$

$$b(1, 2, 2) = \min\{0.1, 0.4\} = 0.1,$$

$$b(1, 2, 3) = \min\{0.7, 0.3\} = 0.3,$$

together imply that

$$B(1, 2) = 0.2 + 0.1 + 0.3 = 0.6.$$

In this way, all the $B(., .)$ terms can be calculated. If you work out the values of all these terms, from its definition in Lemma 13.18, you can work out the value of α to be less than 1.

In case of recurrent states, α is likely to be less than 1. (To this there is an exception. For the case where all the elements of the matrix are equal, $\alpha = 1$. However, this case has an easy solution — any action is equally good.)

When $\alpha = 1$, we do not have a span contraction. Now since each of the \mathbf{P}_i ($i = 1, 2$) matrices are the transition probability matrices raised to some finite power, we may not obtain $\alpha < 1$ in case of Markov chains with transient states or absorbing states, since α may be 1.

Let us now turn our attention to the second lemma that will be needed for analyzing the convergence of value iteration.

LEMMA 13.19 *Let L denote the Bellman optimality operator for average reward (the transformation applied in the value iteration algorithm) and let M be a positive finite integer.*

We next define some notation. For any $i \in S$,

$$d_{x^k}(i) = \arg \max_{a \in A(i)} [\bar{r}(i, a) + \sum_{j \in S} p(i, a, j)x^k(j)], \quad (13.43)$$

$$L_{d_{x^k}} y^k(i) = [\bar{r}(i, d_{x^k}(i)) + \sum_{j \in S} p(i, d_{x^k}(i), j)y^k(j)], \quad (13.44)$$

and

$$Lx^k(i) \equiv L_{d_{x^k}} x^k(i). \quad (13.45)$$

Consider two vectors \vec{v}^1 and \vec{u}^1 in a given vector space.

Let α be defined as in Lemma 13.18 with the matrix \mathbf{Q} of Lemma 13.18 replaced by the following stacked matrix

$$\begin{bmatrix} \mathbf{P}_{d_{u^1}} \\ \mathbf{P}_{d_{v^1}} \end{bmatrix},$$

where \mathbf{P}_μ denotes the transition probability matrix associated with $\hat{\mu}$. Then,

$$sp(L^M \vec{v}^1 - L^M \vec{u}^1) \leq \alpha sp(\vec{v}^1 - \vec{u}^1).$$

Proof Let states s^* and s_* be defined as follows:

$$s^* = \arg \max_{s \in S} \{L^M v^1(s) - L^M u^1(s)\},$$

and

$$s_* = \arg \min_{s \in S} \{L^M v^1(s) - L^M u^1(s)\}.$$

Now to follow the rest of the proof, one must understand the following:

$$\vec{u}^2 = L\vec{u}^1 = L_{d_{u^1}} \vec{u}^1, \vec{u}^3 = L^2 \vec{u}^1 = L_{d_{u^2}} L_{d_{u^1}} \vec{u}^1$$

and so on. Similarly,

$$\vec{v}^2 = L\vec{v}^1 = L_{d_{v^1}} \vec{v}^1, \vec{v}^3 = L^2 \vec{v}^1 = L_{d_{v^2}} L_{d_{v^1}} \vec{v}^1.$$

Then for any i ,

$$L^M u^1(i) = L_{d_{u^M}} L_{d_{u^{M-1}}} \dots L_{d_{u^2}} L_{d_{u^1}} u^1(i) \quad (13.46)$$

and

$$L^M v^1(i) \geq L_{d_{u^M}} L_{d_{u^{M-1}}} \dots L_{d_{u^2}} L_{d_{u^1}} v^1(i). \quad (13.47)$$

The \geq relation in Equation (13.47) follows from the fact that

$$L^M v^1(i) \geq L_{d_u n} v^1(i) \text{ for any integer value of } n.$$

The above should be clear from the definitions in (13.43, 13.44, and 13.45).

From (13.46) and (13.47), it follows that

$$\begin{aligned} & L^M v^1(s^*) - L^M u^1(s^*) \\ & \geq [L_{d_u M} L_{d_{uM-1}} \dots L_{d_{u1}} v^1(s^*)] - [L_{d_u M} L_{d_{uM-1}} \dots L_{d_{u1}} u^1(s^*)] \\ & = [\bar{r}(s^*, d_{u1}(s^*)) + \bar{r}(s^*, d_{u2}(s^*)) + \dots + \bar{r}(s^*, d_{uM}(s^*)) + \\ & \quad \mathbf{P}_{d_u M} \mathbf{P}_{d_{uM-1}} \dots \mathbf{P}_{d_{u1}} v^1(s^*)] - \\ & \quad [\bar{r}(s^*, d_{u1}(s^*)) + \bar{r}(s^*, d_{u2}(s^*)) + \dots + \bar{r}(s^*, d_{uM}(s^*)) + \\ & \quad \mathbf{P}_{d_u M} \mathbf{P}_{d_{uM-1}} \dots \mathbf{P}_{d_{u1}} u^1(s^*)] \\ & = \mathbf{P}_{d_u} (v^1 - u^1)(s^*). \end{aligned} \tag{13.48}$$

In (13.48), please make note of the following.

1. We have replaced the product of matrices, using the following notation:

$$\mathbf{P}_{d_u M} \mathbf{P}_{d_{uM-1}} \dots \mathbf{P}_{d_{u1}} \equiv \mathbf{P}_{d_u}.$$

2. $\mathbf{P}_{d_u} (v^1 - u^1)(s^*)$ denotes the s^* th component of the vector $\mathbf{P}_{d_u} (\vec{v}^1 - \vec{u}^1)$. This vector is obtained by multiplying matrix \mathbf{P}_{d_u} by the vector $\vec{v}^1 - \vec{u}^1$.

Using logic similar to that used above:

$$L^M v^1(s_*) - L^M u^1(s_*) \leq \mathbf{P}_{d_v} (v^1 - u^1)(s_*), \tag{13.49}$$

where once again

$$\mathbf{P}_{d_v M} \mathbf{P}_{d_{vM-1}} \dots \mathbf{P}_{d_{v1}} \equiv \mathbf{P}_{d_v}.$$

Then,

$$\begin{aligned} & sp(L^M \vec{v}^1 - L^M \vec{u}^1) \\ & = \{L^M v^1(s^*) - L^M u^1(s^*)\} - \{L^M v^1(s_*) - L^M u^1(s_*)\} \\ & \leq \mathbf{P}_{d_{v1}} (v^1 - u^1)(s^*) - \mathbf{P}_{d_{u1}} (v^1 - u^1)(s_*) \text{ from (13.48) and (13.49)} \\ & \leq \max_{s \in S} \mathbf{P}_{d_{v1}} (v - u)(s) - \min_{s \in S} \mathbf{P}_{d_{u1}} (v - u)(s) \\ & \leq \max_{s \in S} \frac{[\mathbf{P}_{d_{v1}}]}{[\mathbf{P}_{d_{u1}}]} (v - u)(s) - \min_{s \in S} \frac{[\mathbf{P}_{d_{v1}}]}{[\mathbf{P}_{d_{u1}}]} (v - u)(s) \end{aligned}$$

$$\begin{aligned}
 & \text{(where } \frac{[\mathbf{A}]}{[\mathbf{B}]} \text{ denotes a stacked matrix)} \\
 &= sp \left(\frac{[\mathbf{P}_{d_{v1}}]}{[\mathbf{P}_{d_{u1}}]} (\vec{v}^1 - \vec{u}^1) \right) \\
 &\leq \alpha sp(\vec{v}^1 - \vec{u}^1).
 \end{aligned}$$

The last inequation follows from Lemma 13.18. ■

The following result formalizes the convergence of value iteration.

PROPOSITION 13.20 *Value iteration converges in the span if all states in every Markov chain in the MDP are recurrent under every policy, and every Markov chain in the MDP is regular; the policy \hat{d} generated by value iteration is ϵ -optimal.*

Proof Under the condition of recurrence for every policy and under the condition that the Markov chain in the MDP is regular for any given policy, the associated transition probability matrix, when raised to a sufficiently large number M , will yield an α — defined in Lemma 13.18 — that will be less than 1. Then from Lemma 13.19 and from Theorem 13.17, it follows that value iteration converges in the span. ■

Note that we have not proved that the iterates (the elements of the value function vector) will converge. In fact, the iterates in value iteration, themselves, may not converge and may actually become unbounded. It has been seen that *relative* value iteration keeps the iterates bounded. Convergence of relative value iteration, under certain conditions, has been established in Bertsekas [20]. We refer the interested reader to the latter for more material on this topic.

8. Convergence of DP schemes for SMDPs

The convergence of DP schemes such as policy iteration and value iteration for DTMDPs in the discounted case follows from the proofs of the MDP versions. The contraction factor in DTMDPs has a different form. It is $e^{\gamma t(i,a,j)}$, but the MDP proofs can be extended to it.

For the average reward case, also, the proof of policy iteration can be extended from the MDP case very easily. However, for value iteration, the method itself is approximate. The convergence analysis of DP algorithms for both average reward and discounted reward SMDPs are discussed in Puterman [140].

9. Convergence of Reinforcement Learning Schemes

RL (reinforcement learning) schemes are derived from dynamic programming Schemes, and yet they need a separate convergence analysis because mathematically the two schemes are quite different. The main difference of course

is in the use of the step size in RL, which is absent in the DP schemes. The use of the step size makes any RL scheme of the form of the so-called “stochastic approximation scheme,” which is often abbreviated as the s.a. scheme.

Convergence analysis of many RL algorithms uses the idea of an ordinary (deterministic) differential equation underlying a stochastic approximation scheme. It can be shown that many stochastic approximation schemes are *tracked* by ordinary differential equations (ODEs). In other words, it can be shown that, associated with a stochastic approximation scheme, there exists an ODE. Now, this is an old finding. What is more interesting is that several properties of the associated ODE can be exploited to prove the convergence of RL schemes. In the RL literature, the use of ODEs in proving convergence was pioneered by Borkar and his colleagues via a series of seminal papers. We will not delve into the details of the ODE analysis.

We will begin with a discussion on some background material that will be needed to understand the convergence analysis of RL algorithms.

10. Background Material for RL Convergence

This section is divided into a few subsections. The first and second subsections will respectively discuss the notion of non-expansive mappings and Lipschitz continuity — concepts that we will need in the subsequent convergence analysis.

10.1. Non-Expansive Mappings

DEFINITION 13.2 A mapping (or transformation) F is said to be a non-expansive mapping in \mathcal{R}^n if:

$$\|F\vec{v} - F\vec{u}\| \leq \|\vec{v} - \vec{u}\|$$

for all \vec{v}, \vec{u} in \mathcal{R}^n .

10.2. Lipschitz Continuity

First, read the definition of continuity (Definition 12.1) of functions from Chapter 12. Lipschitz continuity can be formally defined as follows [112].

DEFINITION 13.3 The continuous function $f(x)$ is said to Lipschitz continuous on the set \mathcal{X} if a constant K (a finite number) exists such that for all $x_1, x_2 \in \mathcal{X}$

$$\|f(x_2) - f(x_1)\| \leq K\|x_2 - x_1\|.$$

It can be shown that linear functions are Lipschitz continuous.

10.3. Convergence of a sequence with probability 1

The notion of convergence with probability 1 needs to be understood at this stage. This concept is somewhat different from the idea of convergence of a sequence discussed in Chapter 11.

DEFINITION 13.4 A sequence $\{x^k\}_{k=0}^{\infty}$ of random variables is said to converge to a real number x^* with probability 1, if for a given $\epsilon > 0$ and a given $\delta > 0$, there exists an N such that

$$\text{Prob}[|x^k - x^*| < \epsilon] > 1 - \delta$$

for all $k \geq N$. Alternatively,

$$\text{Prob}\left[\lim_{k \rightarrow \infty} x^k = x^*\right] = 1.$$

In the above definition, note that N may depend on both ϵ and δ . The definition implies that if a sequence tends to a limit with probability 1, then you can make δ as small as you want, and still obtain a finite value for N , at which the sequence is arbitrarily close (ϵ -close) to the limit. This type of convergence is also called “almost sure” convergence.

The definition given above could be extended to a vector via replacement of $|x^k - x^*|$ by the norm $\|\vec{x}^k - \vec{x}^*\|$.

Note that in the definition above, the sequence does not have to be a sequence of *independent* random variables.

The strong law of large numbers uses this concept of convergence. However, in the strong law of large numbers, the sequence is defined to be one of independent random variables.

When we have convergence of this kind, for a given value of $\epsilon > 0$, the probability with which the sequence value can deviate from the limit by an absolute amount greater than ϵ can be made as small as one wants. However, there will always be that small probability (at most δ) with which the value may differ from the limit by a distance *greater* than ϵ . Note that in our previous definition of convergence in Chapter 11, there is no such probability.

11. Key Results for RL convergence

We will next present some key results in the context of convergence of RL schemes. The first result that we present is related to synchronous conditions. Now, in a simulator, RL is, almost surely, never synchronous. However, we will first show below (in Theorem 13.21) that, if used synchronously (as in DP schemes), the scheme converges and then in the next result (Proposition 13.26), we will show that the behavior of the asynchronous scheme can be made to approximate that of its synchronous counterpart, under certain conditions.

11.1. Synchronous Convergence

THEOREM 13.21 Consider a stochastic approximation scheme given by:

$$x_l^{k+1} = x_l^k + \alpha^k (H(\vec{x}^k) - x_l^k + w_l^k), \quad l = 1, 2, \dots, n,$$

where x_l denotes the Q -factor associated with the l th state-action pair, n denotes the number of state-action pairs, α^k is the step size (learning rate) at the k th iteration, H denotes a function on the vector \vec{x}^k from R^n to R^n and w_l^k denotes the noise generated in the k th iteration when the Q -factor of the l th state-action pair is updated. Assume that the scheme is used synchronously. In addition, assume the following conditions to hold.

Condition 1. The function H is Lipschitz continuous.

Condition 2. The step size α^k satisfies the following conditions:

$$\sum_{k=0}^{\infty} \alpha^k = \infty,$$

$$\sum_{k=0}^{\infty} (\alpha^k)^2 < \infty.$$

Condition 3. The iterates (\vec{x}^k) are bounded.

Condition 4. For all l ,

$$E[w_l^k | x^0, x^1, \dots, x^k, w_l^0, w_l^1, \dots, w_l^{k-1}] = 0,$$

$$E[||w_l^k||^2 | x^0, x^1, \dots, x^k, w_l^0, w_l^1, \dots, w_l^{k-1}] \leq A_1 + A_2 ||x^k||^2,$$

for some constants A_1 and A_2 , and where $||.||$ indicates some norm.

Condition 5. The function H is non-expansive with respect to the max norm.

Then, with probability 1, the sequence $\{\vec{x}^k\}$ converges to a fixed point of H , provided H has a fixed point.

Several comments are in order.

Comment 1. The result presented above can be interpreted as a combination of two important results — one from Kushner and Clark [100] and the other from Borkar and Soumyanath [29]. In fact, Kushner and Clark's result is more restrictive than Theorem 13.21. In Kushner and Clark's result, Condition 5 would require that H be contractive. Condition 5 requires non-expansiveness, which is a weaker condition than contraction. This is possible due to the result in Borkar and Soumyanath's paper [29].

Comment 2. A version of this result can be found in Abounadi, Bertsekas and Borkar [3].

Comment 3. The term noise in the scheme indicates the fact that the mapping H may be corrupted; this is due to the fact that the scheme is simulation-based. How exactly this term appears in RL schemes will become clear when we use this result in the context of Q -Learning.

Comment 4. The proof of this result makes use of the martingale convergence theorem, the underlying ODE, its stability, and its critical points. We are skipping the proof because of its somewhat technical nature. It is important, however, to understand the statement of Theorem 13.21. Basically, the theorem says that under the assumptions stated, the scheme will converge to a fixed point of the function H , provided a fixed point exists. We will have more to say about the assumptions, when we use this result in the context of RL convergence.

11.2. Asynchronous Convergence

Let us first review the difference between a synchronous and an asynchronous algorithm. The difference is best explained with an example.

Consider a problem with three states, which are numbered 1 through 3. In a synchronous algorithm, one would first go to state 1, update value(s) associated with it, then go to state 2, update its value(s), and then go to state 3 and update its value(s). Then one would return to state 1, and this would continue. In asynchronous updating, the order of updating is haphazard. It could be:

$$1 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow \dots$$

We have shown above that the RL schemes can converge *synchronously*. Now, in this section, we will show that *asynchronously* the values of iterates can be made sufficiently close to the values obtained synchronously — as long as the step size is sufficiently small (and some other conditions are met). We will also show that if the values are sufficiently close, it is possible that the asynchronous algorithm will return a policy identical to that returned by the synchronous algorithm. We refer to this behavior as the *mimicry of the synchronous algorithm by the asynchronous algorithm*.

This mimicry will be proved in Proposition 13.26, which is the main result of this subsection. This result will need Proposition 13.25, which in turn requires Lemmas 13.22, 13.23, and 13.24.

LEMMA 13.22 *An RL scheme can be written in the form:*

$$x^{k+1} = (1 - \alpha^{k+1})x^k + \alpha^{k+1}V^{k+1} \quad (13.50)$$

where V can take on various forms. Now x^k , which is the iterate x that has been updated k times, can be written as:

$$x^k = A^k + B^k, \quad (13.51)$$

if

$$A^k = (1 - \alpha^1)(1 - \alpha^2) \dots (1 - \alpha^{k-1})(1 - \alpha^k)x^0$$

and

$$B^k = \alpha^k V^k + \sum_{i=1}^{k-1} C^k(i),$$

where

$$C^k(i) = \alpha^i V^i (1 - \alpha^{i+1})(1 - \alpha^{i+2}) \dots (1 - \alpha^k).$$

Proof The proof will be by induction. We will first prove it when $k = 1$. Now x is generated by transformation (13.50). Hence

$$x^1 = (1 - \alpha^1)x^0 + \alpha^1 V^1.$$

Now from Equation (13.51) we have : $A^1 = (1 - \alpha^1)x^0$ and $B^1 = \alpha^1 V^1$ thereby satisfying relation (13.51) if $k = 1$. Note that this case is not very difficult to verify because here C terms are 0; however the case in which $k = 2$ can also be similarly verified.

We now assume that our result is true when $k = m$. Therefore $x^m = A^m + B^m$. Now from transformation (13.50), we can write that

$$\begin{aligned} x^{m+1} &= (1 - \alpha^{m+1})x^m + \alpha^{m+1} V^{m+1} \\ &= (1 - \alpha^{m+1})[A^m + B^m] + \alpha^{m+1} V^{m+1} \\ &= (1 - \alpha^{m+1})[(1 - \alpha^1)(1 - \alpha^2) \dots (1 - \alpha^m)x^0] \\ &\quad + (1 - \alpha^{m+1})[\alpha^m V^m + \sum_{i=1}^{m-1} C^m(i)] + \alpha^{m+1} V^{m+1} \\ &= A^{m+1} + \sum_{i=1}^m C^{m+1}(i) + \alpha^{m+1} V^{m+1} \\ &= A^{m+1} + B^{m+1}. \end{aligned}$$

Thus the result is true if $k = m + 1$. ■

The next lemma will use the lemma proved above, and will be needed in the proof of Proposition 13.25.

LEMMA 13.23 *Let x^k denote the iterate that has been updated k times synchronously, y^k denote the same iterate (that is the iterate associated with the same state-action pair) that has been updated k times asynchronously, and let the step size in the stochastic approximation scheme be denoted by α . Let us further assume that: $(\alpha)^\kappa$, which denotes α raised to the power κ , is much smaller than 1 for $\kappa \geq 2$. Then if $x^0 = y^0$ and $f \neq g$, but $|f - g|$ is finite, the difference between x^g and y^f is of the order of α .*

Proof Now let us first assume that:

$$f = g + \gamma, \text{ where } \gamma > 0.$$

Consider the stochastic approximation scheme defined in Equation (13.50). Using Lemma 13.22, we have that

$$y^f - x^g = y^{g+\gamma} - x^g = A_y^{g+\gamma} - A_x^g + B_y^{g+\gamma} - B_y^g,$$

where, from the definitions in Lemma 13.22,

$$A_t^k = (1 - \alpha^1)(1 - \alpha^2) \dots (1 - \alpha^{k-1})(1 - \alpha^k)t^0$$

and

$$B_t^k = \alpha^k V^k + \sum_{i=1}^{k-1} C_t^k(i),$$

in which

$$C_t^k(i) = \alpha^i V^i (1 - \alpha^{i+1})(1 - \alpha^{i+2}) \dots (1 - \alpha^k)$$

for $t = x$ and $t = y$. Now consider the following product.

$$\begin{aligned} & (1 - a_1)(1 - a_2) \dots (1 - a_N) \\ &= 1 - (a_1 + a_2 + \dots + a_N) \end{aligned} \quad (13.52)$$

$$+(a_1 a_2 + a_1 a_3 + \dots) - (a_1 a_2 a_3 + \dots) + \dots + (-1)^N (a_1 a_2 \dots a_N)$$

It is to be noted that all terms in the right hand side *after* line (13.52) are products of two or more of the a terms. If each a term is less than 1 and if we can neglect terms which are products of two or more of these a terms, then:

$$(1 - a_1)(1 - a_2) \dots (1 - a_N) \simeq 1 - (a_1 + a_2 + \dots + a_N). \quad (13.53)$$

Now

$$\begin{aligned} A_y^{g+\gamma} - A_x^g &= [1 - \alpha^1][1 - \alpha^2] \dots [1 - \alpha^{g+\gamma}]y^0 - [1 - \alpha^1][1 - \alpha^2] \dots [1 - \alpha^g]x^0 \\ &\simeq [1 - \alpha^1 - \alpha^2 - \dots - \alpha^{g+\gamma}]x^0 - [1 - \alpha^1 - \alpha^2 - \dots - \alpha^g]x^0 \text{ (using } x^0 = y^0) \\ &= -[\alpha^{g+1} + \alpha^{g+2} + \dots + \alpha^{g+\gamma}]x^0, \end{aligned}$$

where the approximate equality is obtained from use of (13.53). It is thus clear that $A_y^{g+\gamma} - A_x^g$ is of the order of α . Since the B terms, by their definitions, are of the order of α , the difference in the B terms is also of the order of α . Hence the difference $y^{g+\gamma} - x^g$ is of the order of α . The above can be similarly proved when

$$f = g + \gamma, \text{ where } \gamma < 0. \quad \blacksquare$$

The above result says that the difference between the value of an iterate that is updated synchronously k times and the value of the same iterate that is updated asynchronously $k + \gamma$ times is of the order of α . Clearly, if $|\gamma|$ is very large, the difference will be large too, and that it is of the order of α may have no significance.

Finally, we need one more lemma in order to establish Proposition 13.25.

LEMMA 13.24 *The difference between a Q-factor of a given state associated with action u_1 , updated k times, and the Q-factor associated with the same state but a different action u_2 , also updated k times, is of the order of α , neglecting higher order terms of α , which, as usual, denotes the step size.*

Proof The proof will be by induction. The notation $O((\alpha)^\kappa)$ will be used to denote a function that is of the order $(\alpha)^\kappa$ in which each term is of the order of $(\alpha)^\kappa$ (that is, α raised to the power κ) or higher orders of α . Mathematically, our claim in this result is:

$$Q^k(i, u_1) - Q^k(i, u_2) = O(\alpha).$$

Now, in general, any RL scheme follows the following relation:

$$Q^{k+1}(i, u) = Q^k(i, u) + O(\alpha). \quad (13.54)$$

Now, when $k = 1$,

$$Q^1(i, u_1) = Q^1(i, u_2).$$

Hence, using the above and (13.54), we have that:

$$Q^2(i, u_1) - Q^2(i, u_2) = O(\alpha) - O(\alpha) = O(\alpha).$$

Thus the result is true when $k = 1$.

Assume that the result is true when $k = m$. Then

$$Q^m(i, u_1) - Q^m(i, u_2) = O(\alpha).$$

Using the above and (13.54), we have that:

$$\begin{aligned} Q^{m+1}(i, u_1) - Q^{m+1}(i, u_2) &= [Q^m(i, u_1) + O(\alpha)] - [Q^m(i, u_2) + O(\alpha)] \\ &= O(\alpha) \text{ (neglecting higher order terms of } \alpha.) \end{aligned}$$

This proves our claim when $k = m + 1$. ■

We are now at a position to prove Proposition 13.25. We will use the case of Q-Learning for discounted reward MDPs as an example. However, the result can be extended to *some* other RL algorithms. We will discuss one extension. For Proposition 13.25, we need some background that we develop next.

Error and degree of asynchronism . We define the **age** of an iterate to be the number of times it has been updated. In synchronous updating, iterates of identical age are used in the same transformation. However, in asynchronous updating, there may be an “age gap” or “age difference” in the iterates being used in the *same* transformation. Since the value of an iterate may alter with its age, a transformation with all iterates of the same age and a transformation with iterates of different ages may not be equivalent. The difference in the values produced by the two transformations — one with a positive age gap and the other with no age gap will be referred to as the **error of asynchronism**; the age gap itself will be referred to as the **degree of asynchronism**.

Intuitively, our next result rests on the following notion. If the maximum age gap (degree of asynchronism) between iterates used in an update is finite, then by choosing sufficiently small values for the step size, the error of asynchronism can be made arbitrarily small — that is negligible. The next result will also show that the error vanishes in the limit with a diminishing step-size. Hence a diminishing step size can be helpful. Since asynchronous conditions cannot be avoided in the simulator in which RL is performed, a finite age gap between iterates exists. Ways to work around this difficulty must be identified.

Consider the *Q*-Learning algorithm’s main transformation.

$$Q(i, a) \leftarrow Q(i, a) + \alpha[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j, b) - Q(i, a)],$$

where $Q(i, a)$ stands for the *Q*-factor associated with the state-action pair (i, a) . Now, let us rewrite the above transformation using an enhanced notation — in which the age of the iterate is *not* suppressed. Then the above transformation, *under asynchronous conditions*, may be written as:

$$y^{g+1}(i, a) = y^g(i, a) + \alpha[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} y^f(j, b) - y^g(i, a)], \quad (13.55)$$

where $y^k(i, a)$ denotes the *Q*-factor for (i, a) that has been iterated (updated) k times (in other words, its “age” is k) asynchronously. It is to be noted that if $f \neq g$, transformation (13.55) becomes *asynchronous*. A synchronous version would be

$$x^{g+1}(i, a) = x^g(i, a) + \alpha[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} x^g(j, b) - x^g(i, a)]. \quad (13.56)$$

From the above definitions, the error of asynchronism for a given state-action pair (i, a) , in the k th update of the *Q* factor associated with that pair, can be mathematically defined to be:

$$\Delta^k(i, a) \equiv |y^k(i, a) - x^k(i, a)|. \quad (13.57)$$

Remember that x is the synchronously updated iterate, while y is the asynchronously updated iterate.

In this background, we state Proposition 13.25, next.

PROPOSITION 13.25 *Consider the definitions of x^k and y^k in Lemma 13.23 and transformations (13.55) and (13.56). Assume that*

$$x^0(i, a) = y^0(i, a) \quad \forall (i, a).$$

1. *For a given value of $\epsilon > 0$, there exists a rule for the step size α such that the error of asynchronism $\Delta^g(i, a)$, which is defined in Equation (13.57), associated with any state-action pair (i, a) in the g th iteration is less than ϵ .*
2. *With $g \rightarrow \infty$, this error of asynchronism asymptotically converges to 0 with a diminishing step size, as long as the degree of asynchronism is finite.*

Proof From the two transformations (13.55) and (13.56), for a given state-action pair (i, a) , the absolute value of the difference in the Q -factors obtained from these two transformations, which is also the error of asynchronism, can be written as

$$\begin{aligned} \Delta^{g+1}(i, a) &\leq (1 - \alpha)|y^g(i, a) - x^g(i, a)| + \\ &\quad |\alpha^g[\max_{b \in \mathcal{A}(j)} x^f(j, b) - \max_{b \in \mathcal{A}(j)} y^g(j, b)]| \end{aligned} \quad (13.58)$$

$$\begin{aligned} &\leq |y^g(i, a) - x^g(i, a)| + |\alpha^g[\max_{b \in \mathcal{A}(j)} x^f(j, b) - \max_{b \in \mathcal{A}(j)} y^g(j, b)]| \\ &= \Delta^g(i, a) + |\alpha^g X|. \end{aligned} \quad (13.59)$$

The value of X in the above depends on the maximizing action in (13.58). There are, actually, two scenarios here for X .

If

$$\arg \max_{b \in \mathcal{A}(j)} x^f(j, b) = \arg \max_{b \in \mathcal{A}(j)} y^g(j, b) = z,$$

then we have Case (1): in which

$$X = x^f(j, z) - y^g(j, z).$$

Otherwise, we have Case (2): in which

$$X = x^f(j, u_1) - y^g(j, u_2).$$

Let us consider Case (1) first. From Lemma 13.23, $X = [y^g(j, z) - x^f(j, z)]$ itself is of the order of α .

Let us consider Case (2) next.

$$\begin{aligned}
 X &= x^f(j, u_1) - y^g(j, u_2) \\
 &= [x^f(j, u_1) - x^f(j, u_2)] + [x^f(j, u_2) - y^g(j, u_2)] \\
 &= [O(\alpha)] + [x^f(j, u_2) - y^g(j, u_2)] \text{ (using Lemma 13.24)} \\
 &= [O(\alpha)] + [O(\alpha)] \text{ (using Lemma 13.23)} \\
 &= O(\alpha)
 \end{aligned}$$

X is thus shown above to be of the order of α . Using this fact, Inequation (13.59) can be written as:

$$\Delta^{g+1}(i, a) \leq \Delta^g(i, a) + O(\alpha)^2 \quad (13.60)$$

for all (i, a) , since

$$\alpha^g O(\alpha) = O((\alpha)^2)$$

where $(\alpha)^2$ denotes α raised to the power 2.

We now claim that for $g = 0, 1, 2, \dots$,

$$\Delta^{g+1}(i, a) \leq O((\alpha)^2) \quad \forall (i, a).$$

The claim will be proved via an induction argument. From the assumption in the statement of this result, $\Delta^0(i, a) = 0$. Hence, setting $g = 1$ in Inequation (13.60), we have that:

$$\Delta^1(i, a) \leq 0 + O((\alpha)^2) = O((\alpha)^2),$$

which proves our claim for $g = 0$.

Assuming our claim is true when $g = n$, we have that:

$$\Delta^{n+1}(i, a) \leq O((\alpha)^2) \quad (13.61)$$

for all (i, a) . Then using $g = n + 1$ in Inequation (13.60), from Inequation (13.61), we have that

$$\Delta^{n+2}(i, a) \leq \Delta^{n+1}(i, a) + O((\alpha)^2) \leq O((\alpha)^2) + O((\alpha)^2) = O((\alpha)^2)$$

which proves our claim when $g = n + 1$.

This implies that $\Delta^{g+1}(i, a)$ must be less than or equal to a function of the order of $(\alpha)^2$. Now, since α is smaller than 1, $\Delta^{g+1}(i, a)$ must be less than a function of the order of the square of a very small quantity. Now, $\Delta^{g+1}(i, a)$ is the error of asynchronism for (i, a) . Furthermore, for a given value of $\epsilon > 0$, one can select the rule for the step size α in a manner such that $\Delta^{g+1}(i, a) < \epsilon$.

The error may thus be made as small as one wishes by a suitable selection of α , if γ is finite, where $|\gamma| = |g - f|$. The requirement of finiteness of γ

follows from the proof of Lemma 13.23. Note that in that proof of Lemma 13.23, although the difference $x^{k+\gamma} - x^k$ is of the order of α , its absolute value can become infinitely large. The latter happens if γ , which denotes the *degree of asynchronism*, tends to infinity.

Also it follows from the arguments made above that for a *diminishing* step size, when $g \rightarrow \infty$, α tends to 0. But when α tends to 0, $\Delta^{g+1}(i, a)$ tends to 0. In other words, the error vanishes in the limit. This result is true for any state-action pair (i, a) . ■

We next present a result that shows that under certain conditions the asynchronous algorithm can be made to mimic its synchronous counterpart. As stated earlier, by mimicking, we mean that the maximizing actions selected in the synchronous and asynchronous algorithms would be the same. The maximizing action associated with a state i is defined as:

$$\arg \max_{a \in \mathcal{A}(i)} Q(i, a).$$

PROPOSITION 13.26 *The maximizing action in the synchronous and asynchronous algorithms can be made identical in every iteration by selecting a sufficiently small value for α .*

Proof Let us assume that there are two actions associated with a given state i . The proof can be extended easily to multiple actions. We denote by u_1 and u_2 the two actions in question and without loss of generality assume that:

$$x(u_1) > x(u_2), \quad (13.62)$$

where $x(u_i)$ denotes the Q -factor associated with the synchronous algorithm for action u_i in a given state in a given iteration. The proof will be worked out for any state and any iteration. Let $y(u_i)$ denote the corresponding iterate in the asynchronous algorithm. Now, if we can show that

$$y(u_1) > y(u_2), \quad (13.63)$$

when Inequation (13.62) is true, our proof will be complete because it will imply that the maximizing action in the synchronous algorithm (u_1) will also be the maximizing action in the asynchronous algorithm.

Next, we need to define two quantities, which are:

$$e(u_i) \equiv |x(u_i) - y(u_i)| \text{ and } D \equiv x(u_1) - x(u_2).$$

Now, we will show that if

$$e(u_1) < D/2 \quad (13.64)$$

and

$$e(u_2) < D/2, \quad (13.65)$$

then Inequation (13.63) will be true and we will be done.

There are four possible cases here.

Case 1.

$$y(u_1) = x(u_1) + e(u_1) \text{ and } y(u_2) = x(u_2) + e(u_2),$$

Case 2.

$$y(u_1) = x(u_1) + e(u_1) \text{ and } y(u_2) = x(u_2) - e(u_2),$$

Case 3.

$$y(u_1) = x(u_1) - e(u_1) \text{ and } y(u_2) = x(u_2) + e(u_2),$$

and

Case 4.

$$y(u_1) = x(u_1) - e(u_1) \text{ and } y(u_2) = x(u_2) - e(u_2).$$

Let us consider Case 3.

$$\begin{aligned} & y(u_1) - y(u_2) \\ &= x(u_1) - x(u_2) - e(u_1) - e(u_2) \text{ (from Case 3)} \\ &= D - (e(u_1) + e(u_2)) \\ &> D - D \text{ (from Inequalities (13.64) and (13.65))} \\ &= 0 \end{aligned}$$

Thus $y(u_1) > y(u_2)$. This can be proved for each of the Cases, as long as Inequalities (13.64) and (13.65) hold.

We next need to identify conditions under which Inequalities (13.64) and (13.65) hold. By setting $\epsilon = D$ in Proposition 13.25 and noting that $e(u)$ is the error of asynchronism, we have from Proposition 13.25 that, with a suitable value of α in the RL scheme, Inequalities (13.64) and (13.65) hold. We are done because these two inequalities will imply Inequality (13.63). ■

We can thus conclude that by selecting a sufficiently small value for the step size, the difference between the asynchronous iterate and the synchronous iterate can be brought down to a value such that the maximizing action in both algorithms becomes identical. *The actual values of the iterates may be considerably different* since their difference, although proportional to $(\alpha)^2$, may be considerable. This means that in the asynchronous algorithm, the

Q -factors may never converge to the solution of the Q -factor version of the Bellman equation. As such, this notion of optimality is somewhat different from the notion of Bellman optimality that we used in convergence of dynamic programming algorithms. We achieve the Bellman optimal policy, but not the Bellman Q -factors.

The result agrees with our numerical experience, which tells us that if a small enough step size is used, the synchronous algorithm and the asynchronous algorithm converge to the same policy, although the individual values of the iterates are *quite different*. An interesting point, which we feel we should point out, is that in problems which have a treacherous structure — that is, problems in which D is a small quantity for many state-action pairs, the asynchronous algorithm may be fooled and may select the wrong action. Unfortunately, our analysis does not lead us to any upper bound for α , which is a controllable parameter in the system, that will ensure convergence. This is because D is unknown. This means that α must be selected with trial and error. Also, this result suggests that it is good to select a step size that diminishes with iterations. Then in the limit, the algorithm will exhibit good convergence properties because the error of asynchronism will also diminish with iterations.

What happens if D is 0? (We have left that as an exercise at the end of this chapter.)

The approach adopted above (which is from Gosavi [62]) is somewhat restrictive in the sense that one may have to work out the details for every algorithm separately. A more sophisticated approach to proving convergence of asynchronous schemes uses the method developed by Borkar [26]. Borkar has shown that, under certain conditions, the asynchronous algorithm tracks an ODE (Ordinary Differential Equation), similar to the one tracked by its synchronous counterpart and that the Q -factors do converge to the Bellman solution. We do not develop Borkar's asynchronous ODE theory here because it requires background material that we have not presented. Borkar's approach has also been used to prove the asynchronous convergence of Relative Q -learning (see [2]).

12. Convergence of RL based on value iteration

In this section, we will primarily deal with two algorithms of RL based on value iteration — the Q -Learning algorithm for discounted reward MDPs and the Relative Q -Learning algorithm for average reward MDPs.

12.1. Convergence of Q -Learning

We will now prove that Q -Learning (for discounted reward MDPs) converges under synchronous conditions. The convergence will follow from Theorem 13.21.

The core of the Q -Learning algorithm can be expressed by the following transformation:

$$Q^{k+1}(i, a) \leftarrow Q^k(i, a) + \alpha[r(i, a, \xi^k) + \lambda \max_{b \in \mathcal{A}(\xi^k)} Q^k(\xi^k, b) - Q^k(i, a)], \quad (13.66)$$

where ξ is a random variable that depends on (i, a) . Notice that in the past, we have used j in place of ξ . The reason for using ξ is that it is a **random** variable that does not assume a fixed value for given values of (i, a) .

We first need to show that this algorithm is of the form described in Theorem 13.21. To this end, we need to define some transformations, which we do next.

Let us define a transformation H on the vector Q^k as follows:

$$H(Q^k)(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j)[r(i, a, j) + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j, b)]. \quad (13.67)$$

Let us also define a transformation F on the same vector as follows:

$$F(Q^k(i, a)) = [r(i, a, \xi^k) + \lambda \max_{b \in \mathcal{A}(\xi^k)} Q^k(\xi^k, b)],$$

Now, if we define the noise term as:

$$w^k(i, a) = F(Q^k(i, a)) - H(Q^k(i, a)),$$

then we can write the updating transformation in our algorithm (Equation 13.66) as:

$$Q^{k+1}(i, a) = Q^k(i, a) + \alpha[H(\vec{Q}^k) - Q^k(i, a) + w^k(i, a)],$$

which is of the same form as the updating scheme defined for Theorem 13.21 (replace x by Q and l by (i, a)). Having expressed our updating transformation in the same form, from Theorem 13.21, it follows that if all its assumptions are satisfied, then the algorithm will converge to a fixed point of H . Now, a fixed point of H — call it \vec{Q}^* , by its very definition satisfies the following relationship:

$$\vec{Q}^* = H[\vec{Q}^*].$$

From the definition of H , it follows that Q^* must be a solution of the Q -factor version of the Bellman optimality equation, and consequently \vec{Q}^* must be the optimal solution of the Bellman equation. In other words, as long as the assumptions of the theorem are satisfied, convergence to the optimal solution of the MDP is guaranteed. The existence of a solution to the equation (the Bellman equation):

$$\vec{Q} = H[\vec{Q}],$$

which has been proved, implies that there is a fixed point to H .

Let us tackle the conditions of Theorem 13.21 one by one.

Condition 1. The condition of Lipschitz continuity can be shown from the fact that the map H is a linear function of Q .

Condition 2. These conditions can be proved for a step size such as A/k . (See any standard undergraduate analysis text such as Rudin [147] for a proof.) The relations in Condition 2 are the standard stochastic approximation conditions that appear in a variety of step size based s.a. schemes.

Condition 3. This is the boundedness argument, which needs some work. We will present the result as a lemma.

LEMMA 13.27 *In Q-Learning (for discounted reward MDPs), the iterate Q_p^k for any state-action pair p remains bounded.*

Proof We first claim that

$$|Q_p^k| \leq M(1 + \lambda + \lambda^2 + \cdots + \lambda^k), \quad (13.68)$$

where λ is the discounting factor and M is a positive finite number defined as follows:

$$M = \max\{r_{\max}, \max_l Q_l^0\}, \quad (13.69)$$

where

$$r_{\max} = \max_{i,j \in \mathcal{S}, a \in \mathcal{A}(i)} |r(i, a, j)|. \quad (13.70)$$

It is to be noted that since we start (the learning process) with finite values for the Q -factors, Q_l^0 is finite for every state-action pair l . Since the immediate rewards are finite by definition, r_{\max} is also a finite quantity. Then from the way we have defined M above, M too has to be finite.

From the above claim (13.68), boundedness follows since then if $k \rightarrow \infty$,

$$\lim_{k \rightarrow \infty} |Q_p^k| \leq M \frac{1}{1 - \lambda}$$

for all p , since $\lambda < 1$. (In the above, we have used the formula for a convergent infinite geometric series.)

The right hand side of the above is a finite quantity, and so Q_p^k will always be a finite quantity thus proving Lemma 13.27.

So all we need to do to prove the lemma is to prove our claim in (13.68). We will use an induction argument. Let us first present the Q -learning algorithm in a form that will be convenient to us. The form is:

$$Q_p^1 = (1 - \alpha)Q_p^0 + \alpha(R_{p,\xi} + \lambda Q_\xi^0)$$

for all state-action pairs p . Note that $R_{p,\xi}$ denotes the immediate reward earned when from the state-action pair p , the Markov chain goes to a random state ξ . Now, from the definition of r_{\max} in (13.70), we can claim that

$$|R_{p,\xi}| \leq r_{\max}$$

for all p and ξ . Now, from the Q -Learning form presented above:

$$\begin{aligned} |Q_p^1| &\leq (1 - \alpha)|Q_p^0| + \alpha|R_{p,\xi}| + \alpha\lambda|Q_\xi^0| \\ &\leq (1 - \alpha)M + \alpha r_{\max} + \alpha\lambda M \quad (\text{from the definitions of } r_{\max} \text{ and } M) \\ &\leq (1 - \alpha)M + \alpha M + \alpha\lambda M \quad (\text{from the definition of } M) \\ &\leq (1 - \alpha)M + \alpha M + \lambda M \quad (\text{from the fact that } \alpha \leq 1) \\ &= M(1 + \lambda) \end{aligned}$$

From the above, our claim in (13.68) is proved when $k = 1$. Now assuming that the claim is true when $k = m$, we have that

$$|Q_p^m| \leq M(1 + \lambda + \lambda^2 + \cdots + \lambda^m), \quad (13.71)$$

for all values of p .

Now, from the Q -Learning form presented above:

$$\begin{aligned} |Q_p^{m+1}| &\leq (1 - \alpha)|Q_p^m| + \alpha|R_{p,\xi}| + \alpha\lambda|Q_\xi^m| \\ &\leq (1 - \alpha)M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \\ &\quad \alpha r_{\max} + \alpha\lambda M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) \quad (\text{from (13.71)}) \\ &\leq (1 - \alpha)M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \\ &\quad \alpha M + \alpha\lambda M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) - \\ &\quad \alpha M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \\ &\quad \alpha M + \alpha\lambda M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) - \\ &\quad \alpha M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \\ &\quad \alpha M + \alpha M(\lambda + \lambda^2 + \cdots + \lambda^{m+1}) \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) - \\ &\quad \alpha M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \alpha M + \\ &\quad \alpha M(\lambda + \lambda^2 + \cdots + \lambda^m) + \alpha M\lambda^{m+1} \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) - \alpha M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) \\ &\quad + \alpha M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \alpha M\lambda^{m+1} \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + \alpha M\lambda^{m+1} \\ &\leq M(1 + \lambda + \lambda^2 + \cdots + \lambda^m) + M\lambda^{m+1} \quad (\text{since } \alpha \leq 1) \\ &= M(1 + \lambda + \lambda^2 + \cdots + \lambda^m + \lambda^{m+1}). \end{aligned}$$

From the above, the claim in (13.68) is proved for $k = m + 1$, and thus we are done. ■

Condition 4. The noise term w is the difference between a random value and its mean. This can be seen from its definition. Hence its (conditional) mean is 0 (this is a rather heuristic argument, but more sophistication is beyond our scope here). Its (conditional) variance can be bounded by a function of the square of the iterate. But the iterate \vec{Q} itself is bounded and hence the (conditional) variance is also bounded.

Condition 5. The function H is indeed non-expansive with respect to the max norm. It can be shown as follows. Consider two vectors \vec{Q} and \vec{Q}' . From the definition of H (13.67), it follows that:

$$\begin{aligned} HQ(i, a) - HQ'(i, a) &= \lambda \sum_{j=1}^{|S|} p(i, a, j) [\max_{b \in \mathcal{A}(j)} Q(j, b) - \max_{b \in \mathcal{A}(j)} Q'(j, b)] \\ &< \sum_{j=1}^{|S|} p(i, a, j) [\max_{b \in \mathcal{A}(j)} Q(j, b) - \max_{b \in \mathcal{A}(j)} Q'(j, b)]. \end{aligned}$$

From this, we can write that for any (i, a) pair:

$$\begin{aligned} |HQ(i, a) - HQ'(i, a)| &< \sum_{j=1}^{|S|} p(i, a, j) |\max_{b \in \mathcal{A}(j)} Q(j, b) - \max_{b \in \mathcal{A}(j)} Q'(j, b)| \\ &\leq \sum_{j=1}^{|S|} p(i, a, j) \max_{b \in \mathcal{A}(j)} |Q(j, b) - Q'(j, b)| \\ &\leq \sum_{j=1}^{|S|} p(i, a, j) \max_{j \in S, b \in \mathcal{A}(j)} |Q(j, b) - Q'(j, b)| \\ &= \sum_{j=1}^{|S|} p(i, a, j) \|\vec{Q} - \vec{Q}'\|_\infty \\ &= \|\vec{Q} - \vec{Q}'\|_\infty \sum_{j=1}^{|S|} p(i, a, j) \\ &= \|\vec{Q} - \vec{Q}'\|_\infty(1). \end{aligned}$$

Thus for any (i, a) :

$$|HQ(i, a) - HQ'(i, a)| < \|\vec{Q} - \vec{Q}'\|_\infty$$

Since the above holds for all values of (i, a) , it also holds for the values that maximize the left hand side of the above. Therefore

$$\|H\vec{Q} - H\vec{Q}'\|_\infty < \|\vec{Q} - \vec{Q}'\|_\infty,$$

thereby proving the non-expansiveness with respect to the max norm.

That, Q -Learning, under asynchronous conditions, mimics the behavior of its synchronous counterpart has already been established in Proposition 13.26. Then combining the results from Theorem 13.21 and Proposition 13.26, Q -Learning can be said to converge to the *optimal solution* under asynchronous conditions.

12.2. Convergence of Relative Q -Learning

The convergence of Relative Q -Learning in a simulator follows using arguments similar to those of Q -Learning. The proof for synchronous convergence of Relative Q -Learning follows from Theorem 13.21. All the conditions, excepting for the boundedness condition in Theorem 13.21, can be shown in a manner similar to that used for Q -learning. The boundedness of Relative Q -Learning, which we do not discuss, has been shown in [2] using a different approach.

To show that the behavior of the synchronous and asynchronous algorithms can be made similar, we will next prove Proposition 13.25 for Relative- Q -Learning for average reward. The proof will be an extension of the one presented for Proposition 13.25.

Proof The transformation in Relative Q -Learning for average reward is:

$$Q(i, a) \leftarrow Q(i, a) + \alpha[r(i, a, j) - Q(i^*, a^*) + \max_{b \in \mathcal{A}(j)} Q(j, b) - Q(i, a)],$$

where $Q(i, a)$ stands for the Q -factor associated with the state-action pair (i, a) .

In extending the proof of Proposition 13.25, the term X for this algorithm will contain the following additional term:

$$x^f(i^*, a^*) - y^g(i^*, a^*).$$

Now, from Lemma 13.23, it follows that this additional term is also of the order of α . Using this fact, the rest of the proof of Proposition 13.25 for Relative Q -Learning follows in a straightforward manner. ■

This completes the analysis for Relative Q -Learning.

12.3. Finite Convergence of Q -Learning

A feature of the convergence analysis, discussed above, that we cannot overlook is the attainment of convergence after an **infinite** number of iterations.

The natural question is: will the algorithm ever converge in practice and how long will that take? We next prove that it is possible to terminate Q -Learning in a finite number of steps. The result comes from Gosavi [61].

PROPOSITION 13.28 *In Q -Learning for discounted reward, the optimal policy can be obtained from a finite number of iterations with probability 1.*

Proof Let $Q_1^*(i)$ and $Q_2^*(i)$ denote respectively the highest and the second highest *limiting* values of the Q -factors for state i . The existence of limiting values with probability 1 follows from Theorem 13.21. Let $\tilde{Q}_1^k(i)$ and $\tilde{Q}_2^k(i)$ denote respectively estimates, in the k th iteration of Q -Learning, of the highest and the second highest limiting values. Let $e_l^k(i)$ denote the absolute value of the difference in the actual value and the estimate in the k th iteration of $Q_l^*(i)$, where $l = 1, 2$. Then, for $l = 1, 2$,

$$e_l^k(i) = |\tilde{Q}_l^k(i) - Q_l^*(i)|.$$

From the above, we can infer that there are four cases for the relation between the limiting values and the values in the k th iteration. They are:

Case 1.

$$\tilde{Q}_1^k(i) = Q_1^*(i) - e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) + e_2^k(i),$$

Case 2.

$$\tilde{Q}_1^k(i) = Q_1^*(i) - e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) - e_2^k(i),$$

Case 3.

$$\tilde{Q}_1^k(i) = Q_1^*(i) + e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) + e_2^k(i),$$

and

Case 4.

$$\tilde{Q}_1^k(i) = Q_1^*(i) + e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) - e_2^k(i).$$

We will use the trick we used in Proposition 13.26. Let $D(i) \equiv Q_1^*(i) - Q_2^*(i)$ for all $i \in S$.

We will next prove that with probability 1, there exists a value K for k such that for $k \geq K$,

$$\tilde{Q}_1^k(i) > \tilde{Q}_2^k(i) \tag{13.72}$$

for each i . This will imply that the algorithm can be terminated in a finite number of steps with probability 1.

We will now assume that there exists a value K for k such that for each i , both $e_1^k(i)$ and $e_2^k(i)$ are less than $D(i)/2$ when $k \geq K$. We will shortly prove that this assumption holds in Q-Learning. Thus our goal is to show Inequation (13.72) for each case under the assumption made above.

Let us consider Case 1. What we are about to show can be shown for each of the other Cases.

$$\begin{aligned} & \tilde{Q}_1^k(i) - \tilde{Q}_2^k(i) \\ &= Q_1^*(i) - Q_2^*(i) - e_2^k(i) - e_1^k(i) \text{ from Case 1.} \\ &= D(i) - (e_1^k(i) + e_2^k(i)) \\ &> D(i) - D(i) = 0. \end{aligned}$$

Then,

$$\tilde{Q}_1^k(i) > \tilde{Q}_2^k(i) \quad \forall i, k,$$

proving Inequation (13.72). What remains to be shown is that our assumption that for each i , there exists a value K for k such that both $e_1^k(i)$ and $e_2^k(i)$ are less than $D(i)/2$, when $k \geq K$, is true.

Now, Theorem 13.21 implies that for every i , with probability 1,

$$\lim_{k \rightarrow \infty} \tilde{Q}_1^k(i) = Q_1^*(i),$$

and

$$\lim_{k \rightarrow \infty} \tilde{Q}_2^k(i) = Q_2^*(i).$$

Hence, for a given $\epsilon > 0$, there exists a value k_1 for which

$$|\tilde{Q}_1^k(i) - Q_1^*(i)| < \epsilon$$

when $k \geq k_1$. Similarly, for a given $\epsilon > 0$, there exists a value k_2 for which

$$|\tilde{Q}_2^k(i) - Q_2^*(i)| < \epsilon$$

when $k \geq k_2$. Selecting $\epsilon = D(i)/2$, (where $D(i)$ must be greater than 0; note that when $D(i) = 0$, both actions are equally good and the problem is trivial, and by its definition $D(i)$ cannot be less than 0) and $K \equiv \max\{k_1, k_2\}$, we have that for $k \geq K$, with probability 1,

$$|\tilde{Q}_1^K(i) - Q_1^*(i)| < \epsilon = D(i)/2,$$

that is,

$$e_1^k(i) < D(i)/2.$$

Similarly, using the same value for ϵ , one can show that for $k \geq K$, with probability 1,

$$e_2^k(i) < D(i)/2.$$

This proves our assumption and we are done. ■

A related question is: how should the algorithm be terminated? The reader may recall that in case of value iteration also, convergence occurs in the limit; that is when the number of iterations tends to infinity. Fortunately in value iteration, we can use the value of the norm of a difference vector to terminate the algorithm. Unfortunately in RL this is not true. The norm of the corresponding difference vector in RL will be 0 after every iteration since only one Q -factor gets updated in each iteration. In practice, what we do is: we run the algorithm for as long as the Q -factors keep changing. When the step size becomes smaller than some user-defined value, such as 10^{-7} , the algorithm could be stopped. The reason is no more change will be seen in the Q -factors. Another way to terminate the algorithm is when the policy has not changed in the last several iterations.

13. Convergence of Q - P -Learning for MDPs

In this section, we will discuss the convergence of policy iteration based RL schemes for MDPs. The first subsection will discuss discounted reward and the next section will discuss average reward.

13.1. Discounted reward

Let us first recall the steps in Q - P -Learning for discounted reward.

Step 1. Initialize all the P -factors, $P(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$ to arbitrary values. Set all the visit-factors, $V(l, u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$ to 0. Set E , the number of episodes, to 1. Initialize E_{\max} and k_{\max} to large numbers.

Step 2 (Policy Evaluation). Start fresh simulation. Set each Q -factor, $Q(l, u)$, to 0. Let the system state be i . Set k , the number of iterations within an episode, to 1.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 2b. Let the next state encountered in the simulator be j . Increment $V(i, a)$ by 1.

Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j . Set $\alpha = 1/V(i, a)$. Then update $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \lambda Q(j, \arg \max_{b \in \mathcal{A}(j)} P(j, b))].$$

Step 2c. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$ and return to Step 2a. Otherwise go to Step 3.

Step 3 (Policy Improvement). Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$,

$$P(l, u) \leftarrow Q(l, u).$$

Then increment E by 1. If E equals E_{\max} , go to Step 4. Otherwise, go to Step 2.

Step 4. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg \max_{b \in \mathcal{A}(l)} Q(l, b).$$

The policy (solution) generated by the algorithm is \hat{d} . Stop.

The next result formalizes a convergence proof for this algorithm.

PROPOSITION 13.29 *With probability 1, the Q-P-Learning algorithm as described above will generate the optimal solution for the discounted reward MDP.*

Proof Note that the policy evaluation steps — that is Steps 2, 2a, 2b, and 2c can be viewed as Q -Learning. In fact, as discussed in the previous chapter, policy evaluation is indeed a value iteration. Thus, that these steps will converge do not need a separate proof. The proof follows from the convergence of Q -Learning.

The rest of the algorithm is essentially equivalent to policy iteration, which we have shown converges. Now, an important issue is: the policy evaluation steps, in theory, should take an infinite number of iterations.

In other words, the question is how will the algorithm converge when the evaluation of the Q -factors in Step 2b is incomplete? This issue can be resolved by using the proof that Q -factors in Q -Learning, in a finite number of iterations, do reach a stage where the algorithm can be stopped safely. See Proposition 13.28. In a manner very similar to that proof, it can be shown that the Q -Learning in Step 2b can be terminated in a finite number of steps, with probability 1 ensuring that the right policy is selected in the policy improvement step. Of course, the step size will have to be sufficiently small. ■

The above has been worked out in detail in Gosavi [61].

13.2. Average Reward

For a relative value iteration based Q -P-Learning algorithm, the proof of convergence will follow in a manner similar to the proof given in the previous subsection. For a Bellman equation based Q -P-Learning algorithm, for average reward MDPs, see the proof for the corresponding algorithm for SMDPs.

14. SMDPs

In this section, we will discuss the convergence analysis of average reward RL algorithms for SMDPs. We will not discuss the convergence of discounted algorithms because the proofs follow in ways very similar to those for MDPs.

14.1. Value iteration for average reward

Relaxed-SMART, which was discussed in Chapter 9, can be shown to converge to the optimal policy in average reward SMDPs. The difference between Relaxed-SMART and SMART is that the former starts with a good guess of the average reward while the latter starts with 0. The former also uses a controllable step size in the updating of average reward. Then Borkar's two time scale result (see [25]) can be exploited to show convergence. We do not discuss this topic any further, but refer the reader to Gosavi [63] for further reading. In practice, SMART converges on most problems. However, to exploit the the two time scale result of Borkar, it becomes necessary to use a controllable step size in the update of the average reward, as done in Relaxed-SMART.

14.2. Policy iteration for average reward

The convergence of Q - P -Learning for average reward SMDPs can be proved in a style similar to that employed for average reward MDPs. However, there is an additional step in the SMDP version, which is related to the calculation of average reward of the policy that is subsequently evaluated. Please review the steps in the algorithm from Chapter 9.

See Step 2 of the Q - P -Learning algorithm for average reward SMDPs. It is the case that theoretically this step should take an infinite number of iterations. However, in practice we have to terminate the process in a finite number of iterations. This means that the value of ρ used in the subsequent policy evaluation will be an approximation at best. The error in the value of ρ can cause an error in the values of the Q -factors learned in the subsequent policy evaluation in Step 3c.

The error introduced in the Q -factors due to this can be made arbitrarily small. This will complete the proof of convergence of the extension of Proposition 13.29 for average reward SMDPs. We next discuss the proof for this extension.

Proof (Incomplete Evaluation of ρ .) Let us denote the actual average reward of a policy $\hat{\mu}$ by ρ and the estimate with k replications by ρ^k . Let $Q^k(i, a)$ denote the Q -factor associated with state-action pair (i, a) in the k th iteration of policy evaluation of the algorithm. Let $\bar{Q}^k(i, a)$ denote the same factor, when the actual value of ρ is used. Let us define $O((\alpha)^2)$ as a function of the order of $(\alpha)^2$ or higher powers of α . We now claim that:

$$|Q^k(i, a) - \bar{Q}^k(i, a)| \leq O((\alpha)^2) \quad \forall (i, a) \quad (13.73)$$

if $Q^0(i, a) = \bar{Q}^0(i, a)$ for all values of (i, a) . This will ensure that by selecting a suitable small value for α , the difference between the Q -factor that uses an estimate of ρ and the one that uses the actual value of ρ can be made arbitrarily small. We will use an induction argument to prove our claim.

From Step 3c of the algorithm, it follows that:

$$Q^{k+1}(i, a) = (1 - \alpha)Q^k(i, a) + \alpha[r(i, a, j) - \rho t(i, a, j) + Q^k(j, b)]. \quad (13.74)$$

where b depends on the policy being evaluated. If the actual value of ρ , which we denote by ρ^a is used, the transformation in Step 3c can be written as:

$$\bar{Q}^{k+1}(i, a) = (1 - \alpha)\bar{Q}^k(i, a) + \alpha[r(i, a, j) - \rho^a t(i, a, j) + \bar{Q}^k(j, b)]. \quad (13.75)$$

From Equation 13.75 and from the fact that $Q^0(i, a) = \bar{Q}^0(i, a)$ for all values of (i, a) , it follows that

$$Q^1(i, a) - \bar{Q}^1(i, a) = \alpha[\rho - \rho^a]t(i, a, j),$$

which in turn implies that:

$$|Q^1(i, a) - \bar{Q}^1(i, a)| = \alpha|\rho - \rho^a|t(i, a, j). \quad (13.76)$$

In our algorithm, ρ is estimated by sampling (independent replications in a simulator) and using the strong law of large numbers applies (see Theorem 3.1). From the strong law of large numbers, it follows that with probability 1,

$$|\rho - \rho^a|$$

can be made arbitrarily small. Therefore, for a given value of $\alpha > 0$, there exists a number of samples (replications), such that:

$$|\rho - \rho^a| < \alpha. \quad (13.77)$$

Using Equation (13.76) and Inequation (13.77), one can write that:

$$|Q^1(i, a) - \bar{Q}^1(i, a)| = \alpha|\rho - \rho^a|t(i, a, j) < \alpha\alpha t(i, a, j) = (O(\alpha))^2,$$

which proves our claim in Inequation (13.73) for $k = 1$. Now, assuming that the claim is true when $k = n$, we have that:

$$|Q^n(i, a) - \bar{Q}^n(i, a)| \leq (O(\alpha))^2. \quad (13.78)$$

From transformations (13.74) and (13.75), we can write that:

$$Q^{n+1}(i, a) - \bar{Q}^{n+1}(i, a) = (1 - \alpha)[Q^n(i, a) - \bar{Q}^n(i, a)] + \alpha[\rho - \rho^a]t(i, a, j) + \alpha[Q^n(j, b) - \bar{Q}^n(j, b)].$$

The above implies that:

$$\begin{aligned} |Q^{n+1}(i, a) - \bar{Q}^{n+1}(i, a)| &\leq (1 - \alpha)|[Q^n(i, a) - \bar{Q}^n(i, a)]| + \\ &\quad \alpha|[\rho - \rho^a]|t(i, a, j) + \\ &\quad \alpha|[Q^n(j, b) - \bar{Q}^n(j, b)]| \\ &< |[Q^n(i, a) - \bar{Q}^n(i, a)]| + \\ &\quad \alpha|[\rho - \rho^a]|t(i, a, j) + \\ &\quad |[Q^n(j, b) - \bar{Q}^n(j, b)]| \\ &\leq O((\alpha)^2) + \alpha\alpha t(i, a, j) \\ &\quad + O((\alpha)^2) \\ &= O((\alpha)^2). \end{aligned} \tag{13.79}$$

Line (13.79) uses Inequalities (13.78) and (13.77). The above proves our claim for $k = n + 1$. This means our claim is true for all k .

Since α is a small quantity, the square of α is even smaller. By choosing a suitable value for α , the error in the Q -factors (due to the use of an estimate of ρ) can be made arbitrarily small. The proof of Proposition 13.29 for average reward SMDPs is thus complete. ■

Of course, the calculation of ρ itself is an averaging process and in practice, we can terminate the gathering of samples only when more samples do not change (significantly) the value of the statistic being gathered. For instance, in practice, we would not be interested in measuring changes in the average reward beyond the second place after the decimal point if its unit is dollars per unit time. This issue has been discussed in Chapter 4.

15. Convergence of Actor-Critic Algorithms

Although actor-critic algorithms were perhaps the first to be studied by RL researchers, their convergence analysis was studied very recently by Borkar and his colleagues. The actor critic follows policy iteration in spirit and yet differs markedly from the latter. For instance, in Q - P -Learning, which follows the mechanism of policy iteration, we endeavor to evaluate the actual Q -factors of a given policy by performing several iterations. In other words, the policy evaluation phase is done for a number of iterations — the goal being to estimate the Q -factors associated with the policy. In actor-critic algorithms, in contrast, only one update of the Q -factors is done before switching to a policy improvement. This has no parallel in DP theory and seems to be a strategy unique to RL.

theory. Its convergence analysis has been the topic of highly technical recent papers from Borkar and his colleagues. We do not present any convergence analysis, but name some references in the bibliographic remarks.

16. Function approximation and convergence analysis

In this section, we discuss some issues related to convergence analysis of function approximation coupled RL algorithms.

In this book, the convergence analysis of algorithms has assumed that the algorithm is implemented with a look-up table format. When it is implemented with a function approximator, the convergence analysis issues get more complicated.

Function approximation, which is a key area of RL, has remained more an art than a science. There are many reasons for this, and in this context we would like to make a few points.

- Even empirically, several function approximation methods seem not to work in general. For instance, the literature shows that although the non-linear neural network (backpropagation) seems to have worked on *some* problems, it has not done well on many other problems. Of course, with neural networks one has to be careful with learning rates and where divergence is reported, it is not clear if suitable learning rates used. And yet it is the case that researchers have struggled to develop a robust function approximator that works well on a large number of domains.
- Ideally, we should use a visit factor for each state-action pair. But this is possible only with a look-up table. With a function approximator, one has to use the same visit factor for all the state-action pairs (or for a large number number of state-action pairs, if we use some aggregations scheme.) This is unavoidable because keeping track of the visit factors of each state-action pair individually is ruled out with a large number of state-action pairs.
- More importantly, especially with function approximation coupled *Q*-Learning algorithms, there does not seem to be any satisfactory convergence analysis. Read Chapter 6 (section 6.6) of the book written by Bertsekas and Tsitsiklis for more on this. Some amount of convergence analysis seems to exist for the use of the so-called “*TD(0)*” algorithm for a given policy (which resembles the policy evaluation phase of *Q-P*-Learning). This, we believe, is a major obstacle and it is likely that a clearer picture will evolve with more research on this topic in coming years.
- One critical reason for the lack of convergence analysis, it appears to us, is that the value function keeps changing in *Q*-Learning. Perhaps, function approximation proofs should concentrate on showing that they will work if the value function changes slowly.

However, convergence analysis seems to exist for state-aggregation methods [19]. Convergence of the so-called “ $TD(\lambda)$ ” methods, especially in the function approximation context, can be found in Tsitsiklis and Van Roy [176]; note that TD stands for temporal differences [165]. Furthermore there seem to be some convergence guarantees and error bounds on their performance, which makes them all the more interesting. These methods can be incorporated into the policy evaluation phase of Q - P -Learning. Our discussion of the latter used Q -Learning.

$TD(\lambda)$ methods have been proven to converge in the policy iteration context [19]. Extension of $TD(\lambda)$ concepts to value iteration — that is, in Q -Learning — has been attempted (see Watkins [182] and Gosavi, Bandla, and Das [64]), convergence analysis in these scenarios is difficult because of the lack of a multi-step Bellman optimality equation. We have not been able to discuss multi-step issues, that is, temporal difference issues, in RL because of the approach followed in this book. (We first discussed DP algorithms and then developed their RL counterparts.) However, it is the case that encouraging empirical evidence exists with these multi-step algorithms, and in all probability, general convergence properties of these algorithms will be established in future.

17. Bibliographic Remarks

In what follows, we have summarized some references to convergence analysis of DP and RL theory. The following account is not comprehensive. For a more comprehensive account related to DP, the reader is referred to [140] and [20], while for the same related to RL, the reader is referred to [19].

17.1. DP theory

Our accounts, which show that a solution of the Bellman optimality equation is indeed optimal, follow Bertsekas [20]. The convergence of value iteration, via the fixed point theorem, is due to Blackwell [21]. Our account on this issue is based on the material in Puterman [140]. The convergence proof for policy iteration for discounted reward, presented in this book, follows from Bertsekas [20] and the references therein.

The analysis of the convergence of value iteration and policy iteration algorithms for average reward MDPs follows from accounts presented in Puterman [140]. In the policy iteration case, we have changed the notation somewhat using a scalar notation. The convergence of SMDP algorithms was omitted. The reader may refer to Puterman [140] for more material along these lines.

17.2. RL theory

For RL, the main result (Proposition 13.21) related to synchronous conditions follows from Kushner and Clark [100], Borkar and Soumyanath [29] and

Abounadi, Bertsekas, and Borkar [3]. Work related to ODEs in stochastic approximation was initiated by Ljung [105] and developed by Kushner and Clark [100]. The first ODE related result in *asynchronous conditions* for stochastic approximation is from Borkar [26]. Borkar and Meyn [28] used ODEs in the context of RL to show the convergence of a number of RL algorithms under asynchronous conditions. The result which shows that, under asynchronous conditions, the algorithm can mimic the behavior seen under synchronous conditions is from Gosavi [62].

The two time scale result for synchronous conditions is due to Borkar [25] and has been extended to asynchronous conditions in Abounadi [1].

Convergence of Q -Learning has appeared in a number of papers. The first rigorous proof, which exploits the contraction property, is due to Tsitsiklis [175]. This paper pioneered a great deal of convergence related research in RL. The boundedness of the Q -Learning iterate has been shown in our account with a result from Gosavi [62], while finite convergence of Q -Learning has been established with a result from Gosavi [61].

Our analysis of the convergence of Q - P -learning follows Gosavi [61]. The convergence of Relative Q -Learning can be found in Abounadi, Bertsekas, and Borkar [2]. The convergence of Relaxed-SMART has been analyzed in [63].

For a convergence analysis of actor-critic algorithms, the reader should refer to Borkar and Konda [27] and Konda and Borkar [96]. Convergence of function-approximation coupled $TD(\lambda)$ algorithms has been analyzed in Tsitsiklis and Van Roy [176]. More research on function-approximation related behavior of RL algorithms, especially of the Q -Learning type, is needed.

18. Review Questions

1. What happens if the value of D in the proof of Proposition 13.26 is 0?
2. Why can we not extend the proof of Proposition 13.12 to average reward value iteration?
3. Prove Corollary 13.8. The proof should be along the lines of Proposition 13.7 using a given policy instead of using the optimal policy.
4. Prove Lemma 13.14 using vector notation for the limiting probabilities.
5. Does convergence in the span imply that the values of the sequence will converge? Discuss in the context of the proof of convergence of value iteration for average reward.
6. Prove that in average reward policy iteration for MDPs, if the Bellman equation for a given policy is solved setting any one element of the value function to 0, the value of ρ obtained in the solution is the average reward associated with the policy.
7. What is the difference between a non-expansive mapping and a contractive mapping?
8. Define degree of asynchronism and the error of asynchronism.

9. What is “almost sure” convergence? How does it differ from the notion of convergence (of sequences) defined in Chapter 11 ?
10. Why is the vector \vec{J}^* referred to as the *optimal* value function vector?
(Hint: See Proposition 13.3.) How does finding the optimal value function vector help one identify the optimal policy? (Hint: See the last step of any value iteration algorithm.)

Chapter 14

CASE STUDIES

The greater danger for most of us is not that our aim is too high and we miss it, but that it is too low and we reach it.

— Michelangelo (1475-1564)

1. Chapter Overview

In this chapter, we will describe some case studies, which use simulation-based parametric or control optimization techniques to solve large-scale stochastic optimization problems. We will provide a general description of the problem and of the approach used in the solution process, rather than describing numerical details. Some of the problems discussed here can be solved both by control optimization and by parametric optimization methods.

The literature on *model-based* simulation optimization is quite rich. Regular response surface methods that assume the knowledge of the metamodel of the objective function and use regression are also model-based methods. Keeping in mind that the book focuses on methods that do not need elaborate models (for the objective function or the transition probability function), we will discuss simulation-optimization case studies that use **model-free** methods. This includes neuro-response surface methods, meta-heuristics, simultaneous perturbation, and reinforcement learning.

We will discuss case studies related to:

- inventory control,
- airline yield management,
- machine maintenance,

- transfer line buffer optimization,
- inventory control in a supply chain,
- AGV scheduling,
- economic design of control charts, and
- elevator scheduling.

Research related to model-free methods in parametric optimization has seen a flurry of activity in the recent past. There has been a great deal of interest in applying **meta-heuristics** to simulation optimization. A number of papers have been written on this topic. More than 50 papers have already been written on the **simultaneous perturbation** method that appeared in the literature in 1992.

Much of the initial empirical work in the area of reinforcement learning (RL) has originated from the artificial intelligence community — notable examples being elevator scheduling, backgammon, and AGV scheduling. Some of the more recent empirical work has appeared in the industrial engineering literature; examples of this work include machine maintenance and airline yield management.

In this chapter, we will use the following style in our discussions. We will consider a case study, and then discuss the use of simulation optimization, in general, on the problem involved in it.

2. A Classical Inventory Control Problem

The case study we discuss in this section deals with a classical problem from inventory theory (from Law and Kelton [102]) on which the neuro-response surface approach has been applied by Kilmer, Smith, and Shuman [90]. The latter is one of the first papers to have carried out a comprehensive study of the neuro-response surface approach on a real-world problem. The metamodel developed in the neural network for the objective function can be subsequently used for optimization. Next, we present a brief description of the problem.

Consider a retailer that sells a single product. The product is ordered from a supplier. The demand for the product is a Poisson arrival process, while the size of the demand has a discrete generic distribution. The retailer uses an (S, s) policy for inventory control, which is executed at the end of a fixed time period. In other words, at the end of each time period, the inventory is reviewed. If the current inventory (I) equals or exceeds s , then no order is placed. On the other hand, if the current inventory is below s , then an order of the size $(S - I)$ is placed. Backordering is allowed, and hence I can assume negative values. It is assumed that the time taken to receive an order, after it is placed, is a uniformly distributed random variable. All these are realistic assumptions.

The system incurs a flat ordering cost per order and an incremental cost per unit ordered. If a customer arrives to find that the retailer does not have supply, the retailer incurs a stockout cost. Thus it makes sense to have sufficient stocks ready when the demand arrives. However, keeping *excessive* stocks is not a good idea either because it costs money to hold inventory. All these issues make this an interesting stochastic optimization problem; the optimization has to be carried out for the two variables: S and s . The objective function, $f(S, s)$, is the *expected* total cost incurred by the retailer in n time periods.

The neural network used by Kilmer, Smith, and Shuman estimated both the mean and the variance of the total cost. Estimating two parameters with one neural network, via the regular approach of using two output nodes, can sometimes “interfere with the learning of each” parameter. To circumvent this difficulty, they used a neural network with a *parallel* architecture, which is a somewhat different approach. (It has some computational advantages over using two separate neural networks.) The architecture of their neural network has the following features:

- Four input nodes (two of them for the decision variables S and s)
- Two hidden layers
- A sigmoid function
- Backpropagation algorithm
- One output node for each parallel structure — one for the mean and the other for the variance.

A somewhat distinctive aspect of this study is that the mean is not estimated in isolation, but the variance is also estimated as a companion; this raises very interesting possibilities of being able to optimize the mean keeping its variance in a certain range. Also the fact that two hidden layers were needed perhaps points out that the objective function is quite non-linear.

In any neuro-response surface study, one has to deal with a number of difficult computational issues that we enumerate below.

1. Determining the number of nodes in each layer and the number of hidden layers,
2. designing the data set for training,
3. deciding on the number of epochs used for training (over-fitting can be a serious problem), and
4. finally testing whether the metamodel is acceptable (see Twomey and Smith [178] for a discussion on these issues in the context of neural networks).

Many of these issues and some other issues related to 1) the number of simulation replications needed to obtain satisfactory performance and 2) extrapolation of neural networks have been discussed in [90].

Kilmer and Smith [89] contains some earlier empirical results. Padgett and Roppel [125], Pierreval [133], and Pierreval and Huntsinger [134] discussed the use of neural networks in simulation-based response surface methods. Some other related work can be found in Badiru and Sieger [9], Fishwick [45], Madey *et al.* [107], and Meghabghab and Nasr [114].

3. Airline Yield Management

In 1978, the airline industry was deregulated. This meant companies could select their own market segments and routes. Stiff competition ensued in the airline industry as a result of this. It was perceived that to remain in the competition, optimization methods would be needed. This continues today because unless one uses these methods, there are strong chances of losing market share; it is felt that unless you exploit these methods to operate your company in a cost-efficient manner, your competitors will put them to good use and drive you out of business.

Yield management, also called revenue management, is the science underlying maximization of profits via selling seats on a plane in the most effective manner. Seat allocation and overbooking are two of the critical aspects of the yield management problem, and the key to making profit lies in controlling seat allocation and overbooking properly. Our discussion will follow Gosavi [60], Gosavi [61], and Gosavi, Bandla and Das [64].

There are differences in customer demands. For instance, some customers demand direct flights, while some are willing to fly with a few stopovers. Also, it is the case that some book long in advance of their journey, while some (usually business related travelers) do not. Airline companies exploit these differences by selling seats within the same cabin of a flight at different prices. A passenger who desires a lesser number of stopovers or a passenger who arrives late in the booking process is made to pay a higher fare. This results in classifying passengers into different **fare classes** based on their needs and the circumstances. Airlines usually place an upper limit on the number of seats to be sold in each fare class. This ensures that some seats are reserved for higher fare classes, which supply higher revenues. It is to be understood that the demand for lower fare classes is always higher than that for the higher fare classes and unless such limits are imposed, it is likely that the plane will be filled with only the lower fare-class passengers.

The “overbooking” aspect adds to the complexity of this problem. Customers cancel their tickets and sometimes fail to show up at the flight time (no-shows). Airlines, as a result, overbook (sell more tickets than the number of seats available) flights, anticipating such events. This can minimize the chances of flying

planes with empty seats. It may be noted that a seat in an airplane, like a hotel room or vegetables in a supermarket, is a perishable item, and loses all value as soon as the flight takes off. However, the downside of excessive overbooking is the risk of not having enough seats for all the ticket-holders. When this happens, i.e., when the number of passengers who show up exceeds the capacity because of overbooking, airlines are forced to deny (bump) boarding request to the extra ticket-holders and pay a penalty in two ways: directly in the form of financial compensations to the inconvenienced passengers, and sometimes indirectly through loss of customer goodwill. Usually loss of customer goodwill can be avoided by finding volunteers, who are willing to fly on a later flight; nobody is inconvenienced this way, but the volunteers get monetary benefits and the company loses money. Hence, a careful choice of overbooking policy that maximizes the yield is required.

A major factor used in classification is the time of the request for reservation. Passengers, who book in advance, get discount fares; in other words they are made to belong to lower fare classes (nobody has complaints with this, of course). Those who come later have to pay higher fares — that is, they get allocated to the higher fare classes. (Note that a higher or lower fare class does not mean that there is any seating advantage; we are considering passengers in the cabin.) If classification is carried out by the airline company on the basis of this factor alone, the model, which assumes that passengers in different fare classes arrive *sequentially*, i.e., passengers in the lowest classes arrive first, followed by passengers in the next higher class, and so on, is accurate (see Figure 14.1). However, in reality, the itinerary of the passenger is also used in determining fare class. To understand how an origin-and-destination based (also called itinerary-based or number-of-connections-based) passenger classification works, consider a single flight leg from Chicago to New York. The flight, in addition to carrying passengers whose origin is Chicago and destination is New York, is likely to be carrying passengers from other longer itineraries, such as Denver-Chicago-New York, and Colorado Springs-Denver-Chicago-New York. If the passengers, whose itinerary originates from Chicago, form the highest class, those flying from to New York via Denver form the middle class (these passengers are on the second leg of their flight), and those flying from Colorado Springs to New York via Denver and Chicago form the lowest fare class of passengers in the plane (these passengers are on the third leg of their flight).

Consider Figure 14.2. A circle in Figure 14.2 denotes the origin and the symbol inside it indicates the class of the passenger originating at that point. The figure also shows that usually itinerary-based classification yields two or three fare classes and no more. It may be noted that if this happens to be one of the factors used to classify passengers, *sequential* passenger arrival is a poor assumption. Then, as suggested by Robinson [143], a *concurrent* arrival pattern

must be assumed. In a concurrent arrival pattern, passengers of any fare class arrive at any time in the booking horizon. In practice, several factors are taken into account in classifying passengers, and the airline company may have up to 15 fare classes or more.

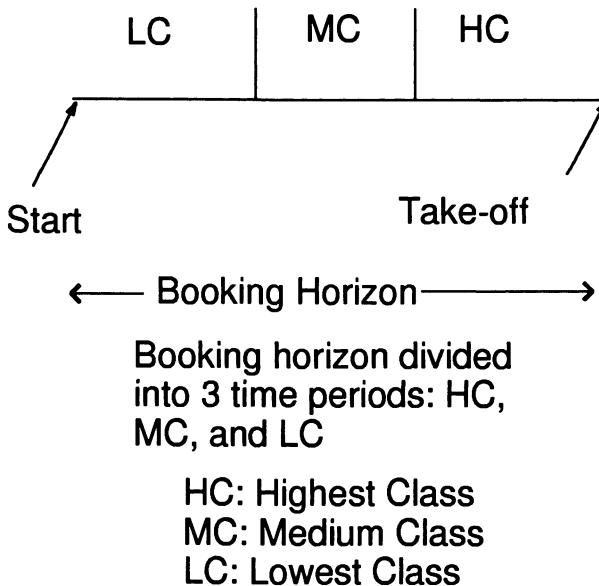


Figure 14.1. A schematic showing classification of customers based on the time of arrival.

McGill and van Ryzin [113] present an excellent overview of the yield management problem, citing a large number of references to the existing literature and defining several terms used in airline industry parlance.

An SMDP model. For an SMDP model, one must come up with a definition of the state space. For n fare classes, a conservative definition of the system state (ϕ) is:

$$\hat{\phi} = (c, s_1, s_2, \dots, s_n, \hat{\psi}_1, \hat{\psi}_2, \dots, \hat{\psi}_n, t),$$

where c represents the class of the current customer (among n possible classes) seeking a ticket, s_i represents the number of seats sold in the i th class, $\hat{\psi}_k$ denotes an s_k -tuple containing the times of arrivals of all the customers in class k , and t represents the time remaining for departure of the flight. This conservative state-space definition satisfies the semi-Markov property. However, it makes the state space too huge, and as such an approximation of the state space is required in practice.

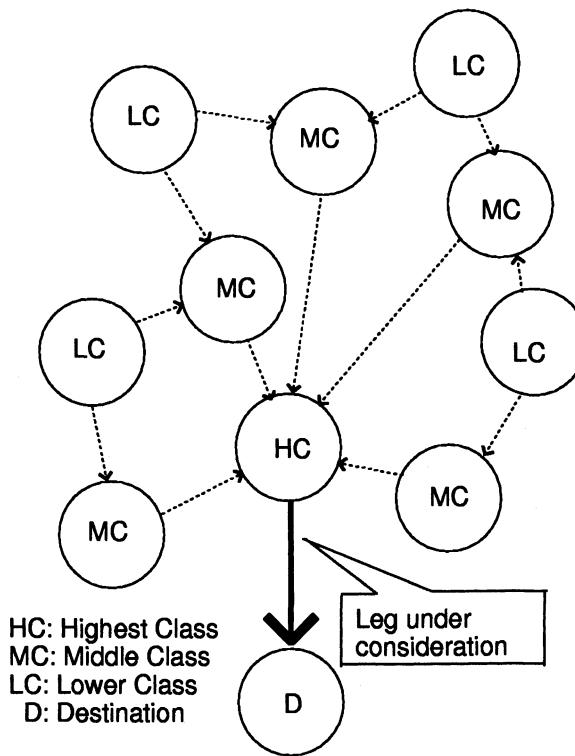


Figure 14.2. A schematic showing classification of customers based on the origin (circle) and the destination, in one particular leg of an airline flight.

Whenever a customer arrives to buy a ticket, a decision needs to be made regarding whether to accept or deny the request. The action space is hence $\{\text{Accept}, \text{Deny}\}$.

The performance metric used in Gosavi *et al.* [64] and Gosavi [60] is long-run average reward, although the problem can also be studied as a finite horizon problem. In both papers, RL approaches have been used to solve the problem.

A Parametric-Optimization Model. The seat-allocation problem can be solved as a parametric-optimization problem. The objective function is the average revenue per unit time or the average revenue per flight. If $x(i)$ denotes the booking limiting for the i th fare class, then the problem is one of

$$\text{Maximizing } f(x(1), x(2), \dots, x(k))$$

where $f(x(1), x(2), \dots, x(k))$, the objective function, is estimated using simulation. The simulator is combined with an optimization algorithm such as

simulated annealing or simultaneous perturbation. See Gosavi [60] for some empirical results obtained by using these algorithms.

The performance of any simulation-optimization model should be calibrated against some existing models in the literature. We discuss one such model. Several other models have been suggested in the literature.

The EMSR Heuristic. In what follows, we present a powerful heuristic algorithm to solve the yield-management problem. This heuristic is widely used in the industry. Belobaba [17] developed the EMSR (Expected Marginal Seat Revenue) model using Littlewood's equation [104]. Belobaba's model is often called the EMSR heuristic. The strategy adopted in the EMSR heuristic comprises of determining *booking limits* for the different fare classes (using Littlewood's equation). When a customer requests a ticket, he/she is given a reservation only if the number of seats sold in the class to which he/she belongs is less than the booking limit for that class. The booking limits for any given class are obtained by solving Littlewood's equation. Littlewood's equation is:

$$P(X_i > S_j^i) = R_j/R_i \quad \text{for } j = 1, 2, \dots, k-1, \quad i = j+1, j+2, \dots, k,$$

where X_i denotes the number of requests for class i that will arrive in the booking horizon, S_j^i denotes the number of seats to be protected from class j for higher class i , R_i and R_j are the fares for the classes i and j respectively, and k is the number of classes. In this equation, the numbering convention that we have used implies that if $i > j$, then $R_i > R_j$.

The unknown quantities in the equation above are: S_j^i . After the values of S_j^i are obtained, the booking limit for a class j is found by:

$$x(j) \leftarrow C - \sum_{i>j} S_j^i,$$

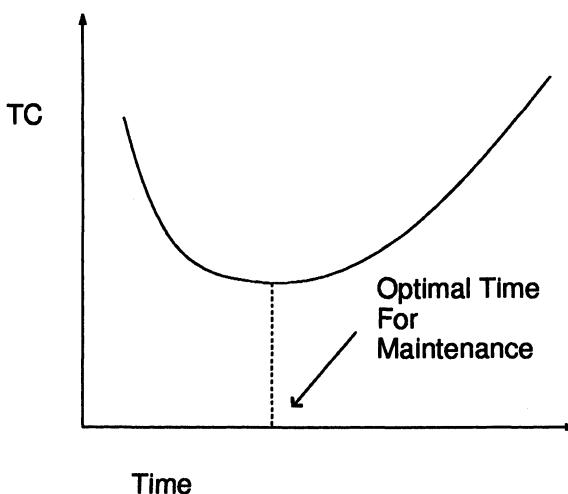
where C is the capacity of the plane. Cancellations and no-shows are accounted for by multiplying the capacity of the aircraft with an overbooking factor. Thus, if C is the capacity of the flight and p is the probability of cancellation, then the overbooking factor is given by $1/(1-p)$ and the modified capacity of the aircraft is given by $C/(1-p)$.

4. Preventive Maintenance

Preventive maintenance has acquired a special place in modern manufacturing management with the advent of the so-called "lean" philosophy. According to the lean philosophy, an untimely breakdown of a machine is viewed as source of *muda* — a Japanese term for waste. Indeed, an untimely breakdown of a machine can disrupt production schedules and reduce production rates. Especially if a machine happens to be a bottleneck, it is very important that the machine be

kept in a working state almost all the time. Total Productive Maintenance and many such *management* philosophies rely on the age-old reliability principle, which states that if a machine is maintained in a preventive manner the up-time of the machine is raised.

It is usually the case that as a machine ages, the probability of its failure increases. If the machine is allowed to age too much, it is likely that it fails, and a **repair** becomes necessary. On the other hand, if the machine is **maintained** too early in its life, the maintenance costs can become excessive. As such, one seeks the optimal time for maintaining the machine. See Figure 14.3 for a graphical demonstration of this concept.



TC: Total Cost (sum of maintenance and repair costs)

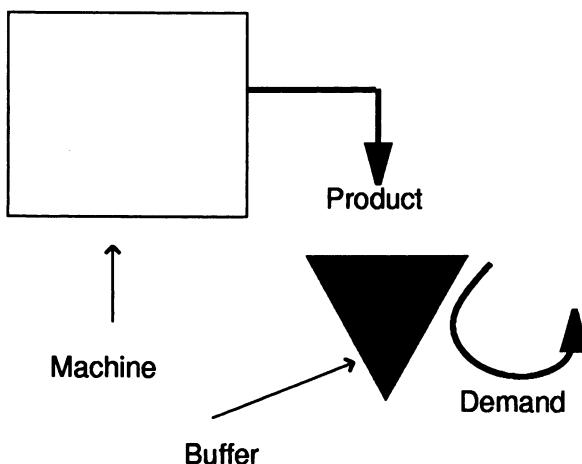
Figure 14.3. The graph shows that there is an optimal time to maintain.

A simple maintenance model assumes that after a repair or a maintenance, the machine is as good as new. In the real world, the problem can become more complex, since the machine cannot be viewed in isolation but must be viewed as a part of the system. A machine in a factory is usually a component of the production-inventory system. We will discuss an SMDP model for solving the problem.

The SMDP model is based on the paper written by Das and Sarkar [39]. They formulated the problem using a semi-Markov model and set up exact expressions for the transition probabilities. The problems they were able to solve had a small state space because of the difficulties in setting up the same expressions for larger state spaces. Also, their expressions were evaluated for some specific

distributions. Nevertheless, the results they obtained on the small problems serve as an important source for benchmarking the performance of various RL algorithms. It is to be noted that on very large problems, since optimal solutions are unavailable, one must turn to heuristic solutions for benchmarking.

Das and Sarkar consider a production-inventory system, that is, a machine with a finished product buffer. See Figure 14.4. The buffer keeps the product till the product is taken away by the customer. The customer could be the actual customer to whom the product is sold (in case the factory has a single machine), or else the customer could be the next machine. The demand, when it arrives, depletes the buffer by 1, while the machine, when it produces 1 unit, fills the buffer by 1 unit. There is a limit to which the buffer may be filled. When this limit is reached, the machine goes on vacation and remains on vacation till the buffer drops down to a predetermined level. The time for producing a part is a random variable; the time between failures, the time for a repair, the time between demand arrivals, and the time for a maintenance are also random variables. The age of the machine can be measured by the number of units produced since last repair or maintenance. (The age can also be measured by the time since last repair or maintenance.)



Machine fills the buffer, while the demand empties it

Figure 14.4. A production-inventory system

An SMDP model. The details of the semi-Markov model can be found in Das and Sarkar. They modeled the state-space as:

$$\hat{\phi} = \{b, c\},$$

where c denotes the number of parts produced since last repair or maintenance, and b denotes the number of parts in the buffer. There are two actions that the decision maker can select from: $\{\text{Produce, Maintain}\}$. The action is to be selected at the end of a production cycle, that is, when one unit is produced.

Value iteration based RL approaches to solve this problem can be found in [37], [63]. RL is able to produce the same results as the optimal-seeking approach of Das and Sarkar. An automata theory based approach to solve this problem can be found in [65].

A parametric-optimization model. A parametric-optimization model, following Das and Sarkar's dynamic model, can be set up easily. Let us assume that for a given value of b , there is a threshold value for c . When the value of c exceeds the threshold, maintenance must be done. For example, if the threshold level for $b = b'$ is 10, the machine must be maintained when the system state reaches $(b', 10)$, while the system must keep producing in states (b', n) , $n \leq 10$. Thus, the problem can be solved with a parametric-optimization model as follows:

$$\text{Minimize } f(c_0, c_1, \dots, c_k),$$

where c_i denotes the threshold level for maintenance when $b = i$. Here $f(\cdot)$ denotes the average cost of running the system using the threshold levels specified inside the round brackets.

The optimality of the threshold policy should be established to use a parametric-optimization approach. Schouten and Vanneste [153] have established the existence of a structure for the optimal policy, under certain assumptions. The objective function can be estimated via simulation, and the simulator can be combined with an optimizer. However, this approach runs into trouble for large buffer sizes, since one has a large parameter space for large values of k . If the limit on the buffer is more than 5, it is best to use the SMDP model described above.

The Age-Replacement Heuristic. With large-scale problems, on which optimal solutions are unknown, it is important to benchmark the performance of these simulation-optimization techniques. An important paradigm, which has been used successfully in reliability applications, goes by the name renewal theory. In fact, renewal theory has strong mathematical roots, and as such provides very robust heuristics in many areas. We next describe the age-replacement heuristic, which is based on renewal theory. (The age-replacement heuristic works excellently for many reliability problems.)

The use of renewal theory is usually based on a strategy that we describe next.

1. Identify a cyclical phenomenon in the stochastic system concerned.
2. Find the expected total cost incurred in one cycle and the expected total time spent in a cycle.
3. The expected total cost divided by the expected total time will, according to renewal theory principles, equal the average cost per unit time in the system.

Every time the machine fails or is maintained, we can assume that a cycle is completed. Then, using the ideas expressed above, an analytical expression for the average cost of maintaining the machine when its age is T can be derived. Thereafter, one can use non-linear programming to find the optimal value of T that minimizes the average cost. See Ross [145] for more details of this model. The average cost, $g(T)$, can be written as

$$g(T) = \frac{EC}{CT},$$

where EC is the expected renewal cost and CT is the expected renewal time. Let x denote the time for failure of the machine. EC can be written as:

$$EC = (1 - F(T))C_m + F(T)C_r,$$

where T is the age of maintenance, $F(x)$ is the cumulative distribution function of the random variable x , C_m is the cost of one maintenance and C_r is the cost of one repair. Now CT can be written as:

$$CT = T_r F(T) + \int_0^T x f(x) dx + (T + T_m)(1 - F(T)),$$

where $f(x)$ denotes the probability density function of the random variable x , T_m denotes the mean time for maintenance, and T_r denotes the mean time for repair.

The age-replacement heuristic has given stiff competition to simulation-optimization techniques described in [37] and [65].

5. Transfer Line Buffer Optimization

The transfer line buffer optimization problem is, perhaps, one of the most widely-studied problems in applied operations research — both in the academic community and in the industry. The driving force behind this problem is the success attributed to the use of “kanbans” in the Japanese automotive industry. Kanbans are buffers placed in between machines in a flow shop (flow line). In a flow shop, unlike a “job shop”, there is little variety in the products being

produced, and all the products maintain a roughly linear flow. In other words, products move in a line from one machine to the next.

Consider Figure 14.5. In between the first and the second machine, there exists a buffer (kanban). A limit is placed on the size of each buffer. When the machine preceding a buffer completes its operation, the product goes into the buffer. The machine following the buffer gets its supply from the buffer. In a kanban controlled system, there is a limit on the buffer size. When the limit is reached, the previous machine is not allowed to produce any parts till there is space in the buffer. The previous machine is then said to be *blocked*. If a machine suffers from lack of material, due to an empty buffer, it is said to be *starved*. A machine is idle when it is starved or blocked. Idleness in machines may reduce the production rate of the line.

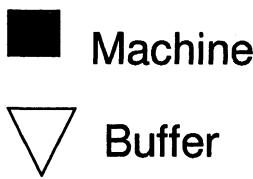
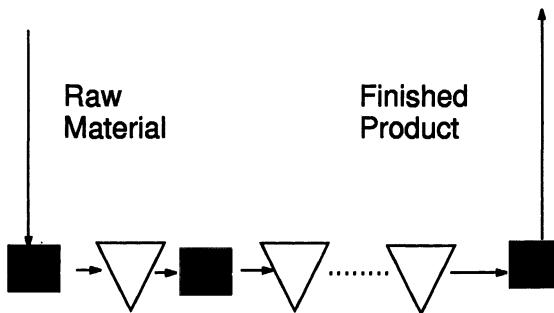


Figure 14.5. A transfer line with buffers

Ideally, starvation and blocking should be minimized. This can be done by placing large buffers to make up for any differences in the speeds of the machines. However, large buffers mean large inventory, which in modern times, is sought to be avoided. This scenario gives rise to a parametric optimization problem — one of finding the sizes of the buffers in between the machines so that the production rate is maximized, while the total inventory in the system is not allowed to rise above a certain level.

In the real world, the problem is more complex. Production times are often random variables. Machines fail and it takes time to repair them. Failures

and repairs add to the randomness in the system. This creates a challenging problem, and a semi-Markov process may be needed to capture its dynamics.

In [7], Altiok and Stidham set up expressions for several objective functions using Markov theory, and then optimized the systems. They had to calculate the transition probabilities of the Markov chain underlying the system towards this end.

Tezcan and Gosavi [171] simulated a general transfer line taking into account

- the randomness in the production times,
- the randomness of failures of machines, and
- the randomness of repair times.

Then they optimized the system using the Learning Automata Search Technique (LAST). The problem can be mathematically described as:

$$\text{Maximize } f(x(1), x(2), \dots, x(k))$$

subject to some constraints. Here $x(i)$ denotes the size of the i th buffer. A possible objective function is the production rate of the system with the following constraint:

$$x(1) + x(2) + \dots + x(k) \leq C = \text{limit on the total inventory.}$$

Another candidate for the objective function is the net average reward per unit time earned from running the system; this objective function should take into account inventory-holding costs and the profits earned from production.

On small problems, the results in [171] are very close to the *optimal* results of Altiok and Stidham [7], thereby pointing to the usefulness of LAST; LAST has been used to solve larger problems as well. Unfortunately, there are no robust heuristics to solve the problem. Some “formulas” are used in the industry to determine the optimal buffer levels, but how close they are to the optimal solution is anybody’s guess.

The problem of buffer optimization can also be cast as an SMDP in which the system state is described by the contents of each buffer, among other things, and the action space is composed of two actions: {*Produce*, *Don’t Produce*}. Thus, whenever a production is completed, the machine in question has two choices: either to produce one more unit or not to produce. For related results, see [110] and [127].

Finding the optimal sizes of kanbans is an important design problem, and several researchers have worked on it. Due to lack of space, we cannot refer to all the literature.

6. Inventory Control in a Supply Chain

A *supply chain* is a collection of facilities that together manufacture and sell a product. It includes the supplier of raw materials needed for making the product, the manufacturer, the retailer stores, and the customers. It also includes the transportation system that is used to transfer material and information flows across all the elements of a supply chain.

Inventory control in a supply chain has assumed a good deal of significance in recent years due to the fact that improper inventory management in a company can lead to reduced market shares. The *lack* of raw material in a production plant, or finished products in the retailer store has caused huge losses to many industries. Carrying inventory can help solve this problem. On the other hand, a company that carries extra inventory has to pay the price of having to store it; the inventory-carrying costs usually raise the price of the product, thereby giving an advantage to the competitor. These are exciting challenges in today's manufacturing world, in which competition is intense and stakes are high. Supply chain management has emerged as an important set of methods that seeks to address some of these issues.

Supply chain management has been defined by Simchi-Levi *et al.* [160] as follows:

Supply chain management is a set of approaches utilized to efficiently integrate suppliers, manufacturers, warehouses, and stores so that merchandise is produced and distributed in the right quantities, to the right locations, and at the right time, in order to minimize system-wide costs while satisfying service requirements.

The supply chain approach is essentially what was known as the "systems approach" to solve production-planning problems. It has been realized in the industry that a systems approach to production planning can make a significant difference to a company's revenue earning ability and to the market share of its product.

In Pontrandolfo, Gosavi, Okogbaa, and Das [138], a problem related to inventory control in a supply chain has been set up as an SMDP and then solved using RL. Next, we describe the problem briefly.

The problem revolves around a set of markets (each market has a customer pool), import warehouses, export warehouses, and production facilities. Some of these facilities are not in the same countries; the setting is global. When the i th market demands a product, the i th import warehouse in the same country is used to satisfy the demand. If the product is not available, the product is backordered and the warehouse places a request to any one of the export warehouses available. Some of the export warehouses may be in different countries. Getting a product from a warehouse in another country may take more time, the transportation costs may be higher, and yet the product itself

may be cheaper. Obtaining a product from the same country usually takes less time but may be more expensive. In any case, when the warehouse decides to replenish its stocks, it needs to make the following decision: which country to choose? The optimal decision is based on the number of units in each warehouse. The problem considered in Pontrandolfo *et al.* assumed that there are three countries involved. The state space can then be described by:

$$\hat{\phi} = \{i_1, i_2, i_3, e_1, e_2, e_3\},$$

where i_k denotes the content in the k th import warehouse, and e_k denotes the content in the k th export warehouse. The action space is made up three actions: $\{Choose\ EW-1, Choose\ EW-2, Choose\ EW-3\}$, where $EW - i$ denotes the export warehouse in the i th country.

In Pontrandolfo *et al.*, a value iteration based RL approach is used to solve the problem. The algorithm is able to outperform some domain-specific heuristics. The model considered there is quite simple; however, it is one of the first RL based approaches to solve a problem of this nature.

7. AGV Routing

Automated Guided Vehicles (AGVs) are material-handling devices, whose movement can be remote controlled. For their motion, one has to use wires or tapes, which are placed on the shop floor. Radio or electronic signals, via the wires or tapes, can then be used to control the movement of the AGVs. AGVs are expensive; it is likely however that the factory of the future will make intensive use of AGVs for transporting material in a shop floor.

A number of design and control related problems related to AGVs have been discussed in Heragu [71]. When the AGV path is fixed and the number of vehicles determined, an important problem that needs to be solved is that of AGV routing. Some typical network flow designs are discussed in Heragu and Rajgopalan [72].

We, next, describe an RL-related case study from Tadepalli and Ok [167]. The network design they used may be uncomplicated, but their work is pioneering and demonstrates the use of RL in finding good routing policies for AGVs. More importantly, the way the state space is defined and the level of detail that is used in modeling the AGV travel make it very useful. The model uses features of the “grid-world” models (see Mahadevan [109]) thereby making it quite different from the approaches for AGV scheduling seen in the industrial engineering literature.

Consider Figure 14.6, which is based on the figure in [167]. There are two machines at two different points, where jobs queue up to be taken away by the AGV to one of the two conveyors. The job *type* produced at each machine is different. The job production process is a random process. The AGV gets a reward when it completes a delivery at the right destination. The reward

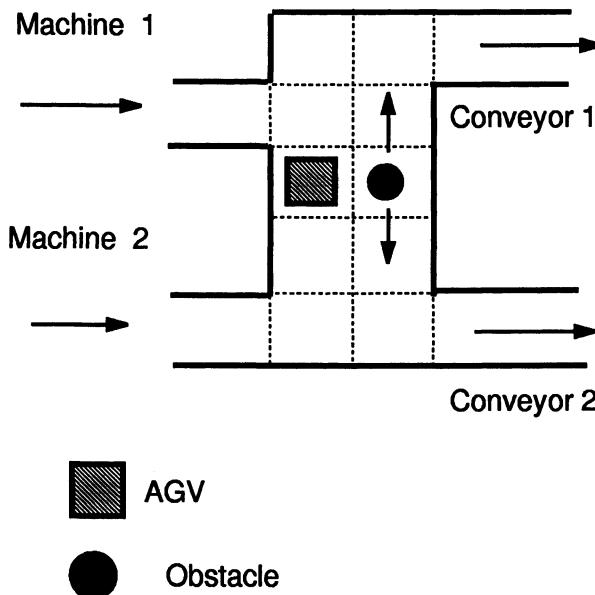


Figure 14.6. An Automated Guided Vehicle System with one AGV

depends on the job type and where it gets delivered. The objective is to deliver as many jobs as possible in unit time. The performance metric used in [167] is average reward per unit time.

The AGV can take any one of 6 possible actions. They are:

1. do nothing
2. load
3. unload
4. move up
5. move down
6. change lane

To unload, the AGV must be next to a conveyor and to load, it should be next to the machine. Tadepalli and Ok have added a moving obstacle to make the AGV's job more difficult. (This is an interesting feature because in many AGV systems, one has more than one AGV and collision-prevention is a major issue.) Any collision with the obstacle results in a cost. The state space is defined as:

$$\hat{\phi} = \{NQ_1, NQ_2, X, Y, X', Y', i\},$$

where NQ_i denotes the number of jobs waiting at the queue near machine i , X and Y represent the x and y coordinates, respectively, of the AGV, X' and Y' stand for the same quantities for the obstacle, and i denotes the job type.

The algorithm used by Tadepalli and Ok [167] is an average reward algorithm, which is based on the average reward Bellman optimality equation. It is a model-building algorithm. (Model-building algorithms have been discussed in Chapter 9).

In summary, it must be said that the task given to the AGV is quite complex. It has to learn the optimal action in each state. When there are a large number of jobs waiting in the queues, it would have to hurry back to the job generators, and once loaded, without wasting much time, it should make its way to the best conveyor. It is not very easy to model this problem as a parametric optimization problem; this is a truly dynamic problem where control optimization techniques must be used. Once the distributions of the inter-arrival time of each generator are identified (along with the distribution of the obstacle trajectories), the system can be run off-line to determine the right action in each state. Then it can be implemented in real time. It is clear that this is a promising approach to solving more complicated problems in AGV routing.

8. Quality Control

Control charts are on-line quality control tools used for controlling the quality of parts produced by a manufacturing process. They can be effectively used to *monitor* a process and thereby control the quality of parts produced. They basically serve as diagnostic tools for detecting problems in the production process. The basic idea — underlying the use of control charts — is to collect samples (of parts) from the process at regular time intervals, and then use the information contained in the samples to make inferences about the mean and variance of the population (process). See Montgomery [118] for more details about control charts.

Designing a control chart, economically, is an important problem from the standpoint of quality control. Some of the basic parameters needed to design a control chart are: the size of the sample used (n), the upper control limit (UCL) and the lower control limit (LCL).

The samples collected are used to compute statistics such as mean, standard deviation, or range. Obtaining values for such statistics outside of the control limits is assumed to be a signal that indicates the process to have gone out of control. When this signal is obtained, the process is stopped. The process is then investigated into — to make sure that the alarm given by the control chart is *not* a false alarm. If the alarm is not a false alarm, the problems in the process are fixed before the process is allowed to produce any more parts. If the alarm turns out to be a false alarm, the process is restarted. False alarms are clearly undesirable, and so is the lack of a signal when the process actually goes out

of control. Even with the best control charts, both phenomena cannot be ruled out.

Generally, the values of n , UCL , and LCL dictate the probability of false alarms and the probability of not detecting a problem when the process actually goes out of control. Using statistical information about the process, one can set up the expressions for these probabilities — in terms of the design parameters: n , LCL , and UCL . These expressions can then be combined to set up an expression for the average cost of running a quality system with a control chart. Excessively lax values for the control limits and small values for the sample size, usually, lead to an increase in the probability of not detecting a problem when it occurs, while excessively stringent values for control limits increase the probability of false alarms.

It turns out that with sophisticated forms of control charts, such as the so-called adaptive control charts, which have several other design parameters, the expression for the average cost can become quite complex (see Das, Jain, and Gosavi [38] for one example that uses Markov chains to construct the closed form of the cost function). When that is the case, numerical methods may be needed for optimization. Simulation is clearly an alternative here, and via simulation *one can avoid having to set up a closed form for the objective function*. Simulation can be used to estimate the cost associated with any given set of values of the design parameters. A method such as simultaneous perturbation can then be used for solving the problem. (See Fu and Hu [48] for an application of perturbation analysis in control chart design.) Simulation has been used in quality control for several years (see Barish and Hauser [10]), but the potential for optimization needed in *sophisticated* control charts is clearly enormous.

9. Elevator Scheduling

The elevator scheduling (or dispatching) case study is due to the independent work of Crites and Barto [35] and Pepyne *et al.* [129]. There is some historical significance to this work; this is perhaps one of the first empirical studies to have used the RL framework on a large-dimensional sequential decision making problem (dynamic decision-making) that has *commercial value*.

In brief, the problem of elevator scheduling can be described as follows. Consider an elevator in a large building. At each floor, people wait in front of the door to go down or to go up. The elevator must decide, when it is approaching a floor, whether to stop or not. The goal of course is to minimize the total amount of time people have to wait. Usually, if there is a passenger who wants to get off the elevator at the approaching floor, the elevator must stop. This is one example of a constraint with which an elevator operates. Typically, elevators have several other constraints. Crites and Barto used a simulator that closely resembles a real-life elevator.

Mathematically, the problem is one of finding the optimal action in each state (floor characterized by the number of customers waiting) visited by the system (elevator). In many states, it will have the choice to select from two actions: $\{\text{Stop}, \text{Don't Stop}\}$. The objective is to reduce a function of the time spent by the passenger waiting for an elevator. The state can be described by the number of people waiting at each floor and as such the state becomes an n -tuple, if n is the number of floors to which the elevator caters. To give an example, consider a 10-floor case and

$$\hat{\phi} = (1, 0, 3, 4, 0, 6, 4, 9, 8, 12), \quad (14.1)$$

where the k th element of $\hat{\phi}$ denotes the number of people waiting on the k th floor. Now, although the information related to the actual number of people waiting may be available in the simulator, in the real-world, such information may be hard to access in real time. This is because after the first person presses the switch to request the elevator, the people who come after the first person do not press the switch again. Thus the state for the example in (14.1) may appear to be as shown below:

$$\hat{\phi} = (1, 0, 1, 1, 0, 1, 1, 1, 1, 1).$$

It is necessary to resolve such problems in RL, especially if the algorithm is trained on data in which perfect state information is available but in real life is made to run on imperfect information. Clearly, if the state information is imperfect, wrong actions will be executed. For example, the algorithm will treat the floor where 10 persons are waiting and the floor where only one person is waiting similarly. We can suggest one way of working around this difficulty, although we are not sure if a commercial scheduler that uses RL is available in the market; the strategy would be to

- install cameras on each floor, and then use the camera-generated images along with a pattern recognition software for determining the number of people waiting on each floor.

That way a near-exact estimate of the actual state would be available in real time, and the controller could use it to select the right action. Of course, the training should be best done off-line.

The goal in the elevator case study is to minimize waiting times. There are other methods for this. The author, as a graduate student, had heard in a class that one effective way to minimize complaints about having to wait too long is to install full-length mirrors in the waiting areas. In fact, this may require a reversal of the sign of the objective function!

On a more serious note, Crites and Barto have actually modeled the problem as a “multi-agent system” using a semi-Markov model. The performance metric

is discounted reward and a Q -learning algorithm has been used. The function approximation scheme is very elaborate. They use a backpropagation neural network with several inputs.

10. Simulation optimization: A comparative perspective

In this book, we have discussed a number of simulation-optimization methods. And in this chapter, we have seen how some of these methods can be used on some real-world problems; in some cases, we have shown the use of more than one method on the same problem. At this point, it will make sense to discuss some relative merits and de-merits of these methods.

Let us first discuss those dynamic problems that can be solved by *both* control and parametric optimization methods. Parametric optimization methods are usually easy to implement because one needs a simple system simulator, which can be connected to an optimizer. Control optimization methods require the optimization algorithm to be an integral part of the simulator. This usually means that greater effort is required in coding the simulator. But on the other hand, control optimization methods may be more efficient in terms of the computer time taken to generate solutions. Remember that parametric optimization formulations of dynamic problems may result in problems of a very high dimension, and searching over the solution space can take a very long time.

Next, we would like to say a few words about problems that can be solved with the parametric-optimization model only. We have discussed several methods in this context, depending on whether the problem has a discrete or continuous solution space. From our review of the literature, we find the genetic algorithm to be the most popular of the meta-heuristics, although there is no general agreement in the literature on whether it is the best. Tabu search, also, has recently seen a large number of useful applications. Although these techniques have some theoretical backing, it should not be overlooked that these remain at best “heuristics.” These heuristics may not generate “the” best solutions, but they usually generate very good solutions that can be used in practice. In comparison to the classical RSM, which is guaranteed to generate near-optimal solutions, these meta-heuristic methods usually take less computational time. In the field of continuous parametric optimization, simultaneous perturbation appears to be a remarkable development. In comparison to finite difference approaches, there is no doubt that simultaneous perturbation takes less computational time.

Finally, regarding problems that can be solved only with the dynamic model, we have presented two types of methods: reinforcement learning and learning automata. The use of both methods on large-scale problems is a recent development, and the literature does not seem to provide sufficient numerical evidence, at this time, to compare the two.

11. Concluding Remarks

As far as parametric optimization is concerned, one can find a very large number of case studies using RSM in the literature. Use of combining neural networks, meta-heuristics, and simulation is a relatively new development. The same is true of RL. The related literature is however growing by the day. Some other empirical work can be found in [161], [170], [164], and the references therein.

Any of these case studies requires a great deal of computational work. A neural network approach requires testing of several parameters such as the number of hidden layers, the number of nodes, the learning rates etc. The same is true of meta-heuristic approaches, which need to be tuned for their tunable parameters such as temperatures, population sizes, or tabu-list sizes.

Getting an RL algorithm to work on a real-life case study usually requires that the simulator be written in a language such as C or MATLAB so that RL-related functions and function approximation routines can be incorporated into the simulator. The reason is in RL, unlike parametric optimization techniques, we do not evaluate the function at a fixed scenario; rather the RL functions have to be incorporated in the middle of trajectories. Furthermore, each problem has its own unique features. In some cases value iteration schemes seem to fare better than policy iteration schemes, and in some cases the reverse is true. Needless to say, the objective function in MDPs or SMDPs cannot be chosen by the analyst and depends on how it is measured in the real world for that particular problem. For instance, if an organization is interested in maximizing the average revenue per unit time, then one must use an average reward RL algorithm.

Choosing the right scheme for function approximation can often prove to be a time-consuming task. Again there is no unique strategy that seems to have worked on all problems. The reason for this is the behavior of RL, when the algorithm is combined with function approximation schemes, is not understood very well even now. When the understanding becomes clearer, it will become easier to choose a function approximation scheme.

12. Review Questions

1. Read the paper written by Kilmer, Smith, and Shuman [90]. Simulate the system for two products. Use a simulation-based neuro-response surface approach to solving the problem of optimizing for (S_1, s_1, S_2, s_2) .
2. Go to your library and get a copy of the paper written by Das and Sarkar [39] from your library. Read how the transition probabilities have been derived. This will give you an idea of how tedious this process can be. Then use a simulation-based parametric optimization approach to solve the problem. Verify your results with those published in [39].

3. Why does a parametric optimization model appear to be inadequate for the AGV case study described in this chapter? (A control optimization model was employed in Tadepalli and Ok [167].) In what way does the AGV case study differ from the other case studies discussed here?

Chapter 15

CODES

*There should be less talk; a preaching point is not a meeting point.
What do you do then? Take a broom and clean someone's
house. That says enough.*

— Mother Teresa (1910-1997)

1. Introduction

This chapter presents computer codes for some algorithms discussed in this book. Program writing is an important technological component of simulation-based optimization. It is true that all the material covered in the previous chapters will serve only as a “preaching point” — unless one learns how to use these algorithms to generate solutions for a potential client. Especially in the engineering community, these skills are as important as understanding the methodology itself.

We believe that unless you learn to write the computer programs needed, you will miss out on a great deal of the fun! *More importantly, developing new algorithms usually needs extensive numerical testing, which cannot be done without computer programming.* Furthermore, computer programming can help one in gaining *a better understanding of the convergence issues and in identifying those theoretical results which are critical for obtaining meaningful numerical results in practice!* We would like to emphasize that the programming aspect of simulation optimization is *not an art*, but a science.

Most of the computer programs in this chapter have been written in C. We assume the reader is familiar with programming in C. C is our choice for programming simulations because it gives the programmer the much needed flexibility that many simulation packages may not give. Some simulation packages

can be linked to C or BASIC or FORTRAN. In such cases, the optimization modules can be written in these languages and the actual simulations in the simulation language. However, writing the *entire* code in C usually makes it possible to reduce the run time of the code. The run time is a very critical issue in simulations. Simulators written in programming languages such as C (or BASIC or FORTRAN) run faster than simulation packages because the latter come with huge overheads. However, it is the case that it is easier to program in simulation packages.

For optimization purposes, where the simulations need to be run over *several* scenarios (each scenario in turn requiring several replications), C makes it possible to get results in a reasonable time interval. As such, although relatively more difficult, it is perhaps a good idea to learn how to write simulation programs in C (see Law and Kelton [102] and Pollatschek [137] for more details).

2. C programming

If you are familiar with any programming language, say Basic or Fortran, making the transition to C should not take much time. Actually, compared to Fortran and BASIC, C has an easier syntax. There are important differences, however. Go-to statements are not allowed in C and for good reasons. Fortunately, by now, several nice and reader-friendly books have been written in C. We recommend Bronson [32] to the beginner. It presents a very gentle introduction to C. After that, the reader may turn to more advanced texts such as Gottfried [66]. From these books, the reader should master, at the least, the following concepts:

- *Local* and *global* variables
- Variable types (*int*, *float*, and *double*)
- Assignment statements (e.g., $a=10$; or $a=a+1$)
- Arrays (multi-dimensional)
- *printf* statement and related syntax
- *If else* statements
- *For* and *while* loops (this is very important: if you are familiar with do loops of FORTRAN, this should be easy)
- *Switch* statements
- Relational operators ($==$, $<=$, $>=$, $<$, $>$, $!=$)
- Simple structures (read Bronson [32]) and the *typedef* statement (read Gottfried [66]).

- *Functions* (these are the analogues of *subroutines* of FORTRAN)

After understanding these concepts, it is important to master the concept of **pointers**. Without pointers, you will not make much progress in writing C programs. The pointer to a variable in a program is its address. Consider the following variable.

```
int gosavi;
```

The address of this variable — that is, the pointer to the variable is given by

```
&gosavi;
```

The & operator in front of a variable makes it a pointer. If you want to see what a pointer looks like, run the following program.

```
# include <stdio.h>
main()
{
    int gosavi;

    printf("The address of this variable is %d\n",&gosavi);

}
```

The printf statement will print something like the following on your screen:

```
The address of the variable gosavi is 1245252
```

The number 1245252 is a number (address) in the memory of the computer, where the variable gosavi is stored. We have little use for the actual value of the address. But, the nice thing about the address of a variable (that is a pointer) is that we can access the actual variable by using the * operator in front of its pointer. Thus, the * operator does the opposite of what the & operator does. Consider the following program:

```
# include <stdio.h>
main()
{
    int gosavi=10;
    int mirror;
    mirror=*&gosavi;
    printf("The mirror is now equal to %d\n",mirror);
```

Notice the output from the printf statement will be:

The mirror is now equal to 10

The reason why, in the above, mirror got assigned the same value as gosavi is that mirror was assigned the value of the variable to which the pointer: & gosavi pointed. Now, & gosavi, of course, pointed to gosavi, and as such mirror got the value of gosavi. If you understand this elementary concept, pointers will be easy and actually fun. We recommend Gottfried [66], among other sources, for mastering pointers.

We can pass the pointer of a variable into a function and then inside the function, we can retrieve the actual variable by using the * operator. Then if we make a change to the variable inside the function, the change occurs to the actual variable, whose pointer was passed. Then when the same variable is accessed outside the function, its value is found to have changed. If instead of passing its pointer, the actual variable is passed into a function, no matter what change one makes to the variable inside, the variable is not changed outside. This is actually a safeguard and prevents unwanted changes to variables.

It is also important to understand the dangers of using *global Variables*. **Global variables should NEVER be used** unless they are used for constants (such as the Gravitational constant). Global variables are extremely hazardous because they lead to logical errors that are very difficult to find. It is true that a global variable does not have to be passed to and back from a function, thereby bypassing the need for pointers, but it is precisely this aspect that makes it so dangerous. The right way to pass a variable into and out of functions is to pass its pointer. It is because of this that pointers are such important tools in C programming.

When a large number of variables are to be passed in and out from a function, it is best to make all those variables members of one or more structures. (In C++ all variables are made members of some class; however we will restrict our discussions to C in this book.) When this is done, one can pass the pointer to the entire structure into and back from the function concerned.

All this will make sense after you have mastered the initial concepts that we have mentioned above. Also, it is important to understand that like swimming cannot be learned without getting wet, programming cannot be learned without typing, compiling, and running programs!

3. Code Organization

Most of the codes in this book are organized in a format that we shall describe next. Every function is written in a separate file. The first file, called: main.c, is

the main file for the program. It is this file that should be compiled and executed. This file will “include” — at the top — a file called fun_list.h, besides other files.

The file called fun_list.h will include the names of all the *relevant* files needed for that particular program. (By relevant files, we mean all the files that contain code to be used for running that particular application.) When you run your program in UNIX or any Windows C compiler, it is best to include all the relevant files in one directory and then compile (and subsequently execute) the file main.c. A few global variables will be used. All letters in a global variable will always be in CAPITALS to distinguish it from other variables. Global variables will be defined in global.h. Some special user-defined types will be used, which will be defined in the file typedefs.h

Prototypes of all functions used in the program will be defined in the file: fun_prots.h. It is necessary to define the prototypes before defining the functions. With this, the functions can be placed in any order. As stated above, all functions are in separate files.

All user defined types (structures) will be defined with `typedef` statements in the `typedefs.h` file. You will see this file in most of the codes presented in this book.

In summary, every `main.c` file will contain the following so-called *header files*. The header files will have a `.h` extension.

- `global.h` (contains definitions of some global variables)
- `typedefs.h` (contains type definitions for structures)
- `fun_prots.h` (contains function prototypes of all relevant functions)
- `fun_list.h` (contains a list of files that contain all the relevant functions)

4. Random Number Generators

For most of the codes given in this chapter, we shall need the following random number generating codes. The next code — that is, `uniformrnd.c`, is a generating code that will be needed for random numbers of every distribution. In other words, it will be needed in all codes that have random numbers.

The code that generates random numbers between two arbitrary values a and b is called `unifrnd.c`. The exponential distribution generating code is called `expornd.c`, while the erlang (loosely referred to as gamma here) distribution code is called `gamrnd.c`.

```
*****uniformrnd.c *****  
double uniformrnd(long* seed)
```

```
{  
/* This routine, uniformrnd, is based on the routine ran0 */  
/* from the book Numerical Recipes in C (Cambridge University Press), */  
/* Copyright (c) 1987-1992 Numerical Recipes Software Used by permission. */  
/* Use of this routine other than as an integral part of uniformrnd */  
/* requires an additional license from Numerical Recipes Software. */  
/* Further distribution in any form is prohibited. */  
/* WHERE TO WRITE FOR PERMISSIONS:*/  
/* Numerical Recipes Software */  
/* P.O. Box 243 */  
/* Cambridge, MA 02238 FAX: 781-863-1739 Email: permissions@nr.com */  
  
/* Code uses Schrage's algorithm for generating uniformly distributed */  
/* random numbers between 0 and 1 */  
  
/* VERY IMPORTANT: seed should never be initialized to 0 */  
  
long k;  
  
k=seed/127773;  
seed=16807>(*seed-k*127773)-2836*k;  
if(*seed<0)  
{  
*seed+=2147483647;  
}  
return((1.0/2147483647)*(*seed));  
  
}  
  
*****unifrnd.c *****  
  
double unifrnd(double a, double b, long* seed)  
{  
/* unif distributed random numbers between a and b */  
double ran;  
  
ran=uniformrnd(seed);  
  
return(a+(b-a)*ran);  
  
}  
  
*****expornd.c *****  
double expornd(double mu, long* seed)  
{  
/* Generates an exponentially distributed random number with a
```

```
mean equal to 1/mu */

double x;

do
x = unifrnd(0.0, 1.0, seed);
while(x == 0.0);

return (-log(x)/mu);
}

/**********gmrnd.c *****/
double gmrnd(int n,double lambda,long* seed)
{
/* generates an n-erlang distributed random no. for parameters :*/
/* (n,lambda); mean=n/lambda*/
/* Using inverse function */
/* if n>8, other methods are recommended; see Law and Kelton */
int count;
double x;

/* adding exponential random numbers */
x=1.0;

for(count=1;count<=n;count=count+1)
{
x=x*uniformrnd(seed);
}
x=-log(x);

return (x/lambda);
}
```

5. Simultaneous Perturbation

This section contains codes for doing simultaneous perturbation. The function estimator.c returns the function value at any given point. This function has to be replaced by the simulator for simulation-optimization purposes. The random number generating codes can be found in the previous section.

```
#define NO_OF_PARAMETERS 2

#include <stdio.h>
#include <math.h>

#include "fun_protos.h"
```

```
#include "fun_list.h"

main()
{
double ran_no, h[NO_OF_PARAMETERS], multiplier, step_size,
step_size_limit, x_current[NO_OF_PARAMETERS], F_plus, F_minus,
x_increased[NO_OF_PARAMETERS], x_decreased[NO_OF_PARAMETERS],
partial_deriv[NO_OF_PARAMETERS], result_current, A, iteration;

long seed;
int binom[NO_OF_PARAMETERS], i;

iteration = 1;
step_size_limit = 0.001;
A = 1;
step_size = A;
seed = 1236537;

x_current[0] = 8.0;
x_current[1] = 7.0;
result_current = estimator(x_current);

while(step_size>step_size_limit)
{
    multiplier = 1/sqrt(iteration);

    for(i=0; i<NO_OF_PARAMETERS; i++)
    {
        ran_no = unifrnd(0, 1, &seed);
if(ran_no>0.5)
{
}
else
{
}
        binom[i] = 1;
    }

    h[i] = binom[i] * multiplier;

    x_increased[i] = x_current[i] + h[i];
    x_decreased[i] = x_current[i] - h[i];
}

F_plus = estimator(x_increased);
F_minus = estimator(x_decreased);

for(i=0; i<NO_OF_PARAMETERS; i++)
{
    partial_deriv[i] = (F_plus - F_minus)/(2*h[i]);
    x_current[i] = x_current[i] - (step_size*partial_deriv[i]);
}
```

```
    }

    result_current = estimator(x_current);

    iteration++;
    step_size = (A/iteration);
}

for(i=0;i<NO_OF_PARAMETERS;i=i+1)
{
    printf("x[%d]=%lf\n", i, x_current[i]);
}
printf("The optimum function value = %lf \n", result_current);
}

*****fun_prots.h*****
```

```
double estimator(double x_current[NO_OF_PARAMETERS]);
double uniformrnd(long* seed);
double unifrnd(double a, double b, long* seed);

*****fun_list.h*****
```

```
#include "estimator.c"
#include "uniformrnd.c"
#include "unifrnd.c"

*****estimator.c*****
```

```
/* function that estimates the objective function value */

double estimator(double x_current[NO_OF_PARAMETERS])
{
return((x_current[0] - 4) * (x_current[0] - 4)) +
((x_current[1] - 2) * (x_current[1] - 2));
}
```

6. Dynamic Programming Codes

In this section, we shall present codes for stochastic dynamic programming — to solve MDPs and SMDPs. The codes are distributed into the following sections.

Section 6.1: Policy Iteration for Average reward MDPs.

Section 6.2: Relative Value Iteration for Average reward MDPs.

Section 6.3: Policy Iteration for Discounted reward MDPs.

Section 6.4: Value Iteration for Discounted reward MDPs.

Section 6.5: Policy Iteration for Average reward SMDPs.

6.1. Policy Iteration for average reward MDPs

This code runs Example A (see Section 5.1). Suitable changes in main.c will be needed for running other examples; the changes will have to be made to the tpm and trm arrays.

```
/* main.c */
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"
main()
{
    double tpm[NA][NS][NS]={

        {
            {0.7,0.3},
            {0.4,0.6}
        },
        {
            {0.9,0.1},
            {0.2,0.8}
        }
    };
    double trm[NA][NS][NS]={

        {
            {6,-5},
            {7,12}
        },
        {
            {10,17},
            {-14,13}
        }
    };

    pia(tpm,trm);

    return;
}

/* global.h */

double SMALL=-100000;
# define NS 2
# define NA 2
```

```
/* fun_prots.h */

void pia(double [NA][NS][NS],double [NA][NS][NS]);
void solver(double [NS][NS+1],double [NS]);

/* fun_list.h */

# include "pia.c"
# include "solver.c"

/* pia.c */

void pia(double tpm[NA][NS][NS],double trm[NA][NS][NS])
{
/* Policy Iteration for Average Reward MDPs. */
/* The strategy adopted for the under-determine linear system is */
/* to replace first value by 0 */

int state,next_state,iteration,done,row,col,policy[NS],action,best_action;
double G[NS][NS+1],x[NS],large,sum,rho;

/* We start with an arbitrary policy. */

for(state=0;state<NS;state++)
{
policy[state]=0;
}

iteration=0;
done=1;

while(1==done)
{/* As long as two consecutive policies do not become identical. */

/* 1. Policy evaluation stage */
/* The linear equations to be solved are Gx=0.*/
/* Initializing a part of the G Matrix. */

for(row=0;row<NS;row++)
{
    for(col=0;col<NS;col++)
    {

        if(col==0)
        { /* because the first value is replaced by rho */
G[row][col]=1;
}
        else
{
```

```

        if(row==col)
        {
            G[row][col]=1-tpm[policy[row]][row][col];
        }
        else
        {
            G[row][col]=-tpm[policy[row]][row][col];
        }
    }
}

/* Initializing the (NS+1)th column of G matrix */

for(state=0;state<NS;state++)
{
    sum=0.0;
    for(next_state=0;next_state<NS;next_state++)
    {
        sum=sum+(tpm[policy[state]][state][next_state]*
                  trm[policy[state]][state][next_state]);
    }
    G[state][NS]=sum;
}

solver(G,x); /* x comes out as the solution */

/* Determine the avg reward */

rho=x[0];

printf("Average reward in iteration % d:%lf\n",iteration,x[0]);

/* The first value is 0 */

x[0]=0;

/* 2. Policy improvement stage */

done=0;

for(state=0;state<NS;state++)
{
    large=SMALL;
    best_action=0;

    for(action=0;action<NA;action++)
    { /* determine the best action for the state */
        sum=0;

```

```
for(next_state=0;next_state<NS;next_state++)
{
    sum=sum+ (tpm[action][state][next_state]*
    (trm[action][state][next_state]+x[next_state]));
}
if(sum>large)
{
    large=sum;
    best_action=action;
}
}

if(policy[state]!=best_action)
/* Policy has improved; record new action */
policy[state]=best_action;
done=1; /* to ensure that one more iteration is done */
}

iteration=iteration+1;
}

printf("Number of iterations needed: %d\n",iteration);

for(state=0;state<NS;state++)
{
printf("Optimal action for state %d is %d\n",state,policy[state]);
}
for(state=0;state<NS;state++)
{
printf("Optimal value for state %d is %lf\n",state,x[state]);
}

return;
}

/* solver.c */

void solver(double G[NS][NS+1],double x[NS])
{
/* Using GAUSS JORDAN ELIMINATION */
int row, col,row1,col1,col2,pivot_row;
double factor,pivot,temp;

/* Find max element */

for(col=0;col<=NS-1;col++)
```

```

{
pivot=-0.1;
/* Find the best pivot */
for(row=col;row<=NS-1;row++)
{
    if(fabs(G[row][col])>pivot)
    {
        pivot=fabs(G[row][col]);
        pivot_row=row;
    }
}
/* To check if solution can be found */
if(pivot<=0.00001)
{ /* pivot is 0 or else too small - will lead to errors */
printf("Error in policy evaluation. Singular matrix.\n");
exit(2);
}

if(pivot_row!=col)
{/* exchange rows to use the best pivot */
    for(col1=col;col1<=NS;col1++)
    {
        temp=G[col][col1];
        G[col][col1]=G[pivot_row][col1];
        G[pivot_row][col1]=temp;
    }
}

/* Gauss elimination*/
for(row1=0;row1<=NS-1;row1++)
{
    if(row1!= col)
    {
        factor=G[row1][col]/G[col][col];
        for(col2=col;col2<=NS;col2++)
        {
            G[row1][col2] -=factor*G[col][col2];
        }
    }
}

/* Find solution */
for(row=0;row<NS;row++)
{
x[row]=G[row][NS]/G[row][row];
}

return;
}

```

```
}
```

6.2. Relative Iteration for average reward MDPs

This code runs Example A (see Section 5.1).

```
/* Codes for solving average reward MDPs using RVI */
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"
main()
{
double epsilon=0.001;
double tpm[NA][NS][NS]={
    {
{0.7,0.3},
{0.4,0.6}
    },
    {
        {0.9,0.1},
        {0.2,0.8}
    }
};
double trm[NA][NS][NS]={
    {
{6,-5},
{7,12}
    },
    {
        {10,17},
        {-14,13}
    }
};

rvia(tpm,trm,epsilon);

return;
}

/* fun_list.h */
# include "rvia.c"

/* fun_protos.h */
void rvia(double [NS][NS][NA],double [NS][NS][NA],double);
```

```
/* global.h */
double SMALL=-100000;
#define NS 2
#define NA 2

/* rvia.c */

void rvia(double tpm[NA][NS][NS],double trm[NA][NS][NS],double epsilon)
{
/* The code assumes state 0 to be the subtraction factor state */
int done,iter=0,state,next_state,action,policy[NS];
double value[NS],value_old[NS],best,sum,span,max_val,min_val,sub_factor;

/* Initialize values */

for(state=0;state<=NS-1;state=state+1)
{
    value[state]=0;
}

done=1;

/********************************************/

while(done!=0)
{ /* main loop of value iteration */
/* Keeping a copy of the old value function */
for(state=0;state<=NS-1;++state)
{
    value_old[state]=value[state];
}

/* Value Update */

for(state=0;state<=NS-1;state=state+1)
{ /* Value updated state by state */
best=SMALL;

    for(action=0;action<=NA-1;action=action+1)
    { /* Find the best value for each state */
sum=0;
        for(next_state=0;next_state<=NS-1;++next_state)
        {
            sum=sum+(tpm[action][state][next_state]*
(trm[action][state][next_state]+value_old[next_state]));
        }
        if(sum>best)
        {
            best=sum;
        }
    }
}
}
```

```
        policy[state]=action;
        value[state]=best;
    }
}

sub_factor=value[0];

for(state=0;state<=NS-1;state=state+1)
{
    value[state]=value[state]-sub_factor; /* RVI subtraction */
}
/* Determine the span of the difference vector */
max_val=value[0]-value_old[0];
min_val=value[0]-value_old[0];

for(state=1;state<=NS-1;state=state+1)
{
    if(value[state]-value_old[state]>max_val)
    {
        max_val=value[state]-value_old[state];
    }
    if(value[state]-value_old[state]<min_val)
    {
        min_val=value[state]-value_old[state];
    }
}

span=max_val-min_val;
/* Determine whether to terminate */
if(span<epsilon)
{ /* terminate */
done=0;
}

for(state=0;state<=NS-1;++state)
{
printf("The value function for state %d=%lf\n",state,value[state]);
}

iter=iter+1;

/*
/* Display policy, and value function */

for(state=0;state<=NS-1;state=state+1)
```

```

{
printf("Epsilon-optimal action for state %d\n",state,policy[state]);
}
for(state=0;state<=NS-1;state=state+1)
{
printf("The value function for state %lf\n",state,value[state]);
}

printf("The number of iterations needed to converge= %d\n",iter);

return;
}

```

6.3. Policy Iteration for discounted reward MDPs

This code runs Example A (see Section 5.1).

```

/* Codes for policy iteration of discounted reward MDPs */
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"
main()
{

double tpm[NA][NS][NS]={
    {
{0.7,0.3},
{0.4,0.6}
    },
    {
        {0.9,0.1},
        {0.2,0.8}
    }
};
double trm[NA][NS][NS]={
    {
{6,-5},
{7,12}
    },
    {
        {10,17},
        {-14,13}
    }
};

```

```
pid(tpm,trm,0.8);

return;
}

/* fun_list.h */

#include "pid.c"
#include "solver.c"

/* fun_protos.h */

void pid(double [NA][NS],double [NA][NS],double);
void solver(double [NS][NS+1],double [NS]);
double SMALL=-100000;

/* global.h */

#define NS 2
#define NA 2

/* pid.c */

void pid(double tpm[NA][NS],double trm[NA][NS],double d_factor)
{
/* Policy Iteration for Discounted Reward MDPs. */
int state,next_state,iteration,done,row,col,policy[NS],action,best_action;
double G[NS][NS+1],x[NS],large,sum;

/* We start with an arbitrary policy. */

for(state=0;state<NS;state++)
{
policy[state]=0;
}

iteration=0;
done=1;

while(1==done)
{/* As long as two consecutive policies do not become identical. */

/* 1. Policy evaluation stage */
/* The linear equations to be solved are Gx=0. */
/* Initializing a part of the G Matrix. */

for(row=0;row<NS;row++)
{
```

```

for(col=0;col<NS;col++)
{
    if(row==col)
    {
        G[row][col]=1-d_factor*(tpm[policy[row]][row][col]);
    }
    else
    {
        G[row][col]=-d_factor*tpm[policy[row]][row][col];
    }
}
}

/* Initializing the (NS+1)th column of G matrix */

for(state=0;state<NS;state++)
{
sum=0.0;
for(next_state=0;next_state<NS;next_state++)
{
    sum=sum+(tpm[policy[state]][state][next_state]*
              trm[policy[state]][state][next_state]);
}
G[state][NS]=sum;
}

solver(G,x); /* x comes out as the solution */

/* 2. Policy improvement stage */

done=0;

for(state=0;state<NS;state++)
{
large=SMALL;
best_action=0;

for(action=0;action<NA;action++)
{ /* determine the best action for the state */
sum=0;

for(next_state=0;next_state<NS;next_state++)
{
    sum=sum+ (tpm[action][state][next_state]*
              (trm[action][state][next_state]+
               (d_factor*x[next_state])));
}
if(sum>large)
{
    large=sum;
}
}
}

```

```
    best_action=action;
}

}

if(policy[state]!=best_action)
/* Policy has improved; record new action */
policy[state]=best_action;
done=1; /* to ensure that one more iteration is done */
}

iteration=iteration+1;
}

printf("Number of iterations needed: %d\n",iteration);
for(state=0;state<NS;state++)
{
printf("Optimal action for state %d is %d\n",state,policy[state]);
}
for(state=0;state<NS;state++)
{
printf("Optimal value for state %d is %lf\n",state,x[state]);
}

return;
}

/* For solver.c, see section on average reward policy iteration */
```

6.4. Value Iteration for discounted reward MDPs

This code runs Example A (see Section 5.1).

```
/* Codes for value iteration for discounted reward MDPs */
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"
main()
{
double epsilon=0.001;
double tpm[NA][NS][NS]={
{
```

```
{0.7,0.3},  
{0.4,0.6}  
},  
{  
    {0.9,0.1},  
    {0.2,0.8}  
}  
};  
double trm[NA][NS][NS]={  
    {  
        {6,-5},  
        {7,12}  
    },  
    {  
        {10,17},  
        {-14,13}  
    }  
};  
  
vi(tpm,trm,0.8,epsilon);  
  
return;  
}  
/* global.h */  
  
double SMALL=-100000;  
# define NS 2  
# define NA 2  
  
/* fun_prots.h */  
  
void vi(double [NS][NS][NA],double [NS][NS][NA],double,double);  
  
/* fun_list.h */  
  
# include "vi.c"  
  
/* vi.c */  
  
void vi(double tpm[NA][NS][NS],double trm[NA][NS][NS],double  
dfactor,double epsilon)  
{  
  
int done,iter=0,state,next_state,action,policy[NS];  
double value[NS],value_old[NS],best,sum,norm;  
  
/* Determine termination factor */  
epsilon=epsilon*(1-dfactor)*0.5/dfactor;
```

```
/* Initialize values */

for(state=0;state<=NS-1;state=state+1)
{
    value[state]=0;
}

done=1;

/********************************************/

while(done!=0)
{ /* main loop of value iteration */
/* Keeping a copy of the old value function */
for(state=0;state<=NS-1;++state)
{
    value_old[state]=value[state];
}

/* Value Update */

for(state=0;state<=NS-1;++state)
{ /* Value updated state by state */
best=SMALL;

for(action=0;action<=NA-1;++action)
{ /* Find the best value for each state */
sum=0;
    for(next_state=0;next_state<=NS-1;++next_state)
    {
        sum=sum+(tpm[action][state][next_state]*
        (trm[action][state][next_state]+
        dfactor*value_old[next_state]));
    }
    if(sum>best)
    {
        best=sum;
        policy[state]=action;
        value[state]=best;
    }
}
}

/* Determine the norm of the difference vector */
norm=-1;
for(state=0;state<=NS-1;++state)
{
    if(fabs(value[state]-value_old[state])>norm)
{
```

```

        norm=fabs(value[state]-value_old[state]);
    }

}

/* Determine whether to terminate */
if(norm<epsilon)
{ /* terminate */
done=0;
}

iter=iter+1;
}

/* Display policy, and value function */

for(state=0;state<=NS-1;++state)
{
printf("Epsilon-optimal action for state %d=%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;++state)
{
printf("The value function for state %d=%lf\n",state,value[state]);
}

printf("The number of iterations needed to converge = %d\n",iter);

return;
}

```

The following two codes : rvi.c and gsvi.c can be used in place of vi.c. To use them the fun_protos.h and fun_list.h would have to be suitably modified.

```

void rvi(double tpm[NA][NS][NS],double trm[NA][NS][NS],double
dfactor,double epsilon)
{
/* The code assumes state 0 to be the subtraction factor state */
int done,iter=1,state,next_state,action,policy[NS];
double value[NS],value_old[NS],best,sum,norm;

/* Determine termination factor */
epsilon=epsilon*(1-dfactor)*0.5/dfactor;

/* Initialize values */

```

```
for(state=0;state<=NS-1;state=state+1)
{
    value[state]=0;
}

done=1;

/********************************************/

while(done!=0)
{ /* main loop of value iteration */
/* Keeping a copy of the old value function */
for(state=0;state<=NS-1;++state)
{
    value_old[state]=value[state];
}

/* Value Update */

for(state=0;state<=NS-1;++state)
{ /* Value updated state by state */
best=SMALL;

    for(action=0;action<=NA-1;++action)
    { /* Find the best value for each state */
sum=0;
        for(next_state=0;next_state<=NS-1;++next_state)
        {
            sum=sum+(tpm[action][state][next_state]*
(trm[action][state][next_state]+
dfactor*value_old[next_state]));
        }
        sum=sum-value_old[0]; /* RVI subtraction */
if(sum>best)
{
    best=sum;
    policy[state]=action;
    value[state]=best;
}
    }
}

/* Determine the norm of the difference vector */
norm=-1;
for(state=0;state<=NS-1;++state)
{
    if(fabs(value[state]-value_old[state])>norm)
    {
        norm=fabs(value[state]-value_old[state]);
    }
}
```

```
        }
        /* Determine whether to terminate */
        if(norm<epsilon)
        { /* terminate */
        done=0;
        }

    iter=iter+1;
}

/* Display policy, and value function */

for(state=0;state<=NS-1;++state)
{
printf("Epsilon-optimal action for state %d=%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;++state)
{
printf("The value function for state %d=%lf\n",state,value[state]);
}

printf("The number of iterations needed to converge= %d\n",iter);

return;
}

/* gsvi.c */

void gsvi(double tpm[NA][NS][NS],double trm[NA][NS][NS],double
dfactor,double epsilon)
{
int done,iter=1,state,next_state,action,policy[NS];
double value[NS],value_old[NS],best,sum,norm;

/* Determine termination factor */
epsilon=epsilon*(1-dfactor)*0.5/dfactor;

/* Initialize values */

for(state=0;state<=NS-1;state=state+1)
{
    value[state]=0;
}

done=1;

/*********************************************/
```

```
while(done!=0)
{ /* main loop of value iteration */
/* Keeping a copy of the old value function */
for(state=0;state<=NS-1;++state)
{
    value_old[state]=value[state];
}

/* Value Update */

for(state=0;state<=NS-1;++state)
{ /* Value updated state by state */
best=SMALL;

for(action=0;action<=NA-1;++action)
{ /* Find the best value for each state */
sum=0;
    for(next_state=0;next_state<=NS-1;++next_state)
    {
        sum=sum+(tpm[action][state][next_state]*
        (trm[action][state][next_state]+
        dfactor*value[next_state]));
    }
    if(sum>best)
    {
        best=sum;
        policy[state]=action;
        value[state]=best;
    }
}
}

/* Determine the norm of the difference vector */
norm=-1;
for(state=0;state<=NS-1;++state)
{
    if(fabs(value[state]-value_old[state])>norm)
    {
        norm=fabs(value[state]-value_old[state]);
    }
}

/* Determine whether to terminate */
if(norm<epsilon)
{ /* terminate */
done=0;
}
for(state=0;state<=NS-1;state++)
{
    printf("The value for state=%d is %lf\n",state,value[state]);
}
```

```

printf("The number of iterations =%d\n",iter);
printf("The number of done=%d\n",done);
printf("The value of norm =%lf\n",norm);

iter=iter+1;
}

/* Display policy, and value function */

for(state=0;state<=NS-1;++state)
{
printf("Epsilon-optimal action for state %d=%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;++state)
{
printf("The value function for state %d=%lf\n",state,value[state]);
}

printf("The number of iterations needed to converge = %d\n",iter);

return;
}

```

6.5. Policy Iteration for average reward SMDPs

This code runs Example B (see Section 9.2.2).

```

/* Codes for policy iteration of average reward SMDPs */
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"
main()
{

double tpm[NA][NS][NS]={
    {
{0.7,0.3},
{0.4,0.6}
    },
    {
        {0.9,0.1},
        {0.2,0.8}
    }
};

};
```

```
double trm[NA][NS][NS]={  
    {  
        {6,-5},  
        {7,12}  
    },  
    {  
        {10,17},  
        {-14,13}  
    }  
};  
  
double ttm[NA][NS][NS]={  
    {  
        {1,5},  
        {120,60}  
    },  
    {  
        {50,75},  
        {7,2}  
    }  
};  
  
pias(tpm,trm,ttm);  
  
return;  
}  
  
/* global.h */  
  
double SMALL=-100000;  
# define NS 2  
# define NA 2  
  
/* fun_prots.h */  
  
void pias(double [NA][NS][NS],double [NA][NS][NS],double [NA][NS][NS]);  
void solver(double [NS][NS+1],double [NS]);  
  
/* fun_list.h */  
  
# include "pias.c"  
# include "solver.c"  
  
/* pias.c */
```

```

void pias(double tpm[NA][NS][NS],double trm[NA][NS][NS],double
ttm[NA][NS][NS])
{
/* Policy Iteration for Average Reward MDPs. */
/* The strategy adopted for the under-determine linear system is */
/* to replace first value by 0 */

int state,next_state,iteration,done,row,col,policy[NS],action,best_action;
double G[NS][NS+1],x[NS],large,sum,sum1,sum2,rho,gain;

/* We start with an arbitrary policy. */

for(state=0;state<NS;state++)
{
policy[state]=0;
}

iteration=0;
done=1;

while(1==done)
{/* As long as two consecutive policies do not become identical. */

/* 1. Policy evaluation stage */
/* The linear equations to be solved are Gx=0. */
/* Initializing a part of the G Matrix. */

for(row=0;row<NS;row++)
{
for(col=0;col<NS;col++)
{
if(col==0)
{/* first column contains coefficients of g*/
sum=0;
for(next_state=0;next_state<NS;next_state++)
{
sum=sum+(tpm[policy[row]][row][next_state]*
ttm[policy[row]][row][next_state]);
}
G[row][0]=sum;
}
else
{
if(row==col)
{
G[row][col]=1-tpm[policy[row]][row][col];
}
else
{
}
}
}
}
}

```

```
{  
    G[row][col]=-tpm[policy[row]][row][col];  
}  
}  
}  
  
/* Initializing the (NS+1)th column of G matrix */  
  
for(state=0;state<NS;state++)  
{  
sum=0.0;  
    for(next_state=0;next_state<NS;next_state++)  
    {  
        sum=sum+(tpm[policy[state]][state][next_state]*  
                  trm[policy[state]][state][next_state]);  
    }  
    G[state][NS]=sum;  
}  
  
solver(G,x); /* x comes out as the solution */  
  
  
/* Determine the avg reward */  
  
rho=x[0];  
  
printf("Average reward in iteration % d:%lf\n",iteration,x[0]);  
  
/* The first value is 0 */  
  
x[0]=0;  
  
/* 2. Policy improvement stage */  
  
done=0;  
  
for(state=0;state<NS;state++)  
{  
large=SMALL;  
best_action=0;  
  
for(action=0;action<NA;action++)  
{ /* determine the best action for the state */  
sum1=0;  
sum2=0;  
  
for(next_state=0;next_state<NS;next_state++)
```

```

{
    sum1=sum1+ (tpm[action][state][next_state]*
    (trm[action][state][next_state]+x[next_state]));
    sum2=sum2+ (tpm[action][state][next_state]*
    ttm[action][state][next_state]);
}

gain=sum1-(rho*sum2);
if(gain>large)
{
    large=gain;
    best_action=action;
}
}

if(policy[state]!=best_action)
/* Policy has improved; record new action */
policy[state]=best_action;
done=1; /* to ensure that one more iteration is done */
}

iteration=iteration+1;
}

printf("Number of iterations needed: %d\n",iteration);

for(state=0;state<NS;state++)
{
printf("Optimal action for state %d is %d\n",state,policy[state]);
}
for(state=0;state<NS;state++)
{
printf("Optimal value for state %d is %lf\n",state,x[state]);
}

return;
}
/* For solver.c see policy iteration for average reward MDPs */

```

7. Codes for Neural Networks

In this section, we shall first present the codes for a neuron and then for the backprop algorithm.

7.1. Neuron

This section provides codes for a batch-mode implementation of the neuron.

```
# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"

main()
{
double weights[NO_INPUTS],input_values[DATA_SIZE][NO_INPUTS],
target_values[DATA_SIZE],bias_weight,x[NO_INPUTS];

init_net(weights,&bias_weight);

reader(input_values,target_values);

neuron(weights,&bias_weight,input_values,target_values);

x[0]=0.2;
x[1]=0.5;

evaluator(weights,bias_weight,x);

return;
}

/** global.h***** */

#define NO_INPUTS 2 /* number of input nodes */
#define DATA_SIZE 10 /* number of training examples in the batch */
int ITERMAX=700;
long SEED1=12;
long SEED2=1324;
long SEED3=3456;
long SEED4=5678;
/** fun_protos.h***** */

int reader(double [DATA_SIZE][NO_INPUTS],double [DATA_SIZE]);
double compute_sigmoid(double);
double uniformrnd(long*);
double unifrnd(double,double,long*);
void neuron(double [NO_INPUTS], double*,
double [DATA_SIZE][NO_INPUTS],double [DATA_SIZE]);
void simu_net(double [DATA_SIZE][NO_INPUTS],double
[NO_INPUTS],double,double [DATA_SIZE]);
void init_net(double [NO_INPUTS],double* );
```

```
void evaluator(double [NO_INPUTS],double, double [NO_INPUTS]);  
  
*****fun_list.h*****  
  
# include "uniformrnd.c"  
# include "unifrnd.c"  
# include "init_net.c"  
# include "simu_net.c"  
# include "neuron.c"  
# include "reader.c"  
# include "evaluator.c"  
  
*****init_net.c*****  
  
void init_net(double weights[NO_INPUTS],double* bias_weight)  
{  
/* initialize weights in the neural net */  
  
int i;  
  
    for(i=0;i<NO_INPUTS;i++)  
    {  
        weights[i]=unifrnd(0.0,0.1,&SEED1);  
    }  
  
    *bias_weight=unifrnd(0.0,0.1,&SEED2);  
  
return;  
}  
  
*****input.dat*****  
  
0.000783  
0.153779  
3.769677  
0.560532  
0.865013  
7.885598  
0.276724  
0.895919  
7.756317  
0.704462  
0.886472  
8.136820  
0.929641  
0.469290  
6.276089  
0.350208  
0.941637  
8.058394
```

```
0.096535
0.457211
5.382590
0.346164
0.970019
8.196260
0.114938
0.769819
6.964032
0.341565
0.684224
6.762684

/****************neuron.c*****/

void neuron(double weights[NO_INPUTS], double*
bias_weight, double input_values[DATA_SIZE][NO_INPUTS],double
target_values[DATA_SIZE])
{
    double output_values[DATA_SIZE],output_deltas[DATA_SIZE];
    double best_sse,best_weights[NO_INPUTS],best_bias_weight;
    double sse,change,denom,error,learning_rate;
    int iter,i,p;

    /* This function trains a neuron net in batch mode; */
    /* a bias unit has been used; */

    learning_rate=0.01;
    best_sse=100000; /* some large number */
    simu_net(input_values,weights,*bias_weight,output_values);

    iter=0;

    while(iter<ITERMAX)
    {/* one iteration of training */

        /* update bias weight */
        change=0;
        for(p=0;p<DATA_SIZE;p++)
            {/* computing deltas for output unit for each point */
            output_deltas[p]=target_values[p]-output_values[p];
            change=change+output_deltas[p];
            }
        *bias_weight=*bias_weight+(learning_rate*change);

        /* updating weights from input units to output*/
        for(i=0;i<NO_INPUTS;i++)
    }
```

```
{  
    change=0;  
    for(p=0;p<DATA_SIZE;p++)  
    {/* sum change over all points */  
        change=change+(output_deltas[p]*input_values[p][i]);  
    }  
    weights[i]=weights[i]+(change*learning_rate);  
  
}  
  
simu_net(input_values,weights,*bias_weight,output_values);  
  
sse=0;  
for(p=0;p<DATA_SIZE;p++)  
{/* finding sum of the SSE */  
error=output_values[p]-target_values[p];  
sse=sse+(error*error);  
}  
printf("# of training iterations=%d\n",iter);  
  
if(sse<best_sse)  
{/* save the weights as the best weights so far */  
best_sse=sse;  
    for(i=0;i<NO_INPUTS;i++)  
    {  
        best_weights[i]=weights[i];  
    }  
best_bias_weight=*bias_weight;  
}  
  
printf("SSE=%lf\n",best_sse);  
iter=iter+1;  
learning_rate=learning_rate*0.99999;  
}  
  
/* return the best weights learned */  
    for(i=0;i<NO_INPUTS;i++)  
    {  
        weights[i]=best_weights[i];  
    }  
*bias_weight=best_bias_weight;  
  
  
return;  
}
```

```
*****reader.c*****  
  
int reader(double input_values[DATA_SIZE][NO_INPUTS],double  
target_values[DATA_SIZE])  
{/* Reads input and target values from a file; */  
/* each line has a different value. */  
/* The first few lines contain the input values and then the next line */  
/* contains the target value and then the sequence repeats */  
  
FILE *fid;  
int p,i;  
fid=fopen("input.dat","r");  
for(p=0;p<DATA_SIZE;p++)  
{  
    for(i=0;i<NO_INPUTS;i++)  
    {  
        fscanf(fid,"%lf \n",&input_values[p][i]);  
    }  
    fscanf(fid,"%lf \n",&target_values[p]);  
}  
fclose(fid);  
  
return;  
}  
  
*****simu_net.c*****  
  
void simu_net(double input_values[DATA_SIZE][NO_INPUTS],double  
weights[NO_INPUTS],double bias_weight,double output_values[DATA_SIZE])  
{  
/* evaluates the values in the net; */  
double sum;  
int i,p;  
  
for(p=0;p<DATA_SIZE;p++)  
{/* for each data point */  
    sum=0.0;  
    for(i=0;i<NO_INPUTS;i++)  
    {/* over all inputs */  
        sum=sum+(weights[i]*input_values[p][i]);  
    }  
    output_values[p]=sum+bias_weight;  
}  
  
return;  
}
```

7.2. Backprop Algorithm — Batch Mode

This subsection provides codes for the batch-mode of the backprop algorithm. We use a virtual bias node in the input layer and a bias node in the hidden layer. This issue has been discussed in Chapter 6. The codes for an incremental implementation are left as an exercise for the student. If you see how this code follows from the algorithm description in Chapter 6 (it follows directly excepting for the additional virtual bias weights from the virtual bias node to each hidden node), writing the codes for the incremental mode will not be difficult.

```

# include <stdio.h>
# include <math.h>
# include "global.h"
# include "fun_protos.h"
# include "fun_list.h"

main()
{
double ih_weights[NO_INPUTS][NO_HIDDEN],ho_weights[NO_HIDDEN],
input_values[DATA_SIZE][NO_INPUTS],hidden_values[NO_HIDDEN],
target_values[DATA_SIZE],x[NO_INPUTS],bias_weight,vbias_weights[NO_HIDDEN];

init_net(ih_weights,ho_weights,&bias_weight,vbias_weights);

reader(input_values,target_values);
backprop_batch(ih_weights,ho_weights,&bias_weight,vbias_weights,
input_values,target_values);

x[0]=0.2;
x[1]=0.5;

evaluator(ih_weights,ho_weights,bias_weight,vbias_weights,x);

return;
}

/*************global.h*******/
#define NO_INPUTS 2 /* number of input nodes */
#define NO_HIDDEN 3 /* number of hidden nodes */
#define DATA_SIZE 10 /* number of training examples in the batch */
int ITERMAX=500;
long SEED1=12;
long SEED2=1324;
long SEED3=3456;
long SEED4=5678;

/* *****fun_protos.h******/

```

```
void reader(double [DATA_SIZE] [NO_INPUTS],double [DATA_SIZE]);
void backprop_batch(double [NO_INPUTS] [NO_HIDDEN],double
[NO_HIDDEN],double*, double [NO_HIDDEN],double
[DATA_SIZE] [NO_INPUTS],double [DATA_SIZE]);
double compute_sigmoid(double);
void init_net(double [NO_INPUTS] [NO_HIDDEN],double
[NO_HIDDEN],double*,double [NO_HIDDEN]);
void simu_net(double [DATA_SIZE] [NO_INPUTS],double
[NO_INPUTS] [NO_HIDDEN],double [NO_HIDDEN],double, double
[NO_HIDDEN],double [DATA_SIZE] [NO_HIDDEN],double [DATA_SIZE]);
double uniformrnd(long*);
double unifrnd(double,double,long*);
void evaluator(double [NO_INPUTS] [NO_HIDDEN],double
[NO_HIDDEN],double,double [NO_HIDDEN],double[NO_INPUTS]);

/* *****fun_list.h*****/
#include "uniformrnd.c"
#include "unifrnd.c"
#include "compute_sigmoid.c"
#include "init_net.c"
#include "simu_net.c"
#include "backprop_batch.c"
#include "reader.c"
#include "evaluator.c"

*****backprop_batch.c*****/

void backprop_batch(double ih_weights[NO_INPUTS] [NO_HIDDEN],double
ho_weights[NO_HIDDEN],double* bias_weight,double vbias_weights[NO_HIDDEN],
double input_values[DATA_SIZE] [NO_INPUTS],double target_values[DATA_SIZE])
{
    double output_values[DATA_SIZE],hidden_values[DATA_SIZE] [NO_HIDDEN];
    double hidden_deltas[DATA_SIZE] [NO_HIDDEN],output_deltas[DATA_SIZE];
    double best_sse,best_ih_weights[NO_INPUTS] [NO_HIDDEN],
    best_ho_weights[NO_HIDDEN],best_vbias_weights[NO_HIDDEN],best_bias_weight;
    double sse,change,denom,error,learning_rate;
    int iter,i,h,p;

    /* This function trains a neural net in batch mode; */
    /* uses a single hidden layer;using a sigmoid transfer function only from*/
    /* input to hidden,not from hidden to output; */
    /* a bias unit has been used; */

    learning_rate=0.01;
    best_sse=100000; /* some large number */
    simu_net(input_values,ih_weights,ho_weights,*bias_weight,vbias_weights,
    hidden_values,output_values);
```

```
iter=0;

while(iter<ITERMAX)
{/* one iteration of training */

    for(p=0;p<DATA_SIZE;p++)
        {/* computing deltas for output unit for each point */
        output_deltas[p]=target_values[p]-output_values[p];
     }

    /* update bias weight */

    change=0;
    for(p=0;p<DATA_SIZE;p++)
        {/* summing over all points */
        change=change+output_deltas[p];
     }
    *bias_weight=*bias_weight+(learning_rate*change);

    /* compute hidden deltas */
    for (h=0;h<NO_HIDDEN;h++)
        {/* computing delta for hidden units for each point */
        for(p=0;p<DATA_SIZE;p++)
            {/* for each point (training example) */
            hidden_deltas[p][h]=output_deltas[p]*ho_weights[h]
            *hidden_values[p][h]*(1-hidden_values[p][h]);
         }
    }

    /* updating weights from input units to hidden units */

    for(i=0;i<NO_INPUTS;i++)
    {
        for(h=0;h<NO_HIDDEN;h++)
        {
            change=0;
            for(p=0;p<DATA_SIZE;p++)
                {/* sum change over all points */
                change=change+
                (hidden_deltas[p][h]*input_values[p][i]);
             }
            ih_weights[i][h]=ih_weights[i][h]+(change*learning_rate);
        }
    }

    /* updating virtual bias weights */
    for(h=0;h<NO_HIDDEN;h++)
    {
        change=0;
```

```
        for(p=0;p<DATA_SIZE;p++)
        {/* sum change over all points */
         change=change+hidden_deltas[p][h];
        }
        vbias_weights[h]=vbias_weights[h]+(change*learning_rate);
    }

/* update weights from hidden to output units */
for(h=0;h<NO_HIDDEN;h++)
{
    change=0;
    for(p=0;p<DATA_SIZE;p++)
    {/* sum over all points */
     change=change+(output_deltas[p]*hidden_values[p][h]);
    }
    ho_weights[h]=ho_weights[h]+(learning_rate*change);
}

simu_net(input_values,ih_weights,ho_weights,*bias_weight,vbias_weights,
hidden_values,output_values);

sse=0;
for(p=0;p<DATA_SIZE;p++)
{/* finding sum of the SSE */
error=output_values[p]-target_values[p];
sse=sse+(error*error);
}
printf("# of training iterations=%d\n",iter);

if(sse<best_sse)
/* save the weights as the best weights so far */
best_sse=sse;
for(h=0;h<NO_HIDDEN;h++)
{
    best_ho_weights[h]=ho_weights[h];
    best_vbias_weights[h]=vbias_weights[h];
    for(i=0;i<NO_INPUTS;i++)
    {
        best_ih_weights[i][h]=ih_weights[i][h];
    }
}
best_bias_weight=*bias_weight;
}

printf("SSE=%lf\n",best_sse);
iter=iter+1;
learning_rate=learning_rate*0.99999;
}
/* return the best weights learned */
```

```
for(h=0;h<NO_HIDDEN;h++)
{
    ho_weights[h]=best_ho_weights[h];
    vbias_weights[h]=best_vbias_weights[h];
    for(i=0;i<NO_INPUTS;i++)
    {
        ih_weights[i][h]=best_ih_weights[i][h];
    }
}
*bias_weight=best_bias_weight;

return;
}

/*************compute_sigmoid.c *****/
double compute_sigmoid(double input)
{
/* this function returns the output of a sigmoid transfer function */

if(input<-709.0)
{
    return (1.0);
}
else if(input>709.0)
{
    return (0.0);
}
else
{
    return (1.0/(1.0+exp(-input)));
}
}

/***********evaluator.c******/
void evaluator(double ih_weights[NO_INPUTS][NO_HIDDEN],double
ho_weights[NO_HIDDEN],double bias_weight,double
vbias_weights[NO_HIDDEN],double x[NO_INPUTS])
{
/* evaluates the output for x*/
/* sigmoid thresholding is done only from input to hidden */
/* layer but not from hidden to output layer */

double sum1,sum2,hidden[NO_HIDDEN];
int h,i;
```

```
for(h=0;h<NO_HIDDEN;h++)
{ /* calculate the sum of all inputs to each hidden node */
    sum1=0.0;
    for(i=0;i<NO_INPUTS;i++)
    { /* over all inputs */
        sum1=sum1+(ih_weights[i][h]*x[i]);
        printf("ih_weight[%d][%d]=%lf\n",i,h,ih_weights[i][h]);
    }
    sum1=sum1+vbias_weights[h];
    hidden[h]=compute_sigmoid(sum1);
}

sum2=0;
for(h=0;h<NO_HIDDEN;h++)
{ /* to calculate output; notice this output is not thresholded
   /* with any sigmoid unit */
    sum2=sum2+(ho_weights[h]*hidden[h]);
    printf("ho_weight[%d]=%lf\n",h,ho_weights[h]);
    printf("vbias_weight[%d]=%lf\n",h,vbias_weights[h]);
}
sum2=sum2+bias_weight;
printf("The bias weight is %lf\n",bias_weight);
printf("The output value is %lf\n",sum2);

return;
}

*****init_net.c*****
```

```
void init_net(double ih_weights[NO_INPUTS][NO_HIDDEN],double
ho_weights[NO_HIDDEN],double* bias_weight,double vbias_weights[NO_HIDDEN])
{
/* initialize weights in the neural net */

int i,h;

for(h=0;h<NO_HIDDEN;h++)
{
    ho_weights[h]=unifrnd(0.0,0.1,&SEED1);
    vbias_weights[h]=unifrnd(0.0,0.1,&SEED2);

    for(i=0;i<NO_INPUTS;i++)
    {
        ih_weights[i][h]=unifrnd(0.0,0.1,&SEED3);
    }
}

/* obo(output bias to output layer weight) */
*bias_weight=unifrnd(0.0,0.1,&SEED4);
```

```
return;
}

*****input.dat*****
```

0.000783
0.153779
3.000602
0.560532
0.865013
5.738535
0.276724
0.895919
4.316186
0.704462
0.886472
6.618693
0.929641
0.469290
6.045585
0.350208
0.941637
4.771489
0.096535
0.457211
3.230002
0.346164
0.970019
4.798756
0.114938
0.769819
3.455619
0.341565
0.684224
4.285201

```
*****reader.c *****
```

```
void reader(double input_values[DATA_SIZE][NO_INPUTS],double target_values[DATA_SIZE])
{/* Reads input and target values from a file; */
/* each line has a different value. */
/* The first few lines contain the input values and then the next line */
/* contains the target value and then the sequence repeats */

FILE *fid;
int p,i;
fid=fopen("input.dat","r");
for(p=0;p<DATA_SIZE;p++)
{
```

```
        for(i=0;i<NO_INPUTS;i++)
        {
            fscanf(fid,"%lf \n",&input_values[p][i]);
        }
        fscanf(fid,"%lf \n",&target_values[p]);
    }
    fclose(fid);

    return;
}

/*************simu_net.c*****/

void simu_net(double input_values[DATA_SIZE][NO_INPUTS],double
ih_weights[NO_INPUTS][NO_HIDDEN],double ho_weights[NO_HIDDEN],double
bias_weight,double vbias_weights[NO_HIDDEN],double
hidden_values[DATA_SIZE][NO_HIDDEN],double output_values[DATA_SIZE])
{
    /* evaluates the values in the net; */
    /* sigmoid thresholding is done only from input to hidden */
    /* layer but not from hidden to output layer */

    double sum1,sum2;
    int h,i,p;

    for(p=0;p<DATA_SIZE;p++)
    {/* for each data point */
        for(h=0;h<NO_HIDDEN;h++)
        {/* calculate the sum of all inputs to each hidden node */
            sum1=0.0;
            for(i=0;i<NO_INPUTS;i++)
            {/* over all inputs */
                sum1=sum1+(ih_weights[i][h])*(input_values[p][i]);
            }
            sum1=sum1+vbias_weights[h];
            hidden_values[p][h]=compute_sigmoid(sum1);
        }
    }

    for(p=0;p<DATA_SIZE;p++)
    {
        sum2=0;
        for(h=0;h<NO_HIDDEN;h++)
        {/* to calculate output; notice this output is not thresholded
           /* with any sigmoid unit */
            sum2=sum2+(ho_weights[h]*hidden_values[p][h]);
        }
        sum2=sum2+bias_weight;
    }
}
```

```

    output_values[p]=sum2;
}

return;
}

```

8. Reinforcement Learning Codes

In this section, we shall present C codes for reinforcement learning and neural networks. The examples used will be two-state MDPs and SMDPs, like in the section on dynamic programming.

Section 8.1: *Q*-Learning for discounted reward MDPs.

Section 8.2: Relative *Q*-Learning for average reward MDPs.

Section 8.3: Relaxed-SMART for average reward SMDPs.

An important note in regards the RL codes is as follows. The simulators used in these codes are simulators for the Markov process (Example A) and the semi-Markov process (Example B). These simulators assume the knowledge of the transition probabilities, rewards, and times. When this information is available, one does not need RL. However, the idea here is to show some basic features of RL in a simple Markov chain. Also with codes such as these, one can *test* the performance of an RL algorithm on an MDP / SMDP whose optimal solution is known. This kind of testing is often done with new algorithms. In practice, RL is used on simulators of systems whose TPMs etc are hard to find. We shall show the use of RL on the problem of preventive maintenance later in this chapter. In that problem, the transition probabilities are unknown, and RL will be integrated within a complex simulator.

8.1. Codes for *Q*-Learning

In this subsection, you will find codes for *Q*-Learning on Example A (of Section 5.1). The simulator.c file, which contains the simulator, will have to be suitably modified to take care of other problems. The simulator used here simulates a Markov chain.

```

#include <stdio.h>
#include <math.h>
#include "global.h"
#include "typedefs.h"
#include "fun_protos.h"
#include "fun_list.h"
main()
{
    simulator_mc();

    return;
}

```

```
}
```

```
*****global.h *****
```

```
long SEED=1; /* Use any positive integer */
int ITERMAY=10000; /* No of iterations of learning */
#define NA 2 /* Number of actions in each state */
#define NS 2 /* Number of states */

double SMALL=-1000000; /* a very small number */
double D_FACTOR=0.8; /* Discounting factor */
double TPM[NA][NS][NS]={
{
    {0.7,0.3},
    {0.4,0.6}
},
{
    {0.9,0.1},
    {0.2,0.8}
}
};

double TRM[NA][NS][NS]={
{
    {6,-5},
    {7,12}
},
{
    {10,17},
    {-14,13}
}
};
```

```
*****fun_prots.h *****
```

```
double uniformrnd(long* );
double unifrnd(double, double, long* );
void simulator_mc(void);
void initialize(seed_data*,statistics*);
int jump_learn(seed_data*,statistics*);
int state_finder(seed_data*,statistics*);
void qlearn(statistics*);
int action_selector(seed_data*);
void pol_finder(statistics,int [NS]);
```

```
*****fun_list.h *****
```

```
# include "unifrnd.c"
# include "jump_learn.c"
# include "simulator_mc.c"
```

```
# include "initialize.c"
# include "state_finder.c"
# include "qlearn.c"
# include "action_selector.c"
# include "pol_finder.c"

/*********************typedefs.h*****************/
typedef struct {
long seed1;
long seed2;
}seed_data;

typedef struct {
double Q[NS][NA];
int iter;
int old_action;
int old_state;
int current_state;
double rimm;
}statistics;

/******************action_selector.c *****/
int action_selector(seed_data* sd)
{
long seed;
double ran,na,sum;
int action,complete,candidate;

seed=sd->seed2;

ran=unifrnd(0,1,&seed);

/* The changed seed has to be saved in the seed data */

sd->seed2=seed;

candidate=0;
na=(double)(NA);
sum=1/na;

complete=0;
/* Selecting each action with equal probability */
while(0==complete)
{
    if(ran<sum)
    /* action selected */
    action=candidate;
    complete=1;
}
```

```
    else
        {/* test if ran is associated with next action */
        candidate=candidate+1;
        sum=sum+(1/na);
    }
}

return(action);
}

***** initialize.c *****

void initialize(seed_data* sd,statistics* stat)
{
/* initialize relevant statistics*/
int state,action;

/* System starts in the following condition */

stat->iter=0;
stat->old_state=0;
stat->old_action=0;

for(state=0;state<=NS-1;state++)
{
    for(action=0;action<=NA-1;action++)
    {
        stat->Q[state][action]=0;
    }
}

/* Initialize seed values */
sd->seed1=SEED+1; /* any seed has to be at least 1*/
sd->seed2=sd->seed1+1;

return;
}

*****jump_learn.c *****

int jump_learn(seed_data* sd,statistics* stat)
{
/* This function simulates a jump and also updates the learning */
```

```
/* statistics. */

int current_state,next_action,old_state,old_action;

/* Extract data related to the old state */

old_state=stat->old_state;
old_action=stat->old_action;

/* Determine current state */

current_state=state_finder(sd,stat);

/* Record Feedback in stat */

stat->current_state=current_state;
stat->rimm=TRM[old_action][old_state][current_state];

/* DO LEARNING */

qlearn(stat);

/* Select next action */

next_action=action_selector(sd);

/* Get ready to get out of this function */

stat->old_state=current_state;
stat->old_action=next_action;

if(stat->iter>=ITERMAX)
{
/* Learning should end */
return(1);
}
else
{
return(0);
}

}

*****pol_finder.c *****

void pol_finder(statistics stat,int policy[NS])
{
int action,state;
double max;
```

```
for(state=0;state<=NS-1;state++)
{/* find policy learned */
max=SMALL;
for(action=0;action<=NA-1;action++)
{
if(stat.Q[state][action]>max)
{
    max=stat.Q[state][action];
    policy[state]=action;
}
}
}

for(state=0;state<=NS-1;state++)
{
printf("Action learned for state %d:%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;state++)
{
for(action=0;action<=NA-1;action++)
{
printf("For state=%d\n",state);
printf("For action=%d\n",action);
printf("Q-factor=%lf\n",stat.Q[state][action]);
}
}
return;
}
*****qlearn.c*****
```

```
void qlearn(statistics* stat)
{
/* Q-Learning */

double q_next,q,learn_rate;
int action;

/* Finding the Max factor in the current state */

q_next=SMALL;

for(action=0;action<=NA-1;action++)
{
if(stat->Q[stat->old_state][action]>q_next)
{
q_next=stat->Q[stat->old_state][action];
}
}
```

```
q=stat->Q[stat->old_state][stat->old_action];

stat->iter=stat->iter+1;

learn_rate=0.1/stat->iter;

q=q*(1-learn_rate)+(learn_rate*(stat->rimm+(D_FACTOR*q_next)));

stat->Q[stat->old_state][stat->old_action]=q;

return;

}

/* simulator.c *****

void simulator_mc(void)
{
/* Simulator for a Markov Chain*/

int policy[NS],done;
seed_data sd;
statistics stat;

/* initialize system */

initialize(&sd,&stat);

done=0; /* Pnemonic for simulation, 1 stands for end*/
/* 0 stands for continue*/

while(0==done)
{
    done=jump_learn(&sd,&stat);
}

/* Done with Learning */
/* Find learned policy */

pol_finder(stat,policy);

return;
}
```

```
*****state_finder.c*****  
  
int state_finder(seed_data* sd,statistics* stat)  
{  
double ran,sum;  
int candidate,old_action,old_state,complete;  
long seed;  
  
seed=sd->seed1;  
  
ran=unifrnd(0,1,&seed);  
  
/* The changed seed has to be saved in the seed data */  
/* for reuse */  
  
sd->seed1=seed;  
  
/* Find current state */  
  
old_action=stat->old_action;  
old_state=stat->old_state;  
  
  
sum=TPM[old_action][old_state][0];  
  
candidate=0;  
  
complete=0;  
  
while(0==complete)  
{  
    if(ran<sum)  
    {  
        complete=1;  
    }  
    else  
    {  
        candidate=candidate+1;  
        sum=sum+TPM[old_action][old_state][candidate];  
    }  
}  
  
return(candidate);  
}
```

8.2. Codes for Relative *Q*-Learning

In this subsection, you will find codes for Relative *Q*-Learning on Example A (of Section 5.1).

```
*****main.c ****
#include <stdio.h>
#include <math.h>
#include "global.h"
#include "typedefs.h"
#include "fun_prots.h"
#include "fun_list.h"
main()
{
    simulator_mc();

    return;
}
*****global.h *****
long SEED=1; /* Use any positive integer */
int NO_REPLICATIONS=30; /* No of replications of simulation */
int ITERMAX=10000; /* No of iterations of learning */
int ITERFIXED=5000; /* No of iterations of simulation with a policy*/
#define NA 2 /* Number of actions in each state */
#define NS 2 /* Number of states */

double SMALL=-1000000; /* a very small number */
double TPM[NA][NS][NS]={
    {
        {0.7,0.3},
        {0.4,0.6}
    },
    {
        {0.9,0.1},
        {0.2,0.8}
    }
};
double TRM[NA][NS][NS]={
    {
        {6,-5},
        {7,12}
    },
    {
        {10,17},
        {-14,13}
    }
};
```

```
*****fun_prots.h*****  
  
double uniformrnd(long* );  
double unifrnd(double, double, long* );  
void simulator_mc(void);  
void initialize(seed_data*,statistics*);  
int jump_learn(seed_data*,statistics*);  
int jump_fixed(seed_data*,statistics*,int [NS]);  
int state_finder(seed_data*,statistics*);  
void rqlearn(statistics*);  
int action_selector(seed_data*);  
void pol_finder(statistics,int [NS]);  
void rho_finder(seed_data*,statistics*,int [NS]);  
*****fun_list.h*****  
  
# include "unifrnd.c"  
# include "jump_learn.c"  
# include "jump_fixed.c"  
# include "simulator_mc.c"  
# include "initialize.c"  
# include "state_finder.c"  
# include "rqlearn.c"  
# include "action_selector.c"  
# include "pol_finder.c"  
# include "rho_finder.c"  
  
*****typedefs.h*****  
  
typedef struct {  
long seed1;  
long seed2;  
}seed_data;  
  
typedef struct {  
double Q[NS][NA];  
int iter;  
int old_action;  
int old_state;  
int current_state;  
double rimm;  
double total_reward;  
}statistics;  
  
*****action_selector.c *****  
  
int action_selector(seed_data* sd)  
{  
long seed;
```

```
double ran,na,sum;
int action,complete,candidate;

seed=sd->seed2;

ran=unifrnd(0,1,&seed);

/* The changed seed has to be saved in the seed data */

sd->seed2=seed;

candidate=0;
na=(double)(NA);
sum=1/na;

complete=0;
/* Selecting each action with equal probability */
while(0==complete)
{
    if(ran<sum)
        {/* action selected */
        action=candidate;
        complete=1;
    }
    else
        {/* test if ran is associated with next action */
        candidate=candidate+1;
        sum=sum+(1/na);
    }
}

return(action);
}

*****initialize.c*****  
  
void initialize(seed_data* sd,statistics* stat)
{
/* initialize relevant statistics*/
int state,action;

/* System starts in the following condition */

stat->iter=0;
stat->old_state=0;
stat->old_action=0;

for(state=0;state<=NS-1;state++)
```

```
/* initialize all Q-factors to 0 */
for(action=0;action<=NA-1;action++)
{
stat->Q[state][action]=0;
}
}

/* Initialize seed values */
sd->seed1=SEED+1; /* any seed has to be at least 1*/
sd->seed2=sd->seed1+1;

return;
}
*****jump_fixed.c*****
```

```
int jump_fixed(seed_data* sd,statistics* stat,int policy[NS])
{
/* This function simulates a jump and also updates the learning */
/* statistics. */

int current_state,next_action,old_state,old_action;

/* Extract data related to the old state */

old_state=stat->old_state;
old_action=stat->old_action;

/* Determine current state */

current_state=state_finder(sd,stat);

/* Record Feedback in stat */

stat->current_state=current_state;
stat->rimm=TRM[old_action][old_state][current_state];
stat->iter=stat->iter+1;
stat->total_reward=stat->total_reward+stat->rimm;

next_action=policy[current_state];

/* Get ready to get out of this function */

stat->old_state=current_state;
stat->old_action=next_action;
```

```
if(stat->iter>=ITERFIXED)
{
/* simulation should end */
return(1);
}
else
{
return(0);
}

}

/*****jump_learn.c*****/

int jump_learn(seed_data* sd,statistics* stat)
{
/* This function simulates a jump and also updates the learning */
/* statistics. */

int current_state,next_action,old_state,old_action;

/* Extract data related to the old state */

old_state=stat->old_state;
old_action=stat->old_action;

/* Determine current state */

current_state=state_finder(sd,stat);

/* Record Feedback in stat */

stat->current_state=current_state;
stat->rimm=TRM[old_action][old_state][current_state];

/* DO LEARNING */

rqlearn(stat);

/* Select next action */

next_action=action_selector(sd);

/* Get ready to get out of this function */

stat->old_state=current_state;
```

```
stat->old_action=next_action;

if(stat->iter>=ITERMAX)
{
/* Learning should end */
return(1);
}
else
{
return(0);
}

}

/*****pol_finder.c*****/

void pol_finder(statistics stat,int policy[NS])
{
int action,state;
double max,rho;

for(state=0;state<=NS-1;state++)
{/* find policy learned */
max=SMALL;
for(action=0;action<=NA-1;action++)
{
if(stat.Q[state][action]>max)
{
max=stat.Q[state][action];
policy[state]=action;
}
}
}

for(state=0;state<=NS-1;state++)
{
printf("Action learned for state %d:%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;state++)
{
for(action=0;action<=NA-1;action++)
{
printf("For state=%d\n",state);
printf("For action=%d\n",action);
printf("Q-factor=%lf\n",stat.Q[state][action]);
}
}
```

```
return;
}

/****************rho_finder.c*****/

void rho_finder(seed_data* sd,statistics* stat,int policy[NS])
{

int done,replicate;
double rho,sum_rho=0,k=0;

for(replicate=1;replicate<=NO_REPLICATIONS;replicate++)
{

/* initialize seeds; making sure every replicate starts with a */
/* different seed that is 100 apart*/

sd->seed1=(long)(replicate*100)*SEED;
sd->seed2=(long)(replicate*100)*SEED+1;

stat->iter=0;
stat->old_state=0;
stat->old_action=0;
stat->total_reward=0;

/* initialize statistics */

done=0; /* Pnemonic for simulation, 1 stands for end*/
          /* 0 stands for continue*/

          while(0==done)
          {
            done=jump_fixed(sd,stat,policy);
          }

rho=stat->total_reward/stat->iter;

sum_rho=sum_rho+rho;

k=k+1.0;
}
printf("The average reward of learned policy is %lf\n",sum_rho/k);

return;
```

```
}

/*****rqlearn.c*****/

void rqlearn(statistics* stat)
{
/* Relative Q-Learning */

double q_next,q,learn_rate;
int action;

/* Finding the Max factor in the current state */

q_next=SMALL;

for(action=0;action<=NA-1;action++)
{
if(stat->Q[stat->current_state][action]>q_next)
{
q_next=stat->Q[stat->current_state][action];
}
}

q=stat->Q[stat->old_state][stat->old_action];

stat->iter=stat->iter+1;

learn_rate=1.0/stat->iter;

/* Q(0,0) is the distinguished factor */

q=q*(1-learn_rate)+(learn_rate*(stat->rimm+q_next-stat->Q[0][0]));

stat->Q[stat->old_state][stat->old_action]=q;

return;
}

/*****simulator.c*****/

void simulator_mc(void)
{
/* Simulator for a Markov Chain*/

int policy[NS],done;
seed_data sd;
statistics stat;

/* initialize system */
```

```
initialize(&sd,&stat);

done=0; /* Pnemonic for simulation, 1 stands for end*/
/* 0 stands for continue*/

while(0==done)
{
    done=jump_learn(&sd,&stat);
}

/* End of learning */
/* To determine policy learned */

    pol_finder(stat,policy);

/* To find the average reward associated with policy */

    rho_finder(&sd,&stat,policy);

return;
}

*****state_finder.c*****
```



```
int state_finder(seed_data* sd,statistics* stat)
{
double ran,sum;
int candidate,old_action,old_state,complete;
long seed;

seed=sd->seed1;

ran=unifrnd(0,1,&seed);

/* The changed seed has to be saved in the seed data */
/* for reuse */

sd->seed1=seed;

/* Find current state */

old_action=stat->old_action;
old_state=stat->old_state;

sum=TPM[old_action][old_state][0];

candidate=0;
```

```
complete=0;

while(0==complete)
{
    if(ran<sum)
    {
        complete=1;
    }
    else
    {
        candidate=candidate+1;
        sum=sum+TPM[old_action][old_state][candidate];
    }
}

return(candidate);
}
```

8.3. Codes for Relaxed-SMART

This subsection presents the codes for Relaxed-SMART on Example B (of Section 9.2.2).

```
/* main.c *****/
#include <stdio.h>
#include <math.h>
#include "global.h"
#include "typedefs.h"
#include "fun_protos.h"
#include "fun_list.h"
main()
{
    simulator_mc();

    return;
}
/*global.h*****/
long SEED=1; /* Use any positive integer */
int NO_REPLICATIONS=30; /* No of replications of simulation */
int ITERMAX=10000; /* No of iterations of learning */
int ITERFIXED=5000; /* No of iterations of simulation with a policy*/
double EXPLORE_PROB_INITIAL=0.1; /* Initial exploration prob */
#define NA 2 /* Number of actions in each state */
#define NS 2 /* Number of states */

double SMALL=-1000000; /* a very small number */
double TPM[NA][NS][NS]={
    {
```

```

        {0.7,0.3},
        {0.4,0.6}
    },
    {
        {0.9,0.1},
        {0.2,0.8}
    }
};

double TRM[NA][NS][NS]={
{
    {6,-5},
    {7,12}
},
{
    {10,17},
    {-14,13}
}
};

double TTM[NA][NS][NS]={
{
    {1,5},
    {120,60}
},
{
    {50,75},
    {7,2}
}
};

/*fun_prots.h******/
double uniformrnd(long* );
double unifrnd(double, double, long* );
void simulator_mc(void);
void initialize(seed_data*,statistics*);
int jump_learn(seed_data*,statistics*);
int jump_fixed(seed_data*,statistics*,int [NS]);
int state_finder(seed_data*,statistics*);
void rsmart(statistics*);
int action_selector(seed_data*,statistics*);
void pol_finder(statistics,int [NS]);
void rho_finder(seed_data*,statistics*,int [NS]);

/*fun_list.h******/

#include "uniformrnd.c"
#include "unifrnd.c"
#include "jump_learn.c"
#include "jump_fixed.c"

```

```
# include "simulator_mc.c"
# include "initialize.c"
# include "state_finder.c"
# include "rsmart.c"
# include "action_selector.c"
# include "pol_finder.c"
# include "rho_finder.c"

/**typedefs.h***** */

typedef struct {
long seed1;
long seed2;
}seed_data;

typedef struct {
double Q[NS][NA];
int iter;
int old_action;
int old_state;
int current_state;
double rimm;
double timm;
double theta;
double total_reward;
double total_time;
double explore_prob;
int flag;
int greedy_iter;
}statistics;

/* action_selector.c***** */

int action_selector(seed_data* sd, statistics* stat)
{
/* This program is written for two actions */

long seed;
double ran,na,sum;
int action,complete,state,act;

seed=sd->seed2;

ran=unifrnd(0,1,&seed);

/* The changed seed has to be saved in the seed data */

sd->seed2=seed;

/* State for which action is being selected */
```

```
state=stat->current_state;

/* Reduce explore prob */

stat->explore_prob=stat->explore_prob*0.9999;

if(stat->Q[state][0]>stat->Q[state][1])
{ /* greedy action is 0 */
if(ran<stat->explore_prob)
{ /* select non-greedy action */
    action=1;
    stat->flag=1;
}
else
{ /* greedy action */
action=0;
    stat->flag=0;
}
}
else
{ /* greedy action is 1 */
if(ran<stat->explore_prob)
{ /* select non-greedy action */
    action=0;
    stat->flag=1;
}
else
{ /* greedy action */
action=1;
    stat->flag=0;
}
}

return(action);
}

/*initialize.c***** */

void initialize(seed_data* sd,statistics* stat)
{
/* initialize relevant statistics*/
int state,action;

/* System starts in the following condition */
```

```
stat->iter=0;
stat->greedy_iter=0;
stat->flag=1;
stat->old_state=0;
stat->old_action=0;
stat->theta=200; /* initial value should be close to actual value */
stat->total_reward=0;
stat->total_time=0;
stat->explore_prob=EXPLORE_PROB_INITIAL;
for(state=0;state<=NS-1;state++)
{
    for(action=0;action<=NA-1;action++)
    {/* initialize all Q-factors to 0 */
        stat->Q[state][action]=0;
    }
}

/* Initialize seed values */
sd->seed1=SEED+1; /* any seed has to be at least 1*/
sd->seed2=sd->seed1+1;

return;
}

/*jump_fixed.c***** */

int jump_fixed(seed_data* sd,statistics* stat,int policy[NS])
{
/* This function simulates a jump and also updates the learning */
/* statistics. */

int current_state,next_action,old_state,old_action;

/* Extract data related to the old state */

old_state=stat->old_state;
old_action=stat->old_action;

/* Determine current state */

current_state=state_finder(sd,stat);

/* Record Feedback in stat */

stat->current_state=current_state;
```

```
stat->rimm=TRM[old_action][old_state][current_state];
stat->timm=TTM[old_action][old_state][current_state];
stat->iter=stat->iter+1;
stat->total_reward=stat->total_reward+stat->rimm;
stat->total_time=stat->total_time+stat->timm;

next_action=policy[current_state];

/* Get ready to get out of this function */

stat->old_state=current_state;
stat->old_action=next_action;

if(stat->iter>=ITERFIXED)
{
/* simulation should end */
return(1);
}
else
{
return(0);
}

}

/*jump_learn.c*****/


int jump_learn(seed_data* sd,statistics* stat)
{
/* This function simulates a jump and also updates the learning */
/* statistics. */

int current_state,next_action,old_state,old_action;
double beta;

/* Extract data related to the old state */

old_state=stat->old_state;
old_action=stat->old_action;

/* Determine current state */

current_state=state_finder(sd,stat);

/* Record Feedback in stat */
```

```
stat->current_state=current_state;
stat->rimm=TRM[old_action][old_state][current_state];
stat->timm=TTM[old_action][old_state][current_state];

if(0==stat->flag)
{ /* greedy action selected */
stat->greedy_iter=stat->greedy_iter+1;
stat->total_reward=stat->total_reward+stat->rimm;
stat->total_time=stat->total_time+stat->timm;

/* update theta */

beta=0.1/stat->greedy_iter;

stat->theta=(1-beta)*stat->theta
+(beta*(stat->total_reward/stat->total_time));

}

/* DO LEARNING of Q Values*/

rsmart(stat);

/* Select next action */

next_action=action_selector(sd,stat);

/* Get ready to get out of this function */

stat->old_state=current_state;
stat->old_action=next_action;

if(stat->iter>=ITERMAX)
{
/* Learning should end */
return(1);
}
else
{
return(0);
}

}

/*pol_finder.c******/
```

```
void pol_finder(statistics stat,int policy[NS])
{
int action,state;
double max,rho;

for(state=0;state<=NS-1;state++)
{/* find policy learned */
max=SMALL;
for(action=0;action<=NA-1;action++)
{
if(stat.Q[state][action]>max)
{
    max=stat.Q[state][action];
    policy[state]=action;
}
}
}

for(state=0;state<=NS-1;state++)
{
printf("Action learned for state %d:%d\n",state,policy[state]);
}
for(state=0;state<=NS-1;state++)
{
for(action=0;action<=NA-1;action++)
{
    printf("For state=%d\n",state);
    printf("For action=%d\n",action);
    printf("Q-factor=%lf\n",stat.Q[state][action]);
}
}

return;
}

/*rho_finder.c******/
void rho_finder(seed_data* sd,statistics* stat,int policy[NS])
{

int done,replicate;
double rho,sum_rho=0,k=0;

for(replicate=1;replicate<=NO_REPLICATIONS;replicate++)
{
```

```
/* initialize seeds; making sure every replicate starts with a */
/* different seed that is 100 apart*/

sd->seed1=(long)(replicate*100)*SEED;
sd->seed2=(long)(replicate*100)*SEED+1;

stat->iter=0;
stat->old_state=0;
stat->old_action=0;
stat->total_reward=0;
stat->total_time=0;

/* initialize statistics */

done=0; /* Pnemonic for simulation, 1 stands for end*/
        /* 0 stands for continue*/

        while(0==done)
{
    done=jump_fixed(sd,stat,policy);
}

rho=stat->total_reward/stat->total_time;

sum_rho=sum_rho+rho;

k=k+1.0;
}
printf("The average reward of learned policy is %lf\n",sum_rho/k);

return;
}

/*rsmart.c******/
void rsmart(statistics* stat)
{
/* Relaxed SMART */

double q_next,q,learn_rate;
int action;

/* Finding the Max factor in the current state */

q_next=SMALL;

for(action=0;action<=NA-1;action++)
{
```

```
if(stat->Q[stat->current_state][action]>q_next)
{
q_next=stat->Q[stat->current_state][action];
}
}

q=stat->Q[stat->old_state][stat->old_action];

stat->iter=stat->iter+1;

learn_rate=0.5/stat->iter;

/* Q(0,0) is the distinguished factor */

q=q*(1-learn_rate)+(learn_rate*(stat->rimm-(stat->timm*stat->theta)
+q_next));

stat->Q[stat->old_state][stat->old_action]=q;

return;

}
/*simulator_mc.c******/



void simulator_mc(void)
{
/* Simulator for a Markov Chain*/



int policy[NS],done;
seed_data sd;
statistics stat;

/* initialize system */

initialize(&sd,&stat);

done=0; /* Pnemonic for simulation, 1 stands for end*/
/* 0 stands for continue*/

while(0==done)
{
done=jump_learn(&sd,&stat);
}

/* Done with Learning */
/* Find the learned policy */

pol_finder(stat,policy);

/* To find the average reward associated with policy */
```

```
        rho_finder(&sd,&stat,policy);

return;
}

/*state_finder.c***** */

int state_finder(seed_data* sd,statistics* stat)
{
double ran,sum;
int candidate,old_action,old_state,complete;
long seed;

seed=sd->seed1;

ran=unifrnd(0,1,&seed);

/* The changed seed has to be saved in the seed data */
/* for reuse */

sd->seed1=seed;

/* Find current state */

old_action=stat->old_action;
old_state=stat->old_state;

sum=TPM[old_action][old_state][0];

candidate=0;

complete=0;

while(0==complete)
{
    if(ran<sum)
    {
        complete=1;
    }
    else
    {
        candidate=candidate+1;
        sum=sum+TPM[old_action][old_state][candidate];
    }
}
```

```
return(candidate);
}
```

9. Codes for the Preventive Maintenance Case Study

In this section, we present the codes used in reinforcement learning to solve the preventive maintenance problem discussed in Chapter 14. The codes simulate the production-inventory system and incorporate the learning function (*update.c*) within the simulator. (For the random number functions, please see Section 4.)

Simulating stochastic systems in C is quite different from writing simulation programs in simulation software. However, as mentioned earlier in this book, simulation-optimization functions can be more easily integrated within simulators written in C than in simulators written in simulation packages. Furthermore, simulators in C run faster, which may be a requirement for simulation optimization. We encourage you to study the C language simulator of the single server, single channel queuing system in Law and Kelton [102], or the same in any other book that deals with this topic — before reading this section. It is important to comprehend the logic used in these programs. We, next, present a quick overview of the logic used in the production-inventory simulator.

Five *types* of events can occur in the simulator. They are: simulation termination, arrival of a demand, completion of a production, completion of a repair, and completion of a maintenance. You will see below that for each of the last four events, we have allocated a function. The function is called in *main.c* *immediately after* the related event takes place. Within each function, one *schedules*, that is, decides the time for future events and stores the time in the *event_clock* (simulation clock — see Chapter 4). For instance, as soon as an arrival occurs, we know from the random numbers for the inter-arrival time, when the next arrival will occur. Thus events of different type are scheduled. After every event, the timer function (in *timer.c*) is called. This function finds the time of the next event. The simulation clock is then advanced to the time of the next event and the next event is simulated (that is the respective function is called). This sequence repeats till the termination event is called, and then the simulation ends.

The codes of this section are presented in two parts. The first subsection contains the codes that are used in *learning* the optimal policy. Now, the learning codes in subsection 9.1 learn the optimal policy; the average reward of the learned policy can then be estimated by re-simulating the system using the learned policy. The codes for the latter are naturally different than those in subsection 9.1 because they do not involve any learning. The codes that used the fixed (learned) policy are given in subsection 9.2.

The learning codes produce a file called *report.dat*, which contains values for all the *Q*-factors learned. From the *Q*-factors, one can read the optimal policy.

Some related input parameters (distributions of governing random variables etc) are placed in global.h. Naturally, these have to be changed with every case example. The parameters are described below. We use $(S, s) = (3, 2)$. For a gamma distribution characterized by (n, λ) , the mean is n/λ . For the exponential distribution characterized by μ , the mean is $1/\mu$. For the uniform distribution characterized by (a, b) , the mean is given by $(a + b)/2$.

Time between arrivals expo. distr.	Time between failures gamma distr.	Time for a repair gamma distr.	Time for a maintenance uniform distr.
μ 1/15	(n, λ) (8, 0.01)	(n, λ) (4, 0.02)	(a, b) (5, 25)

Time for a production: gamma distr (n, λ)
(8, 0.08)

The optimal average reward for this case example (from Das and Sarkar [39]) is 0.57. The reinforcement learning codes use the SMART algorithm (see Das *et al.* [37]) and produce an average reward of 0.56. The value of 0.56 is of course obtained from the fixed policy codes in subsection 9.2. The problem has more than a million states. It can be solved using a look-up table approach.

We must point out that the codes that follow generate a near-optimal policy to a problem without generating the transition probabilities of the underlying semi-Markov process. Generating the transition probability model is extremely tedious (see [39]) in comparison to the coding effort needed here. Furthermore, the transition probability model in [39] depends on some pre-specified distributions, while the distributions in the simulator can be changed very easily.

9.1. Learning Codes

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include "global.h"
# include "typedefs.h"
# include "fun_protos.h"
# include "fun_list.h"

main()
{
/* A simulator for a production-inventory system that learns on its own */

int next_event,done,count1,count2,count3;
double event_clock[5];
system_data syst;
```

```
alg_data alg;
seed_data sd;

/* Initialize variables*/

initialize(event_clock,&syst,&alg,&sd);

done = 0; /* Pnemonic for simulation, 1 stands for end */
           /* 0 stands for continue */

while(0 == done)
{
/* Find out which event should take place next */

next_event=timer(event_clock);

/* Schedule the next event */

switch(next_event)
{
case 0:
done = 1;
break;

case 1:
arrive(event_clock,&syst,&alg,&sd);
break;

case 2:
produce(event_clock,&syst,&alg,&sd);
break;

case 3:
repair(event_clock,&syst,&alg,&sd);
break;

case 4:
maintain(event_clock,&syst,&alg,&sd);
break;
}

/* End of switch statements */

}

/* End of while loop */
reporter(alg);
printf("avg reward %lf\n",alg.total_revenue/TNOW);
printf("exp rate at end %lf\n",alg.exp_rate);
```

```
printf("iterations at end %d\n",alg.iter);

return;
}
*****global.h*****/

/* types of events 0=termination; 1=arrive; 2=produce;3=repair 4=maint */

int NUMBER_OF_EVENTS=5;

double LENGTH=1000000;

double BIG =1000001; /* has to be bigger than LENGTH */

#define PCMAX 100 /* max value for production count */
#define UPPER_LIMIT 3 /* S value for buffer */
#define LOWER_LIMIT 2 /* s value for buffer */

/* the following are the parameters of gamma distribution */
/* for PROD=production time, FAIL=time between failures, */
/* and REPAIR= repair time */
int N_PROD=8;
double LAMBDA_PROD=0.8;

int N_FAIL=8;
double LAMBDA_FAIL=0.01;

int N_REPAIR=4;
double LAMBDA_REPAIR=0.02;

double MU=0.06666666; /* exponential rate of arrival of demands */

/* the following parameters are for the unif dist. for maintenance */

double MAINT_MIN=5;
double MAINT_MAX=20;

double TNOW;

double PROFIT=1;

double MAINT_COST=2;

double REPAIR_COST=5;

*****typedefs.h*****/
```

```
typedef struct
{
double time_for_failure; /* time remaining for failure */
double prod_time; /* time of production */
int buffer; /* current buffer */
int prodcnt; /* current production count */
}system_data;

typedef struct
{
double revenue; /* cumulative revenue since last update */
double time; /* time of last update */
double last_buffer;
double last_prodcnt;
double last_action;
double alpha; /* learning rate for Q-factors */
double rho; /* average reward per unit time */
double Q[UPPER_LIMIT+1][PCMAX+1][2]; /* Q-factors */
int iter; /* number of updates */
double total_revenue;
double total_time;
double exp_rate; /* rate of exploration */
int flag; /* for exploration */
}alg_data;

typedef struct
{
long seed_event;
long seed_prod;
long seed_maint;
long seed_fail;
long seed_rep;
long seed_arrive;
}seed_data;

/**********fun_prots.h*****/

double uniformrnd(long *);
double unifrnd(double, double, long *);
double expornd(double, long *);
double gamrnd(int,double, long *);
int timer(double []);
void arrive(double [],system_data *,alg_data *,seed_data *);
void produce(double [],system_data *,alg_data *,seed_data *);
void repair(double [],system_data *,alg_data *,seed_data *);
void maintain(double [],system_data *,alg_data *,seed_data *);
void update(system_data *,alg_data *);
```

```
void initialize(double [],system_data*,alg_data*,seed_data*);  
double max(double,double);  
void reporter(alg_data*);  
int action_selector(system_data*,alg_data*,seed_data*);  
  
/*****fun_list.h*****/  
  
# include "uniformrnd.c"  
# include "unifrnd.c"  
# include "expornd.c"  
# include "gamrnd.c"  
# include "timer.c"  
# include "arrive.c"  
# include "produce.c"  
# include "repair.c"  
# include "maintain.c"  
# include "update.c"  
# include "initialize.c"  
# include "max.c"  
# include "reporter.c"  
# include "action_selector.c"  
  
/*****action_selector.c*****/  
  
int action_selector(system_data* syst,alg_data* alg, seed_data* sd)  
{  
    int b,c,action;  
    double ran_no;  
  
    b=syst->buffer;  
    c=syst->prodcount;  
  
    ran_no=unifrnd(0,1,&sd->seed_event);  
  
    if(ran_no<alg->exp_rate)  
    {  
        /* must do exploration */  
        alg->flag=1;  
    }  
    else  
    { /* select greedy action */  
        alg->flag=0;  
    }  
  
    if(alg->Q[b][c][0]>=alg->Q[b][c][1])  
    {  
        action=0;  
    }
```

```
else
{
action=1;
}

if(i==alg->flag)
{
/* reverse action */

    if(0==action)
    {
    action=1;
    }
    else
    {
    action=0;
    }
}

return(action);
}

/*****arrive.c*****/

void arrive(double event_clock [],system_data* syst,alg_data* alg,
seed_data* sd)
{ /* This function is called immediately after an arrival */

    double inter_arrival_time,prod_time,maint_time,repair_time;
    int action;

    inter_arrival_time=expornd(MU,&sd->seed_arrive);

    /* schedule next arrival */

    event_clock[1]=TNOW+inter_arrival_time;

    if(syst->buffer>0)
    {
    /* reduce the buffer size by 1 if buffer is not empty */
    /* if unit is sold, increase revenue */

        syst->buffer=syst->buffer-1;
        alg->revenue=alg->revenue+PROFIT;
    }

    /* test if the current arrival ends vacation */
}
```

```
if((UPPER_LIMIT-1)==syst->buffer)
{ /* schedule production or maintenance */
action=action_selector(syst,alg,sd);
if(0==action)
{
/* start a production cycle */

prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);

if(prod_time<syst->time_for_failure)
{ /* production will occur successfully */
event_clock[2]=TNOW+prod_time;
syst->prod_time=prod_time;
}
else
{ /* schedule a repair */
repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
event_clock[3]=TNOW+prod_time+repair_time;
}

}
else
{ /* schedule a maintenance */
maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
event_clock[4]=TNOW+maint_time;
}

}

return;
}

*****initialize.c*****
```

```
void initialize(double event_clock[],system_data* syst, alg_data
*alg,seed_data* sd)
{ /* Initialize the seeds for random number generation */

    int count1,count2,count3;

    sd->seed_event = 10;
    sd->seed_prod = 300;
    sd->seed_arrive= 500;
    sd->seed_rep = 145;
    sd->seed_maint = 600;
    sd->seed_fail = 345;

/* Initialize TNOW, the statistics, the event clock etc */
```

```
TNOW = 0;

/* Initialize the end of simulation event */
event_clock[0]=LENGTH;
/* Initialize the first arrival */
event_clock[1]=expornd(MU,&sd->seed_arrive);
/* Initialize the first production */
event_clock[2]=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);
syst->prod_time=event_clock[2];
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);
syst->prodcount=0;
syst->buffer=0;
event_clock[3]=BIG;
event_clock[4]=BIG;
alg->last_action=0; /* production action */
alg->last_buffer=0;
alg->last_prodcount=0;

alg->revenue=0;
alg->time=TNOW;
alg->total_revenue=0;
alg->total_time=0;

alg->rho=0;
alg->alpha=0.01;
alg->iter=0;
alg->exp_rate=0.02;
for(count1=0;count1<=UPPER_LIMIT;count1=count1+1)
{
    for(count2=0;count2<=PCMAX;count2=count2+1)
    {
        for(count3=0;count3<=1;count3=count3+1)
        {/* initialize the Q-factors to 0 */
        alg->Q[count1][count2][count3]=0;
        }
    }
}
return;

}

*****maintain.c*****
```

```
void maintain(double event_clock[],system_data* syst,alg_data* alg,
seed_data* sd)
{
/* a maintenance has just been finished */

double repair_time,maint_time,prod_time;
int action;
```

```
alg->last_action=1;

syst->prodcount=0;
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);
alg->revenue=alg->revenue-MAINT_COST;

event_clock[4]=BIG;
/* since we have reached a new state, it is time to update */

update(syst,alg);

/* We now store information about the current state in a box. */
/* We will open this box in our next meeting with update. */

alg->last_buffer=syst->buffer;
alg->last_prodcount=syst->prodcount;

/* Now to schedule the next event */
action=action_selector(syst,alg,sd);

if(0==action)
{
    /* start a production cycle, if the buffer has not maxed*/
    if(UPPER_LIMIT!=syst->buffer)
    {
        prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);
        if(prod_time<syst->time_for_failure)
        {/* production will occur successfully */
            event_clock[2]=TNOW+prod_time;
        }
        else
        {/* schedule a repair */
            repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
            event_clock[3]=TNOW+prod_time+repair_time;
        }
    }
    else
    {
        /* schedule a maintenance */
        maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
        event_clock[4]=TNOW+maint_time;
    }
}
```

```
return;
}

/**********max.c*****/

double max(double a,double b)
{
    if (a>=b)
    {
        return (a);
    }
    else
    {
        return (b);
    }
}

/**********produce.c*****/

void produce(double event_clock[],system_data* syst, alg_data*
alg,seed_data* sd)
{

/* A successful production has just been completed */

double repair_time,maint_time,prod_time,ran_no;
int action;

alg->last_action=0;

syst->prodcount=syst->prodcount+1;
syst->buffer=syst->buffer+1;

/* the following is reduced by the age */
syst->time_for_failure=syst->time_for_failure-syst->prod_time;

/* since we have reached a new state, it is time to update */

update(syst,alg);

/* We now store information about the current state in a box. */
/* We will open this box in our next meeting with update. */

alg->last_buffer=syst->buffer;
alg->last_prodcount=syst->prodcount;
```

```
/* Now to schedule the next event */

action=action_selector(syst,alg,sd);

if(0==action)
{
/* start a production cycle, if the buffer has not maxed*/

    if(UPPER_LIMIT==syst->buffer)
    {/* time to take a vacation */
        event_clock[2]=BIG;
    }
    else
    {
        prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);

        if(prod_time<syst->time_for_failure)
        {/* production will occur successfully */
            event_clock[2]=TNOW+prod_time;
            syst->prod_time=prod_time;
        }
        else
        {/* schedule a repair */
            repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
            event_clock[2]=BIG;
            event_clock[3]=TNOW+prod_time+repair_time;
        }
    }
}
else
{
/* schedule a maintenance */
maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
event_clock[2]=BIG;
event_clock[4]=TNOW+maint_time;
}

return;
}

*****repair.c*****
```

```
void repair(double event_clock[],system_data *syst, alg_data*
alg, seed_data *sd)
{
/* repair has just been finished */
```

```
double repair_time,maint_time,prod_time;
int action;

alg->last_action=0;

syst->prodcount=0;
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);
alg->revenue=alg->revenue-REPAIR_COST;

event_clock[3]=BIG;

/* since we have reached a new state, it is time to update */

update(syst,alg);

/* We now store information about the current state in a box. */
/* We will open this box in our next meeting with update. */

alg->last_buffer=syst->buffer;
alg->last_prodcount=syst->prodcount;

/* Now to schedule the next event */

action=action_selector(syst,alg,sd);
if(0==action)
{
/* start a production cycle, if the buffer has not maxed*/

    if(UPPER_LIMIT!=syst->buffer)
    {
        prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);
        if(prod_time<syst->time_for_failure)
            {/* production will occur successfully */
            event_clock[2]=TNOW+prod_time;
            syst->prod_time=prod_time;
            }
        else
            {/* schedule a repair */

                repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
                event_clock[3]=TNOW+prod_time+repair_time;
            }
    }
}
else
/* schedule a maintenance */
```

```
maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
event_clock[4]=TNOW+maint_time;
}

return;
}

/*****reporter.c*****/

void reporter(alg_data alg)
{
int i,j,k;

FILE* report_file;
report_file=fopen("report.dat","a");

for(i=1;i<=UPPER_LIMIT;i=i+1)
{
    for(j=0;j<=PCMAX;j=j+1)
    {
        for(k=0;k<=i;k=k+1)
        {/* initialize the Q-factors to 0 */

fprintf(report_file,"Q[%d] [%d] [%d]=%lf\n",i,j,k,alg.Q[i][j][k]);
        }
    }
}

fprintf(report_file,"number of learning iterations=%d\n",alg.ite);
fclose(report_file);
return;
}

/*****timer.c*****/

int timer(double event_clock[])
{
double minimum=BIG;
int index,next_event;

/* Find out which event is next */

for(index=0;index<NUMBER_OF_EVENTS;index=index+1)
{
    if(event_clock[index]<=minimum)
    {
        minimum=event_clock[index];
        next_event=index;
    }
}
```

```
    }

}

/* advance simulation clock */

TNOW=minimum;

/* return the next event */
return(next_event);
}

*****update.c*****
```

```
void update(system_data *syst,alg_data *alg)
{
    double rimm,timm,rho1,rho2,rho,q,qnext,alpha,beta;
    int action,prodcount,buffer,next_prodcount,next_buffer;

    alg->iter=alg->iter+1;

    rimm=alg->revenue;
    timm=TNOW-alg->time;
    alpha=alg->alpha;

    if(alg->flag==0)
    {
        /* no exploration; so increment total reward and time */
        alg->total_revenue=alg->total_revenue+rimm;
        alg->total_time=alg->total_time+timm;

        alg->rho=alg->total_revenue/alg->total_time;
        rho=alg->rho;
    }

    action=alg->last_action;
    prodcount=alg->last_prodcount;
    buffer=alg->last_buffer;

    next_prodcount=syst->prodcount;
    next_buffer=syst->buffer;
    qnext=max(alg->Q[next_buffer][next_prodcount][0],
              alg->Q[next_buffer][next_prodcount][1]);

    /* actual updating of the concerned Q value */
}
```

```

q=alg->Q[buffer][prodcount][action];
q=((1-alpha)*q)+(alpha*(rimm-(rho*timm)+qnxt));
alg->Q[buffer][prodcount][action]=q;

alg->revenue=0;
alg->time=TNOW;
alg->alpha=alg->alpha*0.9999999;
alg->exp_rate=alg->exp_rate*0.9999;

return;
}

```

9.2. Fixed Policy Codes

In the following codes, the policy is specified in global.h. The policy is contained in T_i , $i = 1, 2, 3$. The policy can be described as follows. (Of course, the policy can be directly implemented from the Q -factors in report.dat; in that case, one would use the maximizing action from the Q -factors associated with a given state.) If the buffer has i units, and the production count is lower than T_i , continue production; if the production count equals T_i , do maintenance.

```

# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include "global.h"
# include "typedefs.h"
# include "fun_protos.h"
# include "fun_list.h"

main()
{
/* A simulator for a production-inventory system using a fixed policy*/

int next_event,done,policy[4];
double event_clock[5],number_of_reps,total_rho;
system_data syst;
alg_data alg;
seed_data sd;

/* specify policy */

policy_setter(&alg);

for(REPLICATE=0;REPLICATE<30;REPLICATE++)
{

```

```
/* Initialize variables*/

initialize(event_clock,&syst,&alg,&sd);

done = 0; /* Pnemonic for simulation, 1 stands for end */
           /* 0 stands for continue */

while(0 == done)
{
/* Find out which event should take place next */

next_event=timer(event_clock);

/* Schedule the next event */

switch(next_event)
{
case 0:
done = 1;
break;

case 1:
arrive(event_clock,&syst,&alg,&sd);
break;

case 2:
produce(event_clock,&syst,&alg,&sd);
break;

case 3:
repair(event_clock,&syst,&alg,&sd);
break;

case 4:
maintain(event_clock,&syst,&alg,&sd);
break;
}

}
/* End of switch statements */

}

/* End of while loop */

total_rho=total_rho+(alg.revenue/TNOW);
printf("Average reward =%lf\n",alg.revenue/TNOW);
```

```
}

*****global.h ****
number_of_reps=(double)(REPLICATE);

printf("Average reward of policy=%lf\n",total_rho/number_of_reps);

return;
}
/* types of events 0=termination; 1=arrive; 2=produce;3=repair 4=maint */

int NUMBER_OF_EVENTS=5;

double LENGTH=1000000;

double BIG =1000001; /* has to be bigger than LENGTH */

#define PCMAX 100 /* max value for production count */
#define UPPER_LIMIT 3 /* S value for buffer */
#define LOWER_LIMIT 2 /* s value for buffer */

/* the following are the parameters of gamma distribution */
/* for PROD=production time, FAIL=time between failures, */
/* and REPAIR= repair time */
int N_PROD=8;
double LAMBDA_PROD=0.8;

int N_FAIL=8;
double LAMBDA_FAIL=0.01;

int N_REPAIR=4;
double LAMBDA_REPAIR=0.02;

double MU=0.06666666; /* exponential rate of arrival of demands */

/* the following parameters are for the unif dist. for maintenance */

double MAINT_MIN=5;
double MAINT_MAX=20;

double TNOW;

double PROFIT=1;
```

```
double MAINT_COST=2;

double REPAIR_COST=5;

int T1=54; /* denotes the threshold for maintenance when buffer=1 */
int T2=56; /* denotes the threshold for maintenance when buffer=2 */
int T3=57; /* denotes the threshold for maintenance when buffer=3 */

*****typedefs.h *****

long REPLICATE;

typedef struct
{
    double time_for_failure; /* time remaining for failure */
    double prod_time; /* time of production */
    int buffer; /* current buffer */
    int prodcnt; /* current production count */
}system_data;

typedef struct
{
    double revenue;
    int policy[UPPER_LIMIT+1][PCMAX];
}alg_data;

typedef struct
{
    long seed_event;
    long seed_prod;
    long seed_maint;
    long seed_fail;
    long seed_rep;
    long seed_arrive;
}seed_data;
*****fun_list.h *****

#include "uniformrnd.c"
#include "unifrnd.c"
#include "expornd.c"
#include "gmrnd.c"
#include "timer.c"
#include "arrive.c"
#include "produce.c"
#include "repair.c"
#include "maintain.c"
#include "initialize.c"
#include "max.c"
```

```
# include "policy_setter.c"

*****fun_prots.h *****

double uniformrnd(long *);
double unifrnd(double, double, long *);
double expornd(double, long *);
double gamrnd(int,double, long *);
int timer(double []);

void arrive(double [],system_data *,alg_data *,seed_data *);
void produce(double [],system_data *,alg_data *,seed_data *);
void repair(double [],system_data *,alg_data *,seed_data *);
void maintain(double [],system_data *,alg_data *,seed_data *);
void initialize(double [],system_data*,alg_data*,seed_data*);
double max(double,double);
void policy_setter(alg_data*);

*****initialize.c *****

void initialize(double event_clock[],system_data* syst, alg_data
*alg,seed_data* sd)
{/* Initialize the seeds for random number generation */

    int count1,count2,count3;

    sd->seed_event = 1000+(REPLICATE*100);
    sd->seed_prod = 1024+(REPLICATE*100);
    sd->seed_arrive= 5678+(REPLICATE*100);
    sd->seed_rep = 2546+(REPLICATE*100);
    sd->seed_maint = 6000+(REPLICATE*100);
    sd->seed_fail = 3456+(REPLICATE*100);

/* Initialize TNOW, the statistics, the event clock etc */

TNOW = 0;

/* Initialize the end of simulation event */
event_clock[0]=LENGTH;
/* Initialize the first arrival */
event_clock[1]=expornd(MU,&sd->seed_arrive);
/* Initialize the first production */
event_clock[2]=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);
syst->prod_time=event_clock[2];
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);
syst->prodcount=0;
syst->buffer=0;
event_clock[3]=BIG;
event_clock[4]=BIG;

alg->revenue=0;
```

```
return;  
}  
  
/************maintain.c *****/  
  
void maintain(double event_clock[],system_data* syst,alg_data* alg,  
seed_data* sd)  
{  
/* a maintenance has just been finished */  
  
double repair_time,maint_time,prod_time;  
  
  
syst->prodcount=0;  
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);  
alg->revenue=alg->revenue-MAINT_COST;  
  
  
event_clock[4]=BIG;  
  
/* Now to schedule the next event */  
  
if(syst->prodcount<alg->policy[syst->buffer][syst->prodcount])  
{  
  
/* start a production cycle, if the buffer has not maxed*/  
  
if(UPPER_LIMIT!=syst->buffer)  
{  
prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);  
if(prod_time<syst->time_for_failure)  
{/* production will occur successfully */  
event_clock[2]=TNOW+prod_time;  
}  
else  
{/* schedule a repair */  
repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);  
event_clock[3]=TNOW+prod_time+repair_time;  
}  
}  
}  
else  
{  
/* schedule a maintenance */  
}
```

```
maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
event_clock[4]=TNOW+maint_time;
}
```

```
return;
}
```

```
*****max.c *****
```

```
double max(double a,double b)
{
```

```
if (a>=b)
{
return (a);
}
else
{
return (b);
}
```

```
}
```

```
*****policy_setter.c *****
```

```
void policy_setter(alg_data* alg)
{
```

```
/* policy is set here using threshold levels from global.h" */
```

```
int count;
```

```
for(count=0;count<PCMAX;count++)
{
```

```
alg->policy[1][count]=T1;
```

```
}
```

```
for(count=0;count<PCMAX;count++)
{
```

```
alg->policy[2][count]=T2;
```

```
}
```

```
for(count=0;count<PCMAX;count++)
```

```
{  
    alg->policy[3][count]=T3;  
}  
  
for(count=0;count<PCMAX;count++)  
{  
    alg->policy[0][count]=100;  
}  
  
return;  
}  
  
/*************produce.c ******/  
  
void produce(double event_clock[],system_data* syst, alg_data*  
alg,seed_data* sd)  
{  
  
/* A successful production has just been completed */  
  
double repair_time,maint_time,prod_time;  
  
syst->prodcount=syst->prodcount+1;  
syst->buffer=syst->buffer+1;  
  
/* the following is reduced by the age */  
syst->time_for_failure=syst->time_for_failure-syst->prod_time;  
  
/* Now to schedule the next event */  
  
if(syst->prodcount<alg->policy[syst->buffer][syst->prodcount])  
{  
/* start a production cycle, if the buffer has not maxed*/  
  
    if(UPPER_LIMIT==syst->buffer)  
    {/* time to take a vacation */  
        event_clock[2]=BIG;
```

```
        }
        else
        {
            prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);

            if(prod_time<syst->time_for_failure)
                /* production will occur successfully */
                event_clock[2]=TNOW+prod_time;
                syst->prod_time=prod_time;
            }
            else
                /* schedule a repair */
                repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
                event_clock[2]=BIG;
                event_clock[3]=TNOW+prod_time+repair_time;
            }
        }
    }
else
{
/* schedule a maintenance */
maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
event_clock[2]=BIG;
event_clock[4]=TNOW+maint_time;
}

return;
}
*****repair.c *****

void repair(double event_clock[],system_data *syst, alg_data*
alg, seed_data *sd)
{
/* repair has just been finished */

double repair_time,maint_time,prod_time;

syst->prodcount=0;
syst->time_for_failure=gamrnd(N_FAIL,LAMBDA_FAIL,&sd->seed_fail);
alg->revenue=alg->revenue-REPAIR_COST;

event_clock[3]=BIG;

/* Now to schedule the next event */
```

```
if(syst->prodcount<alg->policy[syst->buffer] [syst->prodcount])
{
/* start a production cycle, if the buffer has not maxed*/

    if(UPPER_LIMIT!=syst->buffer)
    {
        prod_time=gamrnd(N_PROD,LAMBDA_PROD,&sd->seed_prod);
        if(prod_time<syst->time_for_failure)
        {/* production will occur successfully */
            event_clock[2]=TNOW+prod_time;
            syst->prod_time=prod_time;
        }
        else
        {/* schedule a repair */

            repair_time=gamrnd(N_REPAIR,LAMBDA_REPAIR,&sd->seed_rep);
            event_clock[3]=TNOW+prod_time+repair_time;
        }
    }
}
else
{/* schedule a maintenance */

    maint_time=unifrnd(MAINT_MIN,MAINT_MAX,&sd->seed_maint);
    event_clock[4]=TNOW+maint_time;
}

return;
}

/*************timer.c *****

int timer(double event_clock[])
{
    double minimum=BIG;
    int index,next_event;

    /* Find out which event is next */

    for(index=0;index<NUMBER_OF_EVENTS;index=index+1)
    {
        if(event_clock[index]<=minimum)
        {
            minimum=event_clock[index];
            next_event=index;
        }
    }
}
```

```
/* advance simulation clock */  
  
TNOW=minimum;  
  
/* return the next event */  
return(next_event);  
}
```

10. MATLAB Codes

MATLAB [111] is a very powerful software that makes it very easy for even a novice in programming to write complex programs. At the very outset, we would like to state the following clearly.

We do not sell MATLAB and recommend its use only because we view it as a useful software.

Here are some MATLAB-specific features that we have found very useful.

1. The syntax is very simple. You don't need to define types of variables (no need to worry about float, int etc). MATLAB is smart enough to figure out the type.
2. The syntax is similar to C and that of other programming languages.
2. It readily shows the bug in the program by stopping at the point at which there is a bug.
3. MATLAB can identify a number of bugs on its own. For instance, there is no danger in typing the following in MATLAB:

if(a=b)

when you mean

if(a==b)

In C, the compiler sets *b* to *a*, and this often invisible bug can cause a large number of problems. MATLAB treats only one (=) sign in the *if* statement as a bug and tells you there is a bug, thereby preventing this bug from entering your program.

4. MATLAB can handle matrices very efficiently. If you try to access a third element of a vector, which was defined to have only two elements, MATLAB will see this bug right away and will not return a junk value. Instead, it will

point out the bug to you. Another very useful feature of MATLAB is that it solves linear equations readily. So if the linear system is defined as:

$$\mathbf{A}\vec{x} = \vec{b},$$

you can solve for the vector \vec{x} in MATLAB by writing the following *one* line!

```
x=A\b;
```

This feature is extremely useful in a number of optimization applications. One example is policy iteration.

5. A number of simulation-based optimization applications can be coded in MATLAB. MATLAB has very good random number generators for almost all the distributions. Furthermore, you can change the SEED easily to do a new replication. So it is certainly possible to simulate discrete-event stochastic systems in MATALB. The simulator and the optimization algorithm (such as simulated annealing, tabu search or the genetic algorithm) can be written in the same program. Combining simulations written in simulation-software with an external C program can get really messy. Furthermore, such a system is often unstable and unreliable. Indeed, like with C, we can implement both the simulator and the optimizer in one compact set of programs with MATLAB.
6. A large number of useful “toolboxes” can be purchased with MATLAB. These toolboxes contain codes for a number of algorithms such as neural networks, regression, and interpolation. Function approximation in reinforcement learning may be done with these toolboxes.
7. MATLAB has very good graphical capabilities. For reinforcement learning or any other simulation-optimization technique, graphs are usually necessary to understand the results obtained.
- 8 Finally, if a researcher has a new idea, he or she can try it out *quickly* in MATLAB. Indeed, the time and effort spent in the first trials with a new algorithm are considerably reduced. Since simulation optimization is still a growing field, MATLAB may prove to be a very useful tool in its history.

We next provide a MATLAB program for policy iteration of average reward MDPs.

```
%%%%%%%%%%%%%
% This is the main file: main.m
% This file should be run
```

```
% tpm is the transition probability matrix
% trm is the transition reward matrix
% polita is the function that does policy iteration for average reward
% polita is stored in a separate file called polita.m

tpm(:,:,1)=[0.7,0.3;0.4,0.6];
tpm(:,:,2)=[0.9,0.1;0.2,0.8];
trm(:,:,1)=[6,-5;7,12];
trm(:,:,2)=[10,17;-14,13];
[policy,values]=polita(tpm,trm);

%%%%%%%%%%%%%
%%%%%%%%%%%%%
% This is the file: polita.m

function [policy,v]=polita(tpm,trm)
% function to do policy iteration for MDPs using avg reward
% tpm is a 3-dimensional
% array with (i,j,k) standing for initial state,final state and action
% similarly trm is a 3-d array. policy denotes policy and
% v denotes the value function vector for a given policy

SMALL=-100000; % some small number
[no_states,no_states,no_actions]=size(tpm);
policy=ones(no_states,1);
old_policy=ones(no_states,1);
r=zeros(no_states,1);
P=zeros(no_states,no_states);

done=0;
iteration=0;
while done~=1
% done is set to 1 when policy repeats

iteration=iteration+1;

% Policy Evaluation
% Finding P
for state=1:no_states
    for next_state=1:no_states
        if next_state==1
            P(state,next_state)=1;
        else
            if state==next_state
                P(state,next_state)=1-tpm(state,next_state,policy(state,1));
            else
                P(state,next_state)=-tpm(state,next_state,policy(state,1));
            end
        end
    end
end
```

```

        end
    end
end
end

% Finding r
for state=1:no_states
    sum=0;
    for next_state=1:no_states
        sum=sum+tpm(state,next_state,policy(state,1))*...
            trm(state,next_state,policy(state,1));
    end
    r(state,1)=sum;
end
policy

% finding the values for a policy
v=P\r;
rho=v(1,1)
v(1,1)=0;
% Policy improvement

for state=1:no_states
best=SMALL;
    for action=1:no_actions
        sum=0;
        for next_state=1:no_states
            sum=sum+tpm(state,next_state,action)*...
                (trm(state,next_state,action)+v(next_state,1));
        end
        if sum>best % strictly greater than relation to avoid cycling
            policy(state,1)=action;
            best=sum;
        end
    end
end

if policy==old_policy
done=1;
else
old_policy=policy;
end

end % of while loop

optimal_policy=policy
iterations_needed=iteration
%%%%%%%%%%%%%

```

11. Concluding Remarks

Our objective in this chapter was to demonstrate, via examples, how to write computer programs needed in simulation-based optimization. Our codes were written in a manner to simplify its understanding. As such, at a number of places, we have sacrificed compactness and brevity for the sake of clarity. Many of the codes that we have supplied above can be made more compact. (Of course, the speed of the program decreases with an increase in the number of lines in the code.)

Writing codes in C or MATLAB may appear to be a tedious job in the beginning, and yet it is important to master this skill to be able to do simulation optimization. Writing a simulator in a language such as C can take some time; however, Simulation optimization frequently requires this because it is difficult to integrate reinforcement learning, neural networks or simulated annealing within the simulators written in simulation packages.

12. Review Questions

1. Write the neural network codes for the incremental version of backpropagation. Read the batch and the incremental version of the Widrow-Hoff rule.
2. Simulate the airline system described in Chapter 14 in C.
3. Write programs for simulated annealing, tabu search, and the genetic algorithm. Combine them with the airline simulator of the previous question.
4. Simulate the AGV system described in Chapter 14 in C.
5. Instead of using the look-up table approach of the preventive maintenance RL implementation supplied in Section 9, use a neural network approach. (Use the incremental neural network codes — see Question 1 above.) Instead of using the neural network approach, you may want to use some other function approximation approach described in Chapter 9.

Chapter 16

CONCLUDING REMARKS

*The powerful play goes on.
And you may contribute a verse*

— Walt Whitman (1819-1892)

It is unfortunately true that even now there is a persistent belief that one cannot optimize a system when all one has is its simulation model. It will take some time for the science of simulation-based optimization to get the recognition it deserves in the field of stochastic optimization. One must not forget that simulation-based optimization is a reality today because some people decided to take “the path less traveled by” and made discoveries that changed its topography.

In some sense, simulation optimization has always existed in various forms in the literature, but many of those methods took a great deal of computer time — making it difficult for them to be used on large-scale problems. In this respect, the methods of simultaneous perturbation and reinforcement learning break new ground because they require modest computer time, and in spite of that come up with near-optimal results.

In the field of parametric optimization, the past reluctance to use simulation-based estimates *directly* in non-linear programming algorithms (of the numerical kind) can be attributed to the fact that too many function estimates are needed in regular, numerical, non-linear programming algorithms. With the advent of simultaneous perturbation, it has become clear that methods which need few function evaluations can actually be devised!

In control optimization, the biggest obstacles were the curses of modeling and dimensionality. Reinforcement learning and learning automata methods have made it possible to break these barriers, at least to a certain extent. This

has opened new and exciting possibilities; today one is not limited to the type of problems classical stochastic dynamic programming can solve.

Of course, there is a plethora of exciting challenges in the field of simulation optimization, which still remain for us to face. Some pointers to this are:

- A parametric-optimization algorithm that works on functions *in general* while requiring few function evaluations is still lacking in the literature.
- Many meta-heuristic methods lack sophisticated convergence analysis. Furthermore, convergence rates of some of these algorithms are unknown.
- The behavior of reinforcement learning on many well-known instances of MDPs is still unknown because this methodology is new.
- The lack of robust function approximation methods for reinforcement learning is still an issue that needs a great deal of research and work. Theoretical analysis of convergence methods is yet to be understood well, and this may turn out to be a key to developing robust function approximation schemes.

This book was meant to serve as an introduction to the elementary ideas underlying this topic. But we also hope that it has served as a challenge to the reader; the challenge clearly is to extend the frontiers of simulation-based optimization.

References

- [1] J. Abounadi. Stochastic approximation for non-expansive maps: Application to Q-learning algorithms. Unpublished Ph.D. Thesis, MIT, Department of Electrical Engineering and Computer Science, February, 1998.
- [2] J. Abounadi, D. Bertsekas, and V. Borkar. Learning algorithms for Markov decision processes with average cost. Technical Report, LIDS-P-2434, MIT, MA, USA., 1998.
- [3] J. Abounadi, D. Bertsekas, and V. Borkar. Stochastic approximations for non-expansive maps: Application to Q -learning algorithms. Technical Report, LIDS-P-2433, MIT, MA, USA., 1998.
- [4] J.S. Albus. *Brain, Behavior and Robotics*. Byte Books, Peterborough, NH, USA, 1981.
- [5] M.H. Alrefaei and S. Andradóttir. A simulated annealing algorithm with constant temperature for discrete stochastic optimization. *Management Science*, 45(5):748–764, 1999.
- [6] M.H. Alrefaei and S. Andradóttir. A modification of the stochastic ruler method for discrete stochastic optimization. *European Journal of Operational Research (to appear)*, 2002.
- [7] T. Altıok and S. Stidham. The allocation of interstage buffer capacities in production lines. *IIE Transactions*, 15(4):292–299, 1984.
- [8] S. Andradóttir. Simulation optimization. In *Handbook of Simulation (edited by Jerry Banks), Chapter 9*. John Wiley and Sons, New York, NY, USA, 1998.
- [9] A. B. Badiru and D. B. Sieger. Neural network simulation metamodel in economic analysis of risky projects. *European Journal of Operational Research*, 105:130–142, 1998.
- [10] N. Barish and N. Hauser. Economic design of control decisions. *Journal of Industrial Engineering*, 14:125–134, 1963.
- [11] A.G. Barto, S.J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

- [12] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.
- [13] R.E. Bechhofer, T.J. Santner, and D.J. Goldsman. *Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons*. John Wiley, New York, NY, USA, 1995.
- [14] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60:503–516, 1954.
- [15] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [16] R.E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, NJ, 1962.
- [17] P.P. Belobaba. Application of a probabilistic decision model to airline seat inventory control. *Operations Research*, 37:183–197, 1989.
- [18] D. Bertsekas. *Non-Linear Programming*. Athena Scientific, Belmont, MA, USA, 1995.
- [19] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, USA, 1996.
- [20] D.P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, USA, 1995.
- [21] D. Blackwell. Discrete dynamic programming. *Ann. Math. Stat.*, 33:226–235, 1965.
- [22] J. Boesel and B.L. Nelson. Accounting for randomness in heuristic simulation optimization. Preprint, Northwestern University, Department of Industrial Engineering.
- [23] L.B. Booker. *Intelligent Behaviour as an Adaptation to the Task Environment*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1982.
- [24] V. Borkar and P. Varaiya. Adaptive control of Markov chains I. finite parameter set. *IEEE Transactions on Automatic Control*, 24:953–958, 1979.
- [25] V. S. Borkar. Stochastic approximation with two-time scales. *Systems and Control Letters*, 29:291–294, 1997.
- [26] V. S. Borkar. Asynchronous stochastic approximation. *SIAM J. Control Optim.*, 36 No 3:840–851, 1998.
- [27] V. S. Borkar and V. R. Konda. The actor-critic algorithm as multi-time scale stochastic approximation. *Sadhana (Proc. Indian Academy of Sciences - Eng. Sciences)*, 27(4), 1997.
- [28] V. S. Borkar and S.P. Meyn. The ODE method for convergence of stochastic approximation and reinforcement learning. *SIAM Journal of Control and Optimization*, 38 (2):447–469, 2000.
- [29] V.S. Borkar and K. Soumyanath. A new analog parallel scheme for fixed point computation, part I: Theory. *IEEE Transactions on Circuits and Systems I: Theory and Applications*, 44:351–355, 1997.

- [30] J.A. Boyan and A.W. Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. *Advances in Neural Information Processing Systems*, pages 369–376, 1995.
- [31] S.J. Bradtko and M. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge, MA, USA, 1995.
- [32] G. Bronson. *C For Engineers and Scientists*. West Publishing Company, MN, USA, 1993.
- [33] Y. Carson and A. Maria. Simulation optimization: Methods and applications. *Proceedings of the 1997 Winter Simulation Conference*, pages 118–126, 1997.
- [34] H. Cohn and M. J. Fielding. Simulated annealing: searching for an optimal temperature schedule. *SIAM Journal of Optimization (to appear)*.
- [35] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Neural Information Processing Systems (NIPS)*. 1996.
- [36] C. Darken, J. Chang, and J. Moody. Learning rate schedules for faster stochastic gradient search. In D.A. White and D.A. Sofge, editors, *Neural Networks for Signal Processing 2 - Proceedings of the 1992 IEEE Workshop*. IEEE Press, Piscataway, NJ, 1992.
- [37] T.K. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck. Solving semi-Markov decision problems using average reward reinforcement learning. *Management Science*, 45(4):560–574, 1999.
- [38] T.K. Das, V. Jain, and A.Gosavi. Economic design of dual-sampling-interval policies for x-bar charts with and without run rules. *IIE Transactions*, 29:497–506, 1997.
- [39] T.K. Das and S. Sarkar. Optimal preventive maintenance in a production inventory system. *IIE Transactions*, 31:537–551, 1999.
- [40] S. Davies. Multi-dimensional interpolation and triangulation for reinforcement learning. *Advances in Neural Information and Processing Systems*, 1996.
- [41] L. Devroye, L. Gyorfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, New York, USA, 1996.
- [42] S. A. Douglass. *Introduction to Mathematical Analysis*. Addison-Wesley Publishing Company, Reading, MA, USA., 1996.
- [43] M. J. Fielding. *Optimisation by Simulated Annealing*. PhD thesis, The University of Melbourne, Department of Mathematics, Australia, 1999.
- [44] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, New York, NY, USA, 1997.
- [45] P.A. Fishwick. Neural network models in simulation: A comparison with traditional modeling approaches. *Proceedings of the 1989 Winter Simulation Conference*, pages 702–710, 1989.
- [46] B.L. Fox and G.W. Heine. Probabilistic search with overrides. *Annals of Applied Probability*, 5:1087–1094, 1995.

- [47] M.C. Fu. Optimization via simulation: A review. *Annals of Operations Research*, 53:199–247, 1994.
- [48] M.C. Fu and J. Hu. Efficient design and sensitivity analysis of control charts using Monte Carlo simulation. *Management Science*, 45(3):395–413, 1999.
- [49] E.D. Gaughan. *Introduction to Analysis, 4th edition*. Brooks/Cole Publishing Company, Belmont, CA, USA, 1993.
- [50] S.B. Gelfand and S.K. Mitter. Simulated annealing with noisy or imprecise energy measurements. *Journal of Optimization Theory and Applications*, 62(1):49–62, 1989.
- [51] F. Glover. Tabu Search: A Tutorial. *Interfaces*, 20(4):74–94, 1990.
- [52] F. Glover and S. Hanafi. Tabu search and finite convergence. 2001. Working Paper, University of Colorado, Colorado.
- [53] F. Glover, J.P. Kelly, and M. Laguna. New advances and applications of combining simulation and optimization. *Proceedings of the 1996 Winter Simulation Conference*, pages 144–152, 1996.
- [54] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [55] D. Goldsman and B.L. Nelson. Ranking, selection, and multiple comparisons in computer simulation. *Proceedings of the 1994 Winter Simulation Conference*, pages 192–199, 1994.
- [56] D. Goldsman and B.L. Nelson. Comparing Systems via Simulation. In *Handbook of Simulation (edited by Jerry Banks), Chapter 8*. John Wiley and Sons, New York, NY, USA, 1998.
- [57] D. Goldsman, B.L. Nelson, and B. Schmeiser. Methods for selecting the best system. *Proceedings of the 1991 Winter Simulation Conference (B. L. Nelson, W.D. Kelton, and G.M. Clark, eds.)*, pages 177–186, 1991.
- [58] W.B. Gong, Y.C. Ho, and W. Zhai. Stochastic comparison algorithm for discrete optimization with estimation. *Proc. 31st. Conf. Decision Control*, pages 795–800, 1992.
- [59] A. Gosavi. An Algorithm for solving semi-Markov decision problems using reinforcement learning: Convergence analysis and numerical results. Unpublished Ph.D. dissertation, Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, FL, USA, 1999.
- [60] A. Gosavi. The effect of noise on artificial intelligence and meta-heuristic techniques. In *Proceedings of the Artificial Neural Networks in Engineering Conference (Intelligent Engineering Systems Through Artificial Neural Networks)*, volume 12, pages 981–988. American Society of Mechanical Engineering Press, 2002.
- [61] A. Gosavi. A reinforcement learning algorithm based on policy for average reward: Empirical results with yield management and convergence analysis. *Machine Learning (to appear)*, 2002. (Technical report available with author until paper appears in print).

- [62] A. Gosavi. Asynchronous convergence and boundedness in reinforcement learning. *Proceedings of the 2003 Institute of Industrial Engineering Research Conference in Portland, Oregon*, 2003.
- [63] A. Gosavi. Reinforcement learning for long-run average cost. *European Journal of Operational Research (to appear)*, 2003. (Technical Report available with author until paper appears in print).
- [64] A. Gosavi, N. Bandla, and T. K. Das. A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. *IIE Transactions (Special Issue on Large-Scale Optimization)*, 34(9):729–742, 2002.
- [65] A. Gosavi, T. K. Das, and S. Sarkar. A simulation-based learning automata framework for solving semi-Markov decision problems. *IIE Transactions (to appear)*, 2003.
- [66] B.S. Gottfried. *Programming with C*. McGraw Hill, New York, NY, USA, 1991.
- [67] B. Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13:311–329, 1988.
- [68] C. Harrell, B.K. Ghosh, and R. Bowden. *Simulation Using Promodel*. McGraw Hill Higher Education, Boston, MA, USA, 2000.
- [69] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, NY, USA, 2001.
- [70] S. Haykin. *Neural Networks: A Comprehensive Foundation*. McMillan, New York, NY, USA, 1994.
- [71] S. Heragu. *Facilities Design*. PWS Publishing Company, Boston, MA, USA, 1997.
- [72] S. S. Heragu and S. Rajgopalan. A literature survey of the AGV flowpath design problem. Technical Report No 37-96-406 at the Rensselaer Polytechnic Institute, DSES Department, NY, USA, 1996.
- [73] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research, Seventh Edition*. McGraw Hill, New York, 2001.
- [74] G.E. Hinton. Distributed representations. Technical Report CMU-CS-84-157, Carnegie Mellon University, Pittsburgh, PA, USA, 1984.
- [75] Y.C. Ho and X.R. Cao. *Perturbation Analysis of Discrete Event Dynamic Systems*. Kluwer, 1991.
- [76] Y. Hochberg and A.C. Tamhane. *Multiple Comparison Procedures*. Wiley, 1987.
- [77] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.
- [78] J.H. Holland. Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 593–623. Morgan Kaufmann, San Mateo, CA, USA, 1986.

- [79] T. Homem-de Mello. Variable-sample methods and simulated annealing for discrete stochastic optimization. 2001. Working paper, Department of Industrial, Welding, and Systems Engineering, The Ohio State University, Columbus, OH, USA.
- [80] R. Hooke and T.A. Jeeves. Direct search of numerical and statistical problems. *ACM*, 8:212–229, 1966.
- [81] R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
- [82] S.H. Jacobson and L.W. Schruben. A harmonic analysis approach to simulation sensitivity analysis. *IIE Transactions*, 31(3):231–243, 1999.
- [83] A. Jalali and M. Ferguson. Computationally efficient adaptive control algorithms for Markov chains. In *Proceedings of the 29th IEEE Conference on Decision and Control*, pages 1283 – 1288. 1989.
- [84] S.A. Johnson, J.R. Stedinger, C.A. Shoemaker, Y. Li, and J.A. Tejada-Guibert. Numerical solution of continuous state dynamic programs using linear and spline interpolation. *Operations Research*, 41(3):484–500, 1993.
- [85] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
- [86] P. Kanerva. *Sparse Distributed Memory*. MIT Press, Cambridge, MA, USA, 1988.
- [87] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. van Nostrand-Reinhold, NY, USA, 1960.
- [88] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Stat.*, 23:462–466, 1952.
- [89] R.A. Kilmer and A. E. Smith. Using artificial neural networks to approximate a discrete event stochastic simulation model. *Intelligent Engineering Systems Through Artificial Neural Networks (edited by C.H. Dagli, L.I. Burke, B.R. Fernandez, and J. Ghosh, ASME Press)*, 3:631–636, 1993.
- [90] R.A. Kilmer, A. E. Smith, and L.J. Shuman. Computing confidence intervals for stochastic simulation using neural network metamodels. *Computers and Industrial Engineering (to appear)*, 1998.
- [91] S-H Kim and B.L. Nelson. A fully sequential procedure for indifference-zone selection in simulation. *ACM Transactions on Modeling and Computer Simulation*, 11:251–273, 2001.
- [92] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
- [93] J.P.C. Kleijnen. Sensitivity analysis and optimization in simulation:design of experiments and case studies. In *Proceedings of the 1995 Winter Simulation Conference*, pages 133–140. 1995.
- [94] A. H. Klopf. Brain function and adaptive systems — a heterostatic theory. Technical Report AFCRL-72-0164, 1972.

- [95] D.E. Knuth. *The Art of Computer Programming, Vol 1: Seminumerical Algorithms, 3rd edition.* Addison-Wesley, Reading, MA, 1998.
- [96] V.R. Konda and V. S. Borkar. Actor-critic type learning algorithms for Markov decision processes. *SIAM Journal on Control and Optimization*, 38(1):94–123, 1999.
- [97] E. Kreyszig. *Advanced Engineering Mathematics.* John Wiley and Sons, 1998.
- [98] P.R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329–380, 1985.
- [99] P.R. Kumar and P. Varaiya. *Stochastic Systems: Estimation, Identification, and Adaptive Control.* Prentice Hall, Englewood Cliffs, NJ, USA, 1986.
- [100] H.J. Kushner and D.S. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems.* Springer Verlag, New York, 1978.
- [101] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P.E. Wright. Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions. *SIAM Journal on Optimization*, 9 (1):112–147, 1998.
- [102] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw Hill, Inc., New York, NY, USA, 1999.
- [103] P. L'Ecuyer. Good Parameters for Combined Multiple Recursive Random Number Generators. *Operations Research*, 47:159–164, 1999.
- [104] K. Littlewood. Forecasting and control of passenger bookings. In *Proceedings of the 12th AGIFORS (Airline Group of the International Federation of Operational Research Societies Symposium)*, pages 95–117, 1972.
- [105] L. Ljung. Analysis of recursive stochastic algorithms. *IEEE Transactions on Automatic Control*, 22:551–575, 1977.
- [106] M. Lundy and A. Mees. Convergence of the annealing algorithm. *Mathematical Programming*, 34:111–124, 1986.
- [107] G.R. Madey, J. Wienroth, and V. Shah. Integration of neurocomputing and system simulation for modeling continuous improvement systems in manufacturing. *Journal of Intelligent Manufacturing*, 3:193–204, 1992.
- [108] S. Mahadevan. To discount or not to discount: A case study comparing R-learning and Q-learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 164–172. New Brunswick, NJ, 1994.
- [109] S. Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1):159–95, 1996.
- [110] S. Mahadevan and G. Theodoros. Optimizing production manufacturing with reinforcement learning. *Eleventh International FLAIRS conference*, pages 372–377, 1998.
- [111] Matlab. *The Student edition of MATLAB.* Prentice Hall, Englewood Cliffs, NJ, USA, 1995.

- [112] R. Mattheij and J. Molenaar. *Ordinary Differential Equations in Theory and Practice*. John Wiley and Sons, West Sussex, England, 1996.
- [113] J.I. McGill and G.J. van Ryzin. Revenue management: Research overview and prospects. *Transportation Science*, 33(2):233–256, 1999.
- [114] G. Meghabghab and G. Nasr. Iterative RBF neural networks as metamodels of stochastic simulations. *2nd International Conference on Intelligent Processing and Manufacturing of Materials, Honolulu, Hawaii, USA*, 2:729–734, 1999.
- [115] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.
- [116] T.M. Mitchell. *Machine Learning*. McGraw Hill, Boston, MA, USA, 1997.
- [117] D.G. Montgomery, G.C. Runger, and N.A. Hubele. *Engineering Statistics, Second Edition*. John Wiley and Sons, New York, NY, USA, 2001.
- [118] D.C. Montgomery. *Introduction to Statistical Quality Control (Fourth Edition)*. John Wiley and Sons, New York, NY, USA, 2001.
- [119] R.H. Myers and D.C. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, New York, NY, USA, 1995.
- [120] K.S. Narendra and M.A.L. Thathachar. *Learning Automata: An Introduction*. Prentice Hall, Englewood Cliffs, NJ, USA, 1989.
- [121] K.S. Narendra and R.M. Wheeler. An N-Player sequential stochastic game with identical payoffs. *IEEE Transactions Systems, Man, and Cybernetics*, 13:1154–1158, 1983.
- [122] J.F. Nash. Equilibrium points in n-person games. *Proceedings, Nat. Acad. of Science, USA*, 36:48–49, 1950.
- [123] J.A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308–313, 1965.
- [124] B.L. Nelson. Designing efficient simulation experiments. *Proceedings of the 1992 Winter Simulation Conference. (J.J. Swain, D. Goldsman, R.C. Crain and J.R. Wilson eds.)*, pages 126–132, 1992.
- [125] M.L. Padgett and T.A. Roppel. Neural networks and simulation: modeling for applications. *Simulation*, 58:295–305, 1992.
- [126] S.K. Park and K.W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.
- [127] C.D. Paternina and T.K. Das. Intelligent dynamic control policies for serial production lines. *IIE Transactions*, 33(1):65–77, 2001.
- [128] J. Peng and R.J. Williams. Efficient learning and planning with the DYNA framework. *Adaptive Behavior*, 1:437–454, 1993.
- [129] D.L. Pepyne, D.P. Looze, C.G. Cassandras, and T.E. Djaferis. Application of Q-learning to elevator dispatching. Unpublished Report, 1996.

- [130] G. C. Pflug. *Optimization of Stochastic Models: The Interface between simulation and optimization*. Kluwer Academic, 1996.
- [131] D.T. Pham and D. Karaboga. *Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks*. Springer-Verlag, New York, USA, 1998.
- [132] C.R. Philbrick and P. K. Kitanidis. Improved dynamic programming methods for optimal control of lumped-parameter stochastic systems. *Operations Research*, 49(3):398–412, 2001.
- [133] H. Pierreval. Training a neural network by simulation for dispatching problems. *Proceedings of the Third Rensselaer International Conference on Computer Integrated Manufacturing*, pages 332–336, 1992.
- [134] H. Pierreval and R.C. Huntsinger. An investigation on neural network capabilities as simulation metamodels. *Proceedings of the 1992 Summer Computer Simulation Conference*, pages 413–417, 1992.
- [135] E. L. Plambeck, B.R. Fu, S.M. Robinson, and R. Suri. Sample path optimization of convex stochastic performance functions. *Mathematical Programming*, 75:137–176, 1996.
- [136] B.T. Poljak and Y.Z. Tsyplkin. Pseudogradient adaptation and training algorithms. *Automation and Remote Control*, 12:83–94, 1973.
- [137] M. A. Pollatschek. *Programming Discrete Simulations*. Research and Development Books, Lawrence, KS, USA., 1995.
- [138] P. Pontrandolfo, A. Gosavi, O.G. Okogbaa, and T.K. Das. Global supply chain management: A reinforcement learning approach. *International Journal of Production Research (to appear)*, 40(6):1299–1317, 2002.
- [139] W. H. Press, S.A.Tuckolsky, W.T. Vetterling, and B.P.Flannery. *Numerical Recipies in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1992.
- [140] M. L. Puterman. *Markov Decision Processes*. Wiley Interscience, New York, NY, USA, 1994.
- [141] Y. Rinott. On two-stage selection procedures and related probability-inequalities. *Communications in Statistics: Theory and Methods*, A7:799–811, 1978.
- [142] H. Robbins and S. Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22:400–407, 1951.
- [143] L. W. Robinson. Optimal and approximate control policies for airline booking with sequential nonmonotonic fare classes. *Operations Research*, 43:252–263, 1995.
- [144] S. M. Ross. *Stochastic Processes*. John Wiley and Sons, New York, NY, 1996.
- [145] S. M. Ross. *Introduction to Probability Models*. Academic Press, San Diego, CA, USA, 1997.
- [146] R. Y. Rubinstein and A. Shapiro. *Sensitivity Analysis and Stochastic Optimization by the Score Function Method*. John Wiley and Sons, New York, NY, 1983.

- [147] W. Rudin. *Real Analysis*. McGraw Hill, N.Y., USA, 1964.
- [148] G. Rudolph. Convergence of evolutionary algorithms in general search spaces. *Proceedings of the Third IEEE Conference on Evolutionary Computation, Piscataway, NJ: IEEE Press*, pages 50–54, 1996.
- [149] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Micro-structure of Cognition*. MIT Press, Cambridge, MA, 1986.
- [150] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, 1994.
- [151] A.L. Samuel. Some studies in machine learning using the game of checkers. In E.A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 1959.
- [152] S. Sarkar and S. Chavali. Modeling parameter space behavior of vision systems using Bayesian networks. *Computer Vision and Image Understanding*, 79:185–223, 2000.
- [153] F.A.V. Schouten and S.G. Vanneste. Maintenance optimization with buffer capacity. *European Journal of Operational Research*, 82:323–338, 1992.
- [154] L. Schrage. A more portable random number generator. *Assoc. Comput. Mach. Trans. Math. Software*, 5:132–138, 1979.
- [155] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. *Proceeding of the Tenth Annual Conference on Machine Learning*, pages 298–305, 1993.
- [156] L.I. Sennott. The computation of average optimal policies in denumerable state Markov control processes. *Adv. Appl. Prob.*, 29:114–137, 1997.
- [157] L.I. Sennott. *Stochastic Dynamic Programming and the Control of Queueing Systems*. John Wiley and Sons, New York, NY, USA, 1999.
- [158] S. Sethi and G.L. Thompson. *Optimal Control Theory: Applications to Management Science and Economics, Second Edition*. Kluwer Academic Publishers, Boston, USA, 2000.
- [159] L.S. Shapley. Stochastic games. *Proc. Nat. Acad. Sci., USA*, 39:1095–1100, 1953.
- [160] D. Simchi-Levi, P. Kaminsky, and E. Simchi-Levi. *Designing and Managing a Supply Chain*. McGraw Hill, Boston, MA, USA, 2000.
- [161] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems (1996)*, pages 974–980. 1997.
- [162] J.C. Spall. Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation. *IEEE Transactions on Automatic Control*, 37:332–341, 1992.
- [163] R. Sutton. Reinforcement Learning. *Machine Learning (Special Issue)*, 8(3), 1992.

- [164] R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.
- [165] R.S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, USA, May 1984.
- [166] R.S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.
- [167] P. Tadepalli and D. Ok. Model-based Average Reward Reinforcement Learning Algorithms. *Artificial Intelligence*, 100:177–224, 1998.
- [168] H.A. Taha. *Operations Research: An Introduction*. Prentice Hall, NJ., USA, 1997.
- [169] H. Taylor and S. Karlin. *An Introduction to Stochastic Modeling*. Academic Press, New York, 1984.
- [170] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3), 1992.
- [171] T. Tezcan and A. Gosavi. Optimal buffer allocation in production lines using an automata search. *Proceedings of the 2001 Institute of Industrial Engineering Research Conference in Dallas, Texas*, 2001.
- [172] M.A.L. Thathachar and K.R. Ramakrishnan. A cooperative game of a pair of learnign automata. *Automatica*, 20:797–801, 1894.
- [173] M.A.L. Thathachar and P.S. Sastry. Learning optimal discriminant functions through a cooperative game of automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 17:73–85, 1987.
- [174] J. Tsitsiklis. Markov chains with rare transitions and simulated annealing. *Mathematics of Operations Research*, 14:70–90, 1989.
- [175] J. Tsitsiklis. Asynchronous stochastic approximation and Q -learning. *Machine Learning*, 16:185–202, 1994.
- [176] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.
- [177] A. Turgeon. Optimal operation of multi-reservoir power systems with stochastic inflows. *Water Resources Research*, 16(2):275–283, 1980.
- [178] J.M. Twomey and A.E. Smith. Bias and variance of validation models for function approximation neural networks under conditions of sparse data. *IEEE Transactions on Systems, Man., and Cybernetics*, 28(3):417–430, 1998.
- [179] P. van Laarhoven and E. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, 1987.
- [180] J.A. E. E. van Nunen. A set of successive approximation methods for discounted Markovian decision problems. *Z. Operations Research*, 20:203–208, 1976.
- [181] J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, 1944.

- [182] C.J. Watkins. *Learning from Delayed Rewards*. PhD thesis, Kings College, Cambridge, England, May 1989.
- [183] P. J. Werbös. *Beyond Regression: New Tools for Prediction and Analysis of Behavioral Sciences*. PhD thesis, Harvard University, Cambridge MA, USA, May 1974.
- [184] P. J. Werbös. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man., and Cybernetics*, 17:7–20, 1974.
- [185] R. M. Wheeler and K. S. Narendra. Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, 31(6):373–376, 1986.
- [186] D. J. White. Dynamic programming, Markov chains, and the method of successive approximations. *J. Math. Anal. Appl.*, 6:373–376, 1963.
- [187] B. Widrow and M.E. Hoff. Adaptive Switching Circuits. In *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4*, pages 96–104. 1960.
- [188] W.L. Winston. *An Introduction to Mathematical Programming*. Duxbury Press, CA, USA, 1995.
- [189] I.H. Witten. An adaptive optimal controller for discrete time Markov environments. *Information and Control*, 34:286–295, 1977.
- [190] D. Yan and H. Mukai. Stochastic discrete optimization. *SIAM Journal of Control and Optimization*, 30:594–612, 1992.

Index

- accumulation point, 303
- acronyms, 13
- actor-critic algorithms, 252
 - convergence, 404
- address, 435
- age replacement, 419
- AGV, 424
- airline revenue management, 412
- approximating sequence, 206
- asynchronism, 387
- automated guided vehicles, 424
- average, 24
- average reward, 280

- backpropagation, 73, 411
- backward recursion, 205
- BASIC, 434
- Bayesian Learning, 126
- behavior, 29
- Bellman equation, 162, 181
 - average reward, 162
 - discounted reward, 173
 - optimality proof, 349
 - optimality proof for average reward case, 365
- binary trees, 121, 267
- Bolzano-Weierstrass, 304
- bootstrap, 90
- bounded sequence, 300
- buffer optimization, 420

- C language, 430, 433
- cardinality, 11
- Cauchy sequence, 301
- cdf, 22
- central differences, 98, 327
- central limit theorem, 27
- Chapman-Kolmogorov theorem, 142
- closed form, 49, 58
- co-related data, 43

- complex event, 17
- compound event, 17
- computational operations research, 1
- computer programming, 3, 433
- computer programs
 - backpropagation batch mode, 470
 - dynamic programming, 441
 - MATLAB, 531
 - neural networks, 464
 - organization, 436
 - preventive maintenance, 506
 - random number generators, 437
 - reinforcement learning, 478
 - simultaneous perturbation, 439
- continuous functions, 318
- continuous time Markov process, 146
- continuously differentiable, 319
- contraction mapping, 308, 360
- control charts, 426
- control optimization, 4, 51
- convergence, 3
- convergence of sequences, 298
 - with probability 1, 381
- coordinate sequence, 306
- cost function, 48
- cross-over, 119
- cumulative distribution function, 22
- curse of dimensionality, 212
- curse of modeling, 212

- decision-making process, 184
- decreasing sequence, 300
- differential equations, 380
- discounted reward, 171
- distributions, 21
- domain, 291
- DTMDP, 183
- dynamic optimization, 4
- dynamic programming, 4, 52, 161, 170
- dynamic systems, 29

- elevator scheduling, 427
- embedded Markov chain, 183
- EMSR, 213, 416
- ergodic, 146
- Euclidean norm, 10, 324
- event, 16
- exhaustive enumeration, 157, 186
- expected value, 24
- exploration, 230
- feedback, 124, 228, 279
- finite differences, 100, 327
- finite horizon, 203
- finite horizon problems, 259
- fit, 67
- fixed point theorem, 312
- FORTRAN, 434, 435
- forward differences, 99, 327
- function approximation, 260, 272
 - convergence, 405
 - difficulties, 262
 - neural networks, 264
- function fitting, 69
- game, 201
- games, 278
- Gauss elimination, 165
- Gauss-Siedel algorithm, 178
- genetic algorithm, 117
- global optimum, 82, 320
- global variables, 436
- gradient descent, 70, 79
 - convergence, 324
- H-Learning
 - average reward, 255
 - discounted reward, 254
- heuristic, 213, 416
- hyper-plane, 64
- identity matrix, 10
- immediate reward, 154
- increasing sequence, 300
- incremental, 72
- independent, 42
- induction, 294
- infinity norm, 290
- inventory control, 410
- inverse function method, 36
- irreducible, 146
- jackknife, 90
- jumps, 136
- kanban, 421
- kernel methods, 266
- Kim-Nelson method, 109
- LAST, 123
- layer
 - hidden, 74
 - input, 74
 - output, 74
- learning, 228
- learning automata
 - control optimization, 277
 - parametric optimization, 123
- learning rate, 280
- limiting probabilities, 143
- linear programming, 48, 201
- Lipschitz condition, 324, 380
- local optimum, 81, 96, 320
- loss function, 48
- mapping, 292
- Markov chain, 139
- Markov decision problems, 148, 277
 - convergence, 344
 - reinforcement learning, 211
- Markov process, 136
- mathematical programming, 4
- MATLAB, 430
- max norm, 10, 290
- MCAT, 277
- MDPs, 148, 277
 - convergence, 344
 - reinforcement learning, 211
- mean, 24
- memoryless, 137
- meta-heuristics, 110
- metamodel, 58
- model-based, 69, 93, 409
- model-building algorithms, 253, 426
- model-free, 93
- modified policy iteration, 197
- monotonicity, 346
- multiple comparisons, 107
- mutation, 119
- mutually exclusive, 17
- n-step transition probabilities, 140
- n-tuple, 11, 289
- natural process, 184
- nearest neighbors, 265
- neighbor, 111
- neighborhood, 303
- Nelder-Mead, 4, 94
- neural networks, 69, 264
 - backpropagation, 73
 - codes, 464
 - gradient descent, 326
 - linear, 69
 - non-linear, 73
- neuro-dynamic programming, 211, 271
- Neuro-RSM, 69

- neuron, 69, 269
 nodes, 74
 noise due to simulation, 338
 non-derivative methods, 104
 non-expansive mappings, 380
 non-linear programming, 48, 70, 94
 non-terminating, 43
 normalization, 279
 normed vector spaces, 291
 norms, 10, 290
 notation, 9
 - matrix, 10
 - product, 9
 - sequence, 11
 - sets, 11
 - sum, 9
 - vector, 10
 objective function, 48, 155
 off-line, 229
 on-line, 229
 ordinary differential equations, 380
 overbook, 412
 parametric optimization, 4, 47, 48
 - continuous, 94
 - discrete, 106
 partial derivatives, 319
 PDN, 126
 performance metric, 155
 phase, 113
 plane, 63
 pmf, 21, 23
 pointers, 435
 policy iteration, 195
 - average reward MDPs, 163
 - convergence proof for average reward case, 372
 - convergence proof for discounted case, 357
 - discounted reward MDPs, 173
 - SMDPs, 189
 preventive maintenance, 416
 probability, 16
 probability density function, 23
 probability mass function, 21
 Q-factor, 217
 - boundedness, 394
 - definition, 218
 - Policy iteration, 235
 - Value iteration, 219
 Q-Learning
 - convergence, 392
 - model-building, 257
 - steps in algorithm, 225
 - worked-out example, 231
 Q-P-Learning
 - average reward MDPs, 244
 - convergence, 400
 - discounted reward MDPs, 237
 - incomplete evaluation of average reward, 402
 - semi-Markov decision problems, 250
 quality control, 426
 queue, 134
 R-Learning, 241
 random process, 133
 random system, 30
 random variable, 15
 random variables
 - continuous, 22
 - discrete, 21
 range, 291
 ranking and selection, 107
 regression, 265, 269
 - linear, 60, 63, 69
 - non-linear, 66
 - piecewise, 65
 regular Markov chain, 142, 334
 Reinforcement Learning, 53
 - asynchronous convergence, 383
 - average reward MDPs, 238
 - convergence, 379
 - discounted reward MDPs, 224
 - finite convergence, 397
 - introduction, 211
 - SMDPs, 245
 - synchronous convergence, 382
 Relative Q-Learning, 239
 - convergence, 397
 - model-building, 258
 relative value iteration, 168, 179
 Relaxed-SMART, 243, 249
 renewal reward theorem, 187
 renewal theory, 419
 replication, 42
 response, 279
 response surface method, 1, 57, 58
 revenue management, 412
 Reward-Inaction Scheme, 280
 Rinott method, 108
 Robbins-Monro algorithm, 217, 220, 221
 RSM, 57, 58
 scalar, 9
 seed, 34, 42, 241
 Semi-Markov decision problems
 - average reward DP, 186
 - convergence, 379
 - definition, 182
 - discounted reward DP, 194
 - learning automata, 277
 - reinforcement learning, 248

- sequence, 11, 297
- sets, 11
- sigmoid, 76
- simulated annealing
 - algorithm, 111
 - convergence, 333
- simulation, 32
- simulation packages, 433
- simultaneous perturbation, 4, 101, 328
- SMART, 242, 248
- SMDPs
 - average reward DP, 186
 - convergence, 379
 - definition, 182
 - discounted reward DP, 194
 - learning automata, 277
 - reinforcement learning, 248
- span seminorm, 180
- spill-over effect, 263
- standard deviation, 25
- state, 29, 51, 133
- state aggregation, 260
- static optimization, 4
- stationary point, 319
- step size, 95, 102, 125, 223
- stochastic game, 201
- stochastic optimization, 2, 47
- stochastic process, 133
- stochastic system, 30
- straight line, 60
- strong law of large numbers, 27
- sup-norm, 290
- supply chain, 423
- supply chain management, 423
- symmetric neighborhoods, 334
- system, 1, 3–5, 51, 133
 - definition, 29
- tabu search, 119
 - mutations, 119
 - tabu list, 119
- Taylor's theorem, 320, 329
- technology, 3
- temperature, 113
- terminating, 43
- thresholding, 76
- TPM, 139, 152, 214
- transfer line, 420
- transformation, 11, 292
- transition probability matrix, 139
- transition reward matrix, 154
- transition time matrix, 183
- transpose, 10
- trial-and-error, 228
- TRM, 154
- TTM, 183
- uniformization, 183, 193, 196
- validation, 89
- value iteration, 178, 195
 - Average reward MDPs, 165
 - convergence for average reward case, 379
 - convergence proof for discounted case, 363
 - discounted reward MDPs, 175
 - SMDPs, 191
- vector, 9, 288, 289
- vector spaces, 288
- Widrow-Hoff algorithm, 69
- yield management, 412