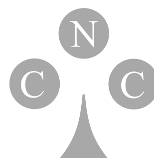
Natural Computing Series

Anthony Brabazon
Michael O'Neill
Seán McGarraghy

Natural Computing Algorithms

Natural Computing Series



Series Editors: G. Rozenberg
Th. Bäck A.E. Eiben J.N. Kok H.P. Spaink

Leiden Center for Natural Computing

Advisory Board: S. Amari G. Brassard K.A. De Jong C.C.A.M. Gielen T. Head L. Kari L. Landweber T. Martinetz Z. Michalewicz M.C. Mozer E. Oja G. Păun J. Reif H. Rubin A. Salomaa M. Schoenauer H.-P. Schwefel C. Torras D. Whitley E. Winfree J.M. Zurada

More information about this series at http://www.springer.com/series/4190

Anthony Brabazon • Michael O'Neill
Seán McGarraghy

Natural Computing Algorithms

Anthony Brabazon
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

Michael O'Neill
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

Seán McGarraghy
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

Series Editors
G. Rozenberg (Managing Editor)

Th. Bäck, J.N. Kok, H.P. Spaink
Leiden Center for Natural Computing
Leiden University
Leiden, The Netherlands

A.E. Eiben
VU University Amsterdam
The Netherlands

ISSN 1619-7127
Natural Computing Series
ISBN 978-3-662-43630-1   ISBN 978-3-662-43631-8 (eBook)
DOI 10.1007/978-3-662-43631-8

Library of Congress Control Number: 2015951433

Springer Heidelberg New York Dordrecht London
© Springer-Verlag Berlin Heidelberg 2015
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer-Verlag GmbH Berlin Heidelberg is part of Springer Science+Business Media (www.springer.com)

To Maria, Kevin and Rose
Tony

To Gráinne, Aoife, Michael, Caoimhe, my father John & to the memory of my mother Jane
Mike

To Milena, Martin and Alex, and my mother Mary; and to the memory of my father Michael
Seán

Preface

The field of natural computing has been the focus of a substantial research effort in recent decades. One particular strand of this concerns the development of computational algorithms using metaphorical inspiration from systems and phenomena that occur in the natural world. These naturally inspired computing algorithms have proven to be successful problem solvers across domains as varied as management science, telecommunications, business analytics, bioinformatics, finance, marketing, engineering, architecture and design, to name but a few. This book provides a comprehensive introduction to natural computing algorithms.

The book is divided into eight main parts, each of which provides an integrated discussion of a range of related natural computing algorithms. The first part covers a family of algorithms which are inspired by processes of evolution (evolutionary computing) as well as introducing pivotal concepts in the design of natural computing algorithms such as choice of representation, diversity generation mechanisms, and the selection of an appropriate fitness function. The second part illustrates a selection of algorithms which are inspired by the social behaviour of individuals (social computing) ranging from flocking behaviours to the food foraging behaviours of several organisms, including bats, insects and bacteria. The third part introduces a number of algorithms whose workings are inspired by the operation of our central nervous system (neurocomputing). The fourth part of the book discusses optimisation and classification algorithms which are metaphorically derived from the workings of our immune system (immunocomputing). The fifth part provides an introduction to developmental and grammatical computing, where the creation of a model or structure results from a development process which is typically governed by a set of rules, a 'grammar'. Physical computing is described in the sixth part of the book, with the primary emphasis being placed on the use of quantum inspired representations in natural computing. Two emerging paradigms in natural computing, chemical computing and plant inspired algorithms are introduced in part seven of the book. The closing chapter of the book looks towards the future of natural computing algorithms.

Of course, given the diverse range of natural computing algorithms which have been developed, we have had to make difficult decisions as to which algorithms to include and which to omit from our coverage. We have generally focussed on the better known algorithms in each part, supplemented by some less known algorithms which we personally found interesting!

This book will be of particular interest to academics and practitioners in computer science, informatics, management science and other application domains who wish to learn more about natural computing. The book is also a suitable accompaniment for a course on natural computing and could be used with graduate and final-year undergraduate students. More generally, the book will be of interest to anyone interested in natural computing approaches to optimisation, clustering, classification, prediction and model induction.

In order to make the book as accessible as possible, no prior knowledge of natural computing is assumed; nor do we assume that the reader has an extensive background in mathematics, optimisation or statistics. In discussing each family of algorithms we concentrate on providing a clear description of the main components of each algorithm and we also include material on the relevant natural background so that readers can fully appreciate the hidden computation that takes place in nature. As the book is intended to provide an introduction to a wide range of algorithms we do not focus on an exhaustive theoretical discussion concerning each of the algorithmic families, as a comprehensive discussion of any of them would require a book in its own right.

Anthony Brabazon
Michael O'Neill
Dublin, March 2015 *Seán McGarraghy*

Acknowledgment

When first conceived, we had little idea that the project of writing this book would take nearly eight years to complete. Over the course of that time, the field of natural computing has seen multiple advances and many people have consequently influenced this work. In particular, we would like to acknowledge a number of people who have directly, and indirectly through their encouragement, contributed to this project.

The greatest single influence has come from our students over the years, both those taking courses that we have taught, and our graduate research students. Students on the Natural Computing (COMP30290), Natural Computing Applications (COMP41190), Numerical Analytics and Software (MIS40530), Network Software Modelling (MIS40550), Data Mining & Applications (MIS40970), and Advanced Specialist Course (Finance) (FIN40960) modules at University College Dublin helped to road test material from the book and many changes were made as a result of their feedback.

The countless conversations and interactions with members of the UCD Natural Computing Research & Applications Group (http://ncra.ucd.ie) have continually challenged and informed our understanding of the world of natural computing and we wish to thank all members (past and present) of this amazing team of researchers who have kept us inspired.

We would also like to thank the School of Business at University College Dublin (http://www.smurfitschool.ie/), and our colleagues in the School, for their support during this project. The Complex & Adaptive Systems Laboratory (http://casl.ucd.ie) at University College Dublin also provided support for this project by fostering the interdisciplinary environment in which our group collaborates and through the provision of physical space for our research team. Anthony Brabazon and Michael O'Neill also acknowledge the support of their research activities provided by Science Foundation Ireland (Grant number 08/SRC/FM1389 – Financial Mathematics and Computation Research Cluster, Grant number 08/IN.1/I1868 – Evolution in Dynamic Environments with Grammatical Evolution (EDGE), and Grant number 13/IA/1850 – Applications of Evolutionary Design (AppED)).

We are very grateful to the anonymous referees for their thorough review of drafts of this book and for their insightful, thought provoking comments. They have notably strengthened this work. For their proof reading, we acknowledge the assistance of Drs. Eoin Murphy, Miguel Nicolau and James McDermott, and we also acknowledge the assistance of Dr. Wei Cui for the production of some of the diagrams in the book. Despite the valiant efforts of all the proof readers and ourselves, errors and omissions will undoubtedly occur, and we accept responsibility for these.

We thank Diego Perez and Miguel Nicolau for their research on the application of Grammatical Evolution (GE) to the Mario AI challenge, which was adapted as a working example of a GE mapping in Chap. 19. We thank Margaret O'Connor for sharing her knowledge on plants, and Owen, Niall and Peggy for listening and keeping Mike sane and firmly grounded at all times. Thanks also to those who in their various ways encouraged our journey into natural computing including Eamonn Walsh and Robin Matthews.

We especially extend our thanks to Ronan Nugent of Springer-Verlag for his on-going support of this project, for his advice on drafts of the manuscript, and especially for his patience with a project that took far longer than we originally planned.

We each extend a special thank you to our families. Without your love, support, patience, and understanding, this project would never have been completed.

Anthony Brabazon
Michael O'Neill
Seán McGarraghy

Contents

1 Introduction . . . 1
1.1 Natural Computing Algorithms: An Overview . . . 2
1.1.1 Biologically Inspired Algorithms . . . 2
1.1.2 Families of Naturally Inspired Algorithms . . . 9
1.1.3 Physically Inspired Algorithms . . . 10
1.1.4 Plant Inspired Algorithms . . . 11
1.1.5 Chemically Inspired Algorithms . . . 11
1.1.6 A Unified Family of Algorithms . . . 11
1.1.7 How Much Natural Inspiration? . . . 12
1.2 Structure of the Book . . . 12

Part I Evolutionary Computing

2 Introduction to Evolutionary Computing . . . 17
2.1 Evolutionary Algorithms . . . 18

3 Genetic Algorithm . . . 21
3.1 Canonical Genetic Algorithm . . . 21
3.1.1 A Simple GA Example . . . 23
3.2 Design Choices in Implementing a GA . . . 24
3.3 Choosing a Representation . . . 25
3.3.1 Genotype to Phenotype Mapping . . . 26
3.3.2 Genotype Encodings . . . 26
3.3.3 Representation Choice and the Generation of Diversity . 28
3.4 Initialising the Population . . . 29
3.5 Measuring Fitness . . . 29
3.6 Generating Diversity . . . 31
3.6.1 Selection Strategy . . . 31
3.6.2 Mutation and Crossover . . . 35
3.6.3 Replacement Strategy . . . 39

3.7 Choosing Parameter Values 40
3.8 Summary 41

4 Extending the Genetic Algorithm 43
4.1 Dynamic Environments 43
4.1.1 Strategies for Dynamic Environments 44
4.1.2 Diversity 44
4.2 Structured Population GAs 48
4.3 Constrained Optimisation 50
4.4 Multiobjective Optimisation 53
4.5 Memetic Algorithms 57
4.6 Linkage Learning 59
4.7 Estimation of Distribution Algorithms 61
4.7.1 Population-Based Incremental Learning 62
4.7.2 Univariate Marginal Distribution Algorithm 63
4.7.3 Compact Genetic Algorithm 65
4.7.4 Bayesian Optimisation Algorithm 66
4.8 Summary 71

5 Evolution Strategies and Evolutionary Programming 73
5.1 The Canonical ES Algorithm 74
5.1.1 $(1 + 1)$-ES 74
5.1.2 $(\mu + \lambda)$-ES and (μ, λ)-ES 75
5.1.3 Mutation in ES 75
5.1.4 Adaptation of the Strategy Parameters 76
5.1.5 Recombination 78
5.2 Evolutionary Programming 80
5.3 Summary 82

6 Differential Evolution 83
6.1 Canonical Differential Evolution Algorithm 83
6.2 Extending the Canonical DE Algorithm 88
6.2.1 Selection of the Base Vector 88
6.2.2 Number of Vector Differences 88
6.2.3 Alternative Crossover Rules 89
6.2.4 Other DE Variants 89
6.3 Discrete DE 90
6.4 Summary 92

7 Genetic Programming 95
7.1 Genetic Programming 95
7.1.1 GP Algorithm 97
7.1.2 Function and Terminal Sets 98
7.1.3 Initialisation Strategy 100
7.1.4 Diversity-Generation in GP 102

7.2 Bloat in GP . . . 105
7.3 More Complex GP Architectures . . . 105
7.3.1 Functions . . . 105
7.3.2 ADF Mutation and Crossover . . . 108
7.3.3 Memory . . . 108
7.3.4 Looping . . . 109
7.3.5 Recursion . . . 111
7.4 GP Variants . . . 112
7.4.1 Linear and Graph GP . . . 112
7.4.2 Strongly Typed GP . . . 112
7.4.3 Grammar-Based GP . . . 112
7.5 Semantics and GP . . . 113
7.6 Summary . . . 113

Part II Social Computing

8 Particle Swarm Algorithms . . . 117
8.1 Social Search . . . 118
8.2 Particle Swarm Optimisation Algorithm . . . 118
8.2.1 Velocity Update . . . 120
8.2.2 Velocity Control . . . 123
8.2.3 Neighbourhood Structure . . . 124
8.3 Comparing PSO and Evolutionary Algorithms . . . 125
8.4 Maintaining Diversity in PSO . . . 127
8.4.1 Simple Approaches to Maintaining Diversity . . . 129
8.4.2 Predator–Prey PSO . . . 130
8.4.3 Charged Particle Swarm . . . 132
8.4.4 Multiple Swarms . . . 134
8.4.5 Speciation-Based PSO . . . 135
8.5 Hybrid PSO Algorithms . . . 136
8.6 Discrete PSO . . . 137
8.6.1 BinPSO . . . 137
8.6.2 Angle-Modulated PSO . . . 138
8.7 Evolving a PSO Algorithm . . . 139
8.8 Summary . . . 139

9 Ant Algorithms . . . 141
9.1 A Taxonomy of Ant Algorithms . . . 142
9.2 Ant Foraging Behaviours . . . 142
9.3 Ant Algorithms for Discrete Optimisation . . . 144
9.3.1 Graph structure . . . 144
9.3.2 Ant System . . . 147
9.3.3 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System . . . 151
9.3.4 Ant Colony System . . . 152

9.3.5 Ant Multitour Systems 153
9.3.6 Dynamic Optimisation 154
9.4 Ant Algorithms for Continuous Optimisation 155
9.5 Multiple Ant Colonies 157
9.6 Hybrid Ant Foraging Algorithms 159
9.7 Ant-Inspired Clustering Algorithms 160
9.7.1 Deneubourg Model 161
9.7.2 Lumer and Faieta Model 162
9.7.3 Critiquing Ant Clustering 166
9.8 Classification with Ant Algorithms 167
9.9 Evolving an Ant Algorithm 169
9.10 Summary 170

10 Other Foraging Algorithms 171
10.1 Honeybee Dance Language 171
10.2 Honeybee Foraging 172
10.2.1 The Honeybee Recruitment Dance 172
10.3 Designing a Honeybee Foraging Optimisation Algorithm 173
10.3.1 Bee System Algorithm 174
10.3.2 Artificial Bee Colony Algorithm 175
10.3.3 Honeybee Foraging and Dynamic Environments 178
10.4 Bee Nest Site Selection 180
10.4.1 Bee Nest Site Selection Optimisation Algorithm 182
10.5 Honeybee Mating Optimisation Algorithm 184
10.6 Summary 186

11 Bacterial Foraging Algorithms 187
11.1 Bacterial Behaviours 187
11.1.1 Quorum Sensing 187
11.1.2 Sporulation 188
11.1.3 Mobility 188
11.2 Chemotaxis in E. Coli Bacteria 189
11.3 Bacterial Foraging Optimisation Algorithm 190
11.3.1 Basic Chemotaxis Model 191
11.3.2 Chemotaxis Model with Social Communication 192
11.4 Dynamic Environments 198
11.5 Classification Using a Bacterial Foraging Metaphor 198
11.6 Summary 199

12 Other Social Algorithms 201
12.1 Glow Worm Algorithm 201
12.2 Bat Algorithm 206
12.2.1 Bat Vocalisations 206
12.2.2 Algorithm 207
12.2.3 Discussion 210

12.3 Fish School Algorithm 211
12.3.1 Fish School Search 212
12.3.2 Summary 214
12.4 Locusts 215
12.4.1 Locust Swarm Algorithm 216
12.5 Summary 218

Part III Neurocomputing

13 Neural Networks for Supervised Learning 221
13.1 Biological Inspiration for Neural Networks 221
13.2 Artificial Neural Networks 222
13.2.1 Neural Network Architectures 222
13.3 Structure of Supervised Neural Networks 224
13.3.1 Activation and Transfer Functions 226
13.3.2 Universal Approximators 228
13.4 The Multilayer Perceptron 228
13.4.1 MLP Transfer Function 230
13.4.2 MLP Activation Function 230
13.4.3 The MLP Projection Construction and Response Regions 231
13.4.4 Relationship of MLPs to Regression Models 233
13.4.5 Training an MLP 234
13.4.6 Overtraining 237
13.4.7 Practical Issues in Modelling with and Training MLPs . 239
13.4.8 Stacking MLPs 243
13.4.9 Recurrent Networks 244
13.5 Radial Basis Function Networks 246
13.5.1 Kernel Functions 246
13.5.2 Radial Basis Functions 247
13.5.3 Intuition Behind Radial Basis Function Networks 248
13.5.4 Properties of Radial Basis Function Networks 249
13.5.5 Training Radial Basis Function Networks 250
13.5.6 Developing a Radial Basis Function Network 251
13.6 Support Vector Machines 252
13.6.1 SVM Method 258
13.6.2 Issues in Applications of SVM 258
13.7 Summary 259

14 Neural Networks for Unsupervised Learning 261
14.1 Self-organising Maps 262
14.2 SOM Algorithm 264
14.3 Implementing a SOM Algorithm 266
14.4 Classification with SOMs 271

14.5 Self-organising Swarm 272
14.6 SOSwarm and SOM 275
14.7 Adaptive Resonance Theory 276
14.7.1 Unsupervised Learning for ART 277
14.7.2 Supervised Learning for ARTs 279
14.7.3 Weaknesses of ART Approaches 279
14.8 Summary 279

15 Neuroevolution 281
15.1 Direct Encodings 282
15.1.1 Evolving Weight Vectors 282
15.1.2 Evolving the Selection of Inputs 283
15.1.3 Evolving the Connection Structure 283
15.1.4 Other Hybrid MLP Algorithms 286
15.1.5 Problems with Direct Encodings 287
15.2 NEAT 289
15.2.1 Representation in NEAT 290
15.2.2 Diversity Generation in NEAT 291
15.2.3 Speciation 292
15.2.4 Incremental Evolution 296
15.3 Indirect Encodings 297
15.4 Other Hybrid Neural Algorithms 297
15.5 Summary 297

Part IV Immunocomputing

---

16 Artificial Immune Systems 301
16.1 The Natural Immune System 302
16.1.1 Components of the Natural Immune System 302
16.1.2 Innate Immune System 302
16.1.3 Adaptive Immune System 304
16.1.4 Danger Theory 309
16.1.5 Immune Network Theory 309
16.1.6 Optimal Immune Defence 310
16.2 Artificial Immune Algorithms 310
16.3 Negative Selection Algorithm 310
16.4 Dendritic Cell Algorithm 315
16.5 Clonal Expansion and Selection Inspired Algorithms 320
16.5.1 CLONALG Algorithm 320
16.5.2 B Cell Algorithm 322
16.5.3 Real-Valued Clonal Selection Algorithm 323
16.5.4 Artificial Immune Recognition System 325
16.6 Immune Programming 330
16.7 Summary 331

Part V Developmental and Grammatical Computing

17 An Introduction to Developmental and Grammatical Computing . . . 335
17.1 Developmental Computing . . . 335
17.2 Grammatical Computing . . . 336
17.3 What Is a Grammar? . . . 337
17.3.1 Types of Grammar . . . 338
17.3.2 Formal Grammar Notation . . . 340
17.4 Grammatical Inference . . . 341
17.5 Lindenmayer Systems . . . 341
17.6 Summary . . . 343

18 Grammar-Based and Developmental Genetic Programming . . . 345
18.1 Grammar-Guided Genetic Programming . . . 346
18.1.1 Other Grammar-Based Approaches to GP . . . 351
18.2 Developmental GP . . . 351
18.2.1 Genetic L-System Programming . . . 351
18.2.2 Binary GP . . . 352
18.2.3 Cellular Encoding . . . 354
18.2.4 Analog Circuits . . . 354
18.2.5 Other Developmental Approaches to GP . . . 354
18.3 Summary . . . 356

19 Grammatical Evolution . . . 357
19.1 A Primer on Gene Expression . . . 358
19.2 Extending the Biological Analogy to GE . . . 360
19.3 Example GE Mapping . . . 361
19.4 Search Engine . . . 368
19.4.1 Genome Encoding . . . 368
19.4.2 Mutation and Crossover Search Operators . . . 368
19.4.3 Modularity . . . 370
19.4.4 Search Algorithm . . . 371
19.5 Genotype–Phenotype Map . . . 372
19.6 Grammars . . . 372
19.7 Summary . . . 373

20 Tree-Adjoining Grammars and Genetic Programming . . . 375
20.1 Tree-Adjoining Grammars . . . 377
20.2 TAG3P . . . 377
20.3 Developmental TAG3P . . . 379
20.4 TAGE . . . 379
20.5 Summary . . . 381

21 Genetic Regulatory Networks 383
21.1 Artificial Gene Regulatory Model for Genetic Programming ... 383
21.1.1 Model Output 385
21.2 Differential Gene Expression 386
21.3 Artificial GRN for Image Compression 389
21.4 Summary 389

Part VI Physical Computing

22 An Introduction to Physically Inspired Computing 393
22.1 A Brief Physics Primer 393
22.1.1 A Rough Taxonomy of Modern Physics 393
22.2 Classical Mechanics 395
22.2.1 Energy and Momentum 395
22.2.2 The Hamiltonian 396
22.3 Thermodynamics 398
22.3.1 Statistical Mechanics 400
22.3.2 Ergodicity 402
22.4 Quantum Mechanics 402
22.4.1 Observation in Quantum Mechanics 403
22.4.2 Entanglement and Decoherence 404
22.4.3 Noncommuting Operators 405
22.4.4 Tunnelling 406
22.4.5 Quantum Statistical Mechanics 406
22.5 Quantum Computing 407
22.5.1 Two-State Systems and Qubits 407
22.5.2 Digital Quantum Computers 408
22.5.3 Quantum Information 409
22.5.4 Adiabatic Quantum Computation 410
22.6 Annealing and Spin Glasses 411
22.6.1 Ising Spin Glasses 412
22.6.2 Quantum Spin Glasses 414
22.7 Summary 415

23 Physically Inspired Computing Algorithms 417
23.1 Simulated Annealing 417
23.1.1 Search and Neighbourhoods 419
23.1.2 Acceptance of 'Bad' Moves 419
23.1.3 Parameterisation of SA 420
23.1.4 Extensions of SA 421
23.1.5 Concluding Remarks 421
23.2 Simulated Quantum Annealing 422
23.2.1 Implementation of SQA 424
23.2.2 SQA Application to TSP-Type Problems 424

23.3 Constrained Molecular Dynamics Algorithm ... 426
23.4 Physical Field Inspired Algorithms ... 429
23.4.1 Central Force Optimisation ... 429
23.4.2 Gravitational Search Algorithm and Variants ... 431
23.4.3 Differences Among Physical Field-Inspired Algorithms ... 435
23.5 Extremal Optimisation Algorithm ... 436
23.6 Summary ... 437

24 Quantum Inspired Evolutionary Algorithms ... 439
24.1 Qubit Representation ... 439
24.2 Quantum Inspired Evolutionary Algorithms (QIEAs) ... 440
24.3 Binary-Valued QIEA ... 440
24.3.1 Diversity Generation in Binary QIEA ... 443
24.4 Real-Valued QIEA ... 446
24.4.1 Initialising the Quantum Population ... 446
24.4.2 Observing the Quantum Chromosomes ... 447
24.4.3 Crossover Mechanism ... 448
24.4.4 Updating the Quantum Chromosomes ... 449
24.5 QIEAs and EDAs ... 449
24.6 Other Quantum Hybrid Algorithms ... 450
24.7 Summary ... 452

Part VII Other Paradigms

25 Plant-Inspired Algorithms ... 455
25.1 Plant Behaviours ... 455
25.2 Foraging ... 456
25.2.1 Plant Movement and Foraging ... 457
25.2.2 Root Foraging ... 459
25.2.3 Predatory Plants ... 461
25.3 Plant-Level Coordination ... 462
25.4 A Taxonomy of Plant-Inspired Algorithms ... 464
25.5 Plant Propagation Algorithms ... 464
25.5.1 Invasive Weed Optimisation Algorithm ... 464
25.5.2 Paddy Field Algorithm ... 467
25.5.3 Strawberry Plant Algorithm ... 468
25.6 Plant Growth Simulation Algorithm ... 469
25.6.1 The Algorithm ... 472
25.6.2 Variants on the Plant Growth Simulation Algorithm ... 474
25.7 Root-Swarm Behaviour ... 475
25.7.1 Modelling Root Growth in Real Plants ... 476
25.7.2 Applying the Root-Swarm Metaphor for Optimisation ... 476
25.8 Summary ... 477

26 Chemically Inspired Algorithms . . . 479
26.1 A Brief Chemistry Primer . . . 479
26.2 Chemically Inspired Algorithms . . . 481
26.2.1 Chemical Reaction Optimisation (CRO) . . . 481
26.2.2 Artificial Chemical Reaction Optimisation Algorithm (ACROA) . . . 482
26.3 The CRO Algorithm . . . 483
26.3.1 Potential and Kinetic Energy and the Buffer . . . 483
26.3.2 Types of Collision and Reaction . . . 484
26.3.3 The High-Level CRO Algorithm . . . 485
26.3.4 On-wall Ineffective Collision . . . 489
26.3.5 Decomposition . . . 489
26.3.6 Intermolecular Ineffective Collision . . . 490
26.3.7 Synthesis . . . 492
26.4 Applications of CRO . . . 493
26.5 Discussion of CRO . . . 494
26.5.1 Potential Future Avenues for Research . . . 496
26.6 Summary . . . 498

Part VIII The Future of Natural Computing Algorithms

27 Looking Ahead . . . 501
27.1 Open Issues . . . 501
27.1.1 Hybrid Algorithms . . . 501
27.1.2 The Power and the Dangers of Metaphor . . . 502
27.1.3 Benchmarks and Scalability . . . 502
27.1.4 Usability and Parameter-Free Algorithms . . . 503
27.1.5 Simulation and Knowledge Discovery . . . 503
27.2 Concluding Remarks . . . 503

References . . . 505

Index . . . 547

1

Introduction

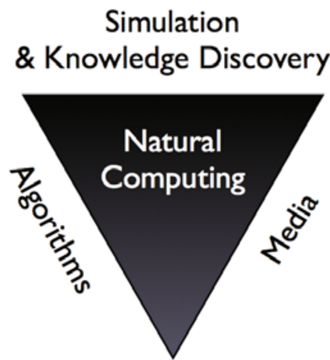


Fig. 1.1. The three facets of natural computing: 1) natural systems as computational media (e.g., DNA and molecular computing), 2) simulation of natural systems with the potential for knowledge discovery, and 3) algorithms inspired by the natural world

Although there is no unique definition of the term natural computing, most commonly the field is considered to consist of three main strands of enquiry: see Fig. 1.1. The first strand concerns the use of natural materials and phenomena for computational purposes such as DNA and molecular computing (*computing in vivo*); the second strand concerns the application of computer simulations to replicate natural phenomena in order to better understand those phenomena (e.g., artificial life, agent based modelling and computational biology); and the third strand, which forms the subject matter of this book, is concerned with the development of computational algorithms which draw their metaphorical inspiration from systems and phenomena that occur in the natural world. These algorithms can be applied to a multiplicity of

© Springer-Verlag Berlin Heidelberg 2015
A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_1

real-world problems including optimisation, classification, prediction, clustering, design and model induction. The objective of this book is to provide an introduction to a broad range of natural computing algorithms.

1.1 Natural Computing Algorithms: An Overview

Natural computing is inherently multidisciplinary as it draws inspiration from a diverse range of fields of study including mathematics, statistics, computer science and the natural sciences of biology, physics and chemistry. As our understanding of natural phenomena has deepened so too has our recognition that many mechanisms in the natural world parallel computational processes and can therefore serve as an inspiration for the design of problem-solving algorithms (defined simply as a set of instructions for solving a problem of interest). In this book we introduce and discuss a range of families of natural computing algorithms. Figure 1.2 provides a high-level taxonomy of the primary methodologies discussed in this book which are grouped under the broad umbrellas of evolutionary computing, social computing, neurocomputing, immunocomputing, developmental and grammatical computing, physical computing, and chemical computing.

1.1.1 Biologically Inspired Algorithms

An interesting aspect of biological systems at multiple levels of scale (ecosystems, humans, bacteria, cells, etc.) which suggests that they may provide good sources of inspiration for solving real-world problems is that the very process of survival is itself a challenging problem! In order to survive successfully organisms need to be able to find resources such as food, water, shelter and mates, whilst simultaneously avoiding predators. It is plausible that mechanisms which have evolved in order to assist survivability of organisms in these settings, such as sensing, communication, cognition and mobility could prove particularly useful in inspiring the design of computational algorithms. Virtually all biological systems exist in high-dimensional (i.e., many factors impact on their survival), dynamic environments and hence biologically inspired algorithms may be of particular use in problem solving in these conditions.

Although many aspects of biological systems are noteworthy, a few characteristics of biological systems which provide food for thought in the design of biologically inspired algorithms include: the importance of the population; the emphasis on robustness (survival) rather than on optimality; and the existence of multilevel adaptive processes.

Populational Perspective

In many biologically inspired algorithms the search for good solutions takes place within a *population* of potential solutions. As in biological settings, indi-

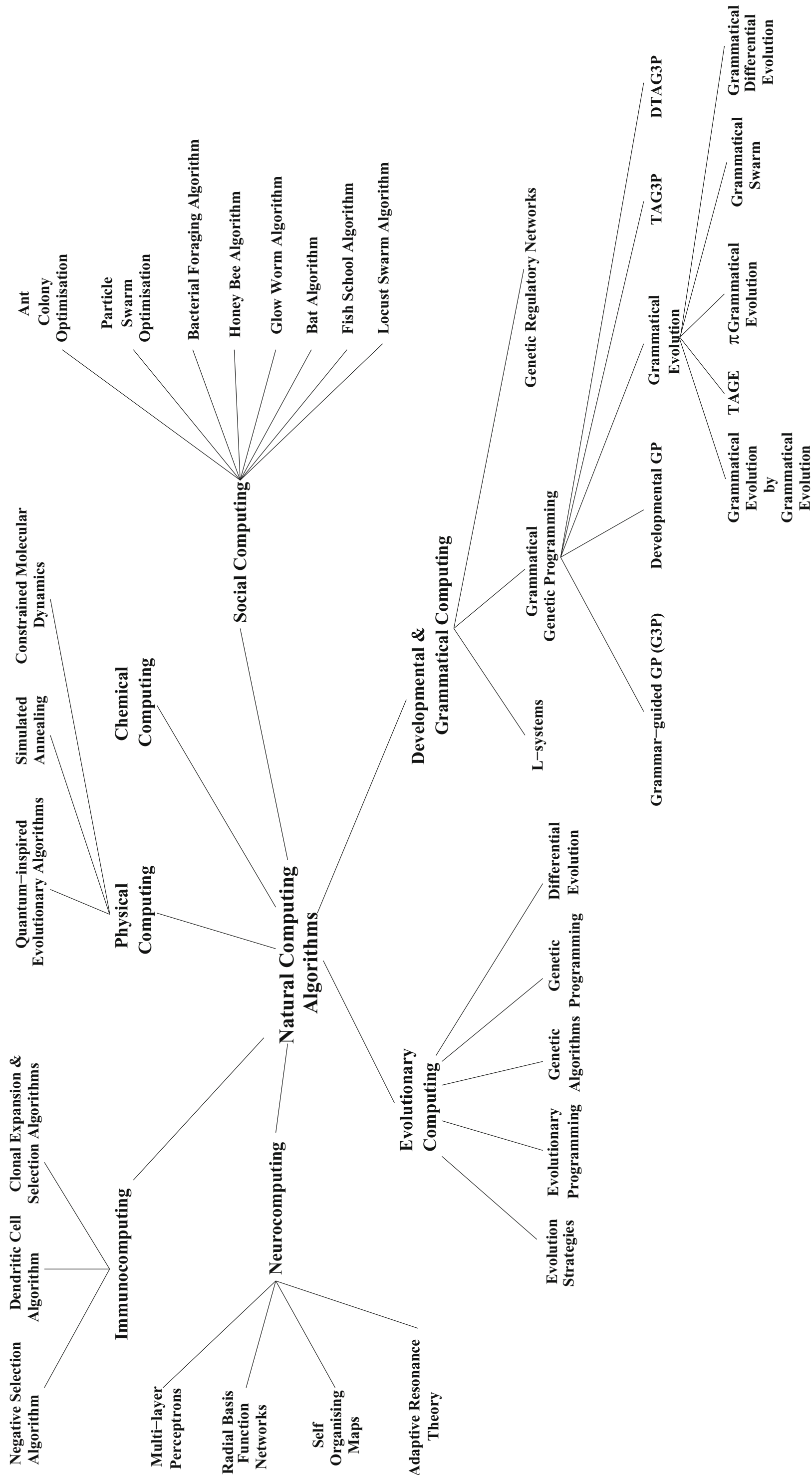


Fig. 1.2. A taxonomy of the nature inspired algorithms discussed in this book

viduals in a population can be considered an *individual hypothesis* (or learning trial) in the game of survival. From a species point of view, maintaining a dispersed population of individuals reduces the chance that environmental change will render the entire species extinct.

In contrast, in many traditional (non-natural-computing) optimisation algorithms, the paradigm is to generate a single trial solution and then iteratively improve it. Consider for example, a simple random hill-climbing optimisation algorithm (Algorithm 1.1) where an individual solution is iteratively improved using a greedy search strategy. In greedy search, any change in a solution which makes it better is accepted. This implies that a hill-climbing algorithm can find a local, but not necessarily the global, optimum (Fig. 1.3). This makes the choice of starting point a critical one. Note that this is so for all hill-climbing algorithms, including those that exploit information found so far (e.g., Newton's algorithm uses slope information, while the Nelder–Mead algorithm does not need slopes but uses several previous points' objective function values).

Algorithm 1.1: Hill Climbing Algorithm

Randomly generate a solution x;
Calculate the objective function value $f(x)$ for the solution;

repeat
  Randomly mutate solution;
  if *new solution is better than the current solution* **then**
    Replace the current solution with the new one
  end
until *terminating condition*;

Many of the biologically inspired algorithms described in this book maintain and successively update a population of potential solutions, which in the ideal case provides a good coverage (or sampling) of the environment in which we are problem-solving, resulting in a form of parallel search. This of course assumes the process that generates the first population disperses the individuals in an appropriate manner so as to maximise coverage of the environment. Ideally as search progresses it might be desirable to maintain a degree of dispersion to avoid premature convergence of the population to local optima. It is the existence of a population which allows these bioinspired algorithms the potential to achieve global search characteristics, and avoid local optima through the populational dispersion of individuals.

Dispersion and Diversity

It is important to highlight this point that we should not confuse the notions of diversity (of objective function values) and dispersion. Often we

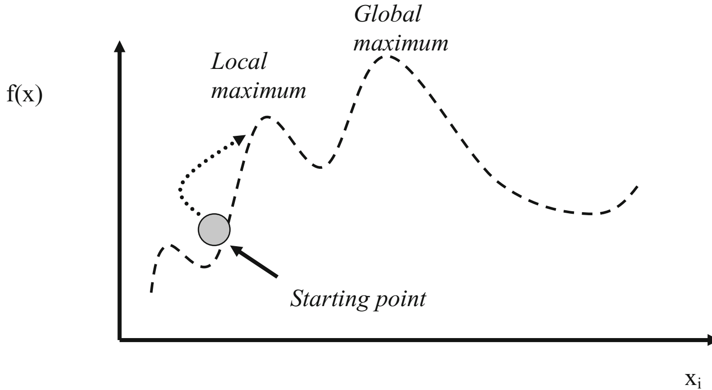


Fig. 1.3. A hill-climbing algorithm will find a local optimum. Due to its greedy search strategy it cannot then escape from this as it would require a 'downhill' move to traverse to the global optimum

(over)emphasise the value of diversity within a population. From a computational search perspective it is arguably more valuable to focus on the dispersion (or coverage) of the population. It is possible to have an abundance of diversity within a population; yet, at the same time, the population could be converged on a local optimum. However, a population which has a high value of dispersion is less likely to be converged in this manner. Figure 1.4 illustrates the difference between dispersion and diversity. The importance of dispersion is brought into sharp focus if we expose the population to a changing environment where the location of the optima might change/move over time. A population which maintains dispersion may have a better chance to adapt to an environment where the global optimum moves a relatively far distance from its current location.

Communication

Another critical aspect of most of the algorithms described in this book is that the members of the population do not search in isolation. Instead they can *communicate* information on the quality of their current (or their previous) solution to other members of the population. Communication is interpreted broadly here to mean exchange of information between members of the population, and so might take various forms: from chemical signals being left in the environment of a social computing algorithm, which can be sensed by individuals in the population; to the exchange of genes in an evolutionary algorithm. This information is then used to bias the search process towards areas of better solutions as the algorithm iterates.

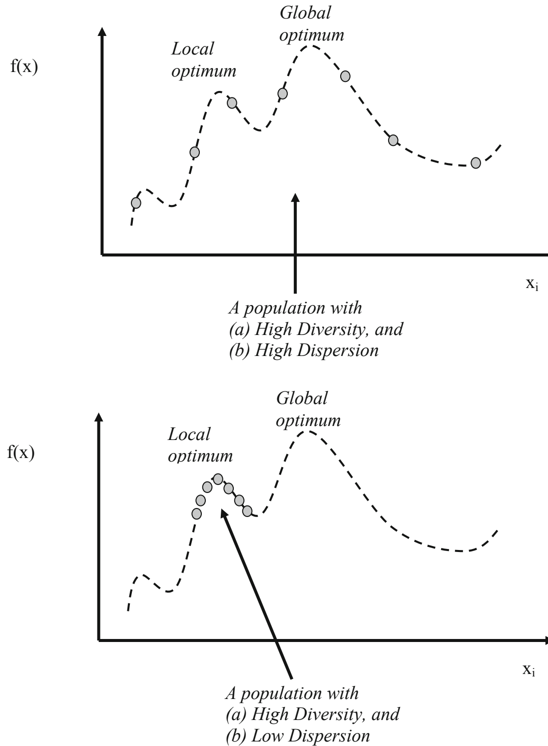


Fig. 1.4. An illustration of the difference between a dispersed (top) and a diverse (bottom) population. It is possible to have a large amount of diversity (e.g., a wide range of objective function values) but still be converged on a local optimum

Robustness

'Survival in a dynamic environment' is the primary aim in many biological systems. Therefore the implicit driver for organisms is typically to uncover and implement survival strategies which are 'good enough' for current conditions and which are *robust* to changing environmental conditions. Optimality is a fleeting concept in dynamic environments as (for example), the location of good food resources last week may not hold true next week.

Adaptiveness

Adaptiveness (new learning) occurs at multiple levels and timescales in biological systems, ranging from (relatively) slow genetic learning to (relatively) fast lifetime learning. The appropriate balance between speed of adaptation and the importance of memory (or *old learning*) depends on the nature of

the dynamic environment faced by the organism. The more dynamic the environment, the greater the need for adaptive capability and the less useful is memory.

An interesting model of adaptation in biological systems is outlined in Sipper et al. [581] and is commonly referred to as the POE model. This model distinguishes between three levels of organisation in biological systems:

i. phylogeny (P),
ii. ontogeny (O), and
iii. epigenesis (E).

Phylogeny concerns the adaptation of genetic code over time. As the genome adapts and differentiates, multiple species, or phylogeny, evolve. The primary mechanisms for generating diversity in genetic codes are mutation, and, in the case of sexual reproduction, recombination. The continual generation of diversity in genetic codings facilitates the survival of species, and the emergence of new species, in the face of changing environmental conditions. Much of the research around evolutionary computation to date exists along this axis of adaptation.

Ontogeny refers to the development of a multicellular organism from a zygote. While each cell maintains a copy of the original genome, it specialises to perform specific tasks depending on its surroundings (cellular differentiation). In recent years there has been increasing interest in developmental approaches to adaptation with the adoption of models such as artificial genetic regulatory networks.

Epigenesis is the development of systems which permit the organism to integrate and process large amounts of information from its environment. The development and working of these systems is not completely specified in the genetic code of the organism and hence are referred to as 'beyond the genetic' or epigenetic. Examples include the immune, the nervous and the endocrine systems. While the basic structure of these systems is governed by the organism's genetic code, they are modified throughout the organism's lifetime as a result of its interaction with the environment. For example, a human's immune system can maintain a memory of pathogens that it has been exposed to (the acquired immune system). The regulatory mechanism which controls the expression of genes is also subject to epigenetic interference. For example, chemical modification (e.g., through methylation) of regulatory regions of the genome can have the effect of silencing (turning off or dampening) the expression of a gene(s). The chemical modification can arise due to the environmental state in which the organism lives. So the environment can effect which genes are expressed (or not) thereby indirectly modifying an organism's genetic makeup to suit the conditions in which it finds itself.

In complex biological organisms, all three levels of organisation are interlinked. However, in assisting us in thinking about the design of biologically inspired algorithms, it can be useful to consider each level of organisation (and their associated adaptive processes) separately (Fig. 1.5). Summarising

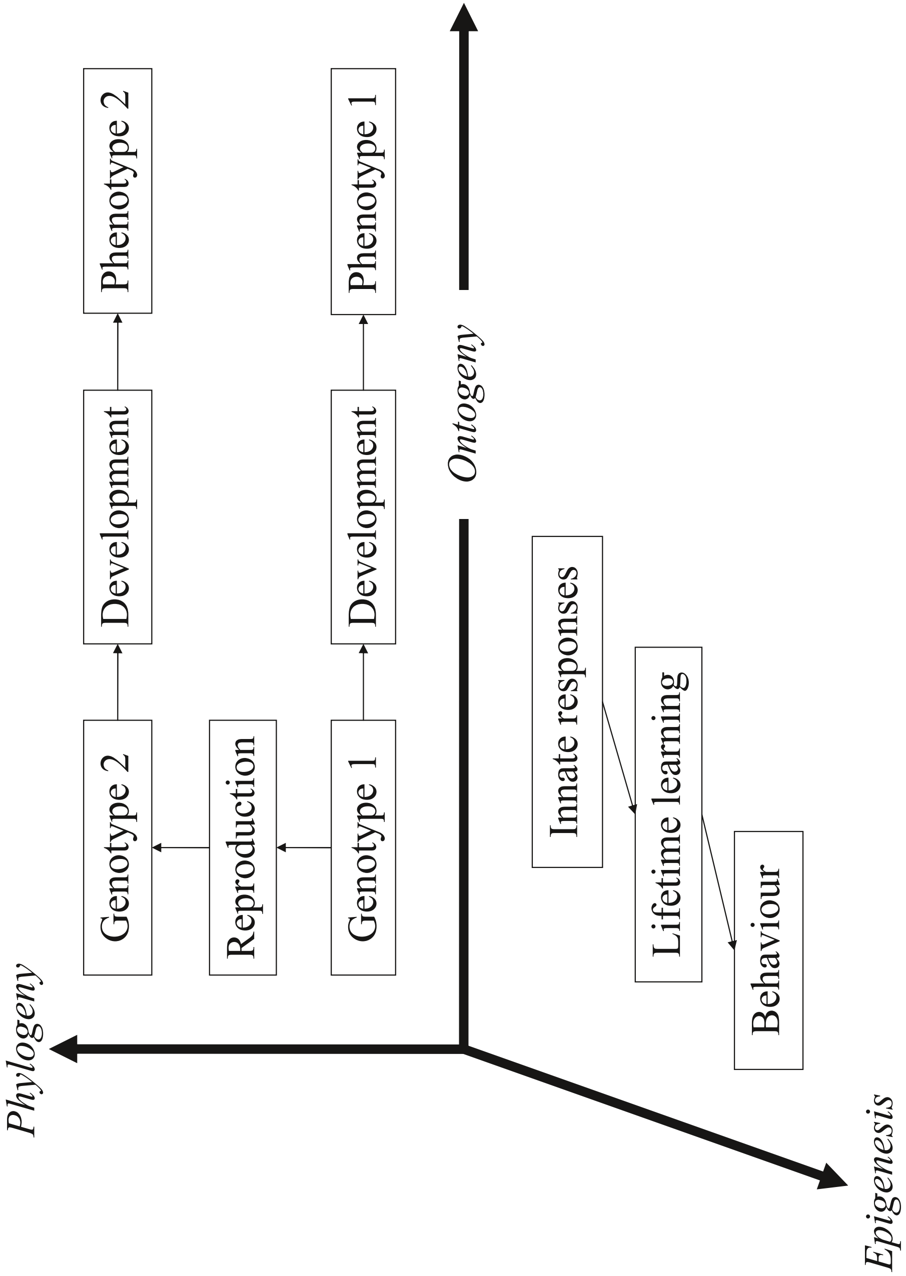


Fig. 1.5. Three levels of organisation in biological systems

the three levels, Sipper et al. [581] contends that they embed the ideas of *evolution*, structural *development* of an individual, and *learning* through environmental interactions.

Most biologically inspired algorithms draw inspiration from a single level of organisation but it is of course possible to design hybrid algorithms which draw inspiration from more than one level. For example, neuroevolution, discussed in Chap. 15, combines concepts of both evolutionary and lifetime learning, and evolutionary–development (evo–devo) approaches (e.g., see Chap. 21) hybridise phylogenetic and ontogenetic adaptation.

1.1.2 Families of Naturally Inspired Algorithms

A brief overview of some of the main families of natural computing algorithms is provided in the following paragraphs. A more detailed discussion of each of these is provided in later chapters.

Evolutionary Computing

Evolutionary computation simulates an evolutionary process on a computer in order to breed good solutions to a problem. The process draws high-level inspiration from biological evolution. Initially a population of potential solutions are generated (perhaps randomly), and these are iteratively improved over many simulated generations. In successive iterations of the algorithm, fitness based selection takes place within the population of solutions. Better solutions are preferentially selected for survival into the next generation of solutions, with diversity being introduced in the selected solutions in an attempt to uncover even better solutions over multiple generations. Algorithms that employ an evolutionary approach include genetic algorithms (GAs), evolutionary strategies (ES), evolutionary programming (EP) and genetic programming (GP). Differential evolution (DE) also draws (loose) inspiration from evolutionary processes.

Social Computing

The social models considered in this book are drawn from a *swarm* metaphor. Two popular variants of swarm models exist, those inspired by the flocking behaviour of birds and fish, and those inspired by the behaviour of social insects such as ants and honey bees. The swarm metaphor has been used to design algorithms which can solve difficult problems by creating a population of problem solvers, and allowing these to communicate their relative success in solving the problem to each other.

Neurocomputing

Artificial neural networks (NNs) comprise a modelling methodology whose inspiration arises from a simplified model of the workings of the human brain. NNs can be used to construct models for the purposes of prediction, classification and clustering.

Immunocomputing

The capabilities of the natural immune system are to recognise, destroy and remember an almost unlimited number of foreign bodies, and also to protect the organism from misbehaving cells in the body. Artificial immune systems (AIS) draw inspiration from the workings of the natural immune system to develop algorithms for optimisation and classification.

Developmental and Grammatical Computing

A significant recent addition to natural computing methodologies are those inspired by developmental biology (developmental computing) and the use of formal grammars (grammatical computing) from linguistics and computer science. In natural computing algorithms grammars tend to be used in a generative sense to construct sentences in the language specified by the grammar. This generative nature is compatible with a developmental approach, and consequently a significant number of developmental algorithms adopt some form of grammatical encoding. As will be seen there is also an overlap between these algorithms and evolutionary computation. In particular, a number of approaches to genetic programming adopt grammars to control the evolving executable structures. This serves to highlight the overlapping nature of natural systems, and that our decomposition of natural computing algorithms into families of inspiration is one of convenience.

1.1.3 Physically Inspired Algorithms

Just as biological processes can inspire the design of computational algorithms, inspiration can also be drawn from looking at physical systems and processes. We look at three algorithms which are inspired by the properties of interacting physical bodies such as atoms and molecules, namely simulated annealing, quantum annealing, and the constrained molecular dynamics algorithm. One interesting strand of research in this area is drawn from a quantum metaphor.

Quantum Inspired Algorithms

Quantum mechanics seeks to explain the behaviours of natural systems that are observed at very short time or distance scales. An example of a system is a

subatomic particle such as a free electron. Two important concepts underlying quantum systems are the *superposition of states* and *quantum entanglement*. Recent years have seen the development of a series of quantum inspired hybrid algorithms including quantum inspired evolutionary algorithms, social computing, neurocomputing and immunocomputing. A claimed benefit of these algorithms is that because they use a quantum inspired representation, they can potentially maintain a good balance between exploration and exploitation. It is also suggested that they could offer computational efficiencies.

1.1.4 Plant Inspired Algorithms

Plants represent some 99% of the eukaryotic biomass of the planet and have been highly successful in colonising many habitats with differing resource potential. Just like animals or simpler organisms such as bacteria (Chap. 11), plants have evolved multiple problem-solving mechanisms including complex food foraging mechanisms, environmental-sensing mechanisms, and reproductive strategies. Although plants do not have a brain or central nervous system, they are capable of sensing environmental conditions and taking actions which are 'adaptive' in the sense of allowing them to adjust to changing environmental conditions. These features of plants offer potential to inspire the design of computational algorithms and a recent stream of work has seen the development of a family of plant algorithms. We introduce a number of these algorithms and highlight some current areas of research in this subfield.

1.1.5 Chemically Inspired Algorithms

Chemical processes play a significant role in many of the phenomena described in this book, including (for example) evolutionary processes and the workings of the natural immune system. However, so far, chemical aspects of these processes have been largely ignored in the design of computational algorithms. An emerging stream of study is beginning to remedy this gap in the literature and we describe an optimisation algorithm inspired by the processes of chemical reactions.

1.1.6 A Unified Family of Algorithms

Although it is useful to compartmentalise the field of Natural Computing into different subfields, such as Evolutionary and Social Computing, for the purposes of introducing the material, it is important to emphasise that this does not actually reflect the reality of the natural world around us. In nature all of these learning mechanisms coexist and interact forming part of a larger natural, complex and adaptive system encompassing physical, chemical, evolutionary, immunological, neural, developmental, grammatical and social processes, which, for example, are embodied in mammals. In much the same

way that De Jong advocated for a unified field of Evolutionary Computation [144], we would favour a unification of all the algorithms inspired by the natural world into the paradigm of Natural Computing and Natural Computing Algorithms. Increasingly we are seeing significant overlaps between the different families of algorithms, and upon real-world application it is common to witness their hybridisation (e.g., neuroevolution, evo–devo etc.). We anticipate that the future of natural computing will see the integration of many of these seemingly different approaches into unified software systems working together in harmony.

1.1.7 How Much Natural Inspiration?

An obvious question when considering computational algorithms which are inspired by natural phenomena is how accurate does the metaphor need to be? We consider that the true measure of usefulness of a natural computing algorithm is not its degree of veracity with (what we know of) nature, but rather its effectiveness in problem solving; and that an intelligent designer of algorithms should incorporate ideas from nature — while omitting others — so long as these enhance an algorithm's problem-solving ability. For example, considering quantum inspired algorithms, unless we use quantum computers, which to date are experimental devices and not readily available to the general reader, it is not possible to efficiently simulate effects such as entanglement; hence such algorithms while drawing a degree of inspiration from quantum mechanics must of necessity omit important, even vital, features of the natural phenomenon from which we derive inspiration.

1.2 Structure of the Book

The field of natural computing has expanded greatly in recent years beyond its evolutionary and neurocomputing roots to encompass social, immune system, physical and chemical metaphors. Not all of the algorithms discussed in this book are fully explored as yet in terms of their efficiency and effectiveness. However, we have deliberately chosen to include a wide range of algorithms in order to illustrate the diversity of current research into natural computing algorithms.

The remainder of this book is divided into eight parts. Part I starts by providing an overview of evolutionary computation (Chap. 2), and then proceeds to describe the genetic algorithm (Chaps. 3 and 4), evolutionary strategies and evolutionary programming (Chap. 5), differential evolution (Chap. 6), and genetic programming (Chap. 7). Part II focusses on social computing and provides coverage of particle swarm optimisation (Chap. 8), insect algorithms (Chaps. 9 and 10), bacterial foraging algorithms (Chap. 11), and other social algorithms (Chap. 12). Part III of the book provides coverage of the main neurocomputing paradigms including supervised learning neural

network models such as the multilayer perceptron, recurrent networks, radial basis function networks and support vector machines (Chap. 13), unsupervised learning models such as self-organising maps (Chap. 14), and hybrid neuroevolutionary models (Chap. 15). Part IV discusses immunocomputing (Chap. 16). Part V of the book introduces developmental and grammatical computing in Chap. 17 and provides detailed coverage of grammar-based approaches to genetic programming in Chap. 18. Two subsequent chapters expose in more detail some of grammar-based genetic programming's more popular forms, grammatical evolution and TAG3P (Chaps. 19 and 20), followed by artificial genetic regulatory network algorithms in Chap. 21. Part VI introduces physically inspired computing (Chaps. 22 to 24). Part VII introduces some other paradigms that do not fit neatly into the earlier categories, namely, plant-inspired algorithms in Chap. 25, and chemically inspired computing in Chap. 26. Finally, Part VIII (Chap. 27) outlines likely avenues of future work in natural computing algorithms.

We hope the reader will enjoy this tour of natural computing algorithms as much as we have enjoyed the discovery (and in some cases rediscovery) of these inspiring algorithms during the writing of this book.

Part I

Evolutionary Computing

2

Introduction to Evolutionary Computing

'*Owing to this struggle for life, variations, however slight and from whatever cause proceeding, if they be in any degree profitable to the individuals of a species, in their infinitely complex relations to other organic beings and to their physical conditions of life, will tend to the preservation of such individuals, and will generally be inherited by the offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born, but a small number can survive. I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection'.* (Darwin, 1859 [127], p. 115)

Biological evolution performs as a powerful problem-solver that attempts to produce solutions that are at least *good enough* to perform the job of survival in the current environmental context. Since Charles Darwin popularised the theory of Natural Selection, the driving force behind evolution, molecular biology has unravelled some of the mysteries of the components that underpinned earlier evolutionary ideas. In the twentieth century molecular biologists uncovered the existence of DNA, its importance in determining hereditary traits and later its structure, unlocking the key to the genetic code. The accumulation of knowledge about the biological process of evolution, often referred to as neo-Darwinism, has in turn given inspiration to the design of a family of computational algorithms known collectively as *evolutionary computation.* These evolutionary algorithms take their cues from the biological concepts of natural selection and the fact that the heritable traits are physically encoded on DNA, and can undergo variation through a series of genetic operators such as mutation and crossover.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_2

2.1 Evolutionary Algorithms

Evolutionary processes represent an archetype, whose application transcends their biological root. Evolutionary processes can be distinguished by means of their four key characteristics, which are [57, 92]:

i. a population of entities,
ii. mechanisms for selection,
iii. retention of fit forms, and
iv. the generation of variety.

In biological evolution, species are positively or negatively selected depending on their relative success in surviving and reproducing in the environment. Differential survival, and variety generation during reproduction, provide the engine for evolution [127, 589] (Fig. 2.1).

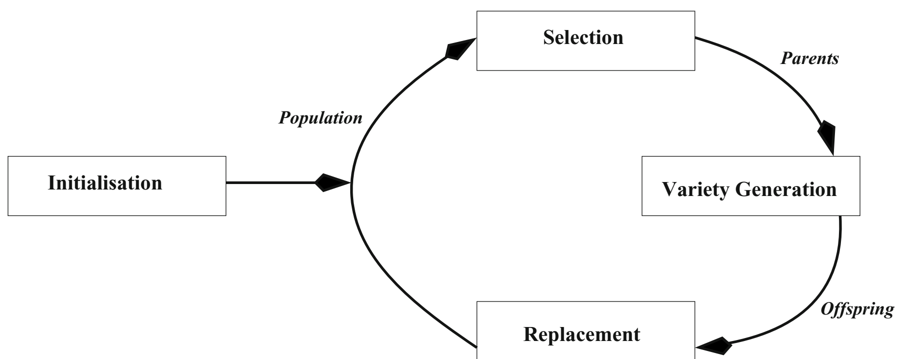


Fig. 2.1. Evolutionary cycle

These concepts have metaphorically inspired the field of evolutionary computation (EC). Algorithm 2.1 outlines the evolutionary meta-algorithm. There are many ways of operationalising each of the steps in this meta-algorithm; consequently, there are many different, but related, evolutionary algorithms. Just as in biological evolution, the selection step is a pivotal driver of the algorithm's workings. The selection step is biased in order to preferentially select better (or 'more fit') members of the current population. The generation of new individuals creates *offspring* or *children* which bear some similarity to their parents but are not identical to them. Hence, each individual represents a trial solution in the environment, with better individuals having increased chance of influencing the composition of individuals in future generations. This process can be considered as a 'search' process, where the objective is to continually improve the quality of individuals in the population.

Algorithm 2.1: Evolutionary Algorithm

Initialise the population of candidate solutions;
repeat
  Select individuals (parents) for breeding from the current population;
  Generate new individuals (offspring) from these parents;
  Replace some or all of the current population with the newly generated individuals;
until *terminating condition*;

Evolutionary Computation in Computer Science

The idea of a (computer) simulated evolutionary process dates back to the very dawn of digital computing, being introduced in the writings of Alan Turing in 1948–1950 [636, 637]. One of the earliest published works on the implementation of an evolutionary-like algorithm was by Friedberg in 1958 [203] and followed by [171, 204]. In the earlier of these studies, random and routine changes were made to binary strings representing machine code, with the performance of individual instructions being monitored in a credit assignment form of learning. Friedberg's studies were subsequently compiled into an edited collection providing a snapshot of the seminal papers that gave rise to the field of Evolutionary Computation [192]. Friedberg's work represents the origin of what is now known as Genetic Programming, which was later popularised via the work of Cramer [119], Dickmans et al. [157] and Koza [339] in the 1980s.

During the 1960s and 1970s two significant, independent, lines of research developing evolutionary algorithms were undertaken in Europe and the US. The Europeans (Rechenberg and Schwefel) developed Evolution Strategies [530, 531, 560, 561] and the Americans (Fogel et al. and Holland) developed Evolutionary Programming [194] and Genetic Algorithms [281]. More recently, Storn and Price have added Differential Evolution [600] to the family of evolutionary algorithms.

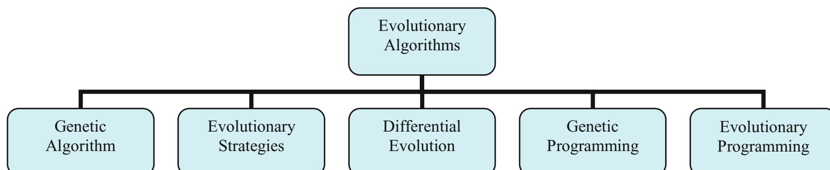


Fig. 2.2. Main branches of evolutionary computation

Although the above strands of research were initially distinct, with each having their own proponents, the lines between all of these evolutionary inspired approaches is becoming blurred with representations and strategies being used interchangeably between the various algorithms. As such, today it is common to use the term evolutionary algorithm to encompass all of the above approaches [144, 175]. Figure 2.2 provides a taxonomy of the more common branches of evolutionary computation.

In the following chapters in Part I of the book, we will introduce three of the main families of evolutionary inspired algorithms in turn, namely, the genetic algorithm (Chaps. 3 and 4), evolutionary strategies (Chap. 5), and differential evolution (Chap. 6). Following these, genetic programming is then discussed in Chap. 7.

3

Genetic Algorithm

While the development of the genetic algorithm (GA) dates from the 1960s, this family of algorithms was popularised by Holland in the 1970s [281]. The GA has been applied in two primary areas of research: optimisation, in which GAs represent a population-based optimisation algorithm, and the study of adaptation in complex systems, wherein the evolution of a population of adapting entities is simulated over time by means of a pseudonatural selection process using differential-fitness selection, and pseudogenetic operators to induce variation in the population.

In this chapter we introduce the canonical GA, focussing on its role as an optimising methodology, and discuss the design choices facing a modeller who is seeking to implement a GA.

3.1 Canonical Genetic Algorithm

In genetics, a strong distinction is drawn between the *genotype* and the *phenotype*; the former contains genetic information, whereas the latter is the physical manifestation of this information. Both play a role in evolution as the biological processes of diversity generation act on the genotype, while the 'worth' or *fitness* of this genotype in the environment depends on the survival and reproductive success of its corresponding phenotype. Similarly, in the canonical GA a distinction is made between the encoding of a solution (the 'genotype'), to which simulated genetic operators are applied, and the phenotype associated with that encoding. These phenotypes can have many diverse forms depending on the application of interest. Unlike traditional optimisation techniques the GA maintains and iteratively improves a *population* of solution encodings. Evolutionary algorithms, including the GA, can be broadly characterised as [193]:

$$x[t+1] = r(v(s(x[t]))) \tag{3.1}$$

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_3

where $x[t]$ is the population of encodings at timestep t, $v(.)$ is the random variation operator (crossover and mutation), $s(.)$ is the selection for mating operator, and $r(.)$ is the replacement selection operator. Once the initial population of encoded solutions has been obtained and evaluated, a reproductive process is applied in which the encodings corresponding to the better-quality, or *fitter*, solutions have a higher chance of being selected for propagation of their genes into the next generation. Over a series of generations, the better adapted solutions in terms of the given fitness function tend to flourish, and the poorer solutions tend to disappear. Just as biological genotypes encode the results of past evolutionary trials, the population of genotypes in the GA also encode a history (or memory) of the relative success of the resulting phenotypes for the problem of interest.

Therefore, the canonical GA can be described as an algorithm that turns one population of candidate encodings and their corresponding solutions into another using a number of stochastic operators. Selection exploits information in the current population, concentrating interest on high-fitness solutions. The selection process is biased in favour of the encodings corresponding to better/fitter solutions and better solutions may be selected multiple times. This corresponds to the idea of *survival of the fittest*. Crossover and mutation perturb these solutions in an attempt to uncover even better solutions. Mutation does this by introducing new gene values into the population, while crossover allows the recombination of fragments of existing solutions to create new ones. Algorithm 3.1 lists the key steps in the canonical genetic algorithm.

An important aspect of the algorithm is that the evolutionary process operates on the *encodings* of solutions, rather than directly on the solutions themselves. In determining the fitness of these encodings, they must first be translated into a solution to the problem of interest, the fitness of the solution determined, and finally this fitness is associated with the encoding (Fig. 3.1).

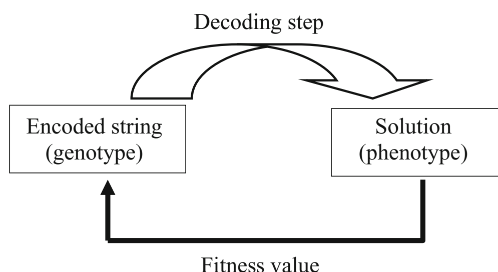


Fig. 3.1. Decoding of genotype into a solution in order to calculate fitness

Algorithm 3.1: Canonical Genetic Algorithm

Determine how the solution is to be encoded as a genotype and define the fitness function;
Create an initial population of genotypes;
Decode each genotype into a solution and calculate the fitness of each of the n solution candidates in the population;

repeat
  Select n members from the current population of encodings (the *parents*) in order to create a mating pool;
  repeat
    Select two parents randomly from the mating pool;
    With probability p_{cross}, perform a crossover process on the encodings of the selected parent solutions, to produce two new (*child*) solutions;
    Otherwise, crossover is not performed and the two children are simply copies of their parents;
    With probability p_{mut}, apply a mutation process to each element of the encodings of the two child solutions;
  until *n new child solutions have been created*;
  Replace the old population with the newly created one (this constitutes a generation);
until *terminating condition*;

3.1.1 A Simple GA Example

To provide additional insight into the workings of the canonical GA, a simple numerical example is now provided. Assume that candidate solutions are encoded as a binary string of length 8 and the fitness function $f(x)$ is defined as the number of 1s in the bit string (this is known as the *OneMax* problem). Let $n = 4$ with $p_{\text{cross}} = 0.6$ and $p_{\text{mut}} = 0.05$. Assume also that the initial population is generated randomly as in Table 3.1.

Table 3.1. A sample initial random population

Candidate	String	Fitness
A	10000110	3
B	01101100	4
C	10100000	2
D	01000110	3

Next, a selection process is applied based on the fitness of the candidate solutions. Suppose the first selection draws candidates B and D and the second

draws B and A. For each set of parents, the probability that a crossover (or recombination) operator is applied is p_{cross}. Assume that B and D are crossed over between bit position 1 and 2 to produce child candidates E and F (Table 3.2), and that crossover is not applied to B and A.

Table 3.2. Crossover applied to individuals B and D from Table 3.1, after the first element of each binary string, to produce the offspring E and F

Initial Parent	**Candidate B**	**Candidate D**
	0 1101100	0 1000110
Resulting Child	**Candidate E**	**Candidate F**
	0 1000110	0 1101100

Crossover is not applied to B and A; hence the child candidates (G and H) are clones of the two parent candidates (Table 3.3).

Table 3.3. No crossover is applied to B and D; hence the child candidates G and H are clones of their parents

Initial Parent	**Candidate B**	**Candidate A**
	01101100	10000110
Resulting Child	**Candidate G**	**Candidate H**
	01101100	10000110

Finally, the mutation operator is applied to each child candidate with probability p_{mut}. Suppose candidate E is mutated (to a 1) at the third locus, that candidate F is mutated (to a 1) at the seventh locus, and that no other mutations take place. The resulting new population is presented in Table 3.4. By biasing selection for reproduction towards more fit parents, the GA has increased the average fitness of the population in this example from 3 $(= \frac{3+4+2+3}{4})$ to 4 $(= \frac{4+5+4+3}{4})$ after the first generation and we can see that the fitness of the best solution F in the second generation is better than that of any solution in the first generation.

3.2 Design Choices in Implementing a GA

Although the basic idea of the GA is quite simple, a modeller faces a number of key decisions when looking to apply it to a specific problem:

Table 3.4. Population of solution encodings after the mutation operator has been applied

Candidate	**String**	**Fitness**
E	01100110	4
F	01101110	5
G	01101100	4
H	10000110	3

- what representation should be used?
- how should the initial population of genotypes be initialised?
- how should fitness be measured?
- how should diversity be generated in the population of genotypes?

Each of these are discussed in the following sections.

3.3 Choosing a Representation

In thinking about evolutionary processes, two distinct mapping processes can be distinguished, one between the genotype and the phenotype, and a second between the phenotype and a fitness measure (Fig. 3.2). In applying the GA, the user must select how the problem is to be *represented*, and there are two aspects to this decision. First, the user must decide how potential solutions (phenotypes) will be encoded onto the genotype. Secondly, the user must decide how individual elements of the genotype will be encoded.

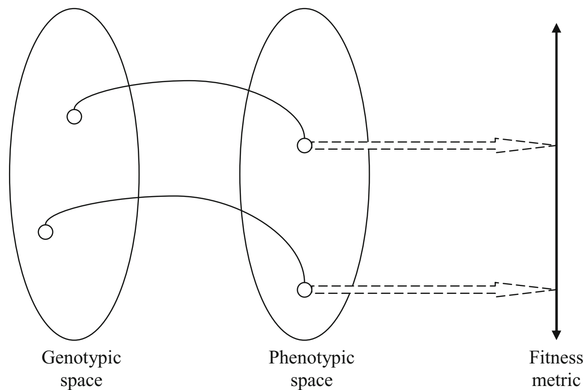


Fig. 3.2. Mapping from genotypic to phenotypic space with each phenotype in turn being mapped to a fitness measure

3.3.1 Genotype to Phenotype Mapping

Suppose a modeller is trying to uncover the relationship between a dependent variable and a set of explanatory variables and that she has collected a dataset of sample values. Let us assume that the relationship between the variables is known to be linear of the form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$. The object is now to uncover the best values for the model coefficients so that the model fits the dataset as well as possible.

In order to apply the GA for this task, a decision is required as to how these parameters should be represented on the genotype. In this case the genotype can simply consist of three elements, each of which encodes a real number (the values for $\beta_0, \beta_1,$ and β_2) and the GA's task is then to uncover the optimal values for these three parameters. In this case, the mapping from the genotype to the phenotype (Fig. 3.3) is straightforward, with three values being plugged directly into the linear model.

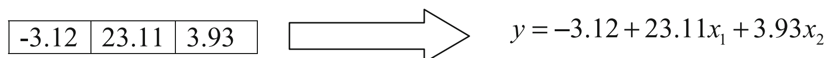


Fig. 3.3. Mapping from real-valued genotype to produce a linear model

The mapping from genotype to phenotype can of course be more complex. Suppose the task was to evolve a classifier of the form:

$$IF\ [x_i\ (<,>)\ VALUE_1]$$
$$(AND,\ OR)\ [x_j\ (<,>)\ VALUE_2]$$
$$THEN\ (Class\ 1)\ ELSE(Class\ 2)$$

Here the phenotype takes the form of a compound conditional statement and the task of GA in uncovering the best classifier is to decide which two explanatory variables to include from the dataset, whether each of these needs to be greater than or less than a trigger value, and how the two logic statements should be compounded. The number of (integer) choices for x_i and x_j depend on the number of explanatory variables in the dataset. Hence, the genotype will need to allow the generation of a series of integer and real values (Fig. 3.4).

3.3.2 Genotype Encodings

In early research on GAs, the canonical GA used binary-valued encodings for genotypes (0101 ... 1). Although this seems very limiting at first glance, binary-valued encodings can be easily used to produce real (or integer) valued

x_i	<>	$Value_1$	AND / OR	x_i	<>	$Value_1$
Integer	Integer	Real	Integer	Integer	Integer	Real

Fig. 3.4. Mixed integer/real genotype, consisting of seven elements, which could encode a classification rule

outputs, thereby allowing the application of a binary-valued GA to a wide range of problems.

The simplest decoding method is to convert the binary string to an integer value, which can in turn be converted into a real value if required. A binary genotype of length n can encode any integer from 0 to $2^n - 1$ (Table 3.5). More generally, it is possible to encode any integer in the range $0, \ldots, L-1$, even when there is no n such that $L = 2^n$.

If a real-valued output is required, the integer value obtained by decoding the binary string can be divided by $2^n - 1$ to obtain a real number in the interval [0,1]. A real number in any interval[1] $[a, b]$ can be obtained by taking the result of the last calculation and rescaling it using the formula $a + x(b - a)$. Taking an example, a binary string which is eight bits long can encode any integer between 0 and 255. If we consider the binary string (00000111), reading from right to left, this can be decoded into the integer value 7 (calculated as: $2^0 \times 1 + 2^1 \times 1 + 2^2 \times 1 + 2^3 \times 0 + 2^4 \times 0 + 2^5 \times 0 + 2^6 \times 0 + 2^7 \times 0$). If instead of an integer value in the range 0 to 255, a real value in the range $[0, 5]$ were required, the integer value could be converted into a real value as follows: $0 + \frac{7}{255} \times (5 - 0) = 0.027451$.

Although the above decoding scheme for a binary string is easy to understand, it can suffer from *Hamming cliffs*, as sometimes a large change in the genotype is required to produce a small change in the resulting integer value. Looking at the change in the binary value required to move from an integer value of 3 to 4 in Table 3.5, it can be seen that the underlying genotype needs to change in all three bit positions. Hamming cliffs can potentially create barriers that the GA could find difficulty in passing.

An alternative encoding system that has been used in some GA systems is that of *Gray coding*. In Gray coding a single integer change only requires a one bit change in the binary genotype. This means that adjacent solutions in the integer search space will be adjacent in the (binary) encoding space as well, requiring fewer mutations to discover. The Gray coding rule starts with a string of all 0s for the integer value 0. To create each subsequent integer in sequence the rule successively flips the right-most bit that produces a new string.

[1]Recall that the shorthand $[a, b]$ denotes the interval of all real numbers $x \in \mathbb{R}$ such that $a \leq x \leq b$, while $[a, b)$ denotes the interval of all $x \in \mathbb{R}$ such that $a \leq x < b$.

Table 3.5. Integer conversion for standard and Gray coding

Integer Value	Canonical Binary Code	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

For many problems, a real-valued genotype encoding is the most natural representation and most current optimisation applications of the GA use real-valued encodings (for example, $2.13, \ldots, -14.56$).

3.3.3 Representation Choice and the Generation of Diversity

The choice of representation is crucial as it determines the nature of the search space traversed by the GA. The choice also impacts on the appropriate design of diversity generation processes such as crossover and mutation.

Ideally, small (big) changes in the genotype should result in small (larger) changes in the phenotype and its associated fitness. This feature is known as *locality*. For example, if there is a good pairing of representation and diversity operators, minor mutations on the genotype will produce relatively small changes in the phenotype and its fitness, whereas a crossover between two very different parents will lead to a larger change in the phenotype. This means that the operators of mutation and crossover will perform distinctly different search processes.

Going back to the previous example of Gray coding, it can be observed from the integer to Gray mapping in Table 3.5, that a small change in the phenotype corresponds to a small change in the genotype. However, the reverse is not true. A single bit-flip in the genotype can lead to a large change in the phenotype (integer value). Hence, even a Gray encoding has poor locality properties and will not necessarily produce better results than the canonical binary coding system.

Raidl and Gottlieb [525] emphasise three key characteristics for the design of quality evolutionary algorithms (EAs):

i. locality,
ii. heritability, and
iii. heuristic bias.

Locality refers to the case where small steps in the search space result in small steps in the phenotypic space. Strong locality increases the efficiency of

evolutionary search by making it easier to explore the neighbourhood of good solutions, whereas weak locality means that evolutionary search will behave more like random search.

Heritability refers to the capability of crossover operators to produce children that utilise the information contained in their parents in a meaningful way. In general, good heritability will ensure that each property of a child should be inherited from one of its parents, and that traits shared by both parents should be inherited by their child. Crossover operators with weak heritability are more akin to macro-mutation operators.

Heuristic bias occurs when certain phenotypes are more likely to be created by the EA than others when sampling genotypes without any selection pressure. In an unbiased case, each item in the phenotypic space has the same chance of occurring if a genotype is randomly generated.

Obviously, inducing heuristic bias can be good if it tends to lead to better solutions (in contrast, random search will tend to have lower heuristic bias), but inducing bias tends to reduce genotypic diversity, which can hinder the search for a global optimum. Heuristic bias arises from the choice of representation and the choice of variation operators.

3.4 Initialising the Population

If good starting points for the search process are known a priori, the efficiency of the GA can be improved by using this information to seed the initial population. More commonly, good starting points are not known and the initial population is created randomly. For binary-valued genotypes, a random number between 0 and 1 can be generated for each element of the genotype, with random numbers ≥ 0.5 resulting in the placing of a 1 in the corresponding locus of the genotype. If a real-valued representation is used, and boundary values for each locus of the genotype can be determined, each element of the genotype can be selected randomly from the bounded interval.

3.5 Measuring Fitness

The importance of the choice of fitness measure when designing a GA cannot be overstressed as this metric drives the evolutionary process. The first step in creating a suitable fitness measure is to identify an appropriate objective function for the problem of interest. This objective function often needs to be transformed into a suitable fitness measure via a transformation (for example, to ensure that the resulting fitness value is always nonnegative); hence:

$$F(x) = g(f(x)) \tag{3.2}$$

where f is the objective function, g transforms the value of the objective function into a nonnegative number, and F is the fitness measure. A simple example of a transformation is the linear rescaling of the raw objective function value:

$$g(x) = af(x) + b \tag{3.3}$$

where a is chosen in order to ensure that the maximum fitness value is a scaled multiple of the average fitness and b is chosen in order to ensure that the resulting fitness values are nonnegative. Using rescaled fitnesses rather than raw objective function values can also help control the selection pressure in the algorithm (Sect. 3.6.1).

In addition to absolute measures of fitness as just described, it is also possible to define fitness in relative rather than absolute terms, thus avoiding having to calculate explicit fitness values for each population member. For example, if our aim is to evolve a chess player we could evaluate the population by allowing individuals to play tournaments against each other where the winner of the tournament is deemed the fittest.

Estimating Fitness

Evaluating the fitness of individual members of the population is usually the most computationally expensive and time-consuming step in a GA. In some cases it is not practical to obtain an exact fitness value for every individual in each iteration of the algorithm.

A simple initial step is to avoid retesting the same individuals, so before testing the fitness of a newly created individual, a check could be made to determine if the same genotype has been tested in a prior generation. If it has been, the known fitness value can simply be assigned to the current individual.

More generally, in cases where fitness function evaluation is very expensive, we may wish to use less costly approximations of the fitness function in order to quickly locate good search regions.

One method of doing this is *problem approximation*, where we replace the original problem statement (fitness function) with a simpler one which approximates the problem of interest, the assumption being that a good solution to the simpler problem would be a good starting point in trying to solve the real problem of interest. An example of this would be the use of crash simulation systems where designs that perform well in computer simulations could then be subject to (expensive) real-world physical testing.

A second approach is to try to reduce the number of fitness function evaluations by estimating an individual's fitness based on the fitness of other 'similar' individuals. Examples of this include *fitness inheritance*, where the fitness of a child is inherited from its parent(s), or *fitness imitation*, where all the individual solutions in a cluster (those close together as defined by some distance metric) are given the same fitness (that of a representative solution of the cluster). Hence, an approximate fitness evaluation is used for much of

the EA run, with the population, or a subset of the better solutions in the population, being exposed to the real (expensive) fitness function periodically during the run. This process entails a trade-off, with the gains from the reduction in the number of fitness evaluations being traded off against the risk that the search process will be biased through the use of fitness approximations.

A practical problem that can arise in applying GAs to real-world problems is that the fitness measures obtained can sometimes be noisy (for example, due to measurement errors). In this case, we may wish to resample fitness over a number of training runs, using an average fitness value in the selection and replacement process.

3.6 Generating Diversity

The process of generating new child solutions aims to exploit information from better solutions in the current population, while maintaining explorative capability in order to uncover even better regions of the search space. Too much exploitation of already-discovered good solutions runs the risk of convergence of the population of genotypes to a local optimum, while too much exploration drives the search process towards random search.

A key issue in designing a good GA is the management of the *exploration vs. exploitation* balance. The algorithm must utilise, or exploit, already-discovered fit solution encodings, while not neglecting to continue to explore new regions of the search space which may contain even better solution encodings. Choices for the selection strategy, the design of mutation and recombination operators, and the replacement strategy, determine the balance between exploration and exploitation. Selection and crossover tend to promote exploitation of already-discovered information, whereas mutation tends to promote exploration.

3.6.1 Selection Strategy

The design of the 'selection for mating' strategy determines the *selection pressure* (the degree of bias towards the selection of higher-fitness members of the population) of the algorithm. If the selection pressure is too low, information from good parents will only spread slowly through the population, leading to an inefficient search process. If the selection pressure is too high, the population is likely to get stuck in a local optimum, as a high selection pressure will tend to quickly reduce the degree of genotypic diversity in the population. Better-quality selection strategies therefore, encourage exploitation of high-fitness individuals in the population, without losing diversity in the population too quickly.

Although a wide variety of selection strategies have been designed for the GA, two common approaches are *fitness proportionate selection* and *ordinal selection.*

Fitness Proportionate Selection

The original method of selection for reproduction in the GA is fitness-proportionate selection (FPS) and under this method the probability that a specific member of the current population is selected for mating is directly related to its fitness relative to other members of the population. The selection process is therefore biased in favour of 'good' (i.e., fit) members of the current population. Given a list of each of the n individuals in the population and their associated fitnesses f_i, a simple way to implement FPS is to generate a random number $r \in \left[0, \sum_{j=1}^{n} f_j\right)$, then select the individual i such that:

$$\sum_{j=1}^{i-1} f_j \leq r < \sum_{j=1}^{i} f_j. \tag{3.4}$$

As a numerical example, suppose $n = 4$, $f_1 = f_2 = 15$, $f_3 = 10$ and $f_4 = 20$. Therefore, $\sum_{j=1}^{4} f_j = 60$. Assume a random draw from $[0, 60)$ produces 29.4. This value falls in the range $[15, 30)$ and hence results in the selection of individual 2. The FPS selection process can be thought of as spinning a roulette wheel, where the fitter individuals are allocated more space on the wheel (Fig. 3.5).

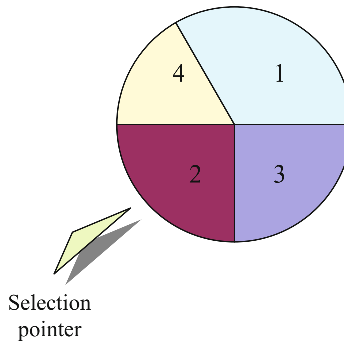


Fig. 3.5. Fitness-proportionate selection with the area on the roulette wheel corresponding to the fitness of each member of the population. Here individual 2 is selected

Although this method of selection is intuitive, it can produce poor results in practice as it embeds a high *selection pressure* in the early stage of the GA. Under FPS, the expected number of offspring for each encoding in the population is given by $\frac{P_{\text{obs}}}{P_{\text{avg}}}$, where P_{obs} is the observed performance (fitness) of the corresponding solution and P_{avg} is the average performance of all solutions in the current population.

Commonly, in the early stage of the search process there is a high variance in the fitness of solution encodings, with a small number of encodings being notably fitter than the others. When FPS is used, these encodings and their descendants can quickly overrun the entire population (better-quality solution encodings may be chosen for replication several times in a single generation), and lead to the *premature convergence* of the population and stagnation of the algorithm in a local optimum.

Conversely, FPS can result in low selection pressure later in the GA run, as the population and the associated fitness values of individuals converge. When fitness values of individuals are very similar, each individual has an almost uniform chance of selection, and hence, slightly better solutions find it difficult to strongly influence future populations.

Ordinal Selection

One approach to overcome the problems of fitness-proportionate selection is to use rank-based selection. In rank-based selection, individuals are ranked from best to worst based on their raw fitness and this rank information is used to calculate a rescaled fitness for each individual. The rescaled fitness values rather than the original fitness values are used in the selection process. An example of a *linear ranking* process is provided by [25]:

$$f_{\text{rank}} = 2 - P + 2(P-1)\frac{(\text{rank}-1)}{(n-1)}. \tag{3.5}$$

In (3.5) rank is the ranking of an individual member of the population (the least fit individual has a rank of 1, and the most fit has a rank of n), there are n members of the population, and P is a scaling factor $\in [1.0, 2.0]$ which determines the selection pressure.

To illustrate the operation of the linear ranking process, assume that $n = 5$ and let $P = 2$. Table 3.6 lists five members of a sample population in order of their raw fitness (the least fit individual is ranked number 1), and also lists their rescaled fitnesses and their selection probabilities. These fitness values are then used in the roulette wheel selection process described above and higher ranking individuals are clearly more likely to be selected.

Table 3.6. Rank ordering and selection fitness

Ranking	1	2	3	4	5
Rescaled fitness	0	0.5	1	1.5	2
Selection probability	0	0.10	0.20	0.30	0.40

An advantage of rank-based selection is that it lessens the risk of biasing the search process as a result of too-intensive selection of the better solutions in the early generations of the GA. Another advantage of rank-based selection

is that it only requires relative (as distinct from absolute) measures of fitness. This could be an advantage if fitness measures are noisy.

Another rank-based method of selection is *truncation selection*. In this scheme, the top $\frac{1}{n}^{\text{th}}$ of the n individuals in the population each get n copies in the mating pool. For example, if there are 100 members of the current population and the truncation rate is set at $\frac{1}{2}$, then the 50 fittest members of the population are each copied twice to create the mating pool.

A commonly used, and computationally efficient, rank-selection method is *tournament selection* (Fig. 3.6). Under tournament selection, k members are chosen randomly without replacement from the population. The fittest of these is chosen as the tournament winner and is 'selected' to act as a parent. Assuming a population of size N, the value of k can be varied from $2, \ldots, N$. Lower values of k provide lower selection pressure, while higher values provide higher selection pressure. For example, if $k = N$, the fittest individual in the current population is always the tournament winner.

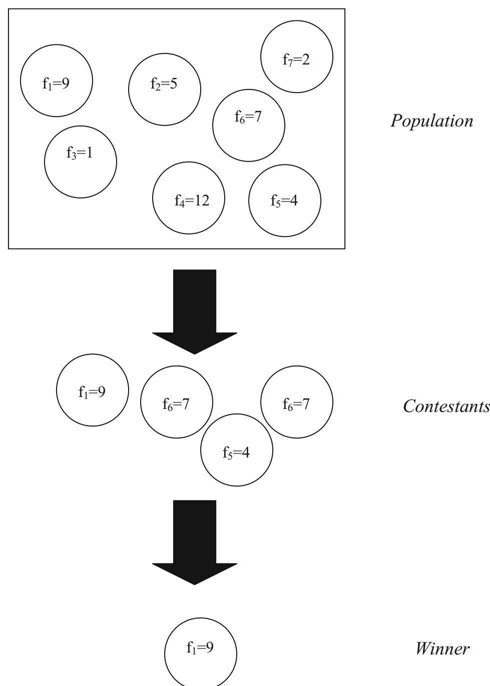


Fig. 3.6. Tournament selection where $k = 4$. Four individuals are randomly chosen from the population with the fittest of these individuals winning the tournament and being selected for reproduction

As selection works on phenotypes (and their related fitness) it is 'representation independent'. This is not the case for the diversity generating operators of mutation and crossover.

3.6.2 Mutation and Crossover

The mutation operator plays a vital role in the GA as it ensures that the search process never stops. In each iteration of the algorithm, mutation can potentially uncover useful novelty. In contrast, crossover, if applied as a sole method of generating diversity, ceases to generate novelty once all members of the population converge to the same genotype.

The rate of mutation has important implications for the usefulness of selection and crossover. If a very high rate of mutation is applied, the selection and crossover operators can be overpowered and the GA will effectively resemble a random search process. Conversely, if a high selection pressure is used, a higher mutation rate will be required in order to prevent premature convergence of the population. In setting an appropriate rate of mutation, the aim is to select a rate which helps generate useful novelty but which does not rapidly destroy good solutions before they can be exploited through selection and crossover. In contrast to mutation, crossover allows for the inheritance of groups of 'good genes' or *building blocks* by the offspring of parents, thereby encouraging more intensive search around already discovered good solutions.

There is a close link between the choice of genotype representation and the design of effective mutation and crossover operators. Initially, mutation and crossover mechanisms for binary encodings are discussed, followed by the consideration of what modifications should be made to these for real-valued encodings.

Binary Genotypes

The original form of crossover for binary-valued genotypes was *single point crossover* (Fig. 3.7). A value p_{cross} is set at the start of the GA (say at 0.7) and for each pair of selected parents, a random number is generated from the uniform distribution $U(0,1)$. If this value is < 0.7, crossover is applied to generate two new children; otherwise crossover is bypassed and the two children are clones of their parents. Crossover rates are typically selected from the range $p_{\text{cross}} \in (0.6, 0.9)$ but, if desired, the rate of crossover can be varied during the GA run.

One problem of single point crossover, is that related components of a solution encoding (schema) which are widely separated on the string tend to be disrupted when this form of crossover is applied. One way of reducing this problem is to implement *two point crossover* (Fig. 3.8), where the two cut positions on the parent strings are chosen randomly and the segments between the two positions are exchanged.

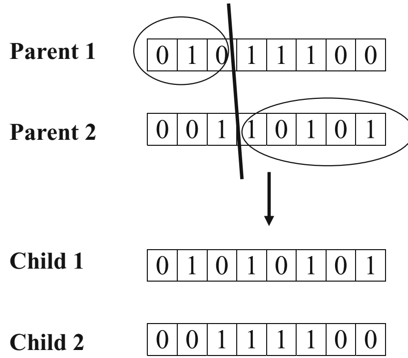


Fig. 3.7. Single point crossover where the cut-point is randomly selected after the third locus on the parent genotypes. The head and tail of the two parents are mixed to produce two child genotypes

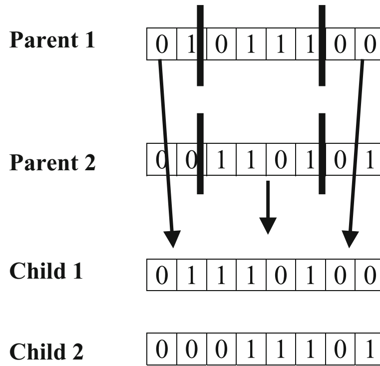


Fig. 3.8. Two-point crossover

Another popular form of crossover is *uniform crossover*. In uniform crossover, a random selection of gene value is made from each parent when filling each corresponding locus on the child's genotype. The process can be repeated a second time to create a second child, or the second child could be created using the values not selected when producing the first child (Fig. 3.9). To implement the latter approach, a random number r is drawn from the uniform distribution $U(0, 1)$ for each locus. If $r < 0.5$, child 1 inherits from parent 1; else, it inherits from parent 2, with child 2 being comprised of the bit values not selected for child 1.

For binary genotypes, a mutation operation can be defined as a bit-flip, whereby a '0' can be mutated to a '1' or a '1' to a '0'. Figure 3.10 illustrates

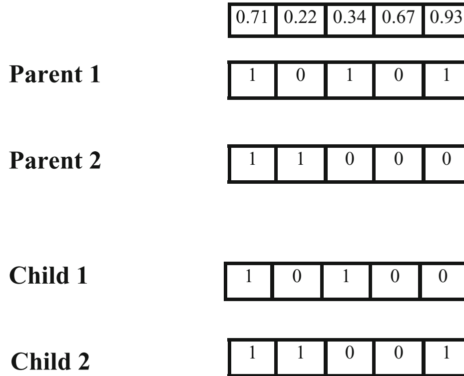


Fig. 3.9. Uniform crossover where a random choice is made as to which parent donates a bit to child 1. Child 2 is then constructed using the bits not selected for inclusion in child 1

an implementation of the mutation process, where $p_{\text{mut}} = 0.1$. Five random numbers (corresponding to a genotype length of five bits) are generated from $U(0, 1)$ and if any of these are < 0.1 then the value of that bit is 'flipped'. This mutation process is repeated for all child solutions generated by the crossover process.

Typical mutation rates for a binary-valued GA are commonly of the order $p_{\text{mut}} = \frac{1}{L}$ where L is the length of the binary string. Of course, there is no requirement that the mutation rate must remain constant during the GA run (Sect. 3.7).

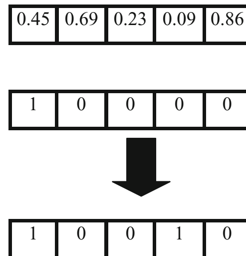


Fig. 3.10. Illustration of mutation, with the bit at the fourth locus being 'flipped'

Real-Valued Genotypes

The crossover operator can be modified for real-valued genotypes so that (for example) elements from the string of each parent are averaged in order to produce the corresponding value in their child(ren) (Fig. 3.11). Figure 3.12 illustrates this geometrically in two dimensions.

More generally, the real values in each locus of the child may be calculated as $P_1 + \alpha(P_2 - P_1)$, where P_1 and P_2 are the real values for that locus in each of the two parents, and α is a scaling factor randomly drawn from some interval (say $[-1.5, +1.5]$). This crossover operator defines a hypercube based on the location of the parents (Fig. 3.13).

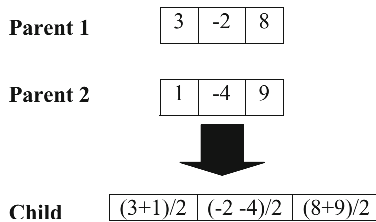


Fig. 3.11. Simple real-valued crossover with two parents producing a single child

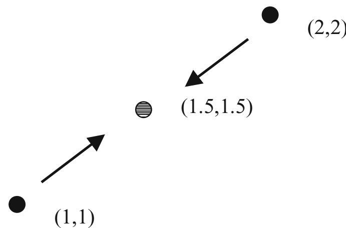


Fig. 3.12. Simple real-valued intermediate crossover with two parents (1,1) and (2,2) producing a single child at (1.5,1.5)

Many alternative mutation and crossover schemes for real-valued encodings exist. For example, a simple strategy for modifying mutation for real-valued encodings is to implement a stochastic mutation operator, where an element of a real-valued string can be mutated by adding a small (positive or negative) real value to it. Each element of the string x_i could be mutated by adding a random number drawn from the normal distribution $N(0, \alpha_i)$, where

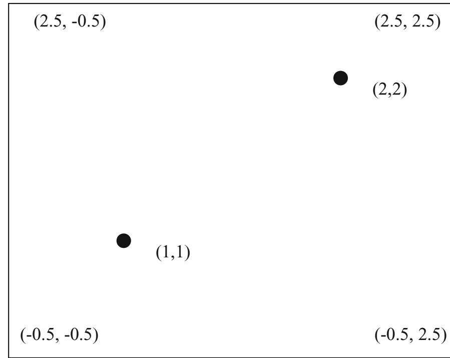


Fig. 3.13. Hypercube defined by crossover operator where parents are (1,1) and (2,2), with $\alpha \in [-1.5, 1.5]$

the standard deviation α_i is defined by the user. This mutation scheme will produce relatively small mutations most of the time, with occasional larger mutation steps.

3.6.3 Replacement Strategy

In deciding which parents and children survive into the next generation a wide variety of replacement strategies can be applied, including:

i. direct replacement (children replace their parents),
ii. random replacement (the new population is selected randomly from the existing population members and their children),
iii. replacement of the worst (all parents and children are ranked by fitness and the poorest are eliminated), and
iv. tournament replacement (the loser of the tournament is selected for replacement).

In the canonical GA, a generational replacement strategy is usually adopted. The number of children produced in each generation is the same as the current population size and during replacement the entire current population is replaced by the newly created population of child encodings.

The ratio of the number of children produced to the size of the current population is known as the *generation gap*. Hence, the generation gap is typically 1.0. It is also possible to create more offspring than members of the current population (generation gap > 1), and then select the best n (where n is the population size) of these offspring for survival into the next generation.

Many variants on the replacement process exist. As already seen, the number of children produced need not equal the current population size, and the automatic replacement of parents by children is not mandatory. A popular

strategy is *steady state* replacement, where only a small number of children (sometimes only one) are created during each generation, with only a small number of the current population, usually the least fit, being replaced during each iteration of the GA. For example, the worst x members of the current population could be replaced by the best x children. Adopting a steady state replacement strategy ensures that successive populations overlap to a significant degree (parents and their children can coexist), requires less memory, and allows the GA to exploit good solutions immediately after they are uncovered.

Generally, fitness-based selection is implemented for *either* parent selection *or* replacement selection, not both. If fitness-based selection is implemented for both, very strong selection pressure will be created, leading to very rapid convergence of the population and poor search of the solution space.

Another common replacement strategy is *elitism*, whereby the best member (or several best members) of the current population always survive into the next population. This strategy ensures that a good individual is not lost between successive generations.

Some GA applications use *crowding operators* to supplement their replacement strategy. In order to encourage diversity in the population of solution encodings, a new child solution is only allowed to enter the population by replacing the current member of the population which is most similar to itself. The objective is to avoid having too many similar individuals (crowds) in the new population.

3.7 Choosing Parameter Values

When applying the GA to real-world problems the user has to choose values for several parameters including the rate of mutation, the rate of crossover, and the size of the population, in addition to selecting the form of selection, the replacement strategy, etc. Even if attention is restricted to the choice of good crossover and mutation rates, the modeller faces a nontrivial problem in selecting these. A common approach in tuning the parameters for a GA application is to undertake a series of trial and error experiments before making parameter choices for the final GA runs (computer simulations). However, this approach is problematic as it can be time-consuming and good choices for these parameters are unlikely to remain constant over the entire GA run.

Rather than selecting static parameter values, an alternative approach is to dynamically adapt the parameters during the run. There are three broad methods of dynamically adapting parameter settings (Fig. 3.14) [175].
Deterministic methods of parameter control vary parameter settings during the GA run, without using any feedback from the search process. An example of a deterministic rule for adapting the mutation rate is:

$$\alpha(t) = \alpha(t_0)(1 - 0.8t/T) \tag{3.6}$$

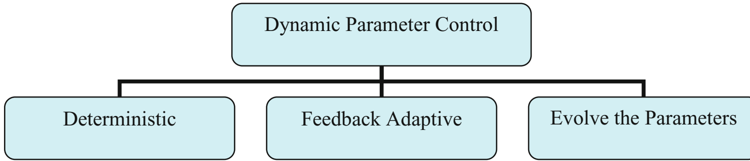


Fig. 3.14. Taxonomy of adaptive parameter control

where t $(0 \leq t \leq T)$ denotes the current generation, $\alpha(t_0)$ is a fixed value and $\alpha(t)$ is the adaptive mutation rate. This rule will reduce the value of the mutation rate during the run, biasing the GA towards increasing exploitation of current solutions as the run progresses.

Under a *feedback* adaptive process, the parameter values are altered based on feedback from the algorithm. If the composition of the population has converged (perhaps measured using the entropy of the population of binary strings) to a threshold level, the mutation rate could be increased by a prespecified rate.

Another possibility is to have the GA *evolve* good choices for its parameters. Under this idea, the GA is double-tasked: both to self-calibrate and to find a good solution to the problem at hand. A comprehensive description of dynamic parameter adaption is provided in [175].

3.8 Summary

This chapter presented an introduction to the best-known evolutionary algorithm, the genetic algorithm. The canonical GA is based on a very simplified abstraction of evolutionary processes, usually employing fixed-size populations, unisex individuals, stochastic mating, and ignoring the child-adult development process [144].

GAs, since their introduction, have been shown to be powerful problem solvers and have been successfully applied to solve a large number of real-world optimisation problems. The methodology has particular utility when traditional techniques fail, either because the objective function is ‘hard’ (for example, noncontinuous), or because the landscape is highly multimodal. The parallel nature of a GA search process makes it less vulnerable to local optima than traditional hill climbing optimisation methods. However, it is important to note that the GA is not a ‘black box’ optimiser. While a canonical GA may obtain good results when applied to a real-world problem, obtaining the best results usually requires a careful design of the algorithm and the careful use of any domain knowledge available.

Despite the good properties of GAs, they, like all optimisation techniques, are subject to limitations. There is no guarantee that an optimal solution

will be found in finite time and progress towards better solutions may be intermittent rather than gradual. Consequently, the time required to find a high-quality solution to a problem is not determinable ex ante. The GA, and indeed all evolutionary optimising methodologies, rely on feedback in the form of fitness evaluations. For some problems, measuring fitness can be difficult (perhaps fitness can only be assessed subjectively by a human) or expensive in terms of cost or computation time. In these cases, GA may not be the most suitable choice of optimising technique.

In this chapter, we have outlined the primary components and principles upon which the GA is based. The next chapter describes a number of extensions of the GA model.

4

Extending the Genetic Algorithm

The previous chapter provided an overview of the main concepts behind the GA. Since the introduction and popularisation of the GA, a substantial body of research has been undertaken in order to extend the canonical model and to increase the utility of the GA for hard, real-world problems. While it is beyond the scope of any single book to cover all of this work, in this chapter we introduce the reader to a selection of concepts drawn from this research. Many of the ideas introduced in this chapter have general application across the multiple families of natural computing algorithms and are not therefore limited to GAs. The chapter concludes with an introduction to Estimation of Distribution Algorithms (EDAs). EDAs are an alternative way of modelling the learning which is embedded in a population of genotypes in an evolutionary algorithm and have attracted notable research interest in the GA community in recent years.

4.1 Dynamic Environments

Many of the most challenging problems facing researchers and decision-makers are those with a dynamic nature. That is, the environment in which the solution exists, and consequently the optimal solution itself, changes over time. Examples of dynamic problems include trading in financial markets, time series analysis of gene expression data, and routing in telecommunication networks. Biological organisms inhabit dynamic environments and mechanisms have arisen to promote the 'survivability' of biological creatures in these environments. These mechanisms are useful sources of inspiration in helping us to design computer algorithms to attack real-world problems in dynamic environments.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_4

4.1.1 Strategies for Dynamic Environments

In designing an evolutionary algorithm for application in a dynamic environment, the nature of the environmental changes will determine the appropriate strategy. For example, if change occurs at a slow pace, adapting the rate of mutation in a GA may be sufficient to allow the population to adjust to a slowly changing location for the global optimum. If the environment alters in a cyclic fashion, a memory of good past solutions may be useful. On the other hand, if the environment is subject to sudden discontinuous change then more aggressive adaptation strategies will be required. Hence, we can adopt a variety of strategies, including [302]:

- restart of the learning process,
- generation of more genotypic diversity if environmental change is detected,
- maintenance of genotypic diversity during the GA run,
- use of a memory mechanism to retain good past solutions (assumes cycling solutions), and
- use of multiple populations.

In an extreme case, it may be necessary to restart the learning process as past learning embedded in the population is no longer useful. More generally, if past learning still provides some guide to finding good solutions in the current environment, the focus switches to how best to adapt the current population in order to track the optimal solution as it changes. The following subsections discuss various aspects of diversity generation and maintenance. The use of multiple populations is discussed in Sect. 4.2.

4.1.2 Diversity

Maintaining diversity in the population of genotypes is important in all EC applications. Even in static environments, a population needs diversity in order to promote a good exploration of the search space. The role of diversity is even more important when faced with a dynamic environment. In the absence of any countermeasures, the canonical GA will tend to lose genotypic diversity during its run as selection and crossover will tend to push the population to a small set of genotypic states; hence the canonical algorithm needs some modification when it is applied in a dynamic environment.

Depending on the expected rate of environmental change, the modeller may decide to maintain a high degree of populational diversity at all times (useful if the environment has high rate of change), or generate it 'on demand' when a change in the environment is detected. At first glance it may appear that the better option is to maintain populational diversity at all times. However, maintaining diversity has a cost, either in terms of having a larger population, or in terms of less intensive exploitation of already discovered good regions. If the environment only changes occasionally, generation of diversity when environmental change is detected may be the better option.

Diversity Generation if Change Is Detected

Cobb's hypermutation strategy [115] was one of the earliest approaches to varying the rate of diversity generation as changes in the fitness landscape are detected. In this approach, if a change in the fitness landscape is detected, the base mutation rate of the GA is multiplied by a hypermutation factor. The size of the factor determines its effect; so if it is very large, it is equivalent to randomly reinitialising the entire population.

A common approach in the detection of environmental change is to use a *sentry strategy*. In a sentry strategy the fitness of a number of fixed genotypes (a form of memory) is monitored throughout the run. If the environment changes, the fitness of some or all of the locations of these sentries will alter and this provides feedback which is used to set the rate of mutation of the GA. If a large change in fitness occurs, indicating that the environment has changed notably, the rate of mutation is increased.

The sentry strategy can be applied in a number of ways. The sentries can remain outside the adapting population of solutions or they can be available for selection and crossover. In the latter case, while the sentries can influence the creation of new child solutions, they remain 'fixed' in location and are not mutated or replaced. Morrison [420] provides a discussion of quality strategies for sentry location, finding that random location often provides good results. In addition to providing information on whether the environment is changing, a sentry strategy can also provide information on where it is changing, thereby providing feedback on whether the changes are local or global.

Diversity Maintenance During Run

Rather than waiting for environmental change to occur and then playing 'catch-up', a strategy of maintaining continual diversity in the population can be followed. A wide variety of methods can be used for this purpose, including:

- weakening selection pressure,
- continual monitoring of populational diversity,
- restricted mating/replacement,
- fitness-sharing/crowding, and
- random immigrants.

Strong selection pressure implies that the GA will intensively sample current high fitness individuals, leading, if unchecked, to a rapid convergence of the population to similar genotypic forms. This can make it difficult for the population to adapt if environmental change occurs, particularly if the change occurs in a region of the landscape which is not currently being sampled by the population of solutions. Hence, the use of a lower selection pressure will help maintain diversity in the population of genotypes. Another related consideration when implementing a GA in a dynamic environment is what form

of selection and replacement strategy to implement. The steady-state GA can offer advantages over the canonical generational GA. It allows a quicker response to a shift in the environment, as high-quality, newly created children are immediately available for mating purposes.

The degree of diversity of a population can be continually monitored in real time as the GA runs. Populational diversity can be defined on many levels, including diversity of fitness values and diversity of phenotypic or genotypic structures. Multiple measures of diversity can be defined for each of these. For example, diversity in a collection of real-valued fitnesses could be measured using the standard deviation of those values. However measured, if population diversity falls below a trigger level, action can be taken to increase diversity by raising the level of mutation or by replacing a portion of the population by newly created random individuals.
Under a *restricted mating* or restricted replacement strategy, individuals which are too similar are not allowed to mate, and in a restricted replacement strategy a newly created child is precluded from entering the population unless it is sufficiently different to existing members of the population. The object in both cases is to avoid convergence of the population to a small subset of genotypes.

A *fitness-sharing* mechanism [213] aims to reduce the chance that a multitude of similar individuals will be selected for reproduction, thereby reducing the genetic diversity of subsequent generations. An example of a fitness-sharing mechanism is:

$$f'(i) = \frac{f(i)}{\sum_{j=1}^{n} s(d(i,j))} \tag{4.1}$$

where $f(i)$ represents the original raw fitness of individual i. If there are a number of individuals which are similar to i in the population, its fitness for use in the selection process is reduced. The shared (reduced) fitness of individual i is denoted as $f'(i)$, and this corresponds to i's original raw fitness, derated or reduced by an amount which is determined by a *sharing function*.

The sharing function s as in (4.2) provides a measure of the *density* of the population within a given neighbourhood of i. For any pair of individuals i, j in the population, the sharing function returns a value of '0' if i and j are more than a specified distance t apart (Fig. 4.1), and a value of '1' if they are identical.

$$s(d) = \begin{cases} 1 - \left(\frac{d}{t}\right)^{\alpha} & \text{if } d < t; \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

where d is a measure of the actual distance between two solutions and α is a scaling constant. To provide intuition on the sharing formula, if two individuals in the current population are virtually identical, the distance between them is close to zero. Consequently, the raw fitness of each individual is reduced by 50%, reducing each individual's chance of being selected for reproduction.

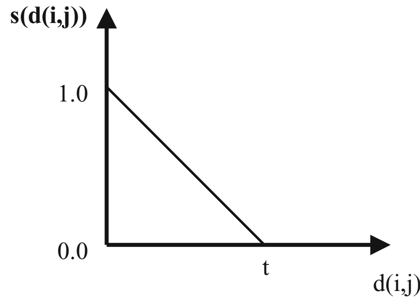


Fig. 4.1. Sharing function where $\alpha = 1$. As the distance between two solutions increases, the degree of fitness sharing between them decreases

In a *random immigrants* strategy [219] a portion of the population is replaced by new randomly created individuals in each generation. This ensures that there is a constant inflow of diverse individuals into the search process. Usually either the worst, or random, members of the current population are replaced by the immigrants.

Memory

If the environment is thought to switch between a number of different 'states' then a sensible adaptation strategy is to build up and maintain a memory of good past solutions. These can be injected into the population if an environmental change is detected. An interesting biological example of this is provided by our acquired immune system, which has the capacity to remember the molecular signature of past invading pathogens, thereby enabling it to respond quickly if the pathogen is subsequently reencountered (Sect. 16.1.3).

Measurement of Performance

An important open question in EAs (and other methods for optimisation) is how best to measure performance in a dynamic environment. Ideally, we want a solution that performs well under the expected environment but which will not fail completely if the environment changes slightly. For example, suppose we are developing a delivery schedule for a large truck fleet. The resulting schedule should be reasonably robust to truck breakdowns or unexpected traffic delays. Similarly, a control program for a machine should be robust to changes in environmental conditions such as temperature or humidity.

A simple way of assessing the brittleness of a proposed solution is to undertake sensitivity analysis on the solution by perturbing it slightly and observing the resulting effect on fitness. Solutions which produce large changes in fitness

when perturbed slightly could therefore be assigned a lower 'adjusted fitness', with this value being used to drive the selection process in the EA.

Summary

The application of GAs, and other natural computing algorithms to dynamic problems has become a major area of natural computing research in recent years and many open issues remain. As noted in the last subsection, the appropriate definition of performance measures for these problems is not a trivial issue. Another open issue is the appropriate definition of diversity.

4.2 Structured Population GAs

Most evolutionary algorithms including the GA are *panmictic* in that any two members of the population can potentially mate with each other. The population consists of a single pool of individuals and the operators of selection and crossover act on the entire pool. An alternative approach is to implement a *structured population* GA where the population is partitioned and mating is constrained to the individuals within each partition. Two popular versions of structured evolutionary algorithms exist, *distributed EAs* (dEAs) and *cellular EAs* (cEAs). Implementing a structured GA can lead to computational efficiencies as the evolutionary process can be parallelised across multiple computers or processors (a good overview of parallelisation techniques in evolutionary algorithms is provided in [11]) and it can also help maintain population diversity.

Distributed EA

The dEA is inspired by the concept of species which are simultaneously evolving on geographically dispersed islands in an ocean. It is also known as the *island model* because of this. In dEA, several separate subpopulations are created and each commences its own evolutionary process. Periodically, fit individuals are allowed to migrate between the subpopulations. The migrations promote the sharing of information from already-discovered good solution encodings, while maintaining genotypic diversity between the subpopulations. Island versions of the GA are natural candidates for parallel implementation as the evolutionary process on each island can be assigned to an individual processor.

In implementing the island model, decisions must be made concerning how often migration events occur between subpopulations, how individuals are selected for migration, how many individuals are selected for each migration event, and what replacement process is applied to refill the subpopulation when members of it migrate to another subpopulation. A variety of migration

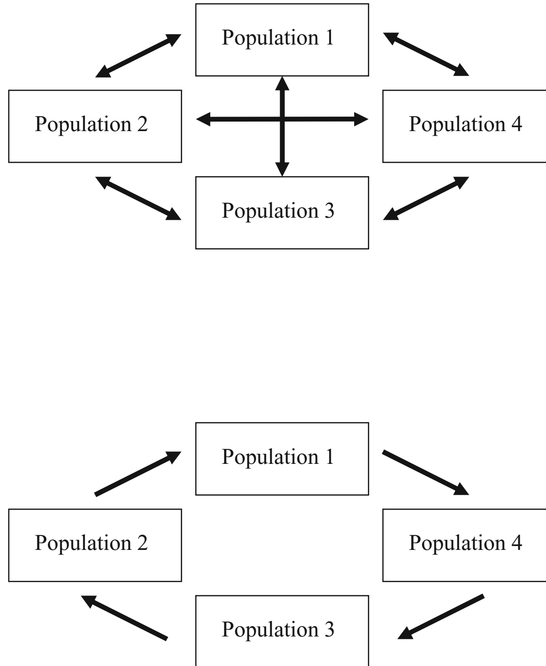


Fig. 4.2. Two examples of an island topology. The top network has unrestricted migration between all islands and the bottom network has a ring migration topology, where individuals can only migrate to one adjacent island

strategies can be used. For example, migration of individuals from one population to another may be unrestricted, or it may be confined to a predefined neighbourhood for each population (Fig. 4.2).

Illustrating one implementation of an island model, suppose there are four subpopulations with an unrestricted migration structure. A *migration pool* can be created for each subpopulation consisting of individuals selected from the other three subpopulations (perhaps their most fit individual). For each subpopulation in turn, a random selection is then made from its migration pool, with this individual replacing the worst individual in the subpopulation it enters.

Cellular EA

In cEA each individual genotype is considered as occupying a cell in a lattice (or graph) structure (Fig. 4.3). The operations of selection and recombination are constrained to take place in a small neighbourhood around each individual. When a cell is being updated, two parents are selected from its surrounding

neighbourhood, and genetic operators are applied to the two parents to produce an offspring, with this offspring replacing the current genotype stored in that cell.

Each cell has its own pool of potential mates and in turn is a member of several other neighbourhoods, as the neighbourhoods of adjoining cells overlap. Therefore, in contrast to dEA, which typically has a few relatively large subpopulations, there are typically many small subpopulations in cEA.

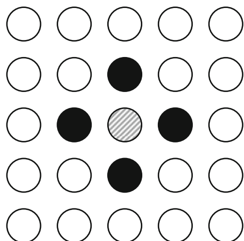


Fig. 4.3. A grid structure where a neighbourhood is defined around an individual (here the shaded cell)

In implementations of cEAs, updates of the state of each cell can be synchronous, where all cells are updated simultaneously using the cell contents in the current lattice. As the new genotypes are created, they are copied across to the next generation's lattice one at a time. Alternatively, the update process can be asynchronous, where the lattice is updated one cell at a time so that new genotypes can influence the update process as soon as they are created. One method of asynchronous update is to update all cells sequentially from left to right, and from line to line, starting from the top left corner (*fixed line sweep*). Another method of asynchronous update is to randomly select (with uniform probability and with replacement) which cell to update during each time step (*uniform choice*).

4.3 Constrained Optimisation

Many important problems consist of attempting to maximise or minimise an objective function subject to a series of constraints. The constraints serve to bound the feasible region of valid solutions, possibly to a very small subset of the entire (unbounded) search space (Fig. 4.4).

More formally, a constrained optimisation problem (assuming that the objective function is to be maximised) can be stated as follows: find the vector $x = (x_1, x_2, \ldots, x_d)^T, x \in \mathbb{R}^d$ in order to:

$$\text{Maximise } f(x) \tag{4.3}$$

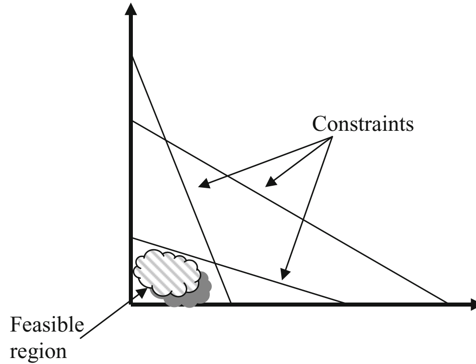


Fig. 4.4. Feasible region for a maximisation problem bounded by the x and y axes and three other constraints

subject to

$$\text{inequality constraints:} \quad g_i(x) \leq 0, \quad i = 1, \ldots, m \tag{4.4}$$

$$\text{equality constraints:} \quad h_i(x) = 0, \quad i = 1, \ldots, r \tag{4.5}$$

$$\text{boundary constraints:} \quad x_i^{\min} \leq x_i \leq x_i^{\max}, \quad i = 1, \ldots, d. \tag{4.6}$$

The boundary constraints can be used to enforce physical conditions such as mass ≥ 0, etc. There are several approaches that can be taken when using a GA for constrained optimisation. A simple approach is to apply the GA as normal and assign zero fitness to any genotypes which generate an illegal solution which breaches one or more constraints. This strategy can be poetically referred to as the *death penalty* [409]. A problem with this approach is that even with low-dimensional problems it can produce a highly inefficient search process. If the problem is highly constrained, many generated genotypes may be illegal (for example, none of the initially randomly generated solutions might be feasible); hence much of the GA's effort is wasted. There is also a risk that there could be over-rapid convergence on the first feasible solution found. As the dimensionality of the problem increases, the above problems are worsened as ratio of invalid solutions outside the feasible area to valid solutions inside the feasible area will rapidly increase. Two key issues arise when applying the GA to a constrained optimisation problem:

i. it may be difficult to generate an initial population of feasible genotypes, and
ii. crossover and mutation may act to convert a legal solution into an illegal one.

Three approaches which can ameliorate these issues include the use of penalty functions, the use of repair operators, and the creation of tailored diversity generation operations.

Penalty Functions

Rather than assign a zero fitness to genotypes which generate infeasible solutions, the objective function from which the fitness value is calculated may be supplemented by appending a penalty function to it. The greater the number of constraints breached, or the further the solution is from the feasible region, the lower the fitness assigned to it.

$$\text{Obj}^*(s) = \text{Obj}(s) - \text{pen}(s) \tag{4.7}$$

$$\text{pen}(s) = \sum_{i=1}^{m} w_i b_i \tag{4.8}$$

$$\text{where } b_i = \begin{cases} 1, & \text{if } s \text{ breaches constraint}_i; \\ 0, & \text{otherwise.} \end{cases}$$

In (4.7) the initial value of the objective function (assuming a maximisation problem) for solution s is reduced by a penalty function $pen(s)$. This penalty is determined by the number of the i constraints that are breached by s, where each of these constraints can have a different weight (w_i) (4.8).

This approach can help the GA to guide the population back to the feasible region. A penalty approach is most likely to work if there are relatively few constraints, as the greater the number of constraints, the harder it is to design an appropriate penalty function. Choices of weight values are important as, if weights are set too low, solutions may violate multiple constraints, if they are set too high, only feasible solutions will effectively be considered. This could lead to a collapse of populational diversity early in the algorithm, resulting in convergence to a locally optimal solution.

In order to overcome the latter problem, an adaptive penalty system could be implemented where low penalty weights are initially applied to facilitate explorative search, with the penalty weights being increased later in the GA run in order to force feasibility.

Repair Operators

Another approach is to attempt to repair infeasible solutions by moving them back into the feasible region. The 'repaired' solution then replaces the illegal solution in the population. Unfortunately, the design of an efficient repair mechanism can become tricky, particularly as the number of constraints increases (a simple repair operation in order to satisfy one constraint may produce a breach of another constraint). Generally, effective repair mechanisms are problem-specific.

Tailored Diversity-Generation Operators

A problem when canonical crossover and mutation operations are applied to a genotypic encoding is that they do not respect the context of the problem domain. Hence, a crossover operation on two parents drawn from the feasible region may produce a child which is infeasible. Similarly, the application of a canonical mutation operator to a feasible genotype may produce one which results in an infeasible solution. One solution to this is to design problem-specific versions of crossover and mutation. An example of this is provided by the NEAT system in Sect. 15.2, which shows how populations of feedforward multilayer perceptrons (neural networks) can be evolved.

4.4 Multiobjective Optimisation

Decision-makers are often faced with having to make trade-offs between multiple, conflicting, objectives. There are usually also constraints on the solutions, such that not all solutions will be considered feasible (Sect. 4.3). Examples of multiobjective problems abound in the real world, ranging from finance to engineering. In the former case, investors typically seek to maximise their return whilst minimising a risk measure (for example, the variability of their return). Hence, the decision faced by the investor is how to allocate her investment funds across multiple assets, so as to attain the highest possible return for a given level of risk. As we would expect, there is no unique solution to this problem, as higher expected returns will typically come at the cost of higher risk. In the case of engineering design problems, there is often a requirement to trade off (for example) the durability of a component and its weight/performance.

The multiobjective problem can be generally formulated as follows. Assume that there are n objectives $f_1, \ldots, f_n$ and d decision variables $x_1, \ldots, x_d$ with $x = (x_1, \ldots, x_d)$, and that the decision-maker is seeking to minimise the multiobjective function $y = f(x) = (f_1(x), \ldots, f_n(x))$. The problem therefore is to find the set (region) $R \subseteq \mathbb{R}^d$ of vectors $x = (x_1, x_2, \ldots, x_d)^T$, $x \in \mathbb{R}^d$ in order to:

$$\text{Minimise } y = f(x) = (f_1(x_1, \ldots, x_d), \ldots, f_n(x_1, \ldots, x_d)) \tag{4.9}$$

subject to

$$\text{inequality constraints:} \quad g_i(x) \leq 0, \quad i = 1, \ldots, m \tag{4.10}$$

$$\text{equality constraints:} \quad h_i(x) = 0, \quad i = 1, \ldots, r \tag{4.11}$$

$$\text{boundary constraints:} \quad x_i^{\min} \leq x_i \leq x_i^{\max}, \quad i = 1, \ldots, d. \tag{4.12}$$

Any specific member x of R (corresponding to a set of decision variable values) will produce a unique y (vector of objective values in objective space)

and the above formulation aims to determine the members of R which are *nondominated* by any other member of R. This set is termed a *Pareto set*. For example, in the investor's problem the aim is to find the investment weights for each of the k possible assets that the investor can include in her portfolio, such that the resulting expected risk-return outcome for that portfolio is nondominated by the expected risk-return output from any other possible portfolio. A member of the Pareto set is nondominated when (in the above example) for a given level of risk no other portfolio with a higher rate of return exists, or for a given level of return no other portfolio with a lower level of risk exists. Once the Pareto set is uncovered, the final choice of investment portfolio is determined by the individual investor's risk preference. Graphically, the Pareto set corresponds to a *Pareto frontier* (usually referred to as the *efficient frontier*) in objective space (Fig. 4.5).

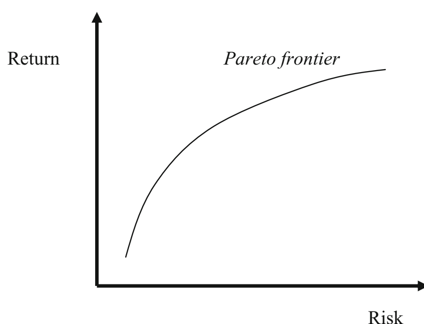


Fig. 4.5. Pareto frontier. The frontier corresponds to the set of investment portfolios that are Pareto optimal

Generally, a solution is termed *Pareto optimal* if an improvement on any one component of the objective function implies a disimprovement on another component. Hence, *all* solutions on the Pareto front are nondominated. Therefore, unlike single criterion problems, a multiobjective problem has multiple, usually an infinite number of, solutions, rendering the calculation of the entire Pareto set infeasible. Typically instead, the object is to find a good approximation of the Pareto set given limited computational resources.

Approaches to Multiobjective Optimisation

A simple approach to multiobjective problems is to attempt to convert them into a single objective problem, to which regular optimisation techniques (including the GA) can be applied. One approach is to assign weights to each objective and then compute a single value for the entire objective function

using a weighted linear combination of the individual components of the objective function for any given solution vector.

Clearly this method is subjective in that it requires the decision-maker to supply the relevant weights. Depending on the problem, the solution obtained can be sensitive to even small changes in these weightings. For a given set of weightings a single solution is obtained; hence, this approach will not identify all nondominated solutions.

Another way of converting a multiobjective problem to a single objective one is to move all but one component of the objective function to the constraint set, again obtaining a single solution for each specification of the constraints.

Multiobjective Optimisation with a GA

In using the GA for multiobjective optimisation, we are trying to closely approximate the true (but unknown) Pareto frontier and this requires that we generate a diverse set of nondominated solutions. A large literature has emerged over the past 20 years on the application of EC methods for multiobjective optimisation.

While a canonical GA can be directly applied to solve a multiobjective problem, a practical issue that arises is that typically the population will converge to a single solution (or small set of solutions) and hence will not uncover a representation of the entire Pareto frontier. One way to overcome this is to restart the GA multiple times and keep a record of the solutions found in each run. While this approach may eventually approximate the Pareto frontier, it is likely to prove computationally expensive as multiple runs may uncover the same points.

Hajela and Lin [239] proposed an alternative approach known as the *weight-based genetic algorithm* (WBGA) which uses a weighted sum approach to convert the multiobjective optimisation problem into a single objective problem. In WBGA each member of the population uses a different weight vector, which is generated randomly at the start of the run (with all weights summing to 1, $\sum_{i=1}^{n} w_i = 1$) and is then fixed. This weight vector is then applied in determining the fitness of each solution. An advantage of this approach is that it allows a standard GA to search for multiple solutions in a single run.

One of the earliest published multiobjective GA applications, called *vector evaluated GA* (or VEGA), was proposed by Schaffer [554]. This algorithm employs a 'switching objective approach' which aims to approximate the Pareto optimal set by a set of nondominated solutions. In the algorithm, the population P_t is randomly divided into K equally sized subpopulations $P_1, P_2, \ldots, P_K$, assuming that there are K objectives. Then, each solution in subpopulation P_i is assigned a fitness value using the objective function z_i corresponding to that subpopulation. Hence, each subpopulation is evaluated using a different objective — thereby removing the difficulty in trying to determine the value of a solution under multiple objectives simultaneously.

Solutions are then selected from these subpopulations, using fitness-based selection, for crossover and mutation. Crossover and mutation are performed on the new population in the same way as for a single objective GA (Algorithm 4.1).

Algorithm 4.1: VEGA Algorithm drawn from [336]

Let the subpopulation size be $N_S = N/K$;
Generate a random initial population P_0;
Set $t = 0$;
repeat
  Randomly sort population P_t;
  repeat
    for $i = 1 + (k-1)N_S, \ldots, kN_S$ **do**
      Assign fitness value $f(x_i) = z_k(x_i)$ to the ith solution in the sorted population;
      Based on the fitness values assigned, select N_S solutions between the $(1 + (k-1)N_S)^{\text{th}}$ and $(kN_S)^{\text{th}}$ solutions of the sorted population to create subpopulation P_k;
    end
  until *complete for each objective* $k = 1, \ldots, K$;
  Combine all subpopulations $P_1, \ldots, P_k$ and apply crossover and mutation on the combined population to create P_{t+1} of size N;
  Let $t = t + 1$;
until *terminating condition*;
Return P_t;

The VEGA approach is relatively easy to implement but suffers from the drawback that it tends to produce solutions which perform well on one objective but poorly on others.

While early approaches to the application of GAs for multiobjective optimisation relied on conversion of the problem into a single objective problem, later algorithms have been developed which directly tackle the multiobjective nature of the problem. In *Pareto ranking* approaches the fitness of a solution depends on its ranking within the current population (in other words, how many individuals is a specific individual dominated by), rather than on its actual objective function value; hence solutions are iteratively improved by focussing on the nondominated solutions in the current population. One of the earliest applications of this idea was Goldberg [211]. See Algorithm 4.2. Note that, here, a lower rank corresponds to a better quality solution. The set F_i are the nondominated fronts, and F_1 is the nondominated front of the population.

Related studies which further developed this method included Fonseca and Fleming's [196] Multiobjective Genetic Algorithm (MOGA), Srinivas and

Algorithm 4.2: Pareto-ranking Algorithm from [336]

repeat
  Set $i = 1$ and $TP = P$;
  Identify nondominated solutions in TP and assign them to the set F_i;
  Set $TP = TPF_i$;
  repeat
    $i = i + 1$;
    Identify nondominated solutions in TP and assign them to the set F_i;
    Set $TP = TPF_i$;
  until $TP = \varnothing$;
  for *every solution* $x \in P$ *at generation* t **do**
    Assign rank $r_1(x, t) = i$ if $x \in F_i$;
  end
until *terminating condition*;
Return F_i;

Deb's Nondominated Sorting Genetic Algorithm (NSGA) [590] and NSGA-II [136]. Zitzler and Thiele's Strength Pareto Evolutionary Algorithm (SPEA and SPEA-2) [683, 684] uses a ranking procedure to assign better fitness values to nondominated solutions in sparser regions of the objective space, thereby encouraging the uncovering of a dispersed set of nondominated solutions.

An excellent concise introduction to evolutionary multiobjective optimisation is provided by Zitzler, Laumanns and Bleuler [682] and Konak, Coit and Smith [336]. Detailed coverage of the field can be found in [116] and [135].

4.5 Memetic Algorithms

Learning in populations of individuals takes place at several levels. At one extreme, the effects of long-term evolutionary learning is encoded in the population of genotypes which make up a species. On a shorter time frame, individuals are also capable of lifetime learning based on their own and on observed experience. In populations of social individuals, learning can also be transmitted by means of a shared culture (e.g. via education systems, legal systems, etc.).

In his famous book, *The Selfish Gene* [134], Richard Dawkins coined the term 'meme' to refer to 'the basic unit of cultural transmission, or imitation'. Dawkins suggested that these memes were selected and processed by individuals and could be improved by the person holding them. As memes could also be passed from person to person, Dawkins argued that they displayed the key characteristics of an evolutionary process, namely, inheritance, variation and selection, thereby leading to a process of cultural evolution, akin to biological evolution.

The concept of a dual evolutionary/lifetime learning mechanism could of course be included in a computational algorithm and the term *memetic algorithm* was first used by Moscato in 1989 [421] to describe an algorithm which combined both genetic (population-based) and individual (or cultural) learning. Early memetic algorithms (MAs) typically consisted of an evolutionary algorithm that included a stage of individual optimisation/learning as part of the search strategy [257, 345, 346, 422]. For example, a local search step could be added to a canonical GA. The local search process could be as simple as periodically performing a hill-climb around a subset of the better solutions in the current population in order to improve these solutions further. Improvements from the local search would be encoded on that individual's genotype and would therefore be potentially transmissible to other members of the population in subsequent generations. Practical design issues in creating these algorithms include choices as to how often individual learning takes place, which members of the population engage in individual learning, how long the individual learning process lasts, and what individual learning method should be used.

These forms of MA are known by a variety of names, including hybrid genetic algorithms, genetic local search algorithms and Lamarckian genetic algorithms. A practical motivation for these hybrid evolutionary algorithms is that while evolutionary algorithms such as the GA tend to be useful in identifying a good (high-fitness) region of a search space, they can be less effective in efficiently searching within this region [452, 486].

Although the early versions of these algorithms were loosely inspired by memetic concepts, they did not explicitly embed the notion of a population of memes which is adapting over time. Over the two decades since the introduction of MAs a wide body of literature has developed on this topic with a variety of algorithms being developed which explicitly include memetic adaptation. In the multimeme MA, memes are directly encoded on an individual's genotype, and determine the nature of the local refinement process which that individual applies. An alternative approach is that a pool of candidate memes compete for survival based on the degree of their past success in producing improvements during the local refinement step with better memes having a higher chance of survival into future generations. Hence, in both of the above approaches, there may be multiple local refinement (individual learning) approaches encoded in the memes. Interested readers are referred to [488]. More generally, a significant body of literature has developed on memetic computation defined by [487] as "a paradigm that uses the notion of meme(s) as units of information encoded in computational representations for the purpose of problem solving". A detailed introduction to this field is provided by [487].

4.6 Linkage Learning

In genetics, multiple genes can interact in producing an effect at the phenotypic level. This is known as epigenesis. Unfortunately, when individual elements of a genotype interact in this manner, it becomes much harder to find good gene values, as the GA implicitly needs to tease out the linkages between the relevant genes and then coevolve good sets of values for them. A particular problem of the canonical GA is that the selection and crossover operators can easily break up promising solutions, where there are epistatic links between dispersed elements of the genotype (Fig. 4.6).

As an example, consider a binary string encoding which is n bits long, where the first and the last bit must both be '1' if the string is to have high fitness. If basic single-point crossover is applied to two parents, one of which already has the correct ('1') value in these locations, it is quite possible that neither child will inherit the good genes from that parent. In other words, because no attention has been paid to the linkage structure between the elements of the string, the crossover operator has acted in a destructive manner.

Recognition of this problem led to the development of literature on *linkage learning*, where the object is to design crossover operators which do not disrupt important emerging partial solutions (building blocks) but which still ensure an effective mixing of partial solutions.

Messy GA

One approach to this problem is to attempt to reorganise the representation of the solution encoding so that functionally related elements will be (re)located close together on the reordered genotype. This reduces the chance that important links between genes will be broken by the crossover operator (Fig. 4.7). GA variants which have employed reordering operators include messy GA (mGA) and the linkage learning genetic algorithm. While the idea is sensible, a general problem with reordering approaches is that they tend to scale poorly as the length of the genotype increases.

Competent GA

An important direction in GA research stems from the recognition of the limited scalability of the canonical GA when it is applied to problems of increasing difficulty. It has been recognised that the success of a GA is dependent upon facilitating the proper growth and mixing of building blocks, which is not achieved by problem-independent recombination operators [212, 621]. The algorithms emerging from this area of research are dubbed *competent GAs*. Competent GAs seek to perform a more intelligent search by respecting the functionally important linkages between the constituent components of a solution in order to prevent the disruption of potentially useful building blocks. More recently, the GP community has applied ideas from competent GA in designing GP algorithms [552, 572].

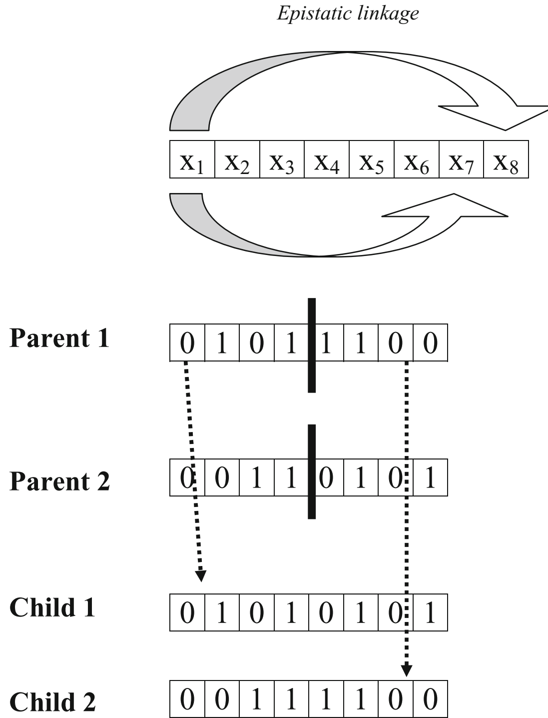


Fig. 4.6. Three of the genes are epistatically linked (x_1, x_7 and x_8). As the genes are widely separated on the genome, application of a single-point crossover will tend to disrupt sets of good choices for these genes

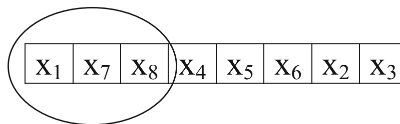


Fig. 4.7. The three epistatically linked genes (x_1, x_7 and x_8) are reordered so they are grouped together at the beginning of the genome. By grouping them together, there is less chance that the linked genes will be broken up by a single-point crossover operator

4.7 Estimation of Distribution Algorithms

Estimation of distribution algorithms (EDAs) are an alternative way of modelling the learning which is embedded in a population of genotypes in an evolutionary algorithm. EDAs are a rapidly growing subfield within evolutionary computing and have several names, including *probabilistic model building algorithms* (PBMAs) and *iterated density estimation evolutionary algorithms* (IDEAs) [361, 500, 504]. Recent years have seen the application of EDAs to a range of problem domains, including multiobjective optimisation [329, 622], and dynamic optimisation [668].

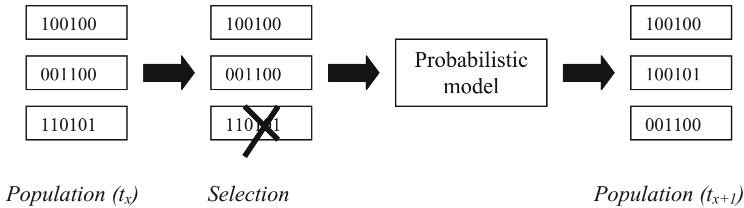


Fig. 4.8. Illustration of EDA with sampling from a probabilistic model replacing the crossover and mutation operators of canonical GA

Rather than maintaining a population of solution encodings from one generation to the next, and manipulating this population using selection, crossover and mutation, global statistical information is extracted from previous iterations of the algorithm. This information is used to construct a posterior probability distribution model of promising solutions, based on the extracted information. New solutions are then sampled from this probability distribution (Fig. 4.8). Hence, EDAs maintain the selection and variation concepts from EAs but generate variation in a different way. Particular advantages of EDAs over genetic algorithms include their lack of multiple parameters (such as crossover and mutation rates) that require tuning and the transparency of the underlying probabilistic model used to guide the search process [360].

Examples of EDAs include population-based incremental learning (PBIL) [26, 27], the compact genetic algorithm (cGA) [252], and the Bayesian Optimisation Algorithm (BOA) [503].

The general EDA methodology can be operationalised in many ways. For example, the design of the model update step depends on the assumptions made concerning the nature of the problem being addressed. Three main groups of EDAs exist; algorithms that assume that all variables are independent (univariate EDA models), those that assume restricted interactions between the variables (for example, bivariate dependencies between variables),

and those that allow unrestricted interactions between the variables. Initially we will consider univariate EDA PBIL and cGA models. Algorithm 4.3 illustrates pseudocode for a canonical univariate EDA.

Algorithm 4.3: Canonical Estimation of Distribution Algorithm

Initialise a probability vector P of length l (assume an l-dimensional problem);

repeat
- Generate n trial solution vectors, using P;
- Evaluate the n trial solutions;
- Select $x < n$ of the better solutions from the population;
- Adapt P using these x solutions;

until *terminating condition*;

Table 4.1. Illustration of probability vectors P_1, P_2, P_3 associated with three different binary-valued populations. The ith component of a probability vector specifies the probability that a bit in position i contains '1' in that particular population

Population 1	*Population 2*	*Population 3*
0101	0100	0000
1010	1010	0000
1010	1110	0000
0101	0010	0000
vector P_1	vector P_2	vector P_3
0.5 0.5 0.5 0.5	0.5 0.5 0.5 0	0.0 0.0 0.0 0.0

4.7.1 Population-Based Incremental Learning

Population-based incremental learning (PBIL) was one of the earliest EDAs developed [26, 27]. Although PBIL can be applied to nonbinary representations, for ease of exposition we will concentrate on the case where the solution encoding is represented as a fixed-length binary string. In PBIL, the algorithm's learning is embedded in a probability vector which describes the probability that a given bit position contains a '1'. Table 4.1 illustrates how a probability vector can embed the information in a population of bit strings. Here, three distinct populations, each consisting of four bit strings, are tabulated along with a probability vector which summarises the information on the relative distribution of 0s and 1s in each population of binary strings. In the case of population 3, it can be seen that the population has converged to

all 0s; hence its probability vector consists of all zeros. Algorithm 4.4 describes a canonical PBIL algorithm.

Algorithm 4.4: PBIL Algorithm

Select the population size n, the probability of mutation p_{mut}, the mutation size size_{mut} and the learning rate R;
Initialise all components of the probability vector P to 0.5;

repeat
- Construct a population of n strings, by undertaking n samplings using the probabilities in P;
- Decode each string into a solution;
- Calculate the fitness of each of the n solutions in the population;
- Determine which solution x^{best} has the highest fitness;
- Update each component P_i of P using the binary encoding corresponding to the best solution, as follows: $P_i(t) = P_i(t-1)(1-R) + x_i^{\text{best}} R$;
- Apply a mutation process, with probability p_{mut}, to each component of P;

until *terminating condition*;

From the description of the algorithm it is evident that unlike the canonical GA, there is no explicit crossover operator in PBIL. One interesting aspect of PBIL is that learning is competitive as only the best encoding in each iteration of the algorithm impacts on the probability vector P. Of course, alterations to the components in the probability vector need not be solely undertaken using x^{best}. An alternative formulation of the vector update step would be to move the vector towards the complement of the least-fit encoding in the current evaluation step, or towards the dominant bit-values of the best 'm' individuals in the current population. When generating the new population of binary strings, a random number r is drawn from $[0, 1]$, and, if $r < P_i$, then locus i in the binary string is set to '1'.

4.7.2 Univariate Marginal Distribution Algorithm

The univariate marginal distribution algorithm (UMDA) was introduced by Mühlenbein [423]. UMDA works with a binary representation and acts to uncover a vector of probabilities P which govern the probability that a '1' will be generated in the corresponding component in a binary genotype. Initially, all components of the probability vector P are initialised to 0.5.

In each iteration of the algorithm, a population of n individuals are sampled from the probability vector using the same approach as in PBIL. The population is evaluated and the best m of these individuals are selected $(m < n)$. The probability vector P is then updated by calculating the average of the values in each locus of the m selected binary strings (Table 4.2). The generate

population and update probability vector steps are iterated until a termination condition is met.

Table 4.2. Update of the probability vector P using the four fittest binary strings from the current population

$m = 4$
1101
1110
1010
0101

P
0.75 0.75 0.5 0.5

Unlike PBIL, several members of the current population, not just the best one, influence the adaptation of the probability vector. More complex variants of the UMDA exist which include mechanisms such as mutation and memory.

Continuous Univariate Marginal Distribution Algorithm

The UMDA can be easily extended to work with real-valued encodings. Assume that we wish to maximise an n-dimensional function f and denote a single solution vector $x \in \mathbb{R}^n$. Initially, a population of potential (real-valued) encodings is generated randomly within the feasible solution region. A fitness-based selection of m of the individuals is performed. The mean and variance of the values in each locus of the selected solutions is then calculated using these individuals.

An assumption is made that the joint distribution of the selected individuals is multinormal and that it can be factorised into n univariate normal distributions. Therefore, all covariances between components of the solution vectors are assumed to be zero. In the next generation, the individual normal distributions corresponding to each individual component of the solution are sampled in order to generate a new population.

Table 4.3 illustrates the calculation of the values used to parameterise the normal distributions used in the sampling process. The four fittest members of the population have been selected and the mean and standard deviation of each of the four loci on their genotypes have been calculated. The mean value for the first locus is 17.75 with a standard deviation of 1.89. When a value is subsequently being generated for the first locus of each individual in the new population, a random sample is drawn from $N(17.75, 1.89)$. The algorithm iterates the sampling, selection and probability distribution updates until termination criteria are met.

Although the workings of the above algorithm are relatively straightforward, real-valued EDAs based on the normal distribution can be prone to

Table 4.3. Calculation of parameters for normal distributions using four fittest binary strings from the current population

Genotype	Locus 1	Locus 2	Locus 3	Locus 4
1	18	220	10	2
2	19	201	11	8
3	15	189	10	23
4	19	178	10	6
Mean	17.75	197.00	10.25	9.75
Std. Dev.	1.89	17.98	0.50	9.18

premature convergence as the value of the parameter for standard deviation can shrink too fast. Alternative variants of the algorithm using *adaptive variance scaling* seek to control the shrinkage of the standard deviation in order to prevent premature convergence [217].

4.7.3 Compact Genetic Algorithm

The compact genetic algorithm (cGA), developed by Harik, Lobo and Goldberg [252], is another univariate EDA. Like PBIL and UMDA it assumes that the variables are independent of each other and it ignores the possibility of epistatic relationships between the variables. It also assumes, in its canonical form, that the solutions are encoded as a binary string.

In implementing the cGA, a probability vector P of length n, where the problem is n-dimensional, is created. As with PBIL and UMDA, this vector represents the probability that a bit in a trial solution will take the value '1'. In generating each individual, n random numbers are generated in the range [0,1]. If a random number is less than the corresponding value in the probability vector P, a '1' is inserted into the corresponding locus of the trial solution being generated. The cGA differs from PBIL and UMDA in the way that the probability vector update is performed.

In updating the probability vector, two trial solutions are sampled using the current values in the probability vector and their fitnesses are calculated. Each component of the probability vector is then adapted in order to make the generation of the better trial solution more likely in the future. Hence, if the better trial solution has a '1' and the worse solution has a '0' in locus i, the value of P_i is adjusted by adding $1/K$ to it. On the other hand, if the better trial solution has a '0' and the worse solution has a '1' in locus i, the value of P_i is adjusted by subtracting $1/K$ from it. If both trial solutions have the same value in locus i, the value of P_i is left unchanged. During the adaptation process, the value of P_i is constrained to lie in the range [0,1] as only these values have meaningful interpretation. The size of the adjustment step depends on the value of K and typical values for this parameter are a function of the dimensionality of the problem. A simple choice suggested by [252] is to set $K = n$. The cGA iterates until all the components of the

probability vector are either 0 or 1, or until a run-time terminating condition is triggered.

Shortcomings of Univariate EDAs

While univariate EDAs (including PBIL, UMDA and cGA) can work well for problems where there are no significant interactions amongst the individual variables, this performance does not carry over to more complex problem settings where such interactions exist. Univariate EDAs only consider building blocks of order 1, and the joint probability distribution that they develop to model the solution vectors consists of the product of the univariate marginal probabilities of all the individual variables. In order to overcome this significant limitation, more complex EDAs have been created which can model multivariate fitness interactions. These include the Bayesian Optimisation Algorithm.

4.7.4 Bayesian Optimisation Algorithm

The Bayesian optimisation algorithm (BOA) [500, 501, 502, 503] is designed to search discrete (including binary) valued spaces in an effort to uncover the dependency structure (epistatic links) between individual elements of the solution. Like univariate EDAs, BOA uses a probability distribution, drawn from the promising solutions uncovered so far, in order to generate new solutions for testing.

The probability distribution is estimated using Bayesian network techniques for modelling the joint distribution of multinomial data. The use of this approach allows the uncovering of interaction effects between elements of the solution and is independent of the ordering of elements in the genotype representing the solution. This is in contrast to the canonical GA which, as outlined in Sect. 4.6, can find it hard to maintain building blocks which are widely separated on the genotype. Another feature of BOA is that it allows for the incorporation of prior information (if any) about likely regions of good solutions into the algorithm. The combination of prior information about the structure of the problem and the set of current solutions is used to guide the search process. The relative importance attached to each set of information can be varied during the algorithm's run.

A Bayesian network is an acyclic, directed graph where the nodes correspond to elements of genotype (for example, a bit) and the edges correspond to the conditional dependencies between the elements. Hence, the network structure encodes the nature of the dependencies or linkages between the elements of the genotype and the parameters of the network encode the conditional probabilities corresponding to the linkages between the elements. Hence, if the value of element y of the genotype depends on the value of element x, the

conditional distribution on the directed edge $x \rightarrow y$ will describe the distribution of values for y given any value of x. Figure 4.9 illustrates a series of nodes with a specific linkage structure.

In applications of Bayesian networks to some real-world problems, for example, the modelling of a medical diagnostic procedure, expert knowledge may exist to identify the nature of the links between variables representing risk characteristics, disease symptoms and the resulting clinical diagnosis. Hence, an expert may be able to identify the structure of the network (the links between the variables) based on domain knowledge. The task is then to identify the conditional probabilities on the edges connecting the nodes, representing for example the conditional probability that a person has disease y given the presence of multiple symptoms or based on the results from diagnostic tests.

More generally, in the context of binary-encoded genotypes, it may not be possible to clearly identify the nature of the links between the elements of a genotype. Hence, the object in applying BOA is to uncover both the correct linkage structure and the conditional probability distributions between the elements of the genotype.

As the algorithm runs, initial networks which attempt to uncover this linkage structure are generated, and the resulting Bayesian models are used to generate new members of the population. In turn, the fitness information in the new population is used to iteratively improve the Bayesian network (by altering it). The power of the Bayesian network approach is that, unlike univariate EDA approaches, it is capable of uncovering and modelling very complex linkage structures between the elements of the genotype. In contrast, a univariate EDA corresponds to a Bayesian network with no edges as all components of the solution vector are independent.

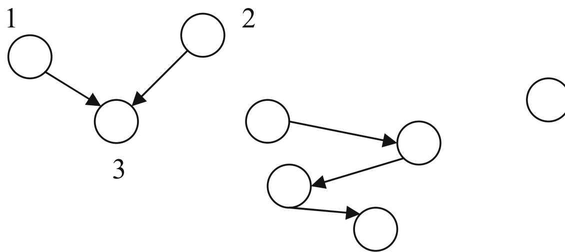


Fig. 4.9. Illustration of Bayesian network between eight variables. The rightmost node (variable) is not dependant on the value of any other variable (it is not linked to any other node). Other nodes display various interdependencies. Node 1 is independent of node 2. Node 3 is dependent on both nodes 1 and 2 (nodes 1 and 2 are *parents* of node 3)

Formally, a Bayesian network encodes a joint probability distribution of the form:

$$p(X) = \prod_{i=1}^{n} p(X_i|\Pi_i) \tag{4.13}$$

where $X = (X_1, \ldots, X_n)$ is a vector of all the variables in the genotype (X_i is the value at the ith position in the genotype), Π_i is the set of all the parents of X_i in the network (i.e., the set of all nodes that have a directed edge to X_i), and $p(X_i|\Pi_i)$ is the conditional probability of X_i given its parents Π_i.

The Algorithm

The canonical Bayesian optimisation algorithm is described in Pelikan, Goldberg and Cantú-Paz [502] (Algorithm 4.5). Initially, a random set of genotypes is generated and a selection process is undertaken to obtain a set $S(t)$ of the better-quality genotypes. A Bayesian network is fitted to these genotypes and a set of new solutions $O(t)$ is generated by sampling from the joint distribution encoded in the Bayesian network. These newly generated solutions are combined with the existing population $P(t)$ and a selection process is used to select the population for the next time step $P(t + 1)$ (Fig. 4.10).

Algorithm 4.5: Bayesian Optimisation Algorithm

Set $t = 0$;
Randomly generate an initial population of solutions $P(0)$;

repeat
    Randomly select a set of promising solutions $S(t)$ from $P(t)$;
    Construct a Bayesian network B that best fits the selected solutions;
    Generate a set of new strings $O(t)$ according to the joint distribution encoded by the Bayesian network B;
    Create a new population $P(t + 1)$ by replacing some strings from $P(t)$ with $O(t)$;
    Let $t = t + 1$;
until *terminating condition*;

Learning a Bayesian Network

There are two key steps in learning a Bayesian network, the learning of its structure and the discovery of the appropriate parameters (the conditional probabilities on the edges) for that structure. The learning of the structure is driven by the calculation of a quality metric or score for a network. The score depends both on the structure of the network and on how well it models the data. The score can also incorporate prior knowledge about the problem, if such knowledge exists. Typical scoring metrics include the Akaike Information

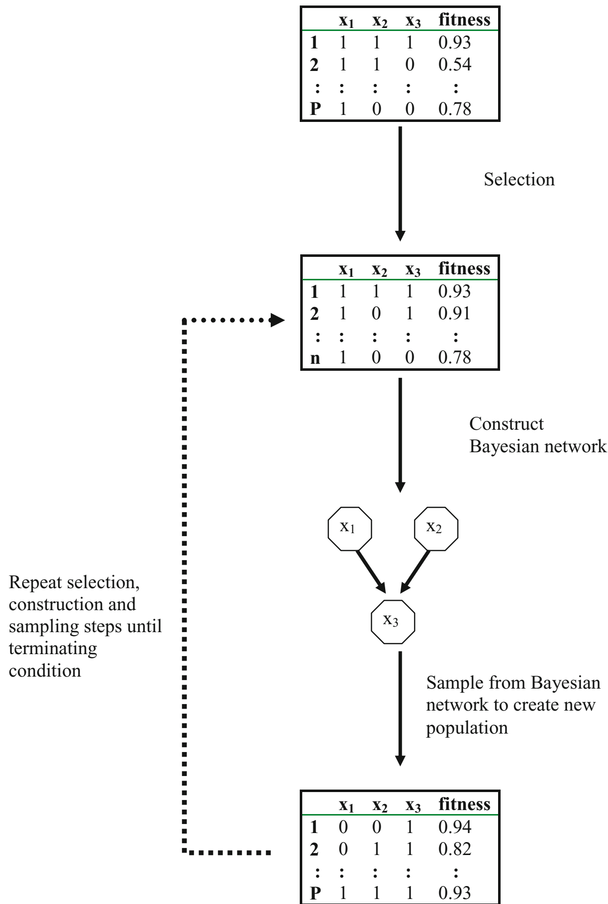


Fig. 4.10. After the initial population is generated (top), n items are randomly selected and used to induce the Bayesian network. A series of solutions is generated from this network; they are then combined with the existing population, and P members are selected to form the next generation. The selection, network induction and sampling steps are iterated until a terminating condition is triggered

Criterion (AIC) and Bayesian–Dirichlet metrics and the minimum description length, which seeks to minimise the number of bits used to describe the network and its associated parameters. Readers are referred to [500, 502] for further details of these.

The learning of the optimal network structure is a difficult problem, as the number of possible networks grows at the rate of $O(n!2^{n(n-1)/2})$ [534]. A six variable problem could therefore produce in excess of 3.7 million networks, making enumerative search for the best network in high-dimensional problem an impractical task. A common practical approach is to adopt a heuristic such as greedy search.

Having selected the scoring metric, an initial network is constructed, starting with an empty network or a randomly generated network. A local search process, employing basic graph operations such as edge additions, removals, or reversals is then applied to the network in an effort to improve its score, using greedy search. In other words, any valid alteration to the network which improves its score is accepted and the search proceeds onwards from the altered network. The graph operations are constrained in order to ensure that the resulting network is acyclic. The network construction process is stopped when the current network cannot be further improved. Once the network structure is determined, the conditional probabilities are calculated for each variable (node) given its parents. The probabilities are calculated using the training data.

Having learnt both the structure and the conditional probabilities, new members of the population can be generated by sampling from the network. In this step, the nodes are ordered so that values for parent nodes (which do not themselves have parents) are generated first. Then the values for nodes with parents are generated using the learnt conditional probabilities and the values already selected for their parent nodes. Table 4.4 illustrates an example of a set of conditional probabilities for the value of binary variable y based on the value of the binary variable x. Therefore, a value for y can be sampled, given a value of x and these conditional probabilities. In subsequent iterations of the BOA, the current network can act as the starting point for learning the network or the network can be constructed from scratch.

Table 4.4. Conditional probabilities for the value of y given the value for x

x	$p(y\|x)$	y
0	0.99	0
0	0.01	1
1	0.91	1
1	0.09	0

More powerful versions of BOA which can uncover and exploit hierarchical decomposition in solutions have been developed. The interested reader is

referred to [500] for a detailed treatment of Hierarchical BOA (hBOA). BOA has also been extended to encompass real-valued problem domains [5].

4.8 Summary

This chapter introduced a range of extensions to the canonical genetic algorithm. Many of these were inspired by the difficulties posed by real-world problems such as dynamic environments, constrained regions of feasibility and/or multiple objectives. Many of the concepts introduced in this chapter have general application across multiple families of natural computing algorithms and are not therefore limited to GAs.

In the following chapters we explore a range of other evolutionary inspired algorithms, including evolution strategies (Chap. 5), differential evolution (Chap. 6), and the automatic programing methodology of genetic programming (Chap. 7).

5

Evolution Strategies and Evolutionary Programming

In this chapter, two significant evolutionary algorithms, *evolution strategies* and *evolutionary programming*, are presented.

Evolution strategies (ES) was developed by Rechenberg and Schwefel [530, 531, 560, 561] in the 1960s and attracted a large following amongst researchers and practitioners, particularly in Europe. ES has been extensively used as a tool for solving real-valued optimisation problems and has also been successfully applied for combinatorial optimisation, for constrained optimisation and for multiobjective optimisation.

Two notable characteristics of ES are that it typically uses a real-valued representation (although binary and integer versions of ES exist) and that it relies primarily on selection and mutation to drive the evolutionary process. Most applications of ES embed *self-adaptation*, in that the algorithm alters its rate of diversity-generation in response to feedback during the optimisation process. Although the idea of adaptive EAs has become widespread in recent years, ES was the first family of EAs to embed self-adaptation as an integral part of its algorithm. A comprehensive history of the development of ES and its theory is provided in [53] and [54].

Evolutionary Programming (EP) has its origins in the work of Fogel during the 1960s [195]. EP adopts a general evolutionary framework and is independent of any particular representation. Unlike the GA there is no concept of a genotype. Instead emphasis is placed on the phenotype, which is iteratively adapted in order to improve its fitness. Consequently, in standard EP there is no concept of 'genetic' crossover, unlike GA where crossover is used to exchange useful building blocks between candidate solutions. Although EP was not initially developed for optimisation applications, it has subsequently been applied for real-valued and combinatorial optimisation.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_5

5.1 The Canonical ES Algorithm

Although people often refer to ES (or indeed the GA) as if it were a single algorithm, it is more accurate to view ES as a family of algorithms. Initially, in Algorithm 5.1 we describe a basic (non-self-adaptive) form of ES, with a number of variants on this algorithm being described in the following sections (see Algorithm 5.2 later in this chapter). Assume that the object is to find the vector of real numbers $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ which is associated with the extremum of a function $f : \mathbb{R}^n \rightarrow \mathbb{R} : x \longmapsto f(x)$.

Algorithm 5.1: Non-Self-adaptive Evolution Strategies $(\mu + \lambda)$

```
Randomly create an initial population {x^1, ..., x^μ} of parent vectors, where
each x^i is of the form x^i = (x^i_1, ..., x^i_n), i = 1, ..., μ;
Evaluate each member of the population;

repeat
    repeat
        Randomly select (individual) parent from {x^i : i = 1, ..., μ};
        Create child vector by applying a mutation operator to the parent;
    until λ children are generated;
    Rank the μ + λ parents and children from best (those producing least
    error on the training dataset) to worst;
    Select the best μ of these to continue into the next generation;
until terminating condition;
```

There are many ways that the above pseudocode can be adapted in order to produce alternative ES algorithms. The population can be initialised in different ways (random or not) and the generation of variety when creating children could arise solely from mutation or could also employ a recombination operator. Multiple selection and replacement strategies could also be applied. The following subsections discuss a number of these possibilities.

5.1.1 (1 + 1)-ES

In the initial studies of evolution strategies [530, 560] attention was focussed on a single parent, single child, scheme known as (1+1)-ES. In this scheme the entire population consists of a single parent which gives rise to a single child. The parent and its child then compete to survive into the next generation, with the poorer solution being eliminated. Hence, this scheme consists of a point-to-point search of the solution space. A practical problem with this approach is that it can result in slow progress towards better regions of the solution space.

5.1.2 $(\mu + \lambda)$-ES and (μ, λ)-ES

After initial experimentation with the $(1 + 1)$ scheme, attention was focussed on single parent, multiple children, schemes $(1, \lambda)$ and $(1 + \lambda)$. In these approaches, the single parent produces λ children. Each of the children are generated by mutating the individual elements of the parent's vector using a random draw from $N(0, \sigma)$, a normal distribution with a mean of 0 and a user-defined standard deviation of σ. In the case of $(1, \lambda)$ the fittest of the λ children survives into the next generation, whereas in the case of $(1 + \lambda)$, the original parent is only replaced if one of its λ children is fitter than it.

More generally, there can be multiple parent, multiple children, ES schemes, denoted as $(\mu + \lambda)$-ES and (μ, λ)-ES. In both of these schemes, parents are randomly selected (with replacement) from μ. After these parents are mutated to produce λ children $(\lambda \gg \mu)$, a selection process is implemented to decide which of the parents and children survive into the next generation. As the value of λ increases the selective pressure of the ES algorithm also increases.

The nature of the selection step depends on which of the schemes is being used. In the first scheme, $(\mu+\lambda)$, the entire population of parents and children compete for survival into the next generation, with the best μ succeeding. In the second scheme, all survivors are selected from the set of children λ so each individual has a single-period life span.

Therefore, $(\mu+\lambda)$ corresponds to an elitist replacement strategy, as a good solution in the initial population can only disappear if replaced by a better child, whereas the generational (μ, λ) scheme does not guarantee the survival of a good parent into the next generation. Despite this, the (μ, λ) replacement scheme is more commonly used in ES applications as the discarding of all parents makes it easier for the population to migrate away from local optima (a temporary reduction in fitness may be required to allow this).

Another potential problem of the $(\mu+\lambda)$ replacement scheme is that it can hinder effective self-adaptation of the mutation rate in an ES (Sect. 5.1.3), as under a $(\mu + \lambda)$ scheme, individuals with good current fitness but poor strategy parameters (for example, mutation rates) can persist for a long time in the population, slowing down the rate of population fitness improvement.

5.1.3 Mutation in ES

Each individual in ES is typically represented as a vector of real values, $x = (x_1, \ldots, x_n)$. Associated with this vector is a set of strategy parameters. This set may contain a single value, or alternatively, one or more values for each element in the solution vector. The elements of the strategy vector guide the mutation process in ES and are endogenous in that they can themselves self-adapt as the algorithm runs.

In the simplest case of mutation, initially ignoring self-adaption, a single strategy vector value σ could be set. In applying the mutation step to a parent's solution vector, a child vector is then formed by additively applying

a random number r drawn from a Gaussian distribution $N(0, \sigma)$ (where the value of σ is chosen by the user) to each element of the parent vector. Hence, the vector $x(t)$, becomes $x(t+1) = x(t) + r$. Assuming that the value of σ is relatively small, most mutations will be correspondingly small, with occasional larger mutations. The parameter σ is referred to as the mutation step size as it plays a critical role in determining the effect of the mutation operator.

While this mutation mechanism is easy to implement, it suffers from two obvious problems:

i. it fails to consider the scaling of each dimension in the solution space, and
ii. the mutation step size is not sensitive to the stage of the search process.

The first problem arises as differing elements of an individual's solution vector may have widely varying scale. Hence, a suitable value of σ for one dimension may not be suitable for another. The second problem also arises due to the fixed value for σ. Early in the search process a large value of σ may be useful in promoting explorative search in order to find a good region of the search space, whereas later in the search process when finer-grained search is required, a smaller value would be preferred. Hence, as discussed in Sect. 3.7, it is likely that the optimal mutation values along each dimension will alter during the optimisation run.

5.1.4 Adaptation of the Strategy Parameters

Two basic approaches can be used in adapting strategy parameters (here governing the mutation size) as the algorithm runs. A deterministic scheme such as the *1/5 success rule* can be applied, or the strategy parameter(s) can be allowed to evolve.

The 1/5 Success Rule

In an early attempt at dynamically varying the mutation parameter, Rechenberg [531] proposed the 1/5 success rule for the $(1+1)$ scheme. Under this rule the ratio (denoted as ϕ) of successful mutations (ones which produce a child of higher fitness than its parent) to the total number of mutations is measured periodically during the optimisation run, with σ being altered as necessary in order to approach a ratio of 1/5. If ϕ was greater than 1/5, the standard deviation of the mutations was increased, and vice versa if ϕ was less than 1/5.

The rationale behind this approach was that if many successful mutations are being created, the current solution is poor, and hence larger step sizes in the search space are warranted. Conversely, if most mutations lead to poorer results, the current solution is at least locally good; hence a finer-grained search around the current solution is indicated. Hence, the adaptive rule for the variance of mutations is:

$$\sigma = \begin{cases} \sigma g & \text{if } \phi(k) < 1/5 \\ \sigma/g & \text{if } \phi(k) > 1/5 \\ \sigma & \text{if } \phi(k) = 1/5 \end{cases} \tag{5.1}$$

where k is the number of generations between each adaptation of σ, and $\phi(k)$ is the portion of successful mutations in the population (here a single individual) over the last k iterations of the ES algorithm. The choice of a suitable value of g depends on the problem at hand. If N (number of dimensions of the solution space) is large (≥ 30), then values of $k = N$ and $0.85 < g < 1$ may be a good starting point [54].

This approach leads to the self-adaptation of mutation step sizes based on global feedback from the performance of the algorithm. However, the approach, while simple, does have drawbacks. It is restricted to the case where there is one strategy parameter, it is designed for the $(1+1)$ scheme, and it does not consider that each dimension of the solution vector may have a different scaling.

Self-adaptive Mutation Step Size

Rather than using a deterministic scheme to alter the mutation step size, the step size can instead be coevolved along with the solution vector as the algorithm iterates [22, 561]. This allows the mutation step to vary in size at different times during the run.

Single-Step-Size Mutation

In the simplest case, each individual in the population is represented as a vector $v = ((x_1, \ldots, x_n), \sigma)$ where $(x_1, \ldots, x_n)$ are the real values corresponding to the solution variables and σ is a real-valued strategy parameter which governs the mutation process for that individual.

In each iteration of this version of ES, the strategy value for each individual is initially mutated using a multiplicative process. This ensures that the resulting value of $\sigma(t)$ is positive. The new value for this parameter, $\sigma(t+1)$, is then additively used to create the new solution values for the child. The updated solution and strategy values for a child (denoted $(x(t+1), \sigma(t+1))$) can be created as follows:

$$\sigma(t+1) = \sigma(t) \cdot \mathrm{e}^{(\tau r)} \tag{5.2}$$

$$x_i(t+1) = x_i(t) + \sigma(t+1) r_i, \quad i = 1, \ldots, n \tag{5.3}$$

where $i = 1, \ldots, n$ are the n elements making up the child, and r and $r_1, \ldots, r_n$ are independent identically distributed standard Gaussian random variables (that is, drawn from $N(0,1)$). The value of τ is defined by the user and this learning parameter critically affects the rate of self-adaptation. Typically, the learning rate is inversely proportional to the square root of the problem size:

$$\tau \propto \frac{1}{\sqrt{n}}. \tag{5.4}$$

In allowing the mutation step size to coevolve with the solution vector, it is important to keep the above ordering of the two steps in the mutation process. First the value of σ is mutated and then this new value is used to create the new solution vector $(x_1, \ldots, x_n)$. Hence, the fitness evaluation of the new solution vector also indirectly evaluates the utility of the mutated strategy vector.

n-Step-Size Mutation

Under the above approach the mutation step size is the same for each dimension. As already discussed, this is likely to produce an inefficient search as the fitness landscape can have different slopes in each direction and hence suitable step sizes along each dimension will usually not be identical.

A solution to this problem is to use a separate strategy parameter for each dimension, so $v = ((x_1, \ldots, x_n), \sigma_1, \ldots, \sigma_n)$ such that a different σ value can be set for each dimension. A similar mutation approach is then applied as in the single step size case, where:

$$\sigma_i(t+1) = \sigma_i(t) \cdot \mathrm{e}^{(\tau' r' + \tau r)} \tag{5.5}$$

$$x_i(t+1) = x_i(t) + \sigma_i(t+1) r_i, \quad i = 1, \ldots, n \tag{5.6}$$

where $\tau' \propto 1/\sqrt{n}$ and $\tau \propto 1/\sqrt{2\sqrt{n}}$; and r', r and $r_1, \ldots, r_n$ are all independent identically distributed random variables drawn from a standard normal distribution (with mean 0 and standard deviation 1). More complex methods of adapting the mutation size have been studied and the interested reader is referred to [562].

5.1.5 Recombination

Although initial implementations of ES concentrated on using a mutation operator for generating diversity, ES can also include a recombination or crossover operator. In this case, the canonical algorithm is altered in that λ children are initially created by iteratively applying the recombination operator and these children are then subject to a mutation operator as already described. The recombination operator in ES typically produces a single child.

ES do not impose a requirement that a child must have two parents. In order to denote the number of parents used, the standard ES notation $(\mu + \lambda)$-ES and (μ, λ)-ES can be extended to $(\mu/\rho, +\lambda)$-ES and $(\mu/\rho, \lambda)$-ES where $\rho \leq \mu$ (ρ is known as the *mixing number*) is the number of parents that combine to produce a single child. In applying the recombination operator repetitively to generate μ children, parents can be selected using a fitness-based methodology such as ranking (Sect. 3.6.1), but usually parents are selected randomly.

Two approaches to recombination are commonly seen in ES, discrete and intermediate. In discrete recombination, one of the parent values is randomly selected for each locus on the string of real values making up the child solution vector. In intermediate recombination, the parental values are averaged in creating the value for each locus on the child. More generally, intermediate recombination can be any linear combination of the two parent values. Figure 5.1 illustrates each form of recombination in the case where a child has two parents.

In Fig. 5.1 it is assumed that two parents ($\rho = 2$) are chosen from the population in order to produce a child. This is referred to as *local recombination*. A second form of recombination called *global recombination* is also used in ES. In global recombination, for each locus of the child's vector, two (or more) parents are randomly drawn from the population and the child's value is obtained using the values from the random parents as already discussed. The process of drawing two random parents is repeated for each locus of the child, until the child vector is filled. Global recombination is commonly used in ES.

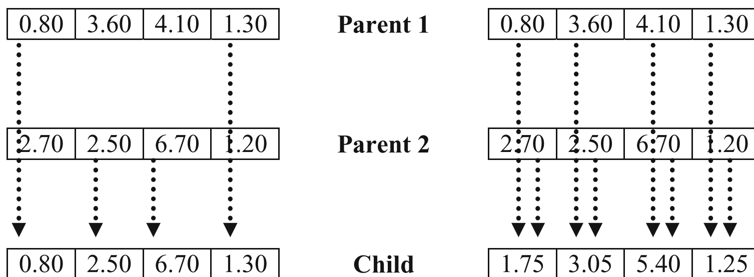


Fig. 5.1. Discrete (left) and intermediate (right) recombination with two parents

Recombination can also be applied to the strategy vectors of individuals. Typically, discrete recombination is applied to the solution variables, with intermediate recombination being applied to the strategy elements of the individual. The rationale for this is to smooth the adaptation of the strategy parameters, rather than having substantial changes in mutation step sizes from one generation to the next. If a recombination operator is added to the ES algorithm, the pseudocode is as presented in Algorithm 5.2.

Covariance Matrix Adaptation Evolution Strategies

One notable variant of ES, developed by Hansen, Ostermeier and Gawelczyk [246, 490], which has produced excellent results in real-valued optimisation problems, is Covariance Matrix Adaptation Evolution Strategies. The essence

Algorithm 5.2: Evolution Strategies with Self-adaptation and Recombination $(\mu + \lambda)$

Create an initial population $\{x^1, \ldots, x^\mu\}$ of parent vectors, each x^i being of the form $x^i = (x^i_1, \ldots, x^i_n)$, $i = 1, \ldots, \mu$;

repeat
  repeat
    Select ρ parents from $\{x^i : i = 1, \ldots, \mu\}$;
    Recombine the ρ parents to form a child, forming both a new solution vector and a new strategy vector;
    Mutate the strategy vector for the child;
    Mutate the solution vector of the child using its newly mutated strategy vector;
  until λ *children are created*;
  Rank the $\mu + \lambda$ parents and children from best (those producing least error on the training dataset) to worst;
  Select the best μ of these to continue into the next generation;
until *terminating condition*;

of the CMA approach is that it allows for correlated mutations in real-valued search spaces. The mutation distribution is generated from a covariance matrix which itself adapts during the evolutionary process. This allows the mutations to better adapt to the fitness landscape, thereby facilitating the optimisation process. For further details readers are referred to [244, 245, 246].

5.2 Evolutionary Programming

Evolutionary programming (EP) was developed by Lawrence Fogel in the early 1960s [195]. The initial application of EP was to simulate an evolutionary process in a population of finite state machines in order to study the development of intelligent, problem-solving behaviour [193]. Hence, unlike ES, the early focus of EP was the adaption of an individual's set of behaviours rather than the evolution of a set of solution variables for an optimisation problem.

While EP does bear similarities with other EAs, it adopts a more abstract view of the evolutionary process. Rather than considering the evolution of an underlying genetic code, EP focusses on applying an evolutionary process directly to the structures of interest (i.e. the phenotypes). EP does not require the use of a specific form of representation (for example, real-valued or integer strings), allowing the user to select the most suitable representation for the problem at hand. In applications of EP these representations have included (amongst many others) finite state machines and graphs [194]. Whatever the choice of representation, EP uses an iterative improvement process whereby

a population of parent structures are perturbed using a suitably defined mutation operator, with a selection process taking place to see which structures survive into the next iteration of the algorithm. An overview of the canonical EP algorithm is provided in Algorithm 5.3.

Algorithm 5.3: Evolutionary Programming Algorithm

Select the representation for the structures which are being evolved;
Let $t := 0$;
Randomly create an initial population of parent structures $P(t)$;
Evaluate each member of the population;

repeat
  Apply a mutation operator to members of $P(t)$ to create a set of child structures $P'(t)$;
  Evaluate fitness of all members of $P'(t)$;
  Apply selection process to obtain $P(t+1)$ from $P(t) \cup P'(t)$;
  Let $t := t + 1$;
until *terminating condition*;

A critical aspect in applying EP to a problem is the design of an appropriate mutation operator. For example, consider the travelling salesman problem (TSP) (further background on the TSP is provided in Sect. 9.3.2). In the TSP there is a network of n cities. Each route, or arc, between two cities has a distance or cost associated with it and the object is to find the tour which minimises the distance travelled in visiting all the cities, returning to the starting city. A TSP tour therefore consists of an ordered list of n integers where each integer corresponds to the index for a city (each integer appears only once on the list). For any ordered list the quality of that solution can be determined by summing the distance between each city pair on the list, including the distance from the last city on the list back to the starting city.

Parent	1	3	4	6	2	5	1	3	4	6	2	5
			X		X			X			X	
Child	1	3	2	6	4	5	1	2	6	4	3	5

Fig. 5.2. Two forms of mutation in a six city TSP, pair swapping (left) and inversion (right)

In order to apply the EP framework to this problem, a mutation operator must be defined. An example of a mutation operator is one that switches pairs of cities in an ordered list; another is to allow route inversion (Fig. 5.2). In pair

switching, two cities are randomly chosen and they are swapped in the tour ordering. In tour inversion, two cities are randomly selected, and the ordering of cities between the two selected points is reversed.

EP implements a very general evolutionary framework and is not closely bound to a biological metaphor. There is no requirement that the population size must remain constant, or that each parent can only generate a single child. As there is no concept of a genotype, crossover is not typically used in EP.

Numerical Optimisation with EP

A stream of literature has emerged over the years applying EP to a wide variety of problems including combinatorial and real-valued optimisation ([193] lists a sample of these studies). If the general EP framework is adapted in order to be applied to real-valued optimisation, the resulting algorithm bears strong similarities with ES without recombination. Individual solution vectors are encoded as real-valued vectors and a natural choice of mutation mechanism for these is a multivariate, zero-mean Gaussian distribution. In contrast to the deterministic, rank-based, selection typical of ES, real-valued optimisation applications of EP often use stochastic, tournament selection. Hence, even poorer solutions have some chance of survival into the next iteration of the algorithm.

5.3 Summary

Evolution strategies and evolutionary programming, along with the genetic algorithm, are amongst the oldest families of evolutionary algorithms. Each of the methodologies has been extensively studied and they have been applied to solve a wide range of real-world problems. While ES and EP bear substantial comparison with the GA, they place emphasis on evolution at a phenotypic rather than at a genotypic level. Canonical versions of ES and EP place emphasis on mutation rather than on crossover as a means of generating diversity.

In Chap. 6, the *Differential Evolution* algorithm, which also draws from an evolutionary metaphor, is introduced.

6

Differential Evolution

Differential evolution (DE) [520, 599, 600, 601] is a population-based search algorithm. It bears comparison with evolutionary algorithms such as the GA as it embeds implicit concepts of mutation, recombination and fitness-based selection. Like the GA, DE iteratively generates good solutions to a problem of interest by manipulating a population of solution encodings. DE also bears comparison with evolutionary strategies (Chap. 5), as its mutation step, when generating diversity in its population of solution encodings, is self-adapting.

6.1 Canonical Differential Evolution Algorithm

Although several variants of the DE algorithms exist, initially we will describe the canonical real-valued version of the algorithm based on the *DE/rand/1/bin* scheme [600].

At the start of the algorithm, a population of n d-dimensional vectors $\{x^j = (x_1^j, x_2^j, \ldots, x_d^j) : j = 1, \ldots, n\}$, each of which encodes a solution, is randomly initialised (assuming there is no domain knowledge which would suggest where the global optimum lies) and evaluated using a fitness function f. (Superscripts indicate the index of vector x^j in the population, while subscripts indicate the component.) During the search process, each individual x^j of the n members of the population is iteratively refined. The modification process has three steps:

i. create a *variant vector* which encodes a solution, using randomly selected members of the population (similar to a mutation step);
ii. create a *trial vector*, by combining the variant vector with x^j's vector (crossover step); and
iii. perform a *selection process* to determine whether the newly created trial vector replaces x^j's current solution vector in the population.

The canonical DE algorithm is outlined in Algorithm 6.1 and is illustrated in Fig. 6.1.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_6

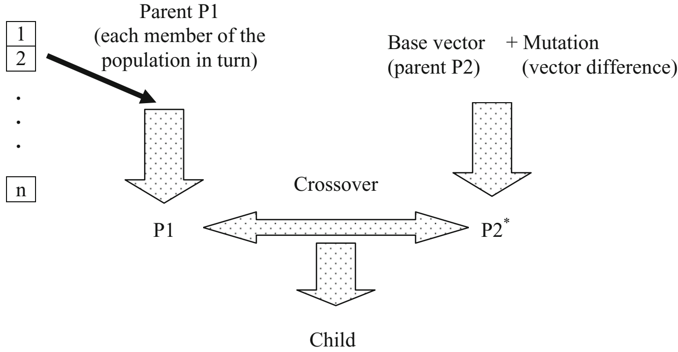


Fig. 6.1. DE variety-generation process

Algorithm 6.1: Differential Evolution Algorithm

Create an initial population $\{x^1, \ldots, x^n\}$ of n random real-valued vectors;
Decode each vector into a solution;
Evaluate fitness of each solution;

repeat
  for *each vector* $x^j \in \{x^1, \ldots, x^n\}$ **do**
    Select three other vectors randomly from the population;
    Apply difference vector to base vector to create variant vector;
    Combine vector x^j with variant vector to produce new trial vector;
    Evaluate the fitness of the new trial vector;
    if *trial vector has higher fitness than* x^j **then**
      Replace x^j with the trial vector;
    end
  end
until *terminating condition*;

Mutation Operator

In the mutation step, a variant vector $v^j(t+1)$ is created from each vector $x^j(t)$ as follows:

$$v^j(t+1) = x^m(t) + F \cdot (x^k(t) - x^l(t)) \tag{6.1}$$

where $k, l, m \in 1, \ldots, n$ are mutually distinct, randomly selected indices (all $\neq j$), x^m is referred to as the base vector, and $x^k(t) - x^l(t)$ is referred to as a difference vector. The variant vector is therefore a mutated version of an existing parent (or *base*) vector. Selecting the three indices randomly implies that all members of the current population have a chance of acting as a parent

and also have a chance of playing a role in the mutation process. The difference between vectors x^k and x^l is multiplied by a scaling parameter F (typically $F \in (0, 2]$). The scaling factor controls the amplification of the difference between x^k and x^l and is used to avoid stagnation of the search process.

Index Number	$X_j(t)$	$V_j(t+1)$		$U_j(t+1)$
1	a	q	rand>CR	a
2	b	w	rand≤CR	w
3	c	e	rand≤CR	e
4	d	r	rnbr=4	r

Fig. 6.2. An example of crossover in DE

A notable attribute of the mutation step in DE is that it is *self-scaling*. The size of mutation along each dimension stems solely from the location of the vectors in the current population. The mutation step self-adapts in size (and direction) as the population converges leading to a finer-grained search.

In contrast the mutation process in the canonical GA is typically based on draws from a separately defined, fixed probability density function. The effective size of the mutation step will usually therefore be different for each element of the solution chromosome, corresponding to the scaling of that element.

Trial Vector

Following the creation of the variant vector, a trial vector $u^j(t+1) = (u_1^j, u_2^j, \ldots, u_d^j)$ is obtained using:

$$u_k^j(t+1) = \begin{cases} v_k^j(t+1), & \text{if } (r \leq r_{\text{cross}}) \text{ or } (j = i_{\text{rand}}) \ ; \\ x_k^j(t), & \text{if } (r > r_{\text{cross}}) \text{ and } (j \neq i_{\text{rand}}). \end{cases} \tag{6.2}$$

where $k = 1, 2, \ldots, d$, the term r is a random number generated in the range (0,1), r_{cross} is a user-specified crossover constant from the range (0,1), and i_{rand} is a randomly chosen index from the range $1, 2, \ldots, d$. The random index is used to ensure that the trial solution differs in at least one locus from $x^j(t)$.

The crossover rate r_{cross} can be considered as a form of mutation rate, as it controls the probability that a component will be inherited from a variant (mutant) vector. Figure 6.2 provides an illustration of the crossover operator in DE.

Selection

As per (6.3) the resulting child (or *trial*) solution replaces its parent if it has higher fitness (a form of greedy selection which also ensures 'elitism' in that the best solution so far is always retained). Otherwise the parent survives unchanged into the next iteration of the algorithm. Figure 6.3 illustrates the diversity-generation process in DE.

$$x^j(t+1) = \begin{cases} u^j(t+1), & \text{if } f(u^j(t+1)) > f(x^j(t)); \\ x^j(t), & \text{otherwise.} \end{cases} \tag{6.3}$$

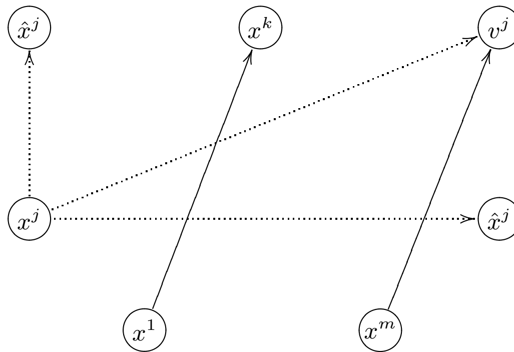


Fig. 6.3. A representation of the differential evolution variety-generation process. The value of F is set at 0.50. In a simple 2-d case (as here), the child of particle x^j can end up in any of three positions. It may end up at either of the two positions $\hat{x}^j$, or at the position of particle $v^j = v^j(t+1)$

Example of DE

Figure 6.4 provides a simple numerical example of DE. The parent vector is $i = 1$, three other vectors are randomly chosen to create the variant vector, and $F = 1$ is assumed. When crossover is applied between the parent and the variant vector, the first and the third elements of the variant vector are assumed to combine with the second element of the parent vector to create the trial vector. Finally, it is assumed that the fitness of the trial vector exceeds that of its parent and therefore it replaces its parent.

Parameters in DE

The DE algorithm has a small number of key parameters: the population size n, the crossover rate r_{cross}, and the scaling factor F. Higher values of

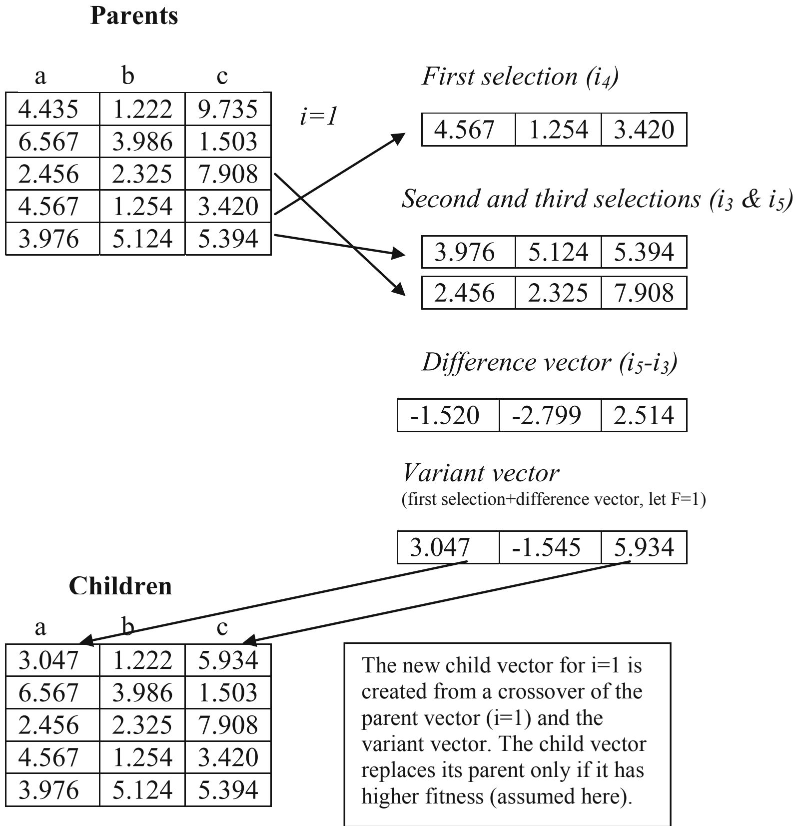


Fig. 6.4. Numerical example of the canonical DE algorithm

r_{cross} tend to produce faster convergence of the population of solutions. As would be expected for any algorithm, good choices of parameter values will be application-specific, but as a starting point, typical values for these parameters are in the ranges, n=50–100 (or five to ten times the number of dimensions in a solution vector), r_{cross}=0.4–0.7 and F=0.4–0.9 for the DE/rand/bin scheme [600]. See [131] for a general discussion on parameter setting in DE. More generally, just as for the GA, it is possible to design variants of the canonical DE algorithm which 'self-adapt' their parameter settings as the algorithm iterates [679].

6.2 Extending the Canonical DE Algorithm

The different variants of the DE algorithm are described using the shorthand DE/x/y/z, where x specifies how the base vector (of real values) is chosen (*rand* if it is randomly selected, or *best* if the best individual in the population is selected), y is the number of difference vectors used in generating a variant vector, and z denotes the crossover scheme (*bin* for crossover based on independent binomial experiments, and *exp* for exponential crossover).

6.2.1 Selection of the Base Vector

In creating the variant vector, a vector difference is applied to a base vector. There are a multitude of ways that the base vector can be selected. One alternative is to use the highest-fitness member of the current population (*DE/best/1*), for example:

$$v^j(t+1) = x^{\text{best}}(t) + F \cdot (x^k(t) - x^l(t)). \tag{6.4}$$

This bears similarity with the use of g^{best} in the particle swarm algorithm (Chap. 8), as the current best member of the population has an impact on the generation of all trial vectors. This implicitly increases the selection pressure in the algorithm and tends to decrease the diversity of the pool of trial vectors. Other selection methods, for example tournament selection, could be used in order to bias the selection of the base vector towards better members of the current population without forcing the selection of the best vector.

6.2.2 Number of Vector Differences

More than one vector difference could be used in the creation of the variant vector. For example, if the *DE/rand/2* scheme is applied, two vector differences and five randomly generated indices are used in calculating the variant vector.

$$v^j(t+1) = x^m(t) + F \cdot (x^k(t) - x^l(t)) + F \cdot (x^q(t) - x^p(t)). \tag{6.5}$$

Alternatively, if the *DE/best/2* scheme is applied, the calculation of the variant vector becomes:

$$v^j(t+1) = x^{\text{best}}(t) + F \cdot (x^k(t) - x^l(t)) + F \cdot (x^q(t) - x^p(t)). \tag{6.6}$$

Increasing the number of vector differences will tend to increase the diversity of the trial vectors generated. A variant on this approach is *DE/current-to-best/1*, where the randomly selected parent is combined with the best vector to create a vector difference, which is then added to a second vector difference

arising from two randomly selected members of the population (K and F are scaling constants).

$$v^j(t+1) = x^m(t) + K(x^{\text{best}}(t) - x^m(t)) + F \cdot (x^r(t) - x^s(t)). \tag{6.7}$$

Hence, in this variant of DE, the best member of the population influences the 'mutation' step in the creation of all child solutions.

6.2.3 Alternative Crossover Rules

Instead of using binary crossover, an exponential crossover operator could be applied (*DE/best/1/exp*). Under this scheme a series of sequential elements are copied from the variant vector to the trial vector, starting from a randomly chosen point in the variant vector. The sequential copying process continues as long as a series of numbers randomly drawn from $U(0,1)$ is $\geq r_{\text{cross}}$ (a threshold value). In Table 6.1, an integer is randomly selected in the range 1 to 4 (say 1). For each indexed element i after this, a random number r_i is drawn from $U(0,1)$. While this number $r_i \leq r_{\text{cross}}$ (or until the end of the vector is reached), $u^j(t+1)$ is selected from the variant vector.

Index No	$x^j(t)$	$v^j(t+1)$	Comment	$u^j(t+1)$
1	a	q	from $x^j(t)$	a
2	b	w	$r_2 \leq r_{\text{cross}}$	w
3	c	e	$r_3 \leq r_{\text{cross}}$	e
4	d	r	$r_4 > r_{\text{cross}}$	d

Table 6.1. Exponential crossover

6.2.4 Other DE Variants

Das et al. [132] suggested that rather than holding the value of F constant as the DE algorithm iterates, it could be allowed to vary. This could be done randomly, for example, $F = 0.5(1+r)$, where r is a random number drawn from the uniform distribution $U(0,1)$ (known as *DE with random scale factor*). Alternatively, it could be decreased linearly during the optimisation run from an upper to a lower bound (known as *DE with time varying scale factor*), for example,

$$F = F_{\max} - F_{\min} \cdot \frac{\text{iter}_{\max} - \text{iter}_{\text{current}}}{\text{iter}_{\max}}.$$

The first approach aims to reduce the chance of the search process stagnating at a local optimum, while the second aims to encourage diverse searching early

in the optimisation run, with a finer degree of search later in the optimisation run. In the Das study, both approaches were found to outperform canonical DE across a range of test functions.

Norma and Iba [453] proposed *Fittest Individual Refinement* (FIR), where the canonical form of DE is supplemented by a crossover-based local search step (XLS) in order to assist in finding the optimum solution. In essence, this results in a memetic variant (see Sect. 4.5) of DE as a local search is undertaken around the best individual after each iteration of the algorithm by selecting it as breeding stock, mating it with a number of newly created variant vectors, and then determining whether any of the child vectors generated have higher fitness.

As for other EAs, it is possible to create many DE variants such as:

- multiple-population DE, whereby several subpopulations search independently, with periodic migration of information between the subpopulations, and
- hybrid DE algorithms which embed (for example) elements taken from other search algorithms.

6.3 Discrete DE

DE was initially developed for real-valued optimisation and the variety-generating operators in the algorithm assume that real-valued encodings are being employed.

DE can be modified and applied to discrete binary or integer-valued problems. One approach to tackling integer-encoded problems is to use quantisation techniques. A simple example of a quantising function is the floor function $\lfloor\ \rfloor$, which converts a real value to an integer by dropping all values to the right of the decimal point, e.g., floor(9.36) = $\lfloor 9.36 \rfloor = 9$. This is an example of uniform quantisation, as continuous values are transformed into a series of evenly spaced integers. Price et al. [521] provide a detailed discussion of how DE can modified and applied to optimise functions with discrete or mixed parameters.

Angle-Modulated DE

An alternative approach, which extends DE to binary encodings, has been developed by Pampara and Engelbrecht [491, 493]. This uses a generating function which produces a binary output from real-valued inputs. The design of the generating function is inspired by the concept of *angle modulation*, which is used in electronic engineering. A trigonometric function can be used to perform angle modulation:

$$g(x) = \sin(2\pi(x-a)b\cos(A)) + d \tag{6.8}$$

where

$$A = 2\pi c(x - a) \tag{6.9}$$

The values of the coefficients a, b, c, d determine the precise shape of the generating function, a represents the horizontal shift of the function, b controls the maximum frequency of the sin function, c controls the frequency of the cos function and d controls the vertical shift of the function. Figure 6.5 illustrates a generating function for the values $a = 0$, $b = 1$, $c = 1$ and $d = 0$.

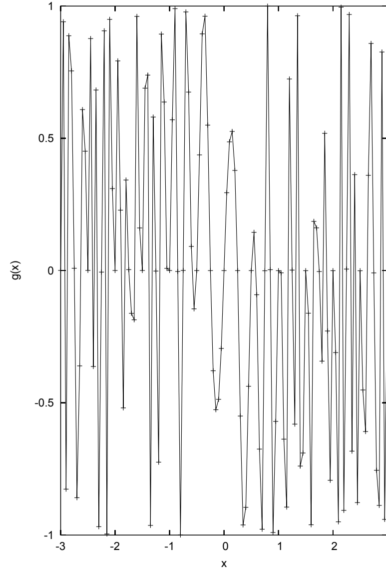


Fig. 6.5. Generating function from (6.9) and (6.8) where $a = 0$, $b = 1$, $c = 1$ and $d = 0$

The value x is taken from evenly spaced intervals drawn from a real number range (say $[-3, 3]$). If the problem of interest requires (for example) binary strings of length 30 bits, then 30 evenly spaced sample values for x are taken from $[-3, 3]$. Each of these values in turn is inserted into (6.9) and (6.8) in order to generate a single output $g(x)$. If the output value $g(x)$ for an individual value of x is positive, a bit value of 1 is assigned. If the output value is negative, the bit value is 0. The binary string is therefore built up, one term at a time, by taking the 30 real values of x and passing them sequentially through the generating function. Once the entire binary string of 30 bits has been generated, its fitness is determined.

Hence, for a specific set of real-values for a, b, c, d, a single binary string is generated. In essence therefore, the original (high-dimensional) binary-valued search space is mapped to a lower-dimensional (4-d) real-valued search space

and DE is applied to the population of four-dimensional tuples, each of which decodes into a binary string. The pseudocode for the algorithm is provided in Algorithm 6.2.

Algorithm 6.2: Angle-Modulated Differential Evolution

```
Create an initial population of m four-dimensional real-valued tuples
randomly from the range [−1, 1];
Select n equally spaced values x_1, ..., x_n from the range (say) [−3, 3];
Use the n values of x and the generating function to transform the m
four-dimensional real-valued tuples into m binary strings of length n;
Evaluate fitness of each binary string and associate this fitness with its
real-valued tuple;

repeat
    Select individuals from population of four-dimensional real-valued tuples
    for generation of a trial vector;
    Produce the (real-valued) trial vector;
    Generate the corresponding binary string;
    Evaluate the fitness of this string;
    if trial tuple has higher fitness than parent tuple then
        Replace parent tuple with the trial tuple;
    end
until terminating condition;
```

Angle modulation transformation is an interesting way of permitting the general application of real-valued optimisation algorithms to binary-valued problems. Thus far, the reported results from using the approach are promising; however, further investigation across a range of binary and discrete optimisation problems is required to fully test the utility of the method. In particular the locality of the real-to-binary transformation in angle modulation needs to be further explored, in order to assess the efficiency of the approach.

6.4 Summary

DE is a simple yet powerful optimising algorithm. It uses a population-based search process to iteratively improve a population of solution encodings by mutating base vectors with scaled population-derived difference vectors. A critical aspect of the algorithm is that these differences adapt during the algorithm to the natural scaling of the problem. Readers interested in further details on the family of differential evolution algorithms are referred to [131].

In the next chapter we continue our development of evolutionary algorithms by introducing genetic programming. In contrast to GA, ES and DE, which are primarily used for optimising purposes, genetic programming

focusses on the evolution of high-quality ‘structures’. These structures are problem-specific and may be as diverse as a computer program, an electronic circuit, a mathematical model, or an engineering design.

7

Genetic Programming

Genetic programming (GP) was initially developed to allow the automatic creation of a computer program from a high-level statement of a problem's requirements, by means of an evolutionary process. In GP, a computer program to solve a defined task is evolved from an initial population of random computer programs. An iterative evolutionary process is employed by GP, where better (fitter) programs for the task at hand are allowed to 'reproduce' using recombination processes to recombine components of existing programs. The reproduction process is supplemented by incremental trial-and-error development, and both variety-generating mechanisms act to generate variants of existing good programs. Over time, the utility of the programs in the population improves as poorer solutions to the problem are replaced by better solutions. More generally, GP has been applied to evolve a wide range of 'structures' (and their associated parameters) including electronic circuits, mathematical models, engineering designs, etc.

7.1 Genetic Programming

Genetic programming (GP) [514] bears similarity with other evolutionary algorithms such as GA and ES in that it operates with a population of potential solutions which are iteratively improved using the mechanisms of selection, crossover/mutation, and replacement. However, there are two critical distinctions between GP and GA:

i. the form of representation used, and
ii. the more open-ended nature of the evolutionary process in GP.

Representation in GP

In the GA a clear distinction is drawn between the genotype and the phenotype, with the evolutionary search process being applied to the population

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_7

of genotypes. In GP, this distinction is lost, and the evolutionary search process, and its associated diversity-generation process, are applied directly to the phenotypes (the solutions).

In the form of GP popularised by John Koza [340, 341, 342, 343] these take the form of Lisp S-expressions. In Lisp, operators precede their arguments (known as a prefix notation), so the expression 2+1 is written as + 2 1. Similarly, $9*((2-1)+4)$ is written as $* + - 2\ 1\ 4\ 9$. More generally, Lisp contains a variety of standard programming operators which also adopt a prefix notation; for example, `(setf x 5)` assigns the value 5 to the variable `x`. S-expressions can be visually represented as a *syntax tree*, which is a graph structure where the nodes correspond to values or operators (Fig. 7.1).

Fig. 7.1. Example S-expression (left) and corresponding syntax tree (right). The syntax tree decodes into the expression $(\sin x) + (x * 3.14) + (y/x)$, where x and y are predefined constants

Taking another example, the following simple C program could be represented as the S-expression $(+(*\ 2 \sin y)(\cos z))$ or as a tree (Fig. 7.2).

```
#include<stdio.h>
#include<stdlib.h>

int main( int argc, char*argv ) {
    float x=0.0, y=0.0, z=0.0, retval;
    x=atof(argv[0]); y=atof(argv[1]); z=atof(argv[2]);
    retval = 2.0*sin(y) + cos(z);
    printf("The answer is: \%f\n", retval);
    return retval;
}
```

Open-Ended Nature of Evolution in GP

The second fundamental difference between algorithms such as GA, ES, DE and GP is that the evolutionary process in GP is more open-ended. Traditionally, with GA (and in DE) we adopt a fixed-length encoding, whereby we fix the number of genes (or bits) that will comprise an individual at the outset of a run. Hence, the maximum complexity of what can be evolved is fixed a priori.

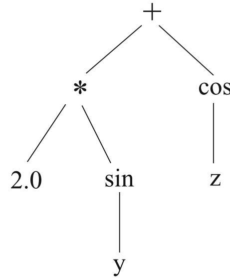


Fig. 7.2. Syntax tree representation of S-expression (+(* 2 sin y)(cos z))

By contrast, GP uses a variable-length representation in that the size of the structure of a solution may not be known. Hence, the number of elements used in the final solution, as well as their interconnections, must be open to evolution. This property allows GP to evolve a simple or a complex structure, depending on the nature of the problem being solved. The ability of GP to evolve structures of differing size will be illustrated in Sect. 7.1.4.

7.1.1 GP Algorithm

Algorithm 7.1 outlines the high-level pseudocode for the GP algorithm. Each of the elements of this is discussed below.

Algorithm 7.1: Genetic Programming Algorithm

Define terminal set, function set and fitness function;
Set parameters for GP run (population size, probabilities for mutation, crossover, selection/replacement strategy, etc.);
Initialise population of solutions;
Calculate fitness of each solution;

repeat
  Select parents;
  Create offspring;
  Calculate fitness of each solution;
  Update population;
until *terminating condition*;
Output optimal solution;

7.1.2 Function and Terminal Sets

When evolving programs or structures in GP, the user must first define the basic building blocks which GP can use. The programs are generated using elements from two sets, namely, the *function set* and the *terminal set*.

The terminal set contains items which have an arity of 0, meaning that they do not require any inputs, or — in programming terms — arguments, for their evaluation. Examples of terminals could include constants (say π) or a defined variable in a computer program. In looking at a GP tree, terminals correspond to the leaf nodes on the syntax tree.

In contrast, the function set contains items that have an arity greater than 0. Hence, function nodes in a GP tree require one or more inputs so that they can be evaluated. For example, the function sin requires a single real-valued input and hence has an arity of 1, whereas the function *AND* requires two Boolean inputs and therefore has an arity of 2.

The inputs to a GP function can in turn be other GP functions once these meet the closure requirement (see below) or they can be terminals. The ability of GP to nest functions within each other enables the creation of programs of varying sizes and shapes.

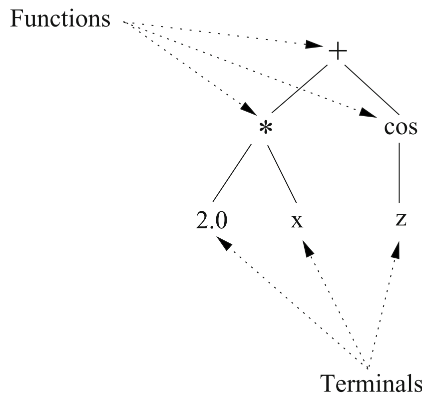


Fig. 7.3. Terminals correspond to the leaves of the tree. Functions can have differing arities and can take either other functions or terminals as inputs

The choice of items for inclusion in the function and terminal sets is user-determined and will be problem-specific. Ideally, the items should be chosen such that together they are *sufficient* for the problem of interest, that is, they are powerful enough to represent a complete solution to the problem at hand. There may be multiple definitions of the function and terminal sets which are sufficient (Fig. 7.4).

The function and terminal sets must also have the property of *closure*. That is, each function should be able to handle gracefully all values it might ever

Fig. 7.4. Example of two equivalent solution trees, generated by different terminal and function sets. The left-hand side tree is generated from $F = \{+\}$, $T = \{a\}$, and the right-hand side tree is generated from $F = \{*\}$, $T = \{a, 2\}$

receive as inputs. To ensure this, all terminals must be allowable inputs for all functions, and the output from any function must in turn be a permitted input to any other function. Closure is important as it ensures that the generated programs will be syntactically correct. For example, a viable function set F and terminal set T for a Boolean problem with three input variables is:

$$F = \{\texttt{and}, \texttt{not}\}$$
$$T = \{\texttt{input0}, \texttt{input1}, \texttt{input2}\}$$

The function and terminal sets hold the property of sufficiency, as it can be shown that all possible Boolean functions on the three input variables can be constructed from the Boolean `and` and Boolean `not` operators alone. Similarly, Boolean function sets { `or`, `not` }, { `nand` } or { `nor` } are possible alternatives for satisfying the property of sufficiency. The closure property is satisfied because the Boolean input values (`input0`, `input1`, `input2`) can all be passed as inputs to each of the functions in the function set F, and the output from each function in F is also a Boolean value that can be passed in turn as input to another function from this set.

Generating Numerical Values

As most real-world solutions will include numerical values, GP needs to be able to generate a variety of values for constants. This gives rise to a practical issue in that it is clearly impossible to include all real numbers in the terminal set. The standard approach to the provision of constants in GP is through *ephemeral random constants* (ERCs). A number of ERCs are generated within a prespecified range at the outset of a run of the GP algorithm. When a node in the growing program is determined to have become a constant, a random value from the set of ERCs is generated. After the initial generation, new constants are created through the recombination of existing ERCs through arithmetic expressions. Even a fairly compact set of functions and ERCs can be used to generate whatever real number is required. Figure 7.5 illustrates how a model parameter of 4.909 (for example) could be generated using the functions sin (calculated in radians here), $+$, $*$ and three real values from an ERC set. Other methods of constant generation for GP also exist.

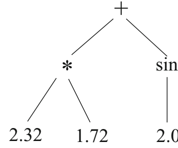


Fig. 7.5. Illustration of the generation of the value 4.909 from a compact set of terminals and ERCs

Incorporating More Complex Structures

In addition to the input variables, constants, and primitive operators specified in the function and terminal sets, it is possible to incorporate standard programming constructs such as conditional statements, parameterised functions, iterations, loops, storage/memory, and recursion into a GP individual. An example GP program containing a conditional expression in both a prefix Lisp-like S-expression and a syntax tree can be seen in Fig. 7.6. Note that the conditional expression, denoted by the `if` function at the root of the subtree, is comprised of three components. The first, left-most component is the condition itself, which can be comprised of a complex logical expression which will return either one of the Boolean `true` or `false` values. In this example, depending on the outcome of the logical expression returning either true or false, the second (the value of x) or third component (value of y) of the conditional expression will be returned, respectively, and subsequently added to the result of `sin(x)`.

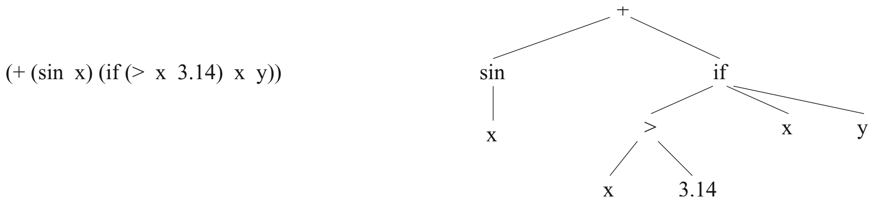


Fig. 7.6. Example GP individual containing a conditional S-expression (left) and its corresponding syntax tree (right)

7.1.3 Initialisation Strategy

Once the function and terminal sets are specified, individuals in the population must be generated using an initialisation strategy. There are two main methods (*Grow* and *Full*). GP implementations typically use both in order to ensure diversity of both structure and content in the initial population. This provides the evolutionary process with initial diversity with regard to tree

(solution) complexity and content in order to help facilitate the efficient uncovering of good solutions. The most common form of initialisation is known as *ramped-half-and-half* initialisation, where half of the initial population of solutions is created using the grow method, and half is created using the full method (Fig. 7.7).[1]

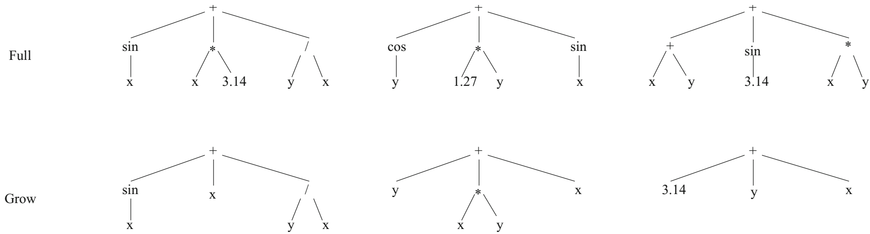


Fig. 7.7. Sample trees from the full method (here depth = 3) and the grow method

Full Method

In the Full method, trees are grown by selecting only functional primitives until a prespecified depth (maxdepth $-$ 1) is reached on all branches, at which point only terminals are selected to complete the tree.

Grow Method

In the Grow method, trees are grown randomly with a terminal or functional primitive being selected randomly at each step, until a branch reaches (maxdepth $-$ 1). At this point, only terminals are selected in order to ensure that the maxdepth limit is not breached.

Comparing the two methods, the Full method produces trees where all branches extend to the predefined maximum depth, whereas the Grow method produces trees of irregular shapes and sizes. Hence, combining both approaches will produce an initial population of trees of varying sizes and internal structure. The structural diversity of the initial population is usually further enhanced during the ramped-half-and-half process by varying the maxdepth limit during initialisation from 2 to $n-1$.

[1] In explaining the process of initialisation (and the operation of evolutionary search operations such as mutation and crossover), we illustrate these processes visually on the relevant syntax trees, although they actually take place on the underlying S-expressions.

7.1.4 Diversity-Generation in GP

Just as for GA, there are many ways that the selection, diversity-generating and replacement operators can be defined in GP. Typically crossover (two parents selected, producing two children) is applied about 90% of time, mutation (one parent selected, one child produced) around 0-1% of time, cloning (one parent selected, one child produced) around 8% of the time. Replacement strategies can vary from generational replacement to steady-state, where only a few children (or in the limit, a single child) are produced in each generation.

Crossover

In the standard crossover operator in GP (subtree crossover), two individuals are selected from the population and copied to become two parents. A subtree in each parent is identified as the crossover site. The subtree in the first parent is replaced with the subtree from the second parent, and vice versa, with the result that two children are created.

Consider, for example, two GP individuals $(+\ x\ y)$ and $(+\ z\ (*\ 1.0\ z))$, and assume a crossover point of y is selected in parent 1 and $*$ in parent 2. Figure 7.8 provides an illustration of one of the two children created.

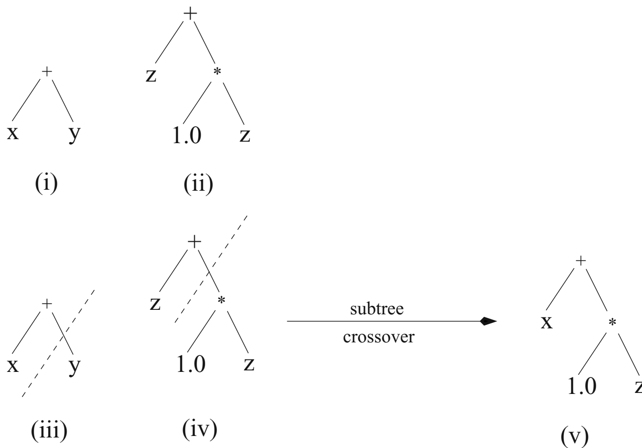


Fig. 7.8. Example syntax trees, (i) and (ii), are copied to become parent 1 and parent 2 respectively. Subtrees are selected as crossover sites on each parent (identified by hashed lines), with the subtree from parent 1 (iii) being replaced with the subtree from parent 2 (iv) to produce a child (v). In this diagram, we only show one child

One issue that arises in GP is that the application of the crossover operator on two identical (or very similar) parents in GP does not usually produce

two identical children (unlike the canonical GA). Figure 7.9 provides an illustration of this, where a randomly chosen cut-point in the two identical parents produces a child which is different from both parents. As can be seen in this simple example, crossover in GP is capable of generating diversity in solution form, even when parents are similar.

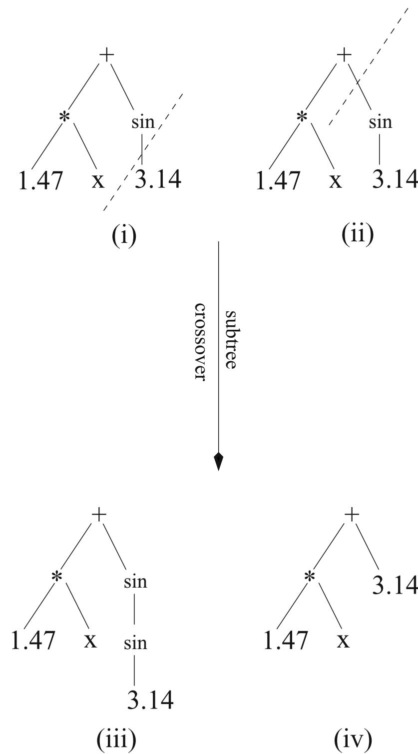


Fig. 7.9. Two identical parent trees (top) producing a different child tree (bottom) as a result of a crossover operation

Another aspect of crossover in GP is that if the terminal set is much larger than the function set (which is common), solution trees will tend to contain a high proportion of leaf (or terminal) nodes. Hence, if the crossover point is chosen randomly in both parents, it can produce 'leaf-node swapping', with children being very similar to their parents. One way to boost the level of diversity generated by the crossover process is to bias the selection of the crossover points so that crossovers occur at function rather than terminal nodes. A common scheme is to select functions 90% of the time, choosing terminal nodes 10% of the time.

In order to enable unrestricted crossovers between parents, the function and terminal set must have the property of closure. Alternatively, if functions can return different data types, then the crossover operator must be restricted such that subtrees can only be swapped when they output the same data type [417]. As will be seen in Chap. 17, grammars can be employed to overcome the problem of closure when functions can return different data types.

Mutation

Historically, mutation was not afforded a significant role in GP, largely because crossover is capable of generating significant diversity on its own, and also because pioneers of the field, such as Koza, wanted to distance themselves from notions of random search that mutation invokes. Koza's early research [340, 341] demonstrates that mutation is not always necessary to solve problems using GP, although in recent years it is more common for researchers to adopt mutation.

Two types of mutation are found in GP systems. In the first type (*subtree mutation*), a random nonterminal is selected, and is deleted along with its subtree. A new subtree is then grown at this point. As an example, consider the S-expression (+(exp z) ($*$ 2 x)). Suppose the nonterminal selected for mutation is $*$, and the new randomly created subtree is (+(sin x) (2)). The resulting expression after the tree mutation is (+(exp z) (+(sin x)(2))) (Fig. 7.10).

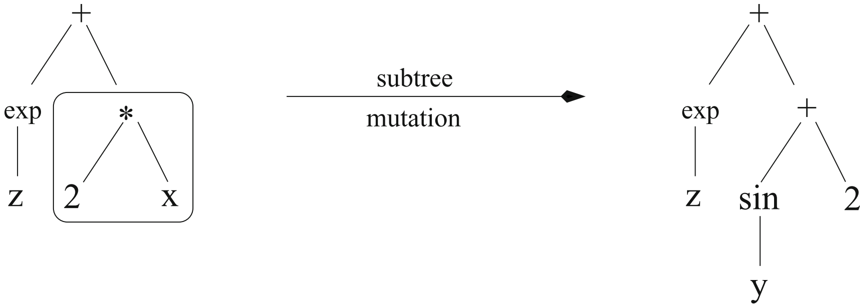


Fig. 7.10. Subtree mutation. A random subtree is selected and deleted, and a new subtree is grown at that cut point

In the second form of mutation operator (point mutation), a single function is replaced by another function of the same arity, or alternatively, a single terminal is replaced by another terminal. Take the S-expression from the last example (+(exp z) ($*$ 2 x)), and assume a point mutation where '$*$' is replaced by another function of the same arity, '+'. The resulting expression after the point mutation is (+(exp z) (+ 2 x)) (Fig. 7.11). As noted for crossover,

mutation can produce a child with a tree depth which is greater than that of its parent.

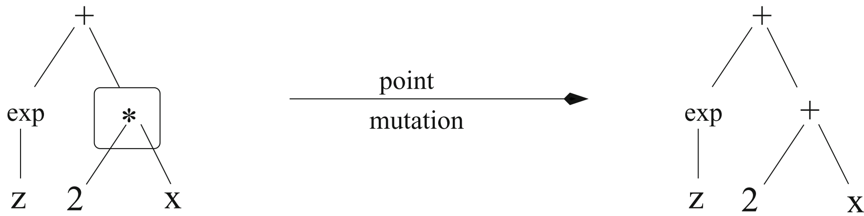


Fig. 7.11. Point mutation. A random terminal/function (here a function) is selected and deleted, and a new terminal/function is inserted at that point

7.2 Bloat in GP

One issue that can arise in GP is that the chromosomes representing individuals tend to grow in size during a GP run without a corresponding improvement in fitness. This can lead to very complex tree structures or *bloat*, which can contain redundant code fragments, and which can be difficult to simplify in order for us to understand the structure of the evolved solution.

Ways of counteracting bloat in GP encompass the inclusion of a penalty term in the fitness function which discourages large trees, and the limiting of the maximum tree depth by forbidding the application of mutation or crossover where the result would produce a tree exceeding these limits. A large number of studies exist in the GP literature on the topic of bloat, and to this date this remains an open issue for the field.

7.3 More Complex GP Architectures

In the illustrative GP individuals we have seen so far in this chapter, the programs are comprised of a single, result-producing function comprised of the whole tree. Most programming languages contain several additional constructs, including functions, memory, looping and recursion. These can be easily incorporated into GP, and each is discussed below.

7.3.1 Functions

In programming, and more generally in problem solving, it is useful to decompose the task at hand into a series of smaller and simpler subtasks, which can be reused to solve the problem as a whole. The ability to reuse parts of

solutions can be incorporated into GP individuals using constructs such as functions, iterations, loops and recursion. To this end it is necessary to introduce a more complex program architecture comprised of multiple branches, including the result-producing branch (RPB). The other branches define, for example, the functions and iterations that the RPB can utilise in the generation of the resulting program output.

The typical method to include functions or subroutines in a GP individual is through *automatically defined functions* (ADFs). ADFs are parameterised functions that can be called in a hierarchical manner, either by the RPB or by another ADF. When ADFs are included in the function and terminal specification of a GP system, the evolutionary process is free to decide whether the main program uses ADFs at all, and to decide what included ADFs actually do. The practical advantage of ADFs is that they allow for the easy (multiple) reuse of good code/solution modules. ADFs also permit the adaptation of the function set in GP 'on-the-fly' during the solution search process.

If ADFs are to be used, they must be defined in a function-defining branch comprised of the function's name, the list of its parameters, and the body of the function. Figure 7.12 outlines the architecture of a GP individual comprised of a single ADF called ADF0 that receives three parameters and sums the parameter values that are passed to it. An ADF is defined using the `DEFUN` function, and `VALUES` is a function that returns whatever value its subtree evaluates to. `DEFUN` simply returns the name of the function to its parent function (`PROGN`). The `PROGN` function evaluates all of its subtrees in succession, returning the result of evaluating the last (the right-most) subtree, which is referred to as the RPB. The RPB can use any of the previously defined ADFs when evaluating the result of the program.

```
(PROGN (DEFUN ADF0
              (LIST ARG0 ARG1 ARG2)
              (VALUES + ARG0 ARG1 ARG2)  )

       (VALUES
              (*  (ADF0  x  y  z) x) )
)
```

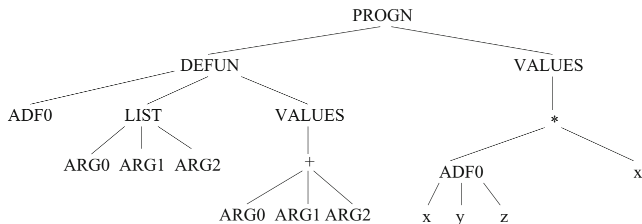


Fig. 7.12. The architecture of an automatically defined function represented in terms of an S-expression (top) and corresponding syntax tree (bottom). The syntax tree decodes to $(x + y + z) * x$

An ADF may nonrecursively call any previously defined ADF from within its own body, thus allowing hierarchical ADF evaluation. A succession of ADFs can thus precede the main RPB in an individual, as outlined in Fig. 7.13.

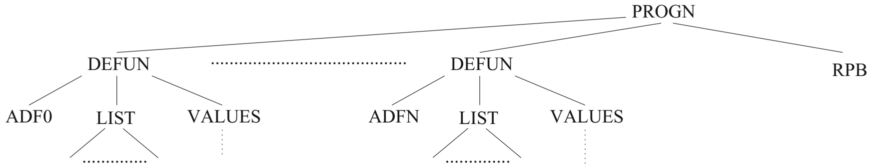


Fig. 7.13. The architecture of a GP individual including a hierarchy of automatically defined functions (ADF0 to ADFN) and the result-producing branch (RPB) represented as a syntax tree

To ensure that architecturally correct (i.e., only permit nonrecursive and hierarchical ADF calls to previously defined ADFs) individuals are generated in the initial population, separate function and terminal sets must be specified for the ADFs and RPBs. Taking an example we could define two ADFs (ADF0 and ADF1), with the following function sets for the RPB, ADF0 and ADF1, respectively.

$$F_{\text{RPB}} = \{\text{if}, *, +, -, /, \text{ADF0}, \text{ADF1}\}$$
$$F_{\text{ADF0}} = \{\text{if}, *, +, -, /\}$$
$$F_{\text{ADF1}} = \{\text{if}, *, +, -, /, \text{ADF0}\}$$

The corresponding terminal sets for a problem with three variables might take the following form where ADF0 is a three-argument function and ADF1 has two arguments.

$$T_{\text{RPB}} = \{x, y, z\}$$
$$T_{\text{ADF0}} = \{\text{ARG0}, \text{ARG1}, \text{ARG2}\}$$
$$T_{\text{ADF1}} = \{\text{ARG0}, \text{ARG1}\}$$

Why Use ADFs?

A particular advantage of using ADFs is that they allow for the easy (multiple) reuse of an already discovered good code module, i.e., GP does not have to 'discover' the same subtree multiple times. For example, suppose that GP is being applied to a symbolic regression problem where one of the terms of the underlying data-generating model is x^{40}, and the function and terminal set are as follows:

$$F = \{+, -, *\}$$
$$T = \{x, \text{ERCs}\}$$

The only way that this function and terminal set can uncover the term x^{40} is by creating a large (deep) tree comprising x's and $*$'s. In contrast, if ADFs are used, the creation of (say) a module which evaluates to x^{10} can be reused efficiently to create a syntax tree which produces x^{40}.

Another advantage of ADFs, particularly in a dynamic environment, is that they can provide an implicit 'memory' structure, whereby good solution fragments captured in an ADF can be maintained and reused as the environment changes.

7.3.2 ADF Mutation and Crossover

A critical aspect of ADFs is that, just like a simple syntax tree, the content of an ADF can be evolved during the GP run. In typical implementations of ADFs, each individual in the population evolves its *own* ADF(s), i.e., they are not shared across multiple members of the population, although it is certainly possible to create a GP system which uses a library of shared ADFs — akin to the concept of a gene library in the immune system (Sect. 16.1.3).

In implementations of GP which contain ADFs, the regular mutation and crossover operations need to be amended slightly in order to ensure that they operate correctly for ADFs.

Mutation

- i. Select parent probabilistically based on fitness.
- ii. Pick a mutation point from either the RPB or an ADF.
- iii. Delete the subtree rooted at the picked point.
- iv. Grow a new subtree at this point composed of the allowable functions / terminals appropriate for the picked point.

Crossover

- i. Select two parents probabilistically based on fitness.
- ii. Pick a crossover point from either RPB or an ADF of the first parent.
- iii. The choice of crossover point in the second parent is restricted to either its RPB or its ADF (depending on the random choice in the first parent).
- iv. Swap the subtrees.

7.3.3 Memory

Memory is implemented in GP in a manner similar to ADFs, using automatically defined storage (ADS), with the addition of two branches to an individual that allow reading and writing to a memory location. Effectively, the additions of a storage writing branch (SWB) and a storage reading branch (SRB) are equivalent to adding a new element to an individual's function set which

allows a newly added memory location to be written to as well as read from (Fig. 7.14). The type (e.g., named memory, stack, queue, two-dimensional array, or list) and dimensionality (number of arguments to address it) are determined (usually randomly) upon creation of the ADS. In Fig. 7.14 a named memory location (ADS0) with zero dimensionality (i.e., the SRB function requires no arguments to retrieve the data stored in ADS0) is created.

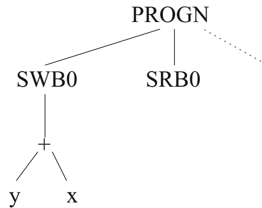


Fig. 7.14. Fragment of an example GP individual containing automatically defined storage (ADS0)

7.3.4 Looping

Iterations and more generally loops can be incorporated into a GP individual using automatically defined iterations (ADIs) and automatically defined loops (ADLs). Similar to ADFs, ADIs and ADLs are defined using a multiple branch architecture, where their branches occur before the RPB. It is common for a simplified form of ADIs and ADLs to be adopted where the defined iterations or loops are invoked only once and prior to the evaluation of the RPB. The result of evaluating the ADI/ADL branch is made available to the RPB indirectly through storage in a named memory location. There may be multiple ADIs and ADLs within an individual, and they can refer to previously defined ADFs.

In the case of ADIs, they are implemented to iterate once over a predefined data structure such as an array, vector or sequence. As such, the size of the data structure to iterate over is known and the possibility of infinite loops is eliminated. A sample ADI in Fig. 7.15 has no arguments, and returns the result of its evaluation indirectly to the result-producing branch by writing to the named memory location M0. The number of elements contained in the data structure being iterated over (V) is built into the ADI function. ADI0 is evaluated as a result of its invocation in the result-producing branch, with the RPB using the result of evaluating ADI0 by accessing M0.

ADLs implement a general form of iteration comprised of loop initialisation (LIB), loop condition (LCB), loop body (LBB), and loop update branches (LUB). Figure 7.16 outlines an example ADL where the LIB sets the memory location M1 to 0; the LCB determines how many iterations over the data

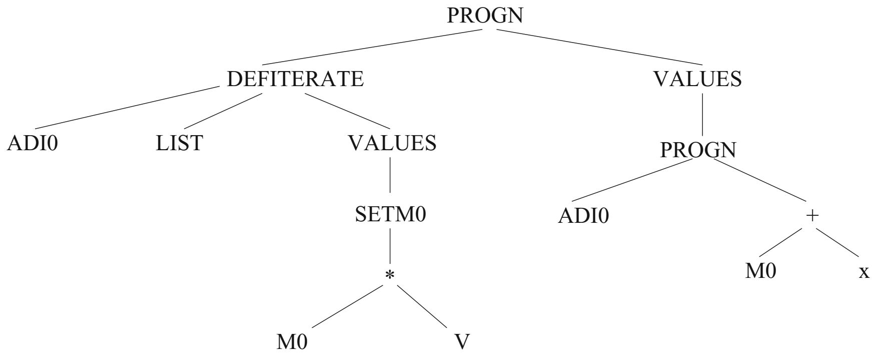


Fig. 7.15. Example GP individual containing an automatically defined iteration (ADI). The result of evaluating ADI0 (multiplying all the values contained in the vector V) is available to the result-producing branch through the named variable memory location M0, which the body of ADI0 wrote to using SETM0

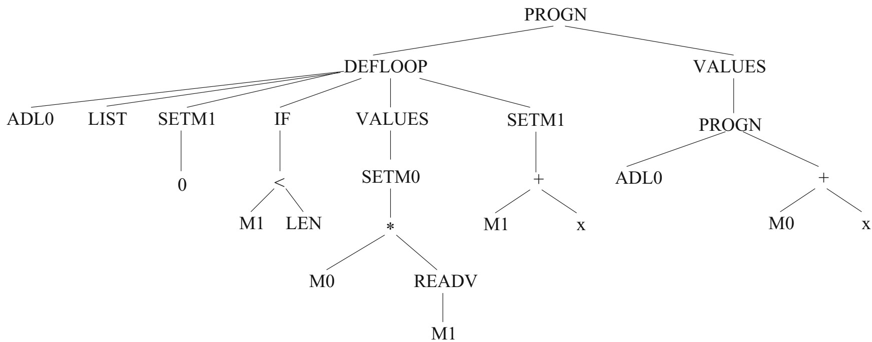


Fig. 7.16. Example GP individual containing an automatically defined loop (ADL). As in the ADI example in Fig. 7.15 the result of evaluating ADL0 is available to the result-producing branch through the named memory location M0, which the body of ADL0 wrote to using SETM0

structure should be conducted. After the LBB is evaluated on each iteration, the LUB is evaluated, which increments the value of M1 (in this example this ensures that an infinite loop will not arise as the LCB is checking the value of M1 to determine when to terminate the loop). In the LBB, `(READV M1)` reads the `M1`$^{\text{th}}$ value of the data structure (V) being looped over. The result of evaluating ADL0 is available to the RPB through the value stored in the named memory location M0. To prevent infinite loops from occuring, generally a timeout strategy is adopted whereby the evaluation of an individual is halted after a predetermined time limit (or a maximum number of iterations) has been reached.

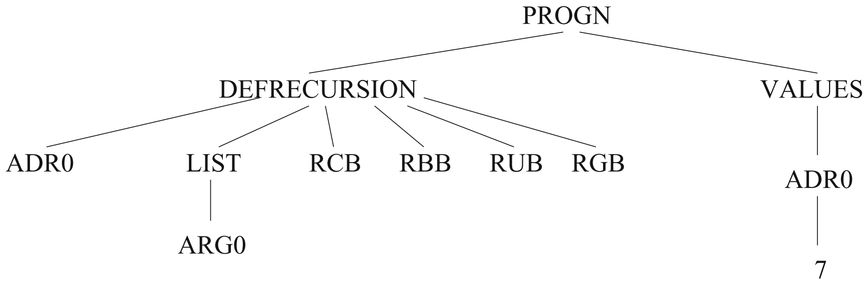


Fig. 7.17. The architecture of automatically defined recursion (ADR)

7.3.5 Recursion

Recursion is made possible in GP through an automatically defined recursion (ADR) architecture (Fig. 7.17). There are four components to ADRs, namely recursion condition (RCB), recursion body (RBB), recursion update (RUB), and recursion ground (RGB) branches. To prevent infinite recursion a limit is placed on the number (or depth) of the recursive calls that are allowed within an individual. When timeout limits in the case of ADIs and ADLs, or depth limits in the case of ADRs, are violated the individual can be selected against by punishment with a large fitness penalty. Figure 7.17 outlines ADR in an individual where the `DEFRECURSION` function is used to define the recursive function `ADR0` that takes a single parameter (`ARG0`). The RCB determines if recursion is continued by returning a positive value, or by returning a negative value recursion is halted. In the event that recursion is halted, the fourth (right-most) RGB branch is evaluated. The RBB branch normally contains a recursive call to the function itself, and when the evaluation of the RBB finishes the RUB branch is evaluated.

During initialisation of a GP population either the architecture of the program is prespecified, that is the presence (or absence) of ADFs, ADSs, ADIs/ADLs and ADRs and their quantities are predetermined, or their incorporation (deletion) can be left open to evolutionary search. In order to allow the search process to add, delete or modify these constructs, architecture-altering operations were introduced specifically for each architecture type [342]. For example, in the case of ADFs, it is possible to create, duplicate or delete an ADF, and even to create, duplicate or delete arguments to an existing ADF. Special attention must also be paid to the crossover operator, which must be implemented to ensure that legal architectures are generated as a consequence of a crossover event.

7.4 GP Variants

A great deal of literature exists on GP and the evolution of programs in general. The interested reader is referred to the following as a good starting point for further investigations: [31, 340, 341, 342, 343, 354, 355, 359]. In this section we outline some of the common variants of GP.

7.4.1 Linear and Graph GP

Earlier in the chapter we introduced a popular form of tree-based GP [340]. However, prior to its introduction a number of alternative representations had been adopted (for example see [119, 203, 204]) and since then a large variety of representations has been examined including graphs, linear structures, grammars and even hybridisations of these. Notable examples include linear GP [30, 454], PADO (Parallel Algorithm Discovery and Orchestration) [618], graph and linear-graph GP [312, 313], Cartesian GP [413], and grammar-based GP systems (e.g., [231, 276, 472, 656, 662]). While the issue of choice of representation is not unique to GP, indeed more broadly it transcends machine learning as a whole, the question as to what makes a *good* representation for EC is an open one and some attempts are now being initiated to formalise this research [540]. In recent years there has been a great deal of research on schema theories for genetic programming, and it is being recognised that these theories demonstrate a commonality between the various representations adopted in EC, with GP schema theories being considered supersets of GA schema theory [359].

7.4.2 Strongly Typed GP

A variation of basic GP called strongly typed GP (STGP) can be used to remove the limitation of closure, that all variables, constants, arguments of functions and values returned by functions must be of the same data type. In STGP, variables, arguments and so on can be of any data type, but STGP requires that each function be strongly typed, in other words that it specify exactly the data types of its arguments and of its returned values. The practical effect of using STGP is that the initialisation strategy of basic GP is altered so that the element chosen at each node of the growing tree must return the data type expected by its parent node. Similarly, the mutation and crossover operators are restricted. For example, if the mutation operator is applied, the new subtree must return the same data type as the tree it replaced. Interested readers are referred to [417] for further details on this variant of GP.

7.4.3 Grammar-Based GP

When one has to deal with the evolution of executable entities, in particular computer code, it is impossible to escape the issue of syntax. As we observed

in strongly typed GP, this approach adopts a tree encoding which explicitly handles data types, as it is necessary to 'manage' the syntax appropriately in order to ensure the property of closure. Grammars provide a formalism through which syntax can be expressed and ultimately controlled. Grammars can be explicitly employed to control syntax when they are used in a generative sense to construct sentences in the language defined by the grammar. Grammars have been used in GP for many years, and even standard tree-based GP implicitly adopts grammars via the specification of the function and terminal sets. Grammar-based GP is an important area of research within the GP community, and it also represents another aspect of the natural world which computer scientists have taken inspiration from to develop problem solving algorithms [403]. Grammar-based GP is discussed in more detail in Chap. 18.

7.5 Semantics and GP

The concept of syntax has played a pivotal role in the field of GP, and lies at the heart of how programs are encoded as individuals in an evolving population. Aside from the use of a fitness measure, the notion of semantics of programs has largely been overlooked by GP until recent years, and now a growing body of research is emerging which focuses on how semantics can be exploited by GP algorithms to improve their performance (e.g., [43, 348, 350]).

At least part of the motivation for the adoption of semantics in GP is to address the credit assignment issue. That is, until recently, in the majority of GP algorithms, a single measure (fitness) has been used to assign credit to the entire GP individual. A GP individual is a complex organism which might be comprised of many branches, and subtrees within each branch, with functional dependencies existing between subtrees in different branches (e.g., with ADFs). Yet few attempts have been made to ascertain how components of an individual contribute to overall fitness. It may be possible to use semantics to measure the contribution of individual subtrees and branches to the overall fitness measure, and direct search towards these fruitful substructures.

Much research in semantics of GP has been directed towards 'semantic aware' search operators (e.g., [42, 44, 349, 440, 419, 441]), whereby operations such as mutation and crossover are applied having taken account of the semantics of the underlying GP 'tree'. This can improve the generalisation properties of the evolving solutions, and improve the efficiency of the GP run.

7.6 Summary

GP is a powerful natural computing algorithm. It is capable of model induction, that is, uncovering and optimising the structure, parameters and contents of the evolving candidate solutions. It is impressively general in its

application, and perhaps this is not surprising given the underlying representation is that of computer programs or functions. Therefore, if the solution to a problem of interest can be cast in the form of a program or function, then GP can potentially be applied in that domain.

A particularly impressive aspect of GP is its success at producing human-competitive performance [344]. The literature presents an ever-increasing list of examples where GP has been used to produce solutions to problems which have either confounded human experts, or produced superior solutions to prior state of the art. In some instances the evolved solutions have been sufficiently novel to be patentable. Notwithstanding the advances in GP over the past couple of decades, there are still many open research areas as outlined in [484]. Part V of this book continues the discussion on GP and its developmental and grammatical variants.

Part II

Social Computing

8

Particle Swarm Algorithms

In this part of the book we discuss algorithms which are metaphorically inspired by a variety of social behaviours. The essence of these behaviours is that individuals can learn from both their own experience and from the experience of others. Hence a group can solve complex tasks which are beyond the capability of any of the individuals in the group. Crucially, this does not occur merely because the task is divided up amongst multiple individuals; rather it occurs because they can communicate with each other and thereby create a shared understanding of the problem. Even populations of individuals with limited information-processing abilities may be able to solve difficult problems, if the individuals communicate and cooperate.

One model of social learning which has attracted particular interest in computer science in recent years is drawn from a swarm metaphor. Three popular variants of swarm models exist, those inspired by:

i. the flocking behaviour of birds or the sociological behaviour of a group of people;
ii. food foraging behaviours; and
iii. behaviours of social insects such as ant colonies.

The essence of these systems is that they exhibit flexibility, robustness and self-organisation [68]. Although the systems can exhibit remarkable coordination of activities between individuals, this coordination does not stem from a centre of control or a *directed* intelligence; rather it is self-organising and emergent.

This chapter introduces the *particle swarm optimisation* (PSO) algorithm. The next two chapters (Chaps. 9 and 10) introduce a range of algorithms which are inspired by the behaviours of two social insects, namely ants and honey bees. Next follows a chapter (Chap. 11) which introduces a range of algorithms inspired by bacterial behaviours, which metaphorically embed a communication mechanism. The final chapter in this part of the book (Chap. 12) introduces a range of emerging algorithms which are inspired by a variety of social communication mechanisms in insects, mammals, and fish.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_8

8.1 Social Search

Suppose we are interested in using a populational search algorithm to find the global optimum in a search space but the individual members of the population do not communicate with each other and do not learn from their past experience. If the 'moves' of the individuals are selected randomly in each iteration of the search algorithm, a random search process results (Fig. 8.1). An obvious drawback of this algorithm is that no use is made of the information gained by any member of the population in order to direct future search efforts. In contrast, the particle swarm algorithm makes use of this information by embedding both a memory of previous search outcomes, and social communication between the particles with regard to the success of their search efforts, in order to bias the future search efforts of all members of the swarm.

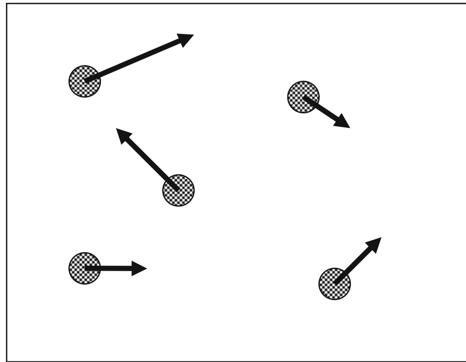


Fig. 8.1. Illustration of a swarm of five individuals (searchers) which have been initialised with random locations and random velocities in the search space

8.2 Particle Swarm Optimisation Algorithm

The particle swarm optimisation algorithm (PSO) was introduced by Kennedy and Eberhart [326] and is described in detail in [177] and [328]. PSO has been applied for two main purposes, as a real-valued optimisation algorithm and as a model of social interaction. In this chapter we concentrate on the application of PSO as an optimisation algorithm.

In PSO a swarm of *particles*, each of which encodes a solution to a problem of interest, move (*fly*) around an n-dimensional search space in an attempt to uncover ever-better solutions to the problem of interest. Each of the particles has two associated properties, a current position and a velocity. Each particle

i also has a memory of the best location in the search space that it has found so far (p_i^{best}), and knows the best location found to date by all the particles in the population (g^{best}). At each iteration of the algorithm, particles are displaced from their current position by applying a velocity (or 'gradient') vector to them. The magnitude and direction of their velocity is influenced by their velocity in the previous iteration of the algorithm, thereby simulating momentum, and the location of a particle relative to the location of its p^{best} and the g^{best}. Therefore, the size and direction of each particle's move is a function of its own history (experience) and the social influence of its peer group. Pseudocode for the canonical version of PSO is provided in Algorithm 8.1.

Algorithm 8.1: Canonical Particle Swarm Algorithm

```
for each particle i in the population do
    Initialise its location by randomly selecting values;
    Initialise its velocity vector to small random values close to zero;
    Calculate its fitness value;
    Set initial p_i^best to the particle's current location;
end
Determine the location of g^best;

repeat
    for each particle i in turn do
        Calculate its velocity using (8.1);
        Update its position using (8.2);
        Measure fitness of new location;
        if fitness of new location is greater than that of p_i^best then
            Revise the location of p_i^best;
        end
    end
    Determine the location of the particle with the highest fitness;
    if fitness of this location is greater than that of g^best then
        Revise the location of g^best;
    end
until terminating condition;
```

Synchronous vs. Asynchronous Updates

In the canonical PSO algorithm the update of the position of g^{best} (if required) is performed at the end of each iteration of the algorithm. An alternative approach is to update g^{best} immediately when a particle finds a position with

higher fitness (an asynchronous update). One advantage of asynchronous updating is that updates in the position of g^{best} are immediately available for use by other particles.

8.2.1 Velocity Update

Each particle i in the swarm has an associated current position in search space x_i, a current velocity v_i, and a personal best position in search space y_i. During each iteration of the algorithm, the velocity and location of each particle are updated using (8.1) and (8.2). Assuming that a function f is to be maximised, that the swarm consists of n particles, and that r_{1d}, r_{2d} are drawn (separately for each dimension in particle i's velocity vector) from a uniform distribution in the range (0,1), the velocity update for each dimension d is:

$$v_{id}(t+1) = v_{id}(t) + c_1 r_{1d}(p_{id}^{\text{best}}(t) - x_{id}(t)) + c_2 r_{2d}(g_d^{\text{best}}(t) - x_{id}(t)) \quad (8.1)$$

where $g^{\text{best}}(t)$ is the location of the global best solution found by all the particles up to iteration t.

Examining the velocity update equation, it is comprised of three parts. The term $v_{id}(t)$ represents momentum, as a particle's velocity on each dimension at time $t+1$ is partly a function of its velocity on that dimension at time t. The term $(p_{id}^{\text{best}}(t) - x_{id}(t))$ represents individual learning by particle i, in that it encourages the particle to return to the location of its p^{best}, the best solution it has found to date. The term $(g_d^{\text{best}}(t) - x_{id}(t))$ represents social learning as the particle is also encouraged to move to the location of the best solution found by any member of the swarm thus far. Hence, the velocity update is a blend of momentum, a tendency to revert to p^{best} and a tendency to move towards g^{best}.

At the start of the algorithm, the p^{best} for each particle is set at the initial location of that particle, and g^{best} is set to the location of the best p^{best}. In each iteration of the algorithm, particles are stochastically accelerated towards their previous best position and towards the global best position, thereby forcing the particles to search around the most promising regions found so far in the solution space.

Once the velocity update for particle i is determined, its position is updated (8.2). The location of p_i^{best} for particle i is also updated if necessary using (8.3–8.4).

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1) \quad (8.2)$$

$$p_i^{\text{best}}(t+1) = p_i^{\text{best}}(t), \text{ if } f(x_i(t)) \le f(p_i^{\text{best}}(t)) \quad (8.3)$$

$$p_i^{\text{best}}(t+1) = x_i(t), \text{ if } f(x_i(t)) > f(p_i^{\text{best}}(t)) \quad (8.4)$$

After all particles have been updated a check is made to determine whether g^{best} needs to be updated: choose

$$\hat{p}^{\text{best}}(t+1) \in \{p_0^{\text{best}}(t+1), \ldots, p_n^{\text{best}}(t+1)\} \quad (8.5)$$

such that

$$f(\hat{p}^{\text{best}}(t+1)) = \max\{f(p_0^{\text{best}}(t+1)), \dots, f(p_n^{\text{best}}(t+1))\}. \quad (8.6)$$

Then

$$g^{\text{best}}(t+1) = g^{\text{best}}(t), \text{ if } f(g^{\text{best}}(t)) > f(\hat{p}^{\text{best}}(t+1)), \quad (8.7)$$

$$g^{\text{best}}(t+1) = \hat{p}^{\text{best}}(t+1), \text{ if } f(g^{\text{best}}(t)) \leq f(\hat{p}^{\text{best}}(t+1)). \quad (8.8)$$

Figure 8.2 provides visual intuition on the workings of the algorithm. A particle i is located at position $x_i(t)$ at time t and has a velocity of $v_i(t)$. The position of the particle at time $t+1$ is determined by $x_i(t) + v_i(t+1)$, with $v_i(t+1)$ being obtained by a stochastic blending of $v_i(t)$, an acceleration towards g^{best} and an acceleration towards p_i^{best}.

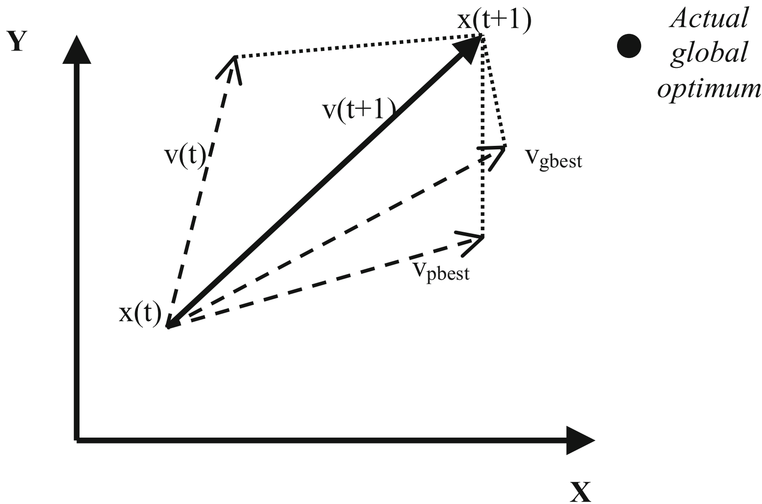


Fig. 8.2. Diagram of the particle position update process

Decomposing the Velocity Update Equation

The core of the PSO algorithm is the way that individual particles move from one iteration of the algorithm to the next. This is governed by the velocity update equation. An interesting aspect of this equation is that none of the component terms of the update equation produces a particularly interesting

search process when considered in isolation. However, when they are combined together a powerful search algorithm results.

For example, if the velocity update equation is reduced to $v_{id}(t+1) = v_{id}(t)$ the search trajectory of individual particles will continue in the same direction that its velocity vector was randomly initialised to in the first iteration of the algorithm, uninfluenced by particle or swarm learning. This will not produce a sensible search process.

If the velocity update equation is reduced to $v_{id}(t+1) = c_1 r_{1d}(p_d^{\text{best}}(t) - x_{id}(t))$, then each particle moves towards the location of its own p^{best}. Again, this produces a very limited search process.

If, instead, the velocity update equation is reduced to $v_{id}(t+1) = c_2 r_{2d}(g_d^{\text{best}}(t) - x_{id}(t))$, each particle will move stochastically in the direction of g^{best}. While this will produce a degree of search activity in the swarm, it is likely to produce a very rapid convergence of the swarm to a single location, before an adequate exploration of the search space has taken place.

When the three terms are combined a much more complex search dynamic is produced. The weight coefficients c_1 and c_2 (and an implicit weight coefficient of 1 before the prior-period velocity) control the relative impacts of prior-period velocity and the p^{best} and g^{best} locations on the search trajectory of particles. Low values for c_1 and c_2 encourage particles to explore far away from already uncovered good points as there is less emphasis on past learning. High values of these parameters encourage more intensive search of regions close to these points. If the prior-period velocity term is excluded from the velocity update equation, the remaining two terms act to move each particle i towards a line connecting p_i^{best} and g^{best} (Fig. 8.3), the exact velocity update depending on the values for the random coefficients r_{1d} and r_{2d}, and the weight coefficients c_1 and c_2. The addition of the prior period velocity term causes the particle to oscillate back and forth beyond this line.

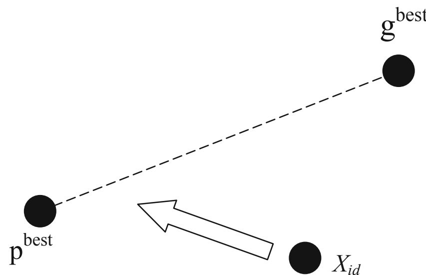


Fig. 8.3. If the prior period velocity term is excluded from the velocity update equation, the p^{best} and g^{best} terms act to move particle i towards a line connecting p_i^{best} and g^{best}

8.2.2 Velocity Control

If the canonical PSO algorithm as described above is applied, particles which are a long distance from p^{best} and g^{best} will tend to produce very large velocity updates and therefore there will be large position oscillations from one iteration of the algorithm to the next. While this facilitates the exploration of wide areas of the search space, large oscillations in particle position will make it difficult for the swarm to intensively search already-discovered high-quality regions of the search space. In order to overcome this problem, a number of methods can be applied to constrain the magnitude of the velocity vector.

Velocity Clamping

In order to limit the velocity that a particle i can attain, on each dimension d the component v_{id} of its velocity vector can be restricted, or *clamped*, to a range $[-v_d^{\max}, v_d^{\max}]$. The value chosen for $v_d^{\max}$ can have an important effect on the efficiency of the algorithm. Small values can result in insufficient exploration of the search space, while large values can result in particles moving past good solutions. The value of $v_d^{\max}$ is typically set in the range $k \cdot (x_d^{\max} - x_d^{\min})$, where $0 < k < 1$ and $x_d^{\max}$ and $x_d^{\min}$ are, respectively, the maximum and minimum allowable values on dimension d.

Related to the issue of velocity clamping, work by Engelbrecht [178] suggests that, particularly in the case of constrained optimisation problems, velocities of particles should be initialised at the start of the PSO algorithm to random values close to 0 (or even to 0) rather than to random values from the entire domain of the optimisation problem as the latter can produce large particle movements in early iterations of the algorithm, resulting in many infeasible solutions and wasted search effort.

Momentum Weight

Another means of controlling particle velocity is to implement a momentum, or inertia, coefficient. In this approach, equation (8.1) is altered by adding an additional coefficient:

$$v_{id}(t+1) = Wv_{id}(t) + c_1 r_{1d}(p_d^{\text{best}}(t) - x_{id}(t)) + c_2 r_{2d}(g_d^{\text{best}}(t) - x_{id}(t)). \quad (8.9)$$

Here, the coefficient W represents an inertia, or *friction*, weight which controls the impact of a particle's prior-period velocity on its current velocity. Higher values of the weight term encourage the search of diverse regions.

From (8.9) it can be seen that the impact of a given choice of value for W on the velocity of a particle also depends on the values of c_1 and c_2. The choice for these parameters determines whether the swarm concentrates on exploration (encouraged by selecting a high value of W relative to the values of c_1 and c_2), or on exploitation of already discovered good solution regions

(encouraged by selecting a low value of W relative to the values of c_1 and c_2). A common approach is to decrease the value of W gradually during the search process. The effect of dampening the value of W over time is to increase the effective influence of p^{best} and g^{best} on the velocity update calculation in an effort to encourage the swarm to converge, leading to more intensive local search of already discovered good regions. A simple method to achieve this is:

$$W = w_{\max} - \frac{w_{\max} - w_{\min}}{\text{iter}_{\max}} \cdot \text{iter}_{\text{curr}} \tag{8.10}$$

where $w_{\max}$ and $w_{\min}$ are the initial and final weight values, respectively (for example, 0.9 and 0.4), $\text{iter}_{\max}$ is the maximum number of iterations of the PSO algorithm, and $\text{iter}_{\text{curr}}$ is the current iteration number.

The methods of velocity clamping and inertia weight are not mutually exclusive. For example, a weight term can be supplemented with velocity clamping in order to encourage the swarm to converge and engage in fine-grained exploration around g^{best}.

Constriction Coefficient Version of PSO

Another method for controlling the magnitude of the velocity update step, the *constriction coefficient*, was proposed in [114]. In this approach, the momentum weight term is dropped and the velocity update for each dimension d is altered to:

$$v_{id}(t+1) = \chi(v_{id}(t) + c_1 r_{1d}(g_{id}^{\text{best}}(t) - x_{id}(t)) + c_2 r_{2d}(g_{id}^{\text{best}}(t) - x_{id}(t))) \tag{8.11}$$

where χ is the constriction coefficient. The value of the constriction coefficient is calculated as $\chi = \frac{2}{|2 - c - \sqrt{(c^2 - 4c)}|}$, where $c = c_1 + c_2$, and $c > 4$, the choice of these values being made to help ensure that the swarm converges to a small region of the search space. A common choice of value for χ is 0.7298, resulting from values of $c_1 = c_2 = 2.05$.

8.2.3 Neighbourhood Structure

A variant on the canonical particle swarm algorithm is to use a local best location (l^{best}) rather than a global best location when performing the velocity updates in (8.1).

In the local best version of the PSO algorithm each particle is notionally *linked* to a subset of the population of particles at the beginning of the algorithm. This linkage structure then remains unchanged during the optimisation process. The term l^{best} replaces g^{best} in (8.1), with l^{best} representing the best location found so far by any particle in that linked group. In defining the nature of the linkages between the particles, a wide range of connection structures could be employed. Figure 8.4 illustrates a three-particle neighbourhood

topology, where each particle is linked to two other particles. Although a subset of the particles are defined as being 'linked' this does not imply that the particles will be spatially proximate throughout the algorithm. It is quite possible, particularly in the early iterations of the algorithm, that the particles could be a considerable distance from each other.

Information Flow in Neighbourhood Structures

The neighbourhood size and structure plays a critical role in determining the flow of information between particles and therefore it impacts directly on the search process itself. At one extreme, if neighbourhood size is set at 1, each particle communicates only with itself. Consequently, the swarm acts as N independent searchers, each of which is anchored by its own p^{best} ($l^{\text{best}}=p^{\text{best}}$ for all particles in this case). On the other hand, if the neighbourhood size is defined as being all N particles, all particles can communicate with each other and we have the g^{best} version of the PSO algorithm (Fig. 8.5). If the neighbourhood regions are defined so that they do not overlap, the swarm behaves as multiple subswarms which simultaneously, and independently, search for good solutions. If the neighbourhoods are defined so that they do overlap, information about high-fitness regions can flow gradually between one neighbourhood and another. When the l^{best} version of the PSO is implemented with a neighbourhood size $< N$, and with overlapping neighbourhood regions, the swarm will tend to maintain more diversity and convergence will be slower than in the g^{best} version.

8.3 Comparing PSO and Evolutionary Algorithms

The PSO algorithm bears some similarity to evolutionary algorithms such as the GA. PSO is population-based, particles encode solutions, search proceeds by updating these encodings over multiple generations (iterations), and the population of particles is typically initialised randomly.

In both algorithms information is shared between members of the population. In the GA, the information-sharing mechanism is between two selected individuals (the parents). In PSO the communication is between an individual and g^{best} or l^{best}. The location of the highest quality solution discovered by any member of the group is broadcast to all members of the population.

Unlike the GA, PSO has no explicit crossover or mutation process. However, the velocity and position update processes can be considered as providing an implicit crossover, as the locations of p^{best} and g^{best} influence the velocity update step. These locations, along with the prior period velocity, are *blended* in producing the current period velocity update. In other words, p^{best} and g^{best} provide a form of 'parent influence' in PSO. The memory embedded in p^{best} and g^{best} can also be thought of as implementing a form of elitism, as

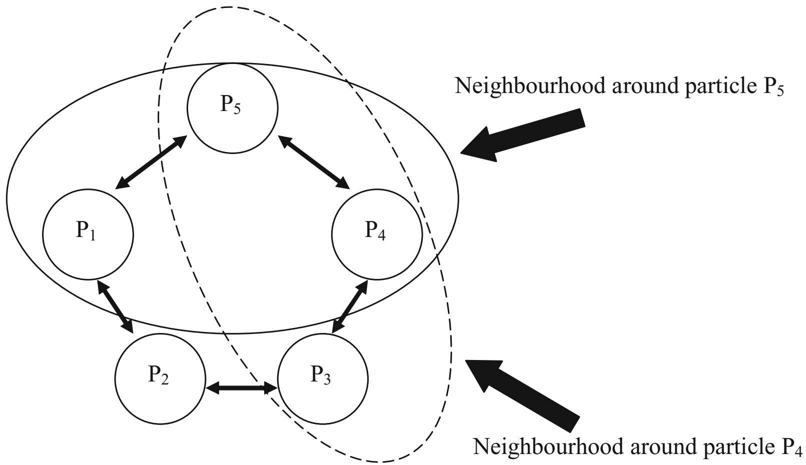


Fig. 8.4. Swarm topology where l^{best} is defined using a three particle, overlapping, neighbourhood. Each particle communicates with itself and with two other particles

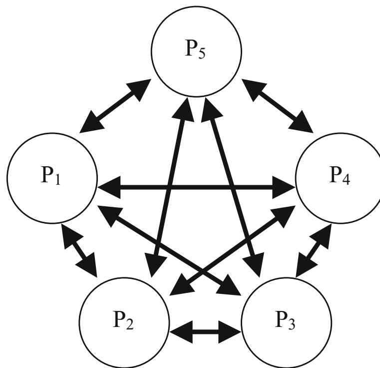


Fig. 8.5. Star topology for swarm. All particles can communicate with one another via g^{best}

knowledge of the best locations found so far in the search space is maintained between iterations of the algorithm.

Although there is no explicit selection or birth and death process in PSO, the attraction of particles towards g^{best} acts as an implicit selection mechanism as it influences the velocity of all particles. Selection pressure in the GA directly impacts on the rate of convergence of the population. Similarly, in PSO, the relative values of W, c_1 and c_2 control the rate of convergence.

8.4 Maintaining Diversity in PSO

Generally, there are three reasons for maintaining diversity in populational search algorithms:

i. to avoid premature convergence of the population,
ii. to cope with a dynamic environment, and
iii. to uncover multiple, equally good, solutions when they exist.

Premature Convergence

Premature convergence arises when a search algorithm stagnates. Highly multimodal environments can pose considerable difficulties for a search algorithm unless it is capable of generating sufficient diversity to allow the population to escape from local optima.

In PSO the potential problem of populational stagnation runs deeper as inspection of the velocity and position update equations of the canonical PSO algorithm points out that the swarm is not guaranteed to converge to a global, or even a local, optimum [177, 640]. If a swarm converges so that for all i we have $x_i = p_i^{\text{best}} = g^{\text{best}}$, and for all i, d we have $v_{id} = 0$, then the swarm will cease moving and freeze. While the swarm will have converged to the best location uncovered during its search there is no guarantee that this location is either a local or a global optimum (Fig. 8.6).

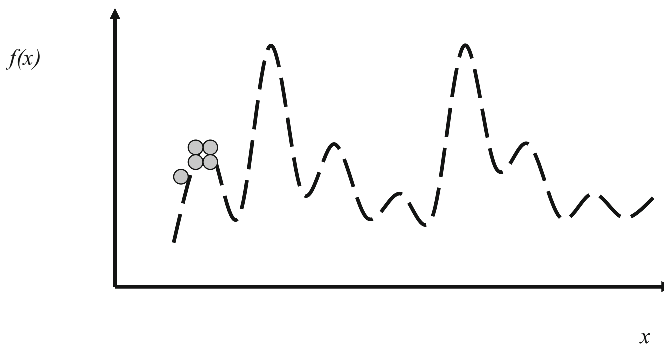


Fig. 8.6. Swarm has converged to a local rather than the global optimum (assuming the objective is maximisation)

One way to gain insight into the degree of convergence of the swarm during a run is to construct a *swarm activity graph*. Swarm activity can be measured in many ways but an intuitive approach is to take the distance moved by each particle in the swarm between two successive iterations of the algorithm and

plot this over time [378]. Hence, if the swarm has substantially converged, the degree of swarm activity will be small (Fig. 8.7).

$$\text{Swarm activity} = \frac{\sum_i^n |x_i(t) - x_i(t-1)|}{nk}. \tag{8.12}$$

In order to make the metric roughly comparable across swarms of different sizes, and problems of differing dimensionality, the Euclidean distance between the location x_i of each particle i at time t and $t+1$ is divided by nk, where n is the swarm size and k is the dimensionality of the problem of interest.

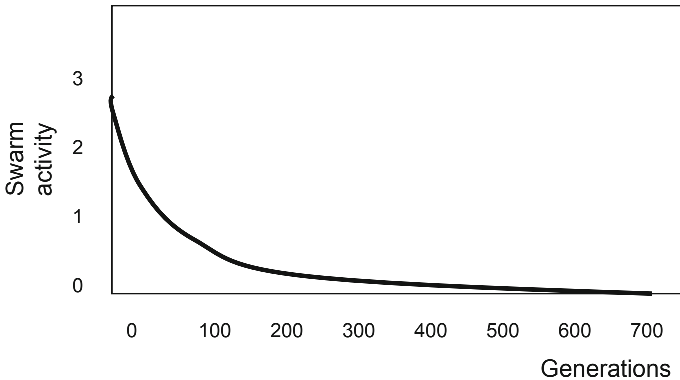


Fig. 8.7. Sample activity graph where the swarm is converging to a small region of the search space

While convergence to a local optimum can be easily guaranteed by modifying the canonical PSO to undertake a local search around the final g^{best} location, there is no simple fix which will ensure that a swarm will efficiently find the global optimum in all search spaces.

Dynamic Environments

Dynamic environments (Sect. 4.1) pose challenges for all optimisation methods, including PSO. If the environment alters substantially, past learning as captured in the memory of the locations of p^{best} and l^{best} or g^{best} can be worthless in guiding the search process in the altered environment. Additionally, if the swarm has collapsed into a compact region of the search space, and the inertia weight has decayed to a low value, the swarm will find it difficult to escape from that region, even if the global optimum moves elsewhere (Fig. 8.8).

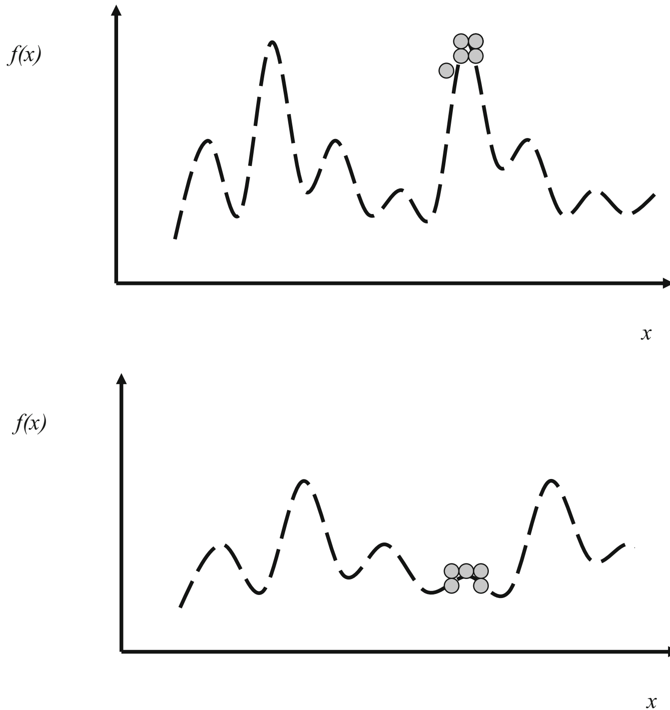


Fig. 8.8. Swarm has successfully uncovered the global optimum (above) but due to loss of diversity it is unable to track the global optimum as the environment changes (below)

Multiple Solutions

In some scenarios, there will be multiple, equally good, solutions, with the aim being to identify all or a subset of these (for example, multiobjective optimisation). Canonical PSO is not suitable for application to these problems due to its convergent nature. The following sections describe a number of approaches for maintaining diversity in PSO.

8.4.1 Simple Approaches to Maintaining Diversity

The simplest approaches to maintaining diversity during a PSO run include deterministic strategies such as random immigrants (Sect. 4.1.2) which help ensure that continual diversity is generated in the population of particles, and the careful choice of PSO parameter settings which encourage exploration rather than exploitation.

Another approach which is suitable for dynamic environments is to use adaptive strategies which boost the level of diversity generation when environmental change is detected. For example, a set of sentry particles (Sect. 4.1.2) at dispersed, fixed locations in the search space can be maintained in memory. Periodically, the fitness of these particles is reassessed and when environmental change is detected via a change in the fitness of these particles, steps can be taken to increase swarm diversity.

For example, the position or p^{best} information of some particles could be reinitialised to new, randomly selected, values. In effect, this implements a 'forgetting' mechanism whereby the past learning of the particle is abandoned. Another approach would be to turn on a mutation mechanism in an effort to generate positional diversity in the swarm's particles. In either case, it may be necessary to reset the value of the inertia weight (W) if it has decayed to a small value during the algorithm.

In addition to simple methods for maintaining diversity in populations of PSO particles, a number of more sophisticated methods, drawing inspiration from a number of metaphors, have also been developed. These are discussed in the following sections.

8.4.2 Predator–Prey PSO

In the canonical PSO algorithm all particles have identical properties. Silva et al. [578] introduce a predator–prey metaphor into PSO and split the population of particles into two mutually exclusive groups. A subset of the particles are considered as *predators* with the remaining particles being classed as *prey*. The predators are attracted to the best individuals in the swarm, whereas the prey particles are repelled by predator particles, thereby generating movement in the swarm (Fig. 8.9).

The biological motivation for the predator–prey model is that prey tend to gather around locations with good resources such as places with plentiful food or water. Prey, who are located at resource-rich locations, therefore have little motivation to seek out alternative resource locations. However, if the flock of prey is attacked and scattered by predators they will be forced to seek out alternative (possibly diverse) predator-free locations. These new locations may turn out to offer even richer resources than the original location. In terms of optimisation, good resource locations can be considered as local optimia.

The predator–prey metaphor could be implemented in a variety of ways. The approach taken in Silva et al. [578] is to have a single predator which is attracted towards the current g^{best} location. The velocity update equation for the predator particles is:

$$v_{\text{predator}}(t+1) = \alpha(g^{\text{best}}(t) - x_{\text{predator}}(t)) \tag{8.13}$$

$$x_{\text{predator}}(t+1) = x_{\text{predator}}(t) + v_{\text{predator}}(t+1) \tag{8.14}$$

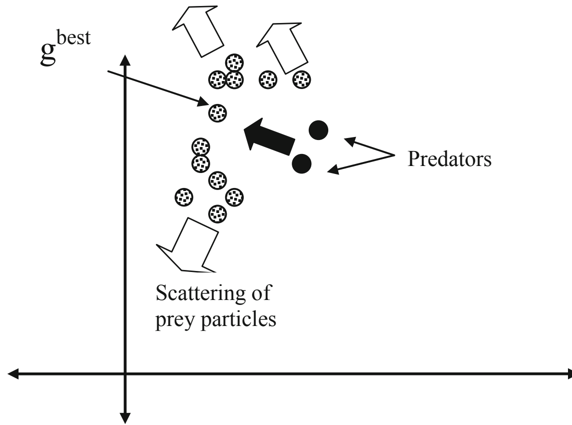


Fig. 8.9. Two predator particles chasing the g^{best} prey particle

where $v_{\text{predator}}(t+1)$ and $x_{\text{predator}}(t+1)$ are the velocity and location of the predator respectively. The parameter α controls how fast the predator moves towards the g^{best} location.

The influence of the predator on each prey particle depends on how close the predator is to the prey particle. The closer the predator the more the prey particle reacts by changing its velocity in order to avoid the predator. To capture this effect, a repulsion term $D(d)$ is added to the velocity update equation for prey particles. Therefore, for each dimension j for each prey particle i, the velocity and position update equations are:

$$v_{ij}(t+1) = Wv_{ij}(t) + c_1 r_{1j}(t)(p_{ij}^{\text{best}}(t) - x_{ij}(t)) + c_2 r_{2j}(t)(g_j^{\text{best}}(t) - x_{ij}(t)) + c_3 r_{3j} D(d)(t) \tag{8.15}$$

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1) \tag{8.16}$$

where the repulsion is calculated using an exponentially decreasing function, defined as $D(d) = a\mathrm{e}^{-bd}$. The parameter d is the Euclidean distance between the predator and the prey, a controls the maximum effect the predator can have on the prey's velocity along any dimension, and b is a scaling factor. The repulsion term produces a more violent reaction by the prey if the predator is very close. For example, if the predator and prey are in the same location, *distance*=0, and the repulsion effect is a (as $\mathrm{e}^0 = 1$). As the *distance* tends to ∞, the repulsive effect tends to 0 (since $\mathrm{e}^{-x} \to 1$ as $x \to \infty$).

Equation 8.15 is used to update each element of a prey's position vector based on a 'fear' threshold, P_f. For each dimension, if $U(0,1) < P_f$, then (8.15) is used to update $x_{ij}(t)$. Otherwise the standard PSO velocity update without the repulsion component is used.

In the early iterations of the PSO algorithm, most particles will not be close to the predator; hence, the predator–prey term in the velocity update vector will tend to have limited effect. As the swarm starts to converge towards the best-so-far g^{best}, the repulsion term helps ensure continued diversity in the swarm. Later in the search process, the influence of the predator should be decreased by reducing the fear threshold (P_f) or by reducing the value of a in order to permit finer search around g^{best}.

A variant on the above predator–prey approach was proposed by [272], in which there are multiple predators which behave as 'normal' PSO particles in that they are both drawn to the g^{best} location and are also influenced by their own p^{best} location. Predator particles employ the standard PSO velocity update equation. In contrast, the velocity update equation of prey contains a repulsion term, whereby a prey particle responds to its nearest predator by moving away from it. Both predators and prey use the same g^{best} information and both particles can update the position and value of g^{best}. Predators therefore tend to search around g^{best}, with the prey particles engaging in more diverse exploration of the search space. The predator–prey model will promote more populational diversity than the canonical PSO model and therefore will offer advantages in multimodal and dynamic environments.

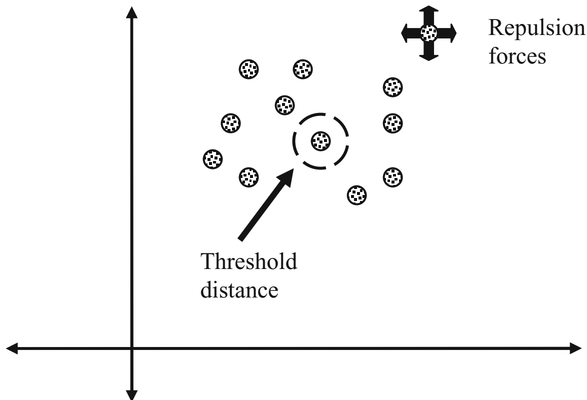


Fig. 8.10. Charged swarm, showing repulsion effect and threshold distance for repulsion effect

8.4.3 Charged Particle Swarm

Charged particle swarm, developed by Blackwell and Bentley [60, 61], represents a synthesis of both social and physical metaphors. In the charged

particle swarm model, a notional electrostatic charge is assigned to each particle. This results in a repulsion force between particles, governed by an inverse square law relationship, reducing the propensity of the swarm to converge and thereby helping to maintain diversity in the population of particles (Fig. 8.10). Charged swarm, like predator–prey PSO, helps ensure that diversity is maintained throughout the algorithm's run, rather than merely reacting to environmental change by attempting to boost populational diversity after it has occurred.

The main alteration required to the equations governing PSO in order to implement charged PSO is to the velocity update equation (8.9) to include a repulsion term r_i, resulting in the update equation:

$$v_{ij}(t+1) = Wv_{ij}(t) + c_1 r_{1j}(p_{ij}^{\text{best}} - x_{ij}(t)) + c_2 r_{2j}(g_j^{\text{best}} - x_{ij}(t)) + r_{ij} \quad (8.17)$$

The repulsion term is calculated at every iteration t for each particle i in the swarm of n particles, using:

$$r_{ij} = \sum_{j=1, j\neq i}^{n} \frac{Q_i Q_j}{|d_{ij}|^3} d_{ij}, \quad p_{\text{core}} < d_{ij} < p \quad (8.18)$$

where $d_{ij} = x_i - x_j$ and each particle i has a charge Q_i. If the particles have no charge ($Q_i = 0$ for all particles), then the repulsion term becomes 0, and the velocity update equation reverts to the standard update equation.

Once particles have a positive charge they are repelled from all other particles that are within a threshold distance p. Particles that are more than p apart do not influence one another. A second parameter is also defined by the modeller, p_{core}, and if particles are within this distance of one another, the repulsion effect is capped at

$$r_i = \sum_{j=1, j\neq i}^{n} \frac{Q_i Q_j d_{ij}}{p_{\text{core}}^2 |d_{ij}|} \quad (8.19)$$

in order to avoid extreme repulsions if the particles are very close together. In empirical testing of the charged swarm model, parameter values of $p_{\text{core}} = 1$, $p = \sqrt{3}\, x_{\max}$ and $Q = 16$ were suggested by [60].

While the charged swarm concept is successful in maintaining diversity in the swarm, it can be less effective in carrying out detailed exploration around the current g^{best} due to the repulsion effects between particles. A variant on the idea of a charged swarm is to split the swarm in two, where half the particles are charged and therefore repel one another and half are neutral particles which carry no change (the *atomic swarm* model) [62]. This combines the benefits of ensuring that there is always diversity in the swarm whilst allowing exploitation around the current g^{best} by the neutrally charged particles. Charged swarm PSO, by promoting diversity, will offer advantages over the canonical PSO in multimodal and dynamic environments.

Charged PSO provides an example of an algorithm whose inspiration is drawn from both a social and a physical metaphor. Later, in Chaps. 22 to 24, we introduce a range of physically inspired natural computing algorithms.

8.4.4 Multiple Swarms

Similarly to the island model in GA (Sect. 4.2), a cooperating multiple swarm system can be implemented where several swarms search independently of each other (Fig. 8.11). Occasionally, particles and/or information is migrated between the swarms. The use of multiple swarms can help encourage populational diversity, which can provide obvious benefits in environments which are dynamic, in environments which have many local optima, or in multiobjective problems where the aim is to uncover a diverse set of potential solutions (for example, a Pareto front). The downside of this approach is that the computational cost of the algorithm increases as the total population size increases.

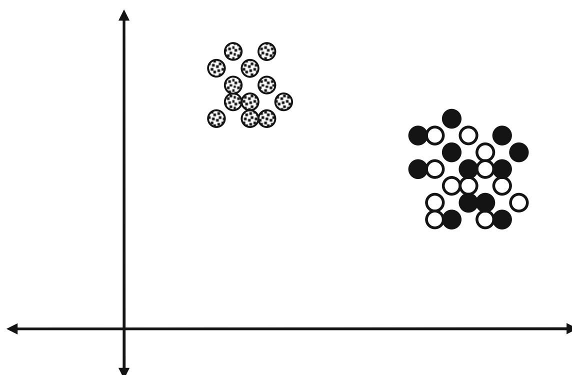


Fig. 8.11. Three independent swarms. Two swarms are separately converging on the same local optimum with the third swarm finding an alternative local optimum

In designing multiple swarm systems, the key decisions include the number of swarms that will be used and how information is to be passed between the individual swarms. One approach is to pass information on g^{best} between the swarms at periodic intervals. Another variant is to periodically replace the p worst particles from swarm A with the p best particles from swarm B (assuming the particles entering the swarm A are better than those they are replacing) and vice versa. An alternative approach is simply to swap randomly selected particles between each swarm.

When the number of swarms is increased beyond two, a migration strategy is required to govern these information flows. Possible strategies include a

sequential migration scheme where each swarm exchanges information and/or particles bilaterally with a predefined swarm, or a random migration scheme where the exchange occurs between two randomly chosen swarms at each migration event.

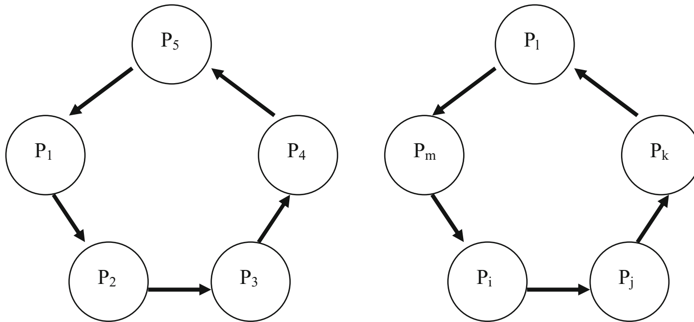


Fig. 8.12. Migration between swarms can be sequential (left) or random (right). In random migration, the value of the indices i, j, k, l, m for the order of migration are randomly selected without replacement at each migration event. Here, it is assumed that there are five swarms. This is an exemplar of a migration strategy between swarms where the order of migration between all swarms is random

8.4.5 Speciation-Based PSO

In some applications the object is to find multiple solutions rather than a single solution. For example, there may be more than one global optimal solution. The standard PSO is not well suited for this task as the algorithm is not specifically designed to capture and maintain information on multiple optima. The easiest way to try to uncover the locations of multiple optima is to undertake multiple sequential PSO searches, and record the g^{best} found by each search. However, this method will not be particularly efficient, as it is possible that several searches will produce the same g^{best}.

An established approach for dealing with the problem of multiple solutions in evolutionary algorithms is to use *niching* (or speciation) strategies. The objective of these strategies is to permit the optimisation algorithm to uncover multiple optimal solutions. Another approach to locating multiple niches is to undertake parallel niching. In parallel niching, a single swarm is initialised and begins searching. As soon as a promising region is identified, a subswarm of the particles which are close to that region is split off the main swarm. The subswarm then behaves as an independent swarm and undertakes its own search in that region in order to uncover the local or possibly the global optimum. Over time, the main swarm shrinks as subswarms are split from it.

Once the subswarms have substantially converged, the g^{best} for each swarm is recovered to form a list of possible solutions. One example of a parallel-niching PSO model is *NichePSO* [74, 177].

8.5 Hybrid PSO Algorithms

Search algorithms have their individual strengths and weaknesses and hybrid algorithms can be employed to improve search quality and efficiency. Two basic approaches to hybridisation are:

i. the operation of multiple algorithms on the same problem either in parallel or sequentially, and
ii. the blending of elements from multiple algorithms in order to design an improved algorithm.

Multiple Algorithm Hybrids

Illustrating the first case, a GA and a PSO could be run in parallel on the same problem, with a periodic exchange of high-quality individuals between the two subpopulations. This corresponds to an island model (Sect. 4.2) where different search strategies are being employed on each island.

Alternatively, different search algorithms could be sequentially applied to the population of solution encodings. For example, Hendtlass [267] describes a PSO-DE system where periodically the population of solution encodings being operated on by the PSO is passed to a DE algorithm. The DE algorithm is then executed for a number of iterations, with the updated locations of each particle being passed back to the PSO algorithm. A variant on this is to design a memetic verison of PSO, where a subset of the solutions in the current population, or perhaps just g^{best}, is refined using local search (Sect. 4.5). The local search step could be employed periodically during the PSO algorithm or alternatively as a final refinement step at the end of the PSO process.

Blended Hybrids

Concepts from PSO can also be blended with other search algorithms. For example, a selection-for-replacement mechanism drawn from the GA could be incorporated into PSO. A simple selection strategy would be to drop the poorest $x\%$ of particles after each iteration of the algorithm, replacing them with newly created, randomly located, particles. A more sophisticated approach is to periodically drop low-fitness particles, replacing their location and velocity vectors with those of higher-fitness particles in the current population, while leaving the p^{best} information for the replaced particle unchanged [16]. This has the effect of intensifying the search in a current good region, while maintaining a memory of historic high-fitness locations uncovered by that particle.

Another hybridisation possibility includes the implementation of a mutation operator. This could assist in maintaining diversity in the swarm of particles, thereby reducing the chance that the swarm gets trapped in a local optimum. An illustration of a mutation mechanism is provided by Higashi and Iba [271], where a particle selected for mutation has the value x_{id} of one of its d position dimensions altered using the function $\text{mutate}(x_{id}) = x_{id} \cdot (1+g(\sigma))$, where $g(\sigma)$ is a random number drawn from a Gaussian distribution with a mean of 0 and a standard deviation of σ. In their study, Higashi and Iba suggest using a value of σ of 0.1 times the range of the particle being mutated. This range can be decreased over time, allowing wider exploration early in the optimisation process and finer-grained search thereafter. The mutation process could implemented stochastically after either the velocity or the position update step. A detailed review of the literature on hybrid PSO models is provided in [177].

8.6 Discrete PSO

So far in this chapter it has been assumed that we are interested in real-valued optimisation. Of course, many real-world problems have integer or binary representations and PSO can be modified and applied to these problems. A simple modification to the PSO in order to apply it to an integer-valued problem is to discretise the position vectors by rounding each element in the vector to the nearest integer.

A number of more sophisticated approaches for applying PSO to binary-encoded problems have been developed. Two of these, *BinPSO* and *Angle Modulated PSO*, are discussed in the following sections. A third method, which draws inspiration from quantum mechanics, *Quantum Binary PSO*, is described in Sect. 24.6.

The first and last of the three binary PSO approaches redesign the velocity update process so that it is appropriate for binary encodings, while the second approach transforms the binary-encoded problem into one which has a continuous encoding. As described in Sect. 3.3, binary encodings can easily be transformed into any desired integer range. Hence versions of the PSO that operate on binary encodings can also be employed for discrete optimisation.

8.6.1 BinPSO

The best-known version of binary PSO, *BinPSO* [327], converts the continuous PSO algorithm to one which operates on binary representations. In BinPSO, the location x_i of each particle i is represented as a binary vector of 0s and 1s, and the search process takes place in binary-valued space. The adapted velocity update equation is virtually unchanged in appearance from (8.1):

$$v_{ij}(t+1) = v_{ij}(t) + c_1 r_{1j}(p_{ij}^{\text{best}} - x_{ij}(t)) + c_2 r_{2j}(g_j^{\text{best}} - x_{ij}(t)) \tag{8.20}$$

where x_{ij} is the value (0 or 1) in dimension j of particle i's location vector. All of the other terms in the update equation are as defined in (8.1). To ensure that each element of the vector $v_i(t+1)$ is binary, a sigmoidal transformation, sig, is performed on each element j of $v_i(t+1)$:

$$\text{sig}(v_{ij}(t+1)) = \frac{1}{1+\exp(-v_{ij}(t+1))}. \tag{8.21}$$

(The sigmoid used here is the logistic function (Sect. 13.4.1).) The value of $x_{ij}(t+1)$ is determined by comparing $\text{sig}(v_{ij}(t))$ with a random number drawn from $U(0,1)$:

$$x_{ij}(t+1) = \begin{cases} 1 & \text{if } U(0,1) < \text{sig}(v_{ij}(t+1)); \\ 0 & \text{otherwise.} \end{cases} \tag{8.22}$$

Although (8.22) looks similar to the standard velocity update equation for continuous PSO, it has a quite different interpretation in BinPSO. The velocity update vector v_{ij} is interpreted as particle i's predisposition to set the value in dimension j of its position vector to '1'. The higher the value of v_{ij} for an individual element of i's position vector, the more likely that $x_{ij} = 1$, with lower values of v_{ij} favouring the choice of $x_{ij} = 0$. $\text{sig}(v_{ij})$ represents the probability of bit x_{ij} taking the value 1 [327]. Therefore, if $\text{sig}(v_{ij}) = 0.3$ there is a 30% chance that $x_{ij} = 1$ and a 70% chance it is 0.

One point to note is that the use of an inertia weight may not be appropriate in BinPSO as it can have unexpected consequences. For example, if W decays towards 0 as the algorithm executes, this will tend to push v_{ij} towards 0, resulting in sig(0) which produces 0.50. This in turn implies that each bit position has a 50:50 chance of change, turning the algorithm into a random search.

Another related issue that can arise is saturation of the sigmoid function. This will occur if velocity values are either very large or very small. In either case, the probability of a bit change becomes very small and exploration will effectively cease. A simple way of reducing this problem is to implement velocity clamping such that $|v_{id}| < V_{\max}$. Therefore, in BinPSO, $V_{\max}$ acts to limit the probability that bit x_{id} takes a value of 0 or 1. If a low value is set for $V_{\max}$, this will increase the (random) generation of diversity in the algorithm, even once the population has started to converge. For example, if $V_{\max}$ is clamped to 3, the values for $\text{sig}(v_{ij})$ will be limited to the range 0.047 to 0.952. Consequently, there is still a good chance that diverse binary vectors will be generated even once the population has substantially converged. Hence, $V_{\max}$ acts as a mutation control knob in BinPSO, with smaller values allowing a higher mutation rate.

8.6.2 Angle-Modulated PSO

An alternative methodology for applying PSO to binary-encoded problems, *Angle Modulated PSO*, is outlined in [493]. In this approach the problem is

transformed so that the real-valued version of the PSO algorithm can be applied. The key step is the use of a generating function which produces a binary output from a real-valued input. PSO is used to tune the real-valued parameters of the generating function, rather than to search directly in the binary-valued problem space. This methodology has also been used to extend differential evolution to binary-encoded problems and a discussion of the approach is provided in Sect. 6.3.

8.7 Evolving a PSO Algorithm

While the canonical PSO algorithm has proven useful for a wide variety of real-world problems, the design of the algorithm should ideally be tailored to the specific problem at hand. Although a multitude of PSO variants exist it is not always apparent which variant should be used for a given application. An interesting alternative to making this decision via trial and error testing of PSO variants is to *breed* a good PSO algorithm for the problem of interest.

Work by Poli and Langdon [512, 513] illustrates how genetic programming (GP) (Chap. 7) can be used to evolve the velocity update equation for PSO. In these studies the function set for the GP system included the functions $+$, $-$, $*$ and $\%$ (protected divide). The terminal set consisted of the position x_i of particle i, its velocity v_i, the best location p_i^{best} previously visited by particle i, and the best location g^{best} uncovered by the entire swarm. The terminal set also included a set of numerical constants and a zero-arity function which returns a real number in the range $[-1, 1]$. Even with this compact function and terminal set, a wide variety of velocity update equations can be evolved. Figure 8.13 illustrates a tree representation of the velocity update equation $v_{id}(t+1) = v_{id}(t) + c_1(p_{id}^{\text{best}}(t) - x_{id}(t)) + c_2(g_d^{\text{best}}(t) - x_{id}(t))$.

A standard GP approach is taken whereby each member of the GP population is decoded into a PSO velocity update equation, the utility of that update equation is tested using some or all of the available data and the resulting measure of fitness is used to drive the evolutionary GP process. The ultimate output from the GP run is a problem-specific PSO update equation.

More complex function and terminal sets could be defined, including additional location memory structures and alternative local neighbourhood structures. This opens up the possibility of crafting highly tailored PSO algorithms. More generally, it is of course possible to apply an evolutionary methodology to 'evolve' other (i.e., non-PSO) natural algorithms [160].

8.8 Summary

The key learning mechanisms in the PSO algorithm are driven by a social learning mechanism so that good solutions uncovered by one member of a

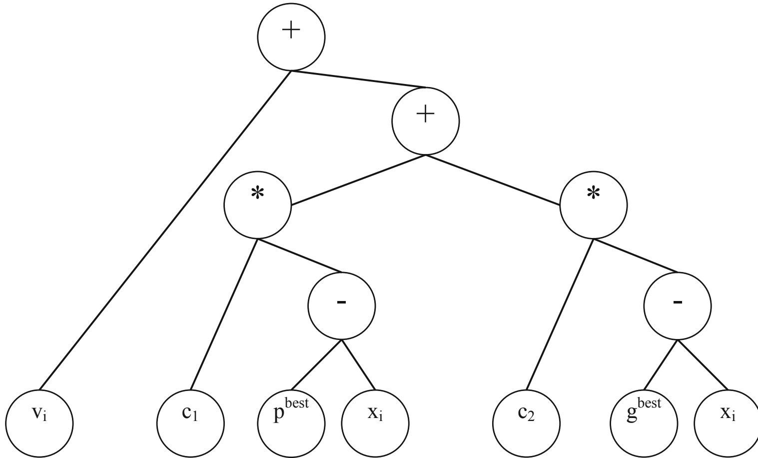


Fig. 8.13. Illustration of a tree representation of a velocity update equation

population are observed by, or communicated to, other members of the population that imitate them. Despite the simplicity of the PSO algorithm, it has shown itself to be a powerful, robust optimisation algorithm, having been successfully applied to a wide range of real-world problems. Particular advantages of the algorithm include its simplicity, its speed, and the relatively small number of parameters that the user is required to set.

PSO is a very active research area at present. Research efforts are clustered into the following areas: investigation of extensions to the basic PSO algorithms (for example, PSO for multiobjective optimisation, extending PSO to binary and integer encodings, developing PSO algorithms for dynamic environments), investigation of differing swarm topologies, investigation of hybrid PSO algorithms, and novel applications of PSO.

9

Ant Algorithms

At first glance the activities of insects do not appear to be an obvious source of inspiration for natural computing algorithms. However, on closer inspection it becomes apparent that many insects are capable of exceedingly complex behaviours. They can process a multitude of sensory inputs, modulate their behaviour according to these stimuli, and make decisions on the basis of a large amount of environmental information. Yet the complexity of individual insects is not sufficient to explain the complexity that many societies of insects can achieve [68]. Although only 2% of all insect species are social, these species have been remarkably successful at earning a living in their environment and comprise more than 50% of the global total insect biomass [19]. This suggests that the social nature of these species could be contributing to their relative success in colonising the natural world.

Three primary mechanisms of communication are observed in social insects:

i. indirect or *stigmergic* communication,
ii. direct interaction of individuals, where the actions of one individual influence those of another, and
iii. direct (nonphysical) communication between individuals.

Stigmergic communication arises where individual members of a group communicate indirectly, by altering the environment faced by their peers. This alteration in the environment has the effect of influencing the subsequent behaviour of other members of the group. Direct interaction occurs via mechanisms such as touch (for example, *antennation*, in which insects rub antennae against each other in order to communicate information about food sources, hunger levels, nestmate recognition and sexual identification etc. [46]) or phenomena such as *stridulation*, whereby individual ants can use sound signals to communicate with other ants, for example to recruit them for a specific task [282]. Ants of the species *Aphaenogaster cockerelli* and *Atta cephalotes* [535] use this mechanism to recruit other ants in order to facilitate prey retrieval when large prey are found.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_9

The best developed family of insect-inspired algorithms are drawn from a variety of ant behaviours and these form the focus of this chapter. More recently, the range of insect-inspired algorithms has been extended to encompass communication mechanisms of other species such as honeybees and these are described in the next chapter (Chap. 10).

9.1 A Taxonomy of Ant Algorithms

Ant algorithms [68, 165, 168, 169] constitute a family of population-based optimisation and clustering algorithms that are metaphorically based on the activities of social ants. Many species of social ants live in colonies. Despite the high degree of organisation of these colonies, there is no overt top-down hierarchial structure. Each individual insect follows a fairly limited set of rules, usually with only local awareness of its environment. In spite of this, the interaction of the activities of these individuals gives rise to a complex *emergent*, self-organised structure and provides the colony with the ability to adapt to changes in its environment. Ant algorithms emphasise the importance of communication (or *distributed learning*) between the individuals in a population by permitting the population to adapt successfully over time. In general, ant algorithms are derived from four metaphors of ant behaviour (Fig. 9.1). The first, foraging, inspires optimisation algorithms, while the next two, brood sorting and cemetery formation, inspire clustering algorithms. In this chapter we limit attention to ant food foraging algorithms and to ant-clustering algorithms.

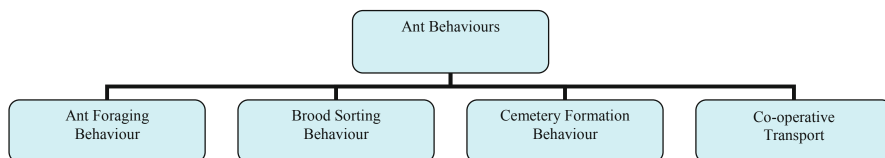


Fig. 9.1. Taxonomy of ant colony algorithms

9.2 Ant Foraging Behaviours

Foraging behaviour, a subfield of behavioural ecology [594], can encompass the activities of a single animal, or, more generally, the cooperative activities of a group of animals (predators), searching an environment for food (prey) or other resources.

Food foraging can be considered as an optimisation process. Broadly speaking, organisms seek to maximise the net energy they obtain from foraging (the energy obtained from food less the energy expended in finding it) per unit of time, subject to constraints such as the physical limitations of the organism. Foraging organisms must decide how to search for food. In other words they must design and implement a *food-foraging strategy* (see [29] for a good overview of a selection of these strategies). These food foraging strategies can serve as a source of inspiration for the design of optimisation algorithms [524].

Many examples of food foraging behaviours can be found in the natural world, ranging from those of herbivores to those of carnivores. Typically food is distributed in regions or patches. Hence food foraging entails searching for good regions on a landscape. For example, the area around a water hole in an arid environment is likely to be a rich hunting ground for carnivores. Of course, the resource endowment of specific regions can change over time in response to environmental conditions, or indeed in response to the success of predator activities.

Food foraging behaviour can be individual, where each individual in a species forages on its own, or social, where foraging is a group behaviour which involves cooperation and direct or indirect communication between individuals.

Many species of social ants use indirect communication mechanisms to assist in their food foraging efforts, for example by providing chemical 'signposts' to discovered food sources. During food foraging some species of ant such as the Argentine ant *Linepithema humile* lay down a trail of chemical attractant known as *pheromone* and subsequent foraging ants are inclined to follow these trails during their own foraging efforts. This simple trail-following behaviour acts to ensure that the ant colony's foraging activities are efficient.

If a group of ants searches randomly around its nest for food, pheromone trails from the nest to close-by food sources will tend to grow more quickly in strength than those to far-away food sources as ants travelling to the closest food source will return quickly to the nest leaving both an outward and inward trail of pheromone. The quality of the food source may also affect the amount of pheromone deposited, with better food sources resulting in higher levels of pheromone deposit. Once subsequent foraging ants tend to follow stronger rather than weaker pheromone trails, *auto-catalytic* behaviour will emerge with an increasing portion of the foraging ants travelling along the strong trail, reinforcing it further (Fig. 9.2). This creates a positive feedback loop between the ants in the search for food. The effect of the pheromone-following behaviour is to create an indirect communication mechanism between ants. As trails emerge over time a collective *memory*, chemically embedded in the environment, is created of the route to the food source. One feature of this communication mechanism is that it is scalable as an individual ant does not need to directly communicate with every other ant in the colony in order to pass on the knowledge it uncovers during its foraging travels.

Of course, one potential drawback of such a positive feedback learning mechanism is that it can lead to *lock-in*, whereby a heavily pheromone-reinforced path continues to be used, even if a rich food source subsequently becomes available closer to the nest.

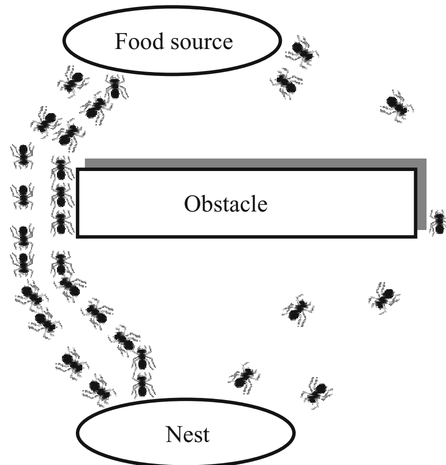


Fig. 9.2. Foraging ants reinforcing the shorter trail to a food source

Although most optimisation applications of ant foraging algorithms rely on positive reinforcement of routes to high quality solutions via a simulated pheromone deposit process, real-world ant foraging pheromone signals can be more complex. For example, Pharaoh's ants (*Monomorium pharaonis*) can deposit repellant pheromone as a 'no entry' signal to mark an unrewarding foraging path [533]. Real-world foraging behaviours may also be influenced by both indirect and direct communication (see [46] for an example of an ant colony optimisation algorithm which uses a combination of concepts including pheromone deposit and antennation).

9.3 Ant Algorithms for Discrete Optimisation

9.3.1 Graph structure

Ant foraging behaviours can be used to inspire the design of algorithms for discrete optimisation. This class of problem can be represented using a graph structure where nodes (or *vertices*) are connected by arcs (or *edges*). The first

step in applying ACO for discrete optimisation is to map the problem onto a suitable *construction graph* so that a path through the graph (corresponding to an ant foraging route) represents a solution to the problem. The task is then to find the optimal path through the graph.

As an example of a construction graph, suppose the objective is to create a timetable, whereby a series of classes (events) are to be assigned to specific time slots. One way to represent this problem as a graph structure is to consider each class (or other event to be scheduled) as a node, and each time slot as an arc (Fig. 9.3). A walk through this graph over specific arcs produces a timetabling of each class to a specific time slot and the object is to produce a timetable which is at least feasible (for example, no more than k classes assigned to a particular time slot), and preferably a timetable which maximises some measure of 'goodness'. In many combinatorial problems, each arc in the graph has some cost associated with it and the object of the problem is to find a low-cost route across the graph.

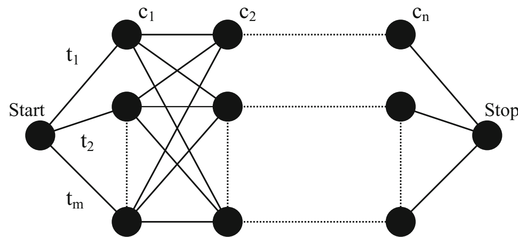


Fig. 9.3. Construction graph for a timetabling problem. Each arc corresponds to a choice of time slot (m time slots exist), and each node corresponds to a class (n classes in total)

Once a construction graph has been designed for the combinatorial optimisation problem of interest, a computer simulation of the activities of artificial ants can be used to uncover a good route across the graph. These artificial ants are released one at a time at the start node. Each ant builds a complete solution to the problem by traversing arcs from one node to the next until a terminating node is reached.

At each node the ant will typically have a choice of several outgoing arcs. In canonical ant colony algorithms the ant has access to two pieces of information when making its decision as to which arc to select. The first piece of information is the quantity of pheromone deposited on each arc, which acts as a guide as to how many previous ants have traversed that arc, and some information concerning the quality of each arc (usually based on a heuristic such as 'pick the outgoing arc with the lowest cost value'). The ant stochastically selects its next arc based on a blending of these pieces of information,

tending to favour arcs which are heavily reinforced with pheromone and arcs which are considered good using the heuristic guide to arc quality.

After each ant in the population has constructed a solution by walking across the construction graph, the quality of each of these solutions is assessed. The pheromone trails on the arcs are then updated, with arcs on the highest-quality solutions having additional pheromone deposited on them. Over multiple iterations of the algorithm, the arcs belonging to the better solutions become more heavily reinforced with pheromone, and consequently more ants tend to follow them. This leads to an intensification of search around those solutions.

The pheromone trails on arcs are also subject to an *evaporation* process during each iteration of the algorithm. This guides the ants by ensuring that defunct, less travelled solution fragments are forgotten over time. The high level pseudocode for an ant foraging algorithm is outlined in Algorithm 9.1.

Algorithm 9.1: Ant Foraging Algorithm

```
Initialise pheromone trails to positive value;
repeat
    for each ant in turn do
        Construct a solution;
        Measure quality of solution;
    end
    Update pheromone trails via pheromone deposit and evaporation;
until terminating condition;
```

Pheromone Matrix as a History

The pheromone information associated with each arc in a construction graph is stored in a *pheromone matrix* (9.1). The entries in the matrix correspond to the quantity of pheromone on the edges between each node, with 0 values on the diagonal of the matrix. The matrix is updated in each iteration of the algorithm, and the values represent a form of memory for the ant system. In (9.1) it is assumed that the pheromone levels are the same regardless of the direction that the arc is traversed. More generally, this need not be the case.

$$\begin{array}{ll} \textbf{Nodes} & \begin{array}{cccc} 1 & 2 & \dots & n \end{array} \\ \begin{array}{l} 1 \\ 2 \\ \vdots \\ n \end{array} & \begin{pmatrix} 0 & 0.23 & \dots & 0.5 \\ 0.23 & 0 & \dots & 0.33 \\ \vdots & \vdots & \ddots & \vdots \\ 0.5 & 0.33 & \dots & 0 \end{pmatrix} \end{array} \tag{9.1}$$

There is a crucial distinction between the role of memory in ant foraging algorithms and the role of memory in other population-based natural computing algorithms such as the GA or PSO. In the latter algorithms, memory of the learning that has occurred during the search process is embedded in the multiple solution encodings stored in the population. As complete encodings are stored, this allows the preservation of interdependencies between solution components.

In contrast, in ant foraging algorithms the memory of past learning does not reside in the individual ants; instead it resides in the pheromone matrix. In addition, the nature of this memory is different from that in GA, PSO, etc. as the matrix does not maintain information on multiple individual solutions found by individual ants. Instead it integrates information from many solutions, over multiple iterations of the algorithm, into a single memory. Hence, the memory structure in ant foraging algorithms has the advantage of durability but it is poorer at capturing interdependencies between solution elements.

Table 9.1. Correspondence between ant systems and the optimisation process

Ant System Component	*Optimisation*
Complete ant trail	Solution
Choosing arcs at each node	Search process
Pheromone trail	Memory of good solution fragment
Updating of pheromone trails	Learning via reinforcement and forgetting

9.3.2 Ant System

The original ant foraging algorithm, known as the *ant system* (AS), was developed by Dorigo [164]. In operationalising the general framework in Algorithm 9.1, a number of decisions must be addressed by the modeller:

i. How are pheromone trails initialised?
ii. How do the ants construct protosolutions?
iii. How are the pheromone trails updated?
iv. Which solutions participate in the deposit of pheromone?

Pheromone Initialisation

Choosing the appropriate levels of pheromone to initialise arcs at t_0 is important as there is a link between the level of pheromone on the arcs of the construction graph at the start of the algorithm and the rate of convergence of the algorithm to a single solution. If the initial levels of pheromone are very low the algorithm will tend to quickly converge on the first good solution which is uncovered, before adequate exploration has occurred. This occurs

because the first solution to receive reinforcing pheromone deposits will be highly favoured by subsequent foraging ants. On the other hand, if the level of pheromone used to initialise arcs is very high, early update steps will have little effect and useful search will be delayed until sufficient evaporation has occurred to decrease pheromone levels to the point where the pheromone deposit can begin to bias the search towards good solutions. Dorigo and Stützle [169] provide guidelines for appropriate parameter settings, including initial pheromone values, for a variety of forms of ant system.

Constructing Protosolutions

The key issue faced by ants when constructing a route through the graph, is which outgoing arc should be selected at each node. The simple approach would be to select the outgoing arc which has the highest pheromone level associated with it. However, this would result in rapid convergence to a single solution, usually producing a poor result.

A more sensible approach would be to allow the ant to stochastically chose from the set of available arcs. For example, the probability of choosing arc ij from amongst the K possible feasible arc choices at a particular construction step could be determined using:

$$P_{ij} = \frac{\tau_{ij}}{\sum_{k=1}^{K} \tau_{ik}} \tag{9.2}$$

where τ_{ik} is the quantity of pheromone associated with arc ik. Suppose there were three arc choices facing an ant at node i, with $P_{i1} = 0.3$, $P_{i2} = 0.4$ and $P_{i3} = 0.3$. A random draw from $U(0, 1)$ producing (say) 0.2 falls into the range $(0 \rightarrow 0.29)$, implying that the ant follows the arc $(i, 1)$. This approach ensures that while arcs which have been part of good solutions in the past are more likely to be selected, an ant still has the potential to explore any arc with a nonzero pheromone value. However, even with the addition of this stochastic choice mechanism, ants tend to quickly lock in on a single route, resulting in poor exploration of the search space.

To combat this problem, the AS algorithm combines pheromone information with a heuristic a priori estimate of the likely quality of each arc when making the choice of which arc to add to the solution being constructed. Adding heuristic information to guide the solution construction process is known as adding *visibility*, or look-ahead, to the construction process.

An illustration of how heuristic information can be used is provided by the well-known travelling salesman problem (TSP). The TSP was one of the earliest applications of AS and is a standard benchmark problem for combinatorial optimisation techniques, as it is an NP-complete problem. In the TSP there is a network of n cities. Each route, or arc, between two cities has a distance or cost associated with it, and the object is to find the tour which minimises the distance travelled in visiting all the cities and returning to the starting city.

In the case of the TSP a simple heuristic for assessing the possible utility of each arc choice facing an ant when it leaves city i is the distance between city i and all other cities to which it is connected, with shorter distances being preferred to longer ones. The ant weighs up both the information from this heuristic and the pheromone information on each arc when selecting which city to visit next.

As the objective is to visit each city once, and once only, during a tour, the choice of which city to visit next should exclude cities already visited. The ant maintains a memory of all cities it has already visited for this purpose. Hence, for the TSP, (9.2) is adapted to produce (9.3). Thus, the probability of ant k travelling from city i to city j at time t, where C_i^k is the set of feasible cities reachable from city i and not yet visited by ant k, is:

$$P_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}(t)]^\beta}{\sum_{c \in C_i^k} [\tau_{ic}(t)]^\alpha \cdot [\eta_{ic}(t)]^\beta}, j \in C_i^k \tag{9.3}$$

The term $\eta_{ij} = 1/d_{ij}$ where d_{ij} is the distance between the cities i and j, and the parameters α and β represent the weight attached to pheromone and heuristic information respectively.

Examining the form of this arc choice rule, if the term $\beta = 0$, then (9.3) effectively reduces to (9.2) and only pheromone information is used in determining the ants' movements. Conversely if $\alpha = 0$, the pheromone information is ignored and only heuristic information is used to guide the search, resulting in a greedy search by ants. While good choices for α and β are problem-specific, parameters of 2 and -2 respectively are not unusual [46].

Commonly in TSP applications, the number of ants is equal to the number of cities, and during each iteration of the algorithm, an ant is started from each city in turn in an effort to ensure that tours starting from all cities are investigated.

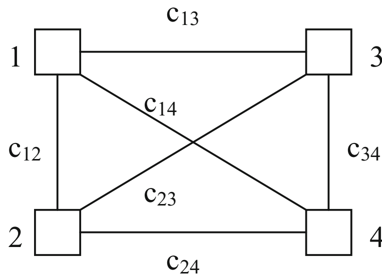


Fig. 9.4. A construction graph for a TSP with four cities. The routes correspond to the arcs/edges of the graph and the four cities are the nodes/vertices

Updating Pheromone Trails

After each of the ants has traversed the graph and has constructed its individual solution to the problem, the quality of each of these solutions is assessed and this information is used to update the pheromone trails. The update process typically consists of an *evaporation* step and a pheromone *deposit* step:

$$\tau_{ij}(t+1) = \tau_{ij}(t)(1-p) + \delta_{ij}. \tag{9.4}$$

In the evaporation step the pheromone associated with every arc is degraded or weakened. The evaporation rate p crucially controls the balance between exploration and exploitation in the algorithm. If p is close to 1 then the pheromone values used in the next iteration of the algorithm are highly dependent on good solutions found in the current iteration, leading to local search around those solutions. Smaller values of p allow solutions from earlier iterations of the algorithm to influence the search process.

The amount of pheromone δ_{ij} deposited on each arc ij during the pheromone update process depends on how the deposit step is operationalised in the algorithm. The deposit step can be performed in many ways depending on which solutions are chosen to participate in the deposit process, what weight is applied to each of these solutions in the deposit process, and how pheromone is deposited on each arc participating in the deposit process. One design of the deposit mechanism for the TSP (called the *ant-cycle* version of AS) is:

$$\tau_{ij}(t+1) = \tau_{ij}(t)(1-p) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t). \tag{9.5}$$

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^k(t)}, & \text{if } (i,j) \in T^k(t); \\ 0, & \text{otherwise.} \end{cases} \tag{9.6}$$

where m is the number of ants, (i,j) is the arc from city i to city j, T^k is the tour done by ant k at iteration t and $L^k(t)$ is the length of this tour. The term $\Delta\tau_{ij}^{k}(t)$ represents the pheromone laid on arc (i,j) by ant k, and Q is a positive constant.

Hence, in (9.6) the amount of pheromone laid on an arc by an ant varies inversely with the length of the tour undertaken by that ant. Arcs on longer tours get less reinforcement than arcs on higher-quality, shorter tours. In the ant-cycle version of AS, every ant lays pheromone at the end of each complete iteration of the algorithm and the total of all these deposits influences ants in the next iteration of the algorithm. Over multiple iterations of the algorithm, solution construction is the result of community learning by all the ants.

If the object is to maximise, rather than minimise an objective function, (9.6) can be recast as:

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} Qf(x^k(t)), & \text{if } (i,j) \in \text{path } x^k(t); \\ 0, & \text{otherwise.} \end{cases} \tag{9.7}$$

where $f(x^k(t))$ is the fitness or quality of ant k's solution.

Which Solutions Participate in the Update Process?

In choosing which solutions participate in the update process, one method (as seen above) is to allow all solutions in the current population to play a role in the deposit process. Another approach is to restrict the number of ants participating and, in the limit, only allow the best solution to deposit pheromone.

Elitist Ant System (EAS) [164] combines these approaches so that while each ant deposits pheromone, the best-so-far (elite) tour discovered is also reinforced in each iteration of the algorithm. In this case, the update step in (9.5) is amended to (9.8) through the addition of an extra term $\Delta\tau_{ij}^*$ (see (9.9)), which increments the pheromone on edge (i, j) if it is on the tour traversed by the elite ant. The tour found by this ant is denoted T^*, the length of this tour is L^* and σ is a scaling constant which controls the degree of reinforcement of the best tour. As the value of σ increases, the ants are encouraged to search intensively around the best-so-far solution, increasing the risk of premature convergence on a locally optimal solution.

$$\tau_{ij}(t+1) = \tau_{ij}(t)(1-p) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t) + \Delta\tau_{ij}^* \tag{9.8}$$

$$\Delta\tau_{ij}^* = \begin{cases} \sigma\frac{Q}{L^*}, & \text{if } (i,j) \in T^*; \\ 0, & \text{otherwise.} \end{cases} \tag{9.9}$$

The *rank-based ant system* (AS_{rank}) developed by Bullnheimer et al. [82] adopts a different approach. Before the pheromone deposit process, all ants are ranked according to the quality of their solution and the amount of pheromone deposited by each ant decreases with its rank. As with elitist ant systems, the best-so-far ant also participates in the pheromone deposit step. The pheromone update (and evaporation) rule is therefore:

$$\tau_{ij}(t+1) = \tau_{ij}(t)(1-p) + \sum_{k=1}^{w-1} (w-k)\Delta\tau_{ij}^k(t) + w\Delta\tau_{ij}^* \tag{9.10}$$

where only the best $(w-1)$ ranked ants participate in the update process, along with the tour of the best-so-far ant (on tour T^*). The value of $\tau_{ij}^k(t) = 1/L^k$ and the value of $\Delta\tau_{ij}^* = 1/L^*$, where L^* is the length of the best-so-far tour. Thus, the best-so-far tour receives a weight of w in the update process with the successive ranked tours receiving a lower weighting.

9.3.3 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System

In order to reduce the risk of premature convergence of the optimisation process to a single solution, and in order to ensure that every arc has a nonzero chance of being selected during the solution construction process,

the pheromone τ_{ij} associated with each arc ij may be constrained so that after the pheromone update process $0 < \tau_{\min} \leq \tau_{ij} \leq \tau_{\max}$ (the values of $\tau_{\min}$ and $\tau_{\max}$ are set by the user). The bounds prevent the relative differences between the pheromone trails on each arc from becoming too extreme.

A variant of AS which adopts this idea is the $\mathcal{MAX}$-$\mathcal{MIN}$ ant system (MMAS) [604]. In MMAS all arcs are initialised with $\tau_{\max}$ pheromone. During the update process only arcs on the global best solution, the *best-so-far* (or the best solution found in the current iteration, the *iteration best*), are updated and pheromone levels are constrained to $0 < \tau_{\min} \leq \tau_{ij} \leq \tau_{\max}$. As MMAS strongly exploits the information in the best tour found so far it is necessary to constrain the deposit of pheromone along this tour in order to ensure that the search process does not stagnate too quickly. Another mechanism often implemented in MMAS in order to encourage continued diversity in the search process is to periodically reinitialise the pheromone levels on all arcs in the network. This step is usually undertaken when stagnation of the optimisation process is detected.

9.3.4 Ant Colony System

The Ant Colony System (ACS) algorithm was developed by Dorigo and Gambardella [166, 167] in order to improve the scalability of the canonical AS algorithm. ACS differs from AS as it uses:

- a different construction rule,
- a different pheromone update rule, and
- a local pheromone update process.

Construction Rule in ACS

In comparison with AS the transition rule in ACS is altered so that an ant (k) at city i moves to city j according to the rule:

$$j = \begin{cases} \text{argmax}_{l \in J_i^k} (\tau_{il}(t)[\eta_{il}(t)]^\beta), & \text{if } q \leq q_0; \\ J, & \text{otherwise.} \end{cases} \tag{9.11}$$

where q is a random variable drawn from a $U(0, 1)$ distribution and $0 \leq q_0 \leq 1$ is a threshold parameter. Hence with probability q_0, the ant exploits the information in the pheromone trails and the decision heuristic, selecting the best possible next arc based on this information. With probability $1 - q_0$, the ant selects its next arc $(J \in J_i^k)$ from the set of cities yet to be visited (or from a candidate list) randomly according to the probability distribution in (9.3) (where $\alpha = 1$).

Therefore, q_0 is a tuning parameter which determines the degree to which the system focusses search attention on exploiting already-discovered good arcs.

Pheromone Update in ACS

In ACS, only the best-so-far ant updates the pheromone trails at the end of each iteration of the algorithm (an *offline pheromone update*). In contrast, in canonical AS, all ants participate in the update process. The pheromone evaporation step is also adjusted so that arcs on the best-so-far trail are subject to an evaporation process. Hence, the level of pheromone at $t+1$ is a weighted average of the pheromone level at t and the new pheromone deposited, with the parameter p governing the relative importance of both. The deposit and evaporation steps are governed by:

$$\tau_{ij}(t+1) = \tau_{ij}(t)(1-p) + p\Delta\tau_{ij}^{*} \tag{9.12}$$

where only the arcs traversed by the best-so-far ant (on tour T^*) participate in the pheromone deposit/evaporation process. The term $\Delta\tau_{ij}^{*}(t) = 1/L^*$, where L^* is the length of the best-so-far tour. As in AS, this pheromone update step is performed after all ants in the population have constructed their solutions.

Local Update in ACS

In addition to the offline pheromone update process on the best-so-far solution, a real-time local update is performed by *each* ant after it traverses an arc. Each ant applies this update only to the last arc traversed. Hence, if an ant traverses an arc (i,j), then the pheromone level on that arc is immediately adjusted as follows:

$$\tau_{ij}(t+1) = (1-\alpha)\tau_{ij}(t) + \alpha\tau_0 \tag{9.13}$$

where $0 < \alpha < 1$ is an evaporation parameter. The parameter τ_0 is typically set to the same value as was used to initialise the pheromone trails at the start of the algorithm. The intent of the local updating rule is to reduce the value of τ_0 each time an arc is traversed by an ant so that subsequent ants are encouraged to try other arcs. Implementing this mechanism also discourages the algorithm from stagnating.

Considering the three ACS mechanisms in their totality, they create a dynamic interplay of exploitation and exploration. Depending on the parameter settings chosen for each mechanism, ACS is capable of a wide range of search behaviours. Although the algorithm does place considerable focus on the best-so-far solution, it counterbalances the convergent effect of this by means of the local updates and through the stochastic transition rule.

Readers interested in studying the ACS algorithm in greater depth are referred to [166, 167, 169].

9.3.5 Ant Multitour Systems

Both of the previous algorithms replace all ants after each iteration and hence there is no concept of internal learning by individual ants in the population.

A contrasting algorithm, the *ant multitour system* (AMTS) [268] embeds a concept of 'prior learning' by individual ants (corresponding to 'learning by experience') in which ants can remember previous choices that they have made. In this algorithm, the ants remember the number of times they have traversed each edge in the previous q tours (q is a user-defined parameter). In order to promote exploration, each ant becomes increasingly unlikely to follow edges that it has previously traversed. Hence the attractiveness of an individual edge (i, j) for ant k is given by:

$$\text{Attractiveness}_{ij}(t) = \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}(t)]^{\beta}}{F_{i,j}} \tag{9.14}$$

where $F_{i,j}$ is determined by $\text{prior}_{i,j}$, the number of times that the specific ant K has traversed the edge i, j. A sample relationship between $F_{i,j}$ and $\text{prior}_{i,j}$ is as follows [46]

$$F_{i,j} = 1 + \sqrt{\text{prior}_{i,j}} \tag{9.15}$$

The term (9.14) replaces the numerator in (9.3).

9.3.6 Dynamic Optimisation

If an ant algorithm has substantially converged before an environmental change occurs, the canonical versions of the algorithms will usually find it difficult to track the new optimum. This arises as the pheromone reinforcement process encourages the population of ants to converge and lock in to a single solution. Hence, applications of ACO to dynamic problems usually require careful design in order to overcome this problem.

As discussed in Sect. 4.1, the optimal response depends on the nature and scale of the environmental change. If a drastic change has occurred in the environment, for example, multiple new cities need to be inserted on a delivery route, it may be necessary to reinitialise the pheromone matrices to their starting values, and then rerun the algorithm, remembering to update the heuristic values for each arc if these have changed.

If the environmental changes are less drastic, one possibility is to maintain parameter settings which promote continued exploration by the ants. For example, by placing higher weight on heuristic information (β) in (9.3), adaption to the altered environment is facilitated. Alternatively, parameter settings (the rate of pheromone evaporation, the values of α and β, etc.) could be adapted dynamically during the execution of the algorithm in response to the detection of environmental change. Another approach which is investigated in Mavrovouniotis and Yang [395] is to use immigrant schemes (Sect. 4.1.2) whereby either random immigrants (randomly created ants) or elitism-based immigrants (immigrants created by mutating the best ant found in the previous iteration of the algorithm) are used to maintain an exploration of the search space. Dorigo and Stützle [169] discuss a variety of other strategies for applying ant algorithms for dynamic environments (also see Sect. 9.5).

9.4 Ant Algorithms for Continuous Optimisation

Although the canonical version of the ant algorithm was designed for discrete optimisation, many optimisation problems involve real-valued parameters. Of course, real-world ant foraging behaviour takes place in a continuous space with pheromone trails diffusing on the landscape once they are laid down.

The earliest approaches applying an ant colony metaphor to continuous optimisation used discretisation whereby the (continuous) range of possible options at each node is reduced to a discrete number using defined grid intervals. A refinement of this approach is to undertake an initial course-grained search, switching to a finer-grained search once a promising solution region is identified [55].

More recently, a number of algorithms for continuous optimisation have been developed which are designed to work directly with real values. One of these is the *continuous ant colony system* (CACS) [517]. In CACS a continuous pheromone model is defined. Pheromone is not considered to be laid along a single discrete 'track'; rather it is laid down spatially. For example, a food source surrounded by multiple ants would be expected to show a high pheromone concentration, with this concentration dropping off as distance to the food source increases.

One way of modelling this phenomenon is to use a normal probability density function (PDF) where the probability of observing a (real-valued) sample point x is defined as:

$$\text{Pheromone}(x) = \mathrm{e}^{\frac{-(x-x_{\min})^2}{2\sigma^2}} \tag{9.16}$$

where $x_{\min}$ is the location of the best point (assuming that the object is to minimise a function) found so far in the range $[a, b]$ during the search, and σ is a measure of the ants' clustering (density) around the best point.

Initially at the start of the algorithm's execution, $x_{\min}$ is randomly chosen from the range $[a, b]$ and σ is set to $3(b - a)$ in order to ensure that the real-valued strings corresponding to each ant are well distributed in the range $[a, b]$. When the solution vector is made up of multiple elements (real values), each element will have its own normal PDF and the values of $x_{\min}$ and σ for each of these individual PDFs will alter during the optimisation run.

Applying the CACS Algorithm

Suppose a problem of interest requires that optimal values for three real-valued parameters are to be uncovered and that a range in which the optimal values lie is known for each of the parameters. Initial values for $x_{\min}$ and σ are selected for each of the three parameters as above. Each ant in turn then completes a 'tour' by generating three random numbers and using each of these in turn to generate a sample value x using the PDFs corresponding to

Algorithm 9.2: Continuous Ant Foraging Algorithm

Select value of $x_{\min}$ randomly from the range of allowable values for each element $(1, \ldots, n)$ of the real-valued solution encoding;
Set σ for each element $(1, \ldots, n)$ of the solution encoding;
repeat
  for *each ant k in turn* **do**
    Construct a solution $(x_1, \ldots, x_n)$ by making a random draw from $\mathrm{PDF}_1, \ldots, \mathrm{PDF}_n$;
    Measure quality of solution;
  end
  Update the $x_{\min}$ values for each PDF if a new best-so-far solution has been found;
  Update the σ values for each PDF;
until *terminating condition*;

each locus of the solution vector. To do this, each PDF is first converted into its corresponding Cumulative Distribution Function (CDF):

$$\mathrm{CDF}_j(x) = \int_a^x \mathrm{PDF}_j(t)\mathrm{d}t \tag{9.17}$$

where b and a are the upper and lower limits of the probability distribution. By generating a random number r from (0,1), the CDF can be used to obtain a value x, where $x = \mathrm{CDF}^{-1}(r)$.

After all ants have completed a tour, the quality of each ant's solution is found using the objective function for the problem. If any of these solutions is better than that found by the best-so-far ant, the value of $x_{\min}$ for each PDF is updated using the values from the best-so-far ant. The value of σ for each PDF is also updated:

$$\sigma^2 = \frac{\sum_{j=1}^{k} \frac{1}{f_i - f_{\min}} (x_j - x_{\min})^2}{\sum_{j=1}^{k} \frac{1}{f_i - f_{\min}}} \tag{9.18}$$

where k is the number of ants and $f_j / f_{\min}$ are the objective function values for each ant j, and the best-so-far ant respectively. As the solutions encoded by each ant converge, the value of σ^2 reduces.

Another way of updating the value of σ for each PDF is to calculate the standard deviation of the values of that parameter across the population of ants during the last iteration of the algorithm [643]. As good solutions are uncovered, a positive reinforcement cycle builds up. More ants are attracted to the best solution string and as the population of ants converges to similar sets of real values, the value of σ automatically decreases, leading to intensification of search in the region of the best solution found so far.

The algorithm iterates until predefined termination criteria are reached, such as a maximum number of iterations, a time limit, or after a set number of iterations without any improvement in the solution.

In order to discourage premature convergence of the population of ants to a single solution vector, an explicit pheromone evaporation mechanism can be applied. Evaporation can be simulated by increasing the value of σ for each PDF, thereby increasing the 'spread' of pheromone in the environment. One method for achieving this suggested by Viana et al. [643] is to apply the following rule to each PDF's σ the end of each iteration:

$$\sigma_{t+1} = \alpha\sigma_t \tag{9.19}$$

where $\alpha > 1$ is the evaporation rate. As the value of α is increased, the level of evaporation also increases.

CACS Algorithm and EDAs

The CACS bears strong similarities with estimation of distribution algorithms (Sect. 4.7) in that both build a probability distribution model of promising solutions based on feedback to earlier solutions (see also the real-valued quantum evolutionary algorithm in Sect. 24.4) [509]. CACS, like univariate EDAs, does not explicitly consider interactions amongst the individual variables.

A good discussion of the links between real-valued ant colony optimisation and EDAs is provided by Socha and Dorigo [583]. This paper also introduces a new algorithm for ant colony optimisation in continuous domains which uses a Gaussian kernel PDF. One advantage of using this method of representing the distribution of a variable is that it allows the description of a case where two (or more) regions of the search space are promising. In contrast, a drawback of using a single Gaussian function to represent a variable's distribution is that the function is unimodal.

A range of other approaches to the design of continuous-valued ant colony optimisation algorithms exist, and the reader is referred to [338] for a short review of some of these.

9.5 Multiple Ant Colonies

As already noted, canonical discrete ant algorithms will tend to converge on a single solution due to the nature of the pheromone reinforcement process. If the environment is multimodal or dynamic this can lead to a suboptimal outcome. One way of promoting solution diversity is by implementing multiple ant colonies [318, 603]. This approach maintains diversity at the level of the colony rather than at the level of individual ants.

In implementing multiple, or parallel, ant colonies, one extreme would be to allow each colony to search independently of others, with each colony maintaining its own pheromone matrix. Alternatively, the ants from each colony could be allowed to interact with each other.

Parallel Implementation

ACO algorithms can be parallelised in two broad ways. *Fine-grained* parallelisation occurs where a few ants are assigned to each subpopulation with frequent exchanges of information between each island. In contrast, *coarse-grained* parallelisation occurs where each island has a larger subpopulation and exchanges of information between islands is infrequent. Fine-grained schemes can impose significant communication overhead and consequently course-grained schemes are more commonly used.

Implementations of multiple ant colonies are amendable to parallelisation as each individual colony can be assigned to a single processor, with a master processor controlling the periodic transfer of information between individual colonies. This permits each colony to engage in independent search whilst allowing the periodic dissemination of good solutions found by other colonies. It also permits the use of different parameter settings in each colony. The key decision choices when using parallel ant colonies are:

i. How often do migration events take place?
ii. How is the migration event structured?

Migration Frequency

There is no simple way to decide how often migration events should occur. If they are very frequent they will tend to produce rapid convergence of the entire population of the colonies to a narrow region of the search space. Conversely, if migration events are very infrequent, knowledge of already discovered good solutions will only slowly disseminate across the colonies.

Migration Structure

Although it is possible to transfer complete pheromone matrices between colonies (for example, where colony A has found a better solution than colony B), a simpler, and an often better, approach is to just transfer information on the best-so-far tour between colonies [410]. In this case, the original pheromone matrices of each colony are left unchanged but subsequent pheromone updates (assuming an elitist strategy is applied where some pheromone is always deposited on the best-so-far solution) are influenced by the transferred information on the best-so-far tour.

In addition to passing information on global and local best-so-far solutions, individual ants can be migrated between colonies. For example, in a migration event, colony A could compare its best n_{best} ants with the n_{best} ants of colony B (A's successor colony on the directed ring), with the n_{best} of these $2n_{\text{best}}$ ants being used to update the pheromone matrix of colony B.

A variety of migration topologies could be employed between the colonies including the fully connected model, where the global best solution found

across all the colonies is periodically broadcast to every colony. A variant on this strategy is to establish a virtual neighbourhood between colonies so that they form a directed ring. In each migration event, a colony sends its best-so-far solution to the next colony in the ring, and that colony's best-so-far solution is updated.

Colony Birth and Death

If a multiple colony approach is adopted there is no requirement that each colony exists for the entire algorithm. One observed phenomenon in nature is that ant colonies can sometimes split due to the birth of a new queen or because of overpopulation of the colony. This fission mechanism bears similarity with the idea of cloning, whereby a copy of the original colony is made, altering some parameters associated with the clone to ensure it is not an identical copy. It is also observed in nature that colonies in resource-poor environments can become extinct. This idea could be used in a multicolony optimisation system to cull poorly performing colonies, or to cull a colony which is too similar to another colony. Hara et al. [247] proposed a multicolony system which embedded mechanisms of colony fission and extinction in order to promote diversity in the search process.

9.6 Hybrid Ant Foraging Algorithms

A common serial hybridisation of ant algorithms is to include a local search phase. Although ant algorithms can hone in on promising solutions, their efficiency can usually be significantly enhanced by adding a local search step. This aims to locally optimise the solutions that the ants have constructed by examining a set of 'neighbouring' solutions and picking the best of these if they are better than the original solution.

Typically, the local search will be undertaken after the population of ants has constructed a set of tours but before the pheromone deposit and evaporation process. In the pheromone update step, pheromone is laid on the arcs of the locally optimised tours, rather than on the arcs of those tours before the local search step was performed.

Although the idea of adding a local search stage is straightforward, there are practical issues in determining how locality and neighbourhoods can be defined for a specific problem. For example, suppose we are considering the TSP. One possible definition of neighbours around a current tour is a k-exchange (or k-opt) neighbourhood, where this is defined as being all the tours which could be constructed from the current tour by exchanging k arcs. For example, to carry out a 2-exchange (or 2-opt move) we delete from the tour two arcs (that are not incident with the same node), giving four vertices of degree 1; we then adjoin two new arcs incident with the degree 1 vertices

to reconnect the parts into a new tour.[1] Figure 9.5 illustrates one member of the set of two-exchange neighbours that could be created from an initial tour.

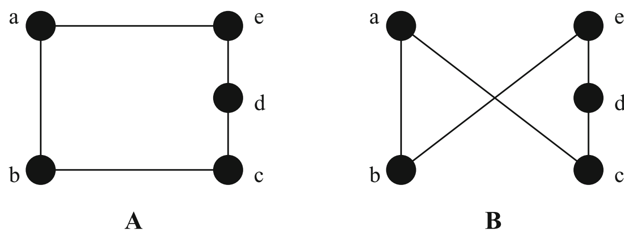


Fig. 9.5. Tour B is obtained by removing arcs (a,e) and (b,c) from tour A, and replacing them with arcs (a,c) and (b,e). Hence, tour B is a two-exchange neighbour of tour A

A parallel hybridisation approach uses multiple search strategies in parallel. Each optimisation technique runs independently, but periodically, information is migrated between each search algorithm. For example, both a GA and ACO could be separately executed on a combinatorial problem of interest. Periodically, the best ACO solution could be migrated to replace one of the GA's solutions if it had higher fitness than the solution being replaced. Conversely, good solutions found by the GA could be allowed to participate in the pheromone deposit process, embedding information uncovered by the GA in the pheromone matrix of the ant algorithm.

A third possibility in developing hybrid ant algorithms is to use another search process to uncover good parameter settings such as α, β and p for the ant algorithm.

9.7 Ant-Inspired Clustering Algorithms

Cluster analysis involves the grouping of similar objects into the same clusters and dissimilar objects into different clusters based on the values of attributes of the objects (see Chap. 14 for a more extensive discussion of clustering). Several behaviours of ants have been used as metaphorical inspiration for the design of clustering algorithms, including the *brood sorting* behaviour of the ant species *Leptothorax unifasciatus*, where ant larvae are sorted by size and clustered at the centre of the brood area in the colony. Another important set of behaviours concern hygiene tasks related to waste disposal as social insects

---

[1] In the case of 2-opt, there is only one way to do this that gives a valid tour of all nodes, rather than two disconnected subtours; for 3-opt and higher, there will be multiple ways to generate a valid new tour.

living in high density colonies (particularly when they are closely genetically-related) are sensitive to pathogen attack [159]. Instances of these behaviours include the spatial segregation of corpses, waste, and diseased individuals. Surprisingly large numbers of social insects can be engaged in these tasks, with some 20% of honey bees and in excess of 30% of ants of the species *Acromyrmex versicolor* being engaged in colony or nest hygiene duties. One example of this is the *cemetery formation* behaviour of the ant species *Lasius niger*, where dead ants are collected from the colony and deposited together. Although these behaviours are not fully understood, a number of models have been created in an attempt to better understand them.

Two clustering models inspired by ant behaviours have been widely examined, the model of Deneubourg et al. [152] and the model of Lumer and Faieta [382]. The idea behind these models is that objects should be picked up if they are not already beside similar objects. They should then be relocated and dropped beside other items of the same type.

9.7.1 Deneubourg Model

Based on their observations of ant corpse clustering and brood sorting phenomena, Deneubourg et al. [152] derived a simple pick-drop model which they believed captured the essence of the ants' clustering behaviour.

In the model, ants traverse an $x \times y$ 2-D grid, randomly moving from their current site to one of four neighbouring sites (up-down-left-right) at each iteration of the algorithm. If an unladen ant encounters an object (for example, a dead ant), it picks it up with probability P_{pick}, and in subsequent iterations may drop the corpse with probability P_{drop}. Assuming there is only one type of object (all other items are classed as dissimilar to this item) in the environment, the pick and drop probabilities can be defined as:

$$P_{\text{pick}} = \left(\frac{k_1}{k_1 + f}\right)^2 \tag{9.20}$$

$$P_{\text{drop}} = \left(\frac{f}{k_2 + f}\right)^2 \tag{9.21}$$

where f is the perceived fraction of all the objects in the neighbourhood of the ant (providing an estimate of the local density of dead ants or equivalently an estimate of the size of the local cluster), and k_1 is a threshold constant.

When an ant encounters a corpse, and $f \ll k_1$, the ant is not considered to be in the vicinity of a large cluster, and therefore should pick up the corpse in order to drop it on a cluster somewhere else (therefore, P_{pick} should be close to 1). Conversely, if an ant encounters a corpse and $f \gg k_1$, the ant is close to a large existing cluster and should not move the corpse as it is already in a large cluster of corpses (therefore P_{pick} should be close to 0).

The probability P_{drop} that a randomly moving loaded ant drops an object is governed by a second threshold constant k_2. When $f \ll k_2$, the ant carrying

a corpse is not close to a cluster of other corpses, and therefore should continue to carry the corpse until a cluster is found (P_{drop} is close to 0).

The value of f is an important parameter in the algorithm as it directly impacts on the probability of both picking up and depositing a corpse. In the Deneubourg et al. algorithm, the value of f is calculated by each ant, based on its personal history. It is assumed that each ant has a T period memory. If we assume that an ant can only encounter zero or one object per time unit, and letting N represent the total number of objects encountered during T time periods, f is calculated as $= N/T$.

The algorithm leads to ant behaviour such that small clusters of dead ants (perhaps of size 1) are emptied, and large clusters grow. In turn, the large clusters will tend to merge. Extending this algorithm to cases where there is more than one object type, f is replaced by a series of f values, each representing the fraction of a type of object encountered during the last T time units [68].

9.7.2 Lumer and Faieta Model

The Deneubourg model was generalised by Lumer and Faieta [382] to encompass objects of more than one type through the inclusion of a distance or *dissimilarity* measure. This resulted in a model which was capable of being applied to complex real-world data clustering problems. Algorithm 9.3 provides the pseudocode for the Lumer and Faieta algorithm.

The algorithm acts by mapping the n-dimensional data objects onto a 2-D grid. The ants traverse this 2-D grid and engage in pick-drop behaviour so as to cluster like items together. Let $d(o_i, o_j)$ be the distance between two objects o_i and o_j in the space of object attributes. Assume that an ant is located at site r on a 2-D grid at time t, and it finds an object o_i at that site. The *local density* $f(o_i)$ with respect to object o_i at site r is given by:

$$f(o_i) = \max\left\{0, \frac{1}{s^2} \sum_{o_j \in \text{Nbd}_{(s\times s)}(r)} \left[1 - \frac{d(o_i, o_j)}{\alpha}\right]\right\} \tag{9.22}$$

where $f(o_i)$ (similar to f in the model of Deneubourg et al.) is a measure of the average similarity of o_i with other objects which are present in its neighbourhood $\text{Nbd}_{(s\times s)}(r)$, defined as the $s \times s$ positions on the 2-D grid around the grid location r of o_i which the ant can 'see'. Therefore, in comparison with the Deneubourg algorithm, which uses a memory to calculate f, the Lumer and Faieta algorithm allows each ant to have a direct perception of the area surrounding its current location. α is a *tuning knob* for the degree of dissimilarity discrimination between objects. If α is large, even quite dissimilar items may be clustered together; if it is small, distances between vectors in the attribute space are amplified, and even similar vectors may end up in different clusters.

Algorithm 9.3: Lumer and Faieta Clustering Algorithm

```
Randomly distribute the data vectors (O_i) on the grid;
Locate the ants randomly on the grid;
repeat
    for each ant i do
        if ant is not loaded and the ant's location is occupied by data vector
        O_i then
            Compute f(O_i) and P_pick(O_i);
            Draw random real number R between 0 and 1;
            if R ≤ P_pick(O_i) then
                Pick up data vector O_i;
            end
        else
            if ant carrying data vector O_i and site is empty then
                Compute f(O_i) and P_drop(O_i);
                Draw random real number R between 0 and 1;
                if R ≤ P_drop(O_i) then
                    Drop data vector O_i;
                end
            end
        end
        Move to randomly selected neighbouring site not occupied by other
        ant;
    end
until terminating condition;
```

Taking two extreme cases to demonstrate the calculation of local density, if all the sites around o_i are occupied by objects which are similar to it then $f(o_i)$=1 and o_i should be picked up with a low probability. If all sites around o_i are occupied by objects which are very dissimilar to it then $f(o_i)$ is small and o_i should be picked up with a high probability. Under the Lumer and Faieta model, the pick and drop probabilities of the Deneubourg et al. model are altered to:

$$P_{\text{pick}}(o_i) = \left(\frac{k_1}{k_1 + f(o_i)}\right)^2 \tag{9.23}$$

$$P_{\text{drop}}(o_i) = 2f(o_i), \text{ if } f(o_i) < k_2 \tag{9.24}$$

$$P_{\text{drop}}(o_i) = 1, \text{ if } f(o_i) \geq k_2 \tag{9.25}$$

In an application of the Lumer and Faieta methodology, each item i is defined by a vector of data, $\text{Data}_i = (r_1, \ldots, r_n)$, where n is the number of elements of the vector.

Each item i is symbolised by an object o_i on the 2-D grid. Initially, these objects are randomly scattered over the 2-D grid, and during the execution of the algorithm they are clustered into heaps of similar items. The distance

between two objects is calculated by the Euclidean distance between the two items' data vectors in $\mathbb{R}^n$. There is no direct link between the position of an object on the 2-D plane and its vector in $\mathbb{R}^n$.

At the start of the algorithm, a fixed number of ants are placed on the 2-D grid. During each iteration of the algorithm, an ant may either be carrying an object or not. In the first case, the ant may:

- drop the object on a neighbouring empty location,
- drop the object on a neighbouring object, if both are similar, or
- drop the object on a neighbouring heap, if the object is similar to other members of the heap (a heap arises when there are multiple objects on a single grid location).

If the ant is not already carrying an object, it may:

- pick up a single object from a neighbouring location, or
- pick up the most dissimilar object from a heap on a neighbouring location.

The algorithm acts to construct clusters, such that the distances between objects of the same cluster are small in comparison with the distances between objects in different clusters. As the algorithm runs, and clusters start to form, the probability of objects being picked up diminishes and $\lim_{t\to\infty} P_{\text{pick}} = 0$, as similar objects are grouped together. When applying the algorithm, the modeller must define the grid size and the number of ants. If the grid is excessively large, clustering will be slow and many small clusters are likely to result. The number of ants must be less than the number of data vectors. If too many ants are used, they may carry the data vectors around for long periods of time as the density of deposited objects will be low. Conversely, if too few ants are used, clustering may be slow. An improved version of the Lumer and Faieta algorithm is described in Handl et al. [243].

Classification Using Ant Clustering Algorithms

Like self-organising maps (SOMs) (Chap. 14.1), ant clustering is unsupervised in that it does not make use of a priori group memberships during the training process. Therefore, once the training process is complete and a number of clusters have been created by the algorithm, the modeller must assign a class label to each cluster if the cluster results are to be used for classification purposes (Sect. 14.4).

A simple scheme is to initially assign the known labels to each training item after clustering has taken place. Then items in each cluster are relabelled based on a majority voting scheme. By comparing these assigned classification labels with the known classification labels for the training data, the in-sample accuracies for the training data used to create the clusters can be obtained.

The out-of-sample data can then be classified by determining which cluster each out-of-sample item is closest to. For example, Fig. 9.6 illustrates an out-of-sample item being labelled as Class 1 as the nearest labelled item to it

(based on Euclidean distance) is from Class 1. However, the nearest neighbour approach can break down if there is noisy or errorful training data. In Fig. 9.7, the unclassified item is labelled as Class 2, due to an outlier training item which has been incorrectly assigned a Class 2 label.

A more robust method of labelling out-of-sample items is to use a *k nearest neighbour* approach where the labelling of the item depends on the predominant label tag amongst its 'k' nearest neighbours. Figure 9.8, illustrates the classification of the out-of-sample item based on its six nearest neighbours.

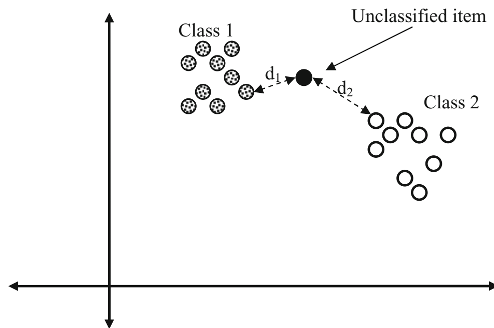


Fig. 9.6. Two class case with one out-of-sample item being classified as Class 1 based on the nearest labeled item

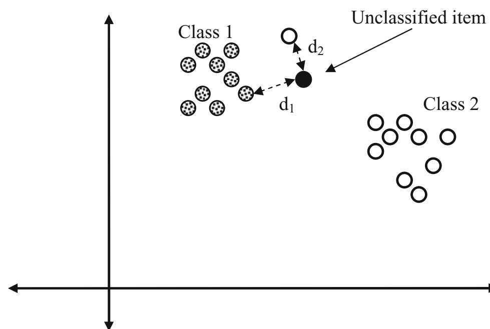


Fig. 9.7. Out-of-sample item being classified incorrectly

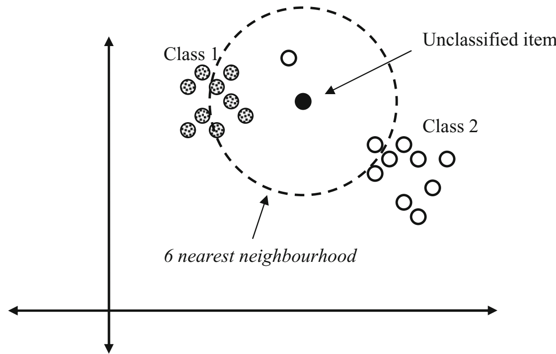


Fig. 9.8. Out-of-sample item being classified based on a $k = 6$ neighbourhood. As four of these neighbours are Class 1, the out-of-sample item is designated as Class 1

The efficiency of the resulting classification and the number of clusters which are identified in the data depend on the choices of the parameters for the algorithm which the user selects. For example, choosing a large grid size will tend to increase the run-time of the algorithm as the ants spend much time traversing empty grid positions. A large grid size will also tend to produce a greater number of clusters, particularly in the earlier stages of the algorithm. Using too few ants results in a very slow clustering process.

9.7.3 Critiquing Ant Clustering

Although the activities of ants in creating cemeteries and in brood sorting have inspired a series of clustering and classification algorithms, the efficiency of these algorithms has come under attack in recent times. Martin et al. [391], in an examination of the Deneubourg model, found that similar clustering behaviour could be produced by a simple model where individual ants had no memory and therefore no ability to calculate the local density of dead ants. The study's results also suggested that there was no collective effect in the Deneubourg model as an individual ant working alone could also create the cemetery, albeit more slowly. Tan et al. [614], based on an examination of other ant inspired clustering models such as Lumer and Faieta, Ant Q and ATTA, suggest that although the algorithms can produce clustering effects, the results do not stem from true collective intelligence in the algorithms. Hence, it remains an open question as to whether efficient, effective, ant-clustering algorithms which embed true collective intelligence can be developed. Interested readers are also referred to [243], which demonstrates that although the ATTA model can produce good clustering solutions, these are only weakly topology-preserving.

9.8 Classification with Ant Algorithms

Although a classification system can be developed using an ant clustering model, a more direct method which follows from ACO is to construct a classification rule by having ants walk across a graph of rule choice fragments. The set of arcs traversed by the ant corresponds therefore to a complete classification rule. The best known example of this constructivist approach is *AntMiner*, developed by Parpinelli, Lopes and Freitas [494]. In AntMiner the object is to uncover a series of rules that explain the training dataset as well as possible. The rules have the following format:

$$\text{IF} < \textit{condition} > \text{THEN} < \textit{Class} >$$

where the condition can contain multiple subparts which are linked using the AND operator. Hence, a classification rule will take the form IF $condition_1$ AND $condition_2$ ... THEN $Class = 1$.

In the canonical version of AntMiner each of the included conditions are constrained to be equalities, each data attribute (explanatory variable) is constrained to only appear once in the classification rule, and the possible values for each data attribute are constrained to be discrete sets. Attributes which take continuous values require preprocessing to be discretised.

Therefore, every attribute in the dataset corresponds to a node on a construction graph and each arc corresponds to a specific state of that attribute. For example, suppose one attribute in the dataset is gender. There will be three arcs corresponding to choices for this attribute which could appear in a classification rule, namely, *Gender=Male*, *Gender=Female* and, to allow a classification rule which was not gender specific, *Gender=Any*. Figure 9.9 illustrates a sample construction graph.

The entire (finite) set of possibilities for a classification rule can be described as a directed, symmetric graph. This formulation allows the classification problem to be recast into a standard ant colony optimisation framework as it produces a combinatorial optimisation problem. Algorithm 9.4 provides an overview of the AntMiner algorithm.

AntMiner Algorithm

In contrast to some classification methodologies, AntMiner develops a *set* of classification rules which explain, or *cover*, most or all of the training data. During each iteration of the **while** loop in Algorithm 9.4, an additional classification rule is uncovered and is added to the set of already discovered classification rules. The data which is explained by this rule is then removed from the training dataset and the pheromone levels on all arcs are returned to their initial values. The process is repeated on the remaining data until either all (or a user-determined portion based on the value chosen for *Max uncovered cases*) of the data is covered by the developed set of classification rules.

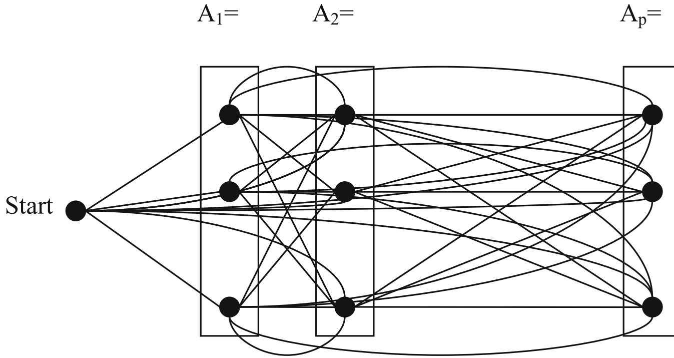


Fig. 9.9. The ant commences at the start node and walks across the (bidirectional) arcs. Each arc corresponds to a choice of value for a specific data attribute $(A_1, \ldots, A_p)$. All nodes are connected and the series of arcs traversed by the ant corresponds to a classification rule

During each iteration of the **repeat** loop three key steps are undertaken. Each ant in turn constructs a rule, the rule is pruned and the pheromone trails on the arcs are updated. At the end of the loop the best of the rules uncovered by the t ants (R_{best}) is added to the set of discovered rules.

Each ant starts the construction process with an empty rule. As an ant walks from one node to the next the classification rule builds up, one term at a time. In selecting an outgoing arc at a node an ant is guided both by the quantity of pheromone on each arc (τ_{ij}) and by a problem-dependent heuristic (η_{ij}) which estimates the quality of arc (i, j). The probability that an ant selects the arc between nodes (i, j) is given by:

$$P_{ij}(t) = \frac{\tau_{ij}(t) \cdot \eta_{ij}}{\sum_{k=1}^{n} x_k \sum_{l=1}^{p_k} (\tau_{kl}(t) \cdot \eta_{kl})} \tag{9.26}$$

where n is the number of attributes in the dataset, p_i is the number of possible values for attribute A_i, and x_i is a binary variable which is set to 1 if attribute A_i is not already included in the rule being constructed by ant_t.

The ant continues to add terms to its growing classification rule until all the data attributes are included in its rule, or until the addition of a term which would make the ant's rule cover fewer than a user-specified number of cases in the dataset (this constraint reduces the chance of overfit). The majority class of all training data covered by the rule is assigned to it.

Next the rule is pruned in order to simplify it and in order to further reduce the chance of overfit. Finally, pheromone is deposited on the arcs chosen by that ant with evaporation occurring on the other arcs. The process is then iterated with the next ant being released to traverse the graph.

Algorithm 9.4: AntMiner Classification Algorithm

```
Let training set = set of all training data;
Initialise Discovered Rule List (initially empty);
while training set size > max uncovered cases do
    Let t = 1 (ant index);
    Let j = 1 (convergence test);
    Initialise pheromone levels on arcs;
    repeat
        Ant_t starts with an empty rule and traverses graph from source to
        sink, incrementally building a rule R_t one term at a time;
        The majority class of all training data covered by the rule is assigned
        to it;
        Prune rule R_t;
        Increase the pheromone along arcs followed by Ant_t (in proportion to
        quality of rule) and decrease pheromone on other arcs;
        if R_t = R_{t-1} then
            Let j = j + 1 (consecutive ants are creating the same rule);
        else
            Let j = 1;
        end
        Let t = t + 1;
    until t ≥ number of ants or j ≥ threshold value;
    Select best rule (R_best) from all rules R_t;
    Add rule R_best to Discovered Rule List;
    Let Training set = Training set \ set of cases correctly covered by R_best;
end
Evaluate performance of extracted rules on out-of-sample data;
Rules are applied in their discovered order and the first rule that covers the
out-of-sample case determines its classification;
Any unclassified out-of-sample data vectors are assigned to a default class;
```

Several studies have extended the original AntMiner model, including [374, 375] (*AntMiner 2 and 3*) and [390] (*AntMiner+*).

9.9 Evolving an Ant Algorithm

A critical component of all discrete ant algorithms is how learnt information is encoded on the edges of the construction graph, in other words how the pheromone trails are updated. Earlier, in Sect. 8.7, an application of genetic programming (GP) to evolve the velocity update equation for a PSO algorithm was illustrated [512, 513]. Of course, this approach has more general application and Tavares and Pereira [617] describe how GP can be used to evolve pheromone trail update strategies. In this study, GP is used to evolve

individuals that encode a trail update strategy, with the fitness of these being tested in an Ant System framework.

9.10 Summary

The communication mechanisms of social insects provide a rich milieu for the development of natural computing algorithms. A key feature of the communication mechanisms is that they enable the societies or swarms of insects to engage in complex, decentralised problem-solving. This facility to solve complex problems is particularly notable given the relative simplicity of the information-processing capabilities of individual insects.

10

Other Foraging Algorithms

Ants are not the only species of insect that use social communication to gather and process information from their environment in order to shape their behaviour. In this chapter we consider a range of mechanisms drawn from the behaviour of a variety of insects, including honeybees, glow worms and locusts, and see how these can stylistically inspire the design of optimisation algorithms.

10.1 Honeybee Dance Language

Honeybees are one of the most studied branches of the insect family. Just as in the case of certain species of ants, their ability to self-organise in complex ways has long attracted the attention of researchers who have examined the question of communication in honeybee societies. Some species of honeybee exhibit a symbolic system of communication based on the performance of a *dance* to transmit information on (amongst other things) the location and quality of resources around the vicinity of the hive. This *dance language* has been extensively studied by ethologists, most notably by Nobel Laureate Karl von Frisch and also by Martin Lindauer and Thomas Seeley. It has been suggested that the dance language of bees "... is on a higher level than the means of communication amongst birds and mammals with the exception of man" [645, p. 540].

Stemming from the ability of individual honeybees to communicate information, a bee colony possesses sophisticated information-gathering and information-processing capabilities. Like ant colonies, decision making is decentralised and parallel, and patterns of bee colony behaviour are self-organising. However there are notable distinctions between the information-sharing mechanisms of ants and those of bees. In the ant foraging models described in Chap. 9.3, communication between ants is primarily indirect and is based on stigmergy. In contrast, the honeybee dance language enables bees to engage in direct communication whereby information is communicated to

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_10

peers symbolically by means of a dance [654]. Direct communication mechanisms offer advantages in the real world as they permit quick reaction to changing environmental conditions. In the following sections we concentrate on three behaviours of honeybees and illustrate how they can provide inspiration for the design of computational algorithms. As will be seen, the first two of these rely heavily on honeybee communication via the dance language.

i. Bee foraging
ii. Nest site selection
iii. Bee mating behaviour

10.2 Honeybee Foraging

Foraging activities of bees involve searching for exploitable resources such as pollen (a source of protein), water, waxy materials for hive building, or nectar from flowers. Nectar is a source of carbohydrate and is converted by bees into honey. It is their most important food resource. A honeybee colony can monitor a large region around a hive for potential food sources (up to several kilometres) and can quickly reallocate its foragers to collect food from new sources which emerge [563]. A particular feature of bee foraging is that the supply of nectar is highly dynamic. The availability and quality of nectar varies with local weather conditions, the foraging activities of the bees themselves (as they exhaust resources), and the blooming cycle of local flora. Seeley notes that the quantity of nectar harvested by a colony of bees can vary by a factor of 100 from one day to the next [563]. Hence, a colony faces a fast-changing allocation problem, whereby foragers need to be dynamically allocated to different food sources so that food intake is maximised.

10.2.1 The Honeybee Recruitment Dance

Most work in a hive is undertaken by the female honeybees. Female bees have four main roles during their lives: cleaner, nurse, food-storer and forager; and they progress through these roles as they age. Forager bees are broadly split between scouts who discover resources around the hive and foragers who transport material from an already-discovered site back to the hive. Typically, scouts account for about 10% of the forager bee population [563].

When a scout or explorer bee discovers a food source of sufficient quality it may undertake a dance on its return to the hive once it has unloaded its nectar into empty honeycomb cells. The objective of the dance is to recruit other foragers who will travel to the food source and exploit it. In turn, the foraging bees may also undertake a dance when they return to the hive if the food resource is of sufficient quality.

The dance language consists of repetitive patterned movements that are intended to communicate information about the exact location and desirability

of the food source. In essence, the dance can be considered as a reenactment of the flight from the hive to the location of interest. The dance is undertaken in a specific location of the hive near the entrance called the *dance floor*. The dance floor consists of a vertical comb in the hive and typically this area of the hive contains multiple potential foraging recruits. The dance is social in that it is never undertaken without an audience [120].

The nature of the dance movements depends on the location of the food source relative to the hive. If the food source is close by (up to about 100 metres from the hive), the bees undertake *round dances* with circular movements predominating. If the food source is further away a *waggle dance* resembling a figure eight is undertaken. The direction to the resource (relative to the sun) is indicated by the angling of the bee's body during the dance. The desirability of the location is communicated by the dance's *liveliness* or *enthusiasm*, with more desirable locations corresponding to livelier dances [564]. The duration of the waggle portion of the dance is a proxy for the distance to the location of food sources.

At any point in time there may be several bees dancing on the dance floor; hence the hive can simultaneously harvest several food sources. This permits quick adaptation by the hive in the event that a particular food resource becomes exhausted and therefore needs to be abandoned. Recruited foragers tend to travel to richer food sources in greater numbers as dances for high-quality food sources tend to be more conspicuous and longer, thereby creating a positive feedback loop and amplification of the exploitation of those food sources. It is also interesting that the dance contains information which allows the relative merits of different food patches to be assessed in terms of both their quality and the energy needed to harvest them, the latter being proxied by the distance the food patch is from the hive. Other forms of dance communication also exist. For example, if it takes a foraging bee too long to unload its nectar, it may start a tremble dance which is designed to recruit storer bees to assist with the unloading process. The above description of foraging behaviour is stylised and omits aspects of the recruitment process such as the role of odours and sounds. Readers requiring details on these aspects are referred to [563, 619, 644].

10.3 Designing a Honeybee Foraging Optimisation Algorithm

Changes in weather, the exhaustion of current food sources and the emergence of new ones ensure that food foraging is a continuous and a dynamic task. A variety of related optimisation algorithms have been designed in recent years which are drawn from a honeybee foraging metaphor, including (as a small sample) [110, 432, 508, 672]. Broadly speaking, a honeybee foraging algorithm can be decomposed into three iterated activities:

i. explore the environment for good food locations

ii. recruit foragers to the better of the discovered locations
iii. harvest these locations

Typically a random search element is also added to the algorithm in order to prevent premature convergence of the search process. A general algorithmic framework that encompasses these steps and applies them for optimisation purposes is outlined in Algorithm 10.1. This basic framework could be operationalised in a wide variety of ways.

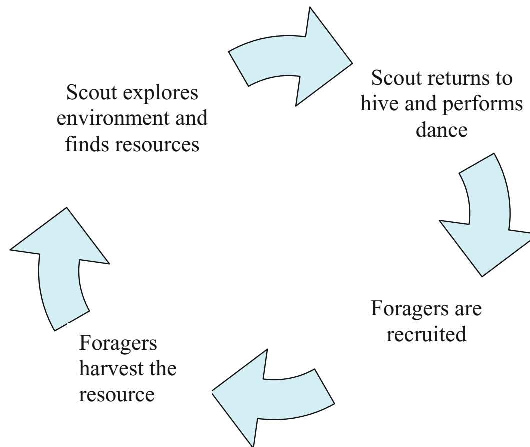


Fig. 10.1. Stylised representation of honeybee foraging behaviour

10.3.1 Bee System Algorithm

One of the earliest papers drawing inspiration from the behaviour of honeybees in order to contribute to the design of a computational algorithm was that of Sato and Hagiwara (1997) [553], which developed an algorithm called *Bee System*. The algorithm applied a two-stage global/local populational search process, with the global search phase drawing heavily on a GA framework.

Initially in the algorithm, a global search step is undertaken using a simple GA. When the best solution found by the GA remains unchanged for x generations, the solution is termed a *superior chromosome* and it is written to memory. At this point, the entire GA search process is restarted from the beginning (with a new random population) and begins searching again. The process iterates until a total of n solutions are stored in the memory.

Then, a 'local' search step is undertaken around each of the n solutions. In this step, a new population of p members is randomly created around each of the n individuals (i.e. n new populations are created). For each of

Algorithm 10.1: Canonical Honeybee Foraging Optimisation Algorithm

Randomly locate n foraging bees in the search space;
Evaluate the fitness of each of these locations;
Store location of best solution;

repeat
    Select the m best locations (patches) for neighbourhood search ($m \leq n$);
    Recruit foragers to search in the vicinity of these locations (number of foragers assigned to each location is proportional to the fitness of the location) and evaluate the fitness for each forager;
    Select the fittest bee for each patch;
    Assign the remaining bees to search randomly and evaluate their fitness;
    Update location of best solution if necessary;
until *terminating condition*;

the populations, all of the members of that population are crossed over with their associated superior chromosome, thereby generating p variants on their superior chromosome, simulating search around it. The resulting p child chromosomes are evaluated. Following this, a standard GA process is applied to the population of child chromosomes using regular crossover. Periodically, a migration step is applied whereby one individual is migrated from a population to its neighbouring population in a ring migration topology (Sect. 4.2). As the search process proceeds, the location of the best solution found thus far is updated as it changes.

From the above description we can see that although the algorithm is a GA variant, it also bears a loose correspondence with the process of honeybee foraging. The initial global search process generates n good candidate solutions, akin to scout bees uncovering a variety of good resource locations in the environment. Following a 'recruitment' process (here all n locations recruit the same number of foragers, p) a second local search process takes place around these locations. This corresponds to the process by which foragers can stochastically locate an even better resource location in the vicinity of the location to which they were originally recruited.

One important feature of real-world honeybee foraging which is omitted from *Bee System* is that of fitness-differential recruitment of foragers. Several subsequent algorithms including [456, 457] and [508] implemented this step, as does the artificial bee colony algorithm [314], which is described next.

10.3.2 Artificial Bee Colony Algorithm

The *artificial bee colony algorithm* (ABC) was proposed by Karaboga in 2005 [314]. In this algorithm the population of bees is divided into employed bees (those who are currently exploiting an already discovered food resource) and

unemployed bees (those who are looking for a food resource). The latter group can be further decomposed into scout bees and onlookers, where scout bees search around the nest for new food sources, and onlooker bees wait at the nest for recruitment to food resources. As before, a specific location in the search space represents a solution to the optimisation problem of interest.

The pseudocode for the artificial bee colony algorithm is outlined in Algorithm 10.2. The search process is undertaken by a population of $2 \times S_N$ artificial bees, where S_N of these are termed employed bees and the remainder are onlooker bees. Initially, each of the S_N employed bees is located on a randomly selected location in the search space (simulating the process whereby these initial food resources have been found by scout bees from the hive), corresponding to the starting food locations from which the search process develops. Each of these locations (or potential solutions) is a D-dimensional vector. The quality of each of these locations is then assessed using (10.2) where fit_i is the fitness of the ith location ($i \in \{1, \ldots, S_N\}$). The effect of (10.2) is to scale the fitness into the range $[0, 1]$ and it is assumed that all raw fitness values are positive ($f_i \in [0, \infty)$). Next, the search process commences.

Initially, each of the S_N employed bees seeks to locate a better food source in its vicinity, simulating the process of a bee using visual cues in order to uncover even better sources of nectar. In this process, assuming that the bee is initially located at $x_{i,j}$, it then takes a randomly generated 'step' from this location to a new location $v_{i,j}$. The taking of this step could be operationalised in a variety of ways. For example, in [317] the process is governed by (10.1) where for $i \in \{1, \ldots, S_N\}$, j is a randomly generated integer in the range $1, \ldots, D$ where D is the number of dimensions, $\phi_{i,j}$ is a randomly generated number $\in [-1, 1]$, and $k \in \{1, \ldots, S_N\}$ is the index of a randomly chosen solution ($k \neq i$). Hence, in essence, the new solution is obtained by 'mutating' the current solution using a stochastic difference vector ($\phi_{i,j}(x_{i,j} - x_{k,j})$) [492]. The difference vector is obtained by comparing the current solution with another solution in the population.

The quality of the resulting v_i is compared with that of x_i and the bee exploits the better location. If the new location ($v_{i,j}$) is better than the old one ($x_{i,j}$) then the new location replaces the old one in the memory of S_N food sources. Otherwise, the location remains unchanged in the memory. A critical aspect of this search process is that as the positions of the best food patches begin to converge as the algorithm runs (and therefore $x_{i,j}$ and $x_{k,j}$ get closer together), the step size in the search process self-adapts, becoming smaller in order to promote more intensive exploitation around the already discovered better solutions.

$$v_{i,j} = x_{i,j} + \phi_{i,j}(x_{i,j} - x_{k,j}) \tag{10.1}$$

After all the employed bees have undertaken a local search step, the onlooker bees are recruited to the S_N food patches and in turn they search around the food patch for which they have been recruited (using (10.1)). In choosing which food patch to investigate, each onlooker bee is driven by

Algorithm 10.2: Artificial Bee Colony Algorithm [314, 316, 317]

```
Randomly assign each of S_N employed bees to initial food sources (locations)
in the search space;
Evaluate the quality of each of these locations;

Let i = 1;
while i < iter_max do
    for each employed bee ∈ {1, ..., S_N} in turn do
        Search for better solutions in the vicinity of its current location (food
        patch) using (10.1);
        Evaluate the quality of the new solution;
        if the new solution is better then
            The bee is relocated to the new position;
        else
            It stays at its current location;
        end
        Calculate the probability of an onlooker bee choosing each of the S_N
        food patches to harvest using (10.3);
        for each onlooker bee i ∈ {1, ..., S_N} in turn do
            Select a food patch to harvest using (10.3) (this chooses a value
            for j);
            Randomly select k ∈ S_N;
            Search for better solutions in the vicinity of that food patch using
            (10.1);
            Evaluate the quality of the new solution using (10.2);
            if the new solution is better then
                The bee is relocated to the new position;
            else
                It stays at its current location;
            end
        end
        if the location of any of the S_N food patches has remained unchanged
        for more than l iterations of the algorithm then
            Abandon that location and replace it with a new randomly
            generated location;
        end
        Record location of best solution found so far;
        Let i = i + 1;
    end
    Return best solution found;
end
```

(10.3), simulating a dance process, where p_i is the probability that any given onlooker bee is recruited to food patch i where $i \in (1, \ldots, S_N)$. This roulette wheel selection mechanism implies that the best food patches are more likely to recruit onlooker bees. If a food source is not improved after a predetermined number of iterations (parameterised as a *limit* l), it is abandoned and replaced by a new randomly created food source, thereby maintaining an exploration capability in the algorithm and helping to prevent premature convergence. This simulates the explorative search behaviour for new food sources by scout bees.

$$\text{fit}_i = \frac{1}{1 + f_i} \tag{10.2}$$

$$p_i = \frac{\text{fit}_i}{\sum_{n=1}^{S_N} \text{fit}_n} \tag{10.3}$$

The critical parameters of the algorithm include the value of S_N, the limit number of iterations l before an unchanged food source location is abandoned, and the maximum number of iterations of the algorithm, iter_{max}.

10.3.3 Honeybee Foraging and Dynamic Environments

One interesting aspect of real-world bee foraging behaviour is that it does not necessarily produce optimal foraging results, in terms of maximising nectar collection per unit of time in a *static* environment [34] where food resources are not depleted via consumption. Nakrani and Tovey [432] note that real world bee colonies operate in a highly dynamic environment, and based on the results of their study suggest that bee foraging strategies are therefore designed to produce best results in dynamic environments rather than in static ones. This reminds us that the application of algorithms inspired by dynamic biological processes to static problems is not necessarily appropriate.

Critiquing the Algorithms

In most honeybee algorithms, and indeed in most discrete ant colony algorithms, the core concept is that of recruitment, whereby bees or ants which have found good food sources recruit conspecifics which travel to and harvest the food resource. In order to maintain some diversity in the search process to avoid premature convergence, ‘forgetting’ mechanisms, such as pheromone evaporation in ant colony optimisation or continual random search by some foragers, are typically included in algorithm implementations. Whilst the resulting algorithms have proven to be highly effective, it can be noted that their design architectures are far simpler than the real-world foraging behaviours of these insects. In the next paragraph we detail some of the foraging process which is omitted in the canonical honeybee algorithm.

Honeybees typically have quite good visual sensory capabilities and are able to identify promising food sources at a distance and alter their flight trajectory to forage at the new resource. The issue of 'in-flight' perception is important as the search process does not likely commence only when a honeybee reaches an 'advertised' patch but is ongoing during the bee's flight.

A second issue is that the recruitment process is much 'noisier' in the real world than is typically suggested in honeybee algorithms, and dances only recruit to an approximation of the location of the food resource. Repeated dances for the same resource often vary in both directional and distance information [8, 233] and a recruit may have to undertake several trips before finding the advertised food source. A side effect of this is that foragers who have been newly recruited to a foraging location may not be sure of the exact distance and therefore of the energy requirements of the foraging flight. Honeybee foragers take small amounts of honey from nestmates via trophallaxis before leaving the hive in order to have food resources for their flight. A study by [248] indicated that dance followers carried a larger amount of honey than dancers but this differential reduced over repeated trips to the same food location. This could be a physical manifestation of the location uncertainty faced by newly recruited foragers. Although, prima facie, a noisy communication mechanism would appear suboptimal, it has been suggested that the imprecision in the honeybee dance could be adaptive as it would allow for the discovery and exploitation of nearby food sources [218]. In essence, it would inject a stochastic element into the foraging process.

A third issue is that the recruitment propensity of a honeybee depends on the quality of its current foraging location. If a foraging honeybee has already found a profitable food source, it is unlikely to be recruited to an alternative food source. Experimental evidence indicates that the majority of foraging bees at any point in time are actually using private rather than social information [233, 664], indicating that while social information and recruitment is important, it does not have the dominant role to which it is assigned in most honeybee algorithms. A more complete picture is that honeybees forage at favoured locations until they become unprofitable, at which time they are more likely to follow dances. In other words they employ a flexible strategy ('copy if dissatisfied') which combines both personal and social learning. This strategy is also relatively simple to implement as it does not require complex cognition such as comparison of the relative costs and benefits of several alternatives. There is also evidence that at least some forager bees maintain a memory of past food sources from which they have previously harvested. These 'inspector bees' continue to make occasional trips to the source to check on its quality and will resume foraging if it again becomes profitable. Hence, these inspector bees act as short-term memory for the bee colony and facilitate the quick reactivation of previously abandoned food sources.

From the discussion above, it is evident that the use of individual perception, social information and private information is nuanced in real honeybee foraging behaviour. This provides several interesting avenues for the design of

new honeybee algorithms which incorporate elements inspired by these mechanisms.

10.4 Bee Nest Site Selection

Another example of cooperative problem solving by honeybees is provided by nest site selection. Typically, in late spring or early summer as a colony outgrows its current hive, the colony will *fission* or 'divide' whereby the queen bee and about half of the population of worker bees will leave the hive and seek to establish a colony at a new nest site, leaving behind a young queen bee and the remainder of the worker bees in the existing hive. Having left the current hive, the swarm usually does not fly far and within about 20 minutes it forms a football-sized cluster of bees, often on the branch of a tree [45]. From this location *scout* bees begin a search for a new nest site and the search process can last for some days. An ideal home for a bee colony is located several metres off the ground, has a cavity volume of more than 20 litres, and has a south-facing entrance hole smaller than 30 square centimeters which is located at the floor of the cavity [566] (p. 222).

During the site selection process, scout bees leave the cluster and search for a new nest site. These scout bees are the most experienced forager bees in the swarm and usually the swarm will have several hundred scouts. As potential nest sites of satisfactory quality are uncovered, the returning scout bees communicate their location to other scout bees by doing a waggle dance on the surface of the swarm. The length of the dance depends on the quality of the site found, with longer dances being undertaken for better-quality sites. If a bee finds a good site, it becomes *committed* to it and will visit it several times. However, the length of its recruitment dance for the site will decrease after each visit. This phenomenon is known as *dance attrition* [566].

A scout bee will only dance for a location if its quality exceeds a threshold value; hence a scout may undertake several trips before uncovering a site of desired quality. Alternatively, a scout bee may be 'recruited' by a returned 'dancing' scout bee and may therefore visit a promising location found by another scout bee. In turn, if the recruited bee also considers the location to be of satisfactory quality, she will dance for that location on her return to the swarm. The net effect of the recruitment and the dance attrition phenomena is that higher-quality sites attract more attention from the searching scouts, creating a positive reinforcement cycle. Dance attrition facilitates the 'forgetting' of nest site locations that are not continually reinforced, akin to pheromone evaporation in ant colony foraging (Sect. 9.3). While multiple nest sites (if several of sufficient quality exist) will be considered in the early stage of the search process, these will be quickly whittled down to a limited number of choices (which could be up to several kilometers from the swarm) from which one is finally chosen. Unlike the foraging process whereby several food

locations may be harvested simultaneously, the nest site selection problem produces a single 'winner'.

The final decision as to nest site location results from a *quorum sensing* process [566]. Once scout bees at a potential nest site sense that the number of other scout bees there has reached a threshold (of approximately 20 bees) they undertake 'buzzing runs' at the new nest site, which triggers the return of all the scouts to the swarm [565]. The scouts then excite the swarm into getting ready to move by doing buzzing runs across the swarm's surface and by moving through the swarm, producing a high-pitched acoustic signal from their wing vibrations (known as *piping*).

When the swarm lifts off, an evident practical problem is that only about 5% of the swarm has previously visited the new nest site [565]; hence most of the swarm do not know the new site's location. The swarm is guided to the correct location by the scout bees who have visited the new site and they signal the correct trajectory by flying rapidly through the swarm in the direction of the new nest site [45]. In addition, some scouts shoot ahead to the nest site where they release marker pheromones.

The nest site selection process of honeybees presents an interesting example of a high-stakes exploration-exploitation trade-off [296]. If 'too fast' a decision is made (thereby exploiting information acquired early in the search process) the swarm runs the risk of selecting a poor location. On the other hand, if the decision-making process is too slow, the swarm is left exposed to the risk of bad weather (for example, rain) and/or to the risk of predation. The selection process also presents an example of decentralised decision making as the final site selection is determined by the actions of multiple, independent, agents (bees). Even where a scout bee is recruited for a potential nest site location, she will travel to the site and inspect it for herself in order to decide whether she will dance (or *vote*) for it. As noted by Passino and Seeley [498], it is plausible that evolution has tuned the process of nest-site selection in order to balance its speed-accuracy trade-off, balancing the chance that the swarm selects a poor site against the energy and time cost associated with searching for a new site.

Recent years have seen a number of studies such as [498] which have used simulation in order to examine the relative importance of various elements of the nest site selection process in determining the success of the search process. As would be expected, values for parameters such as the number of scout bees, the quorum threshold required to make a site decision, the decay rate of dance length following revisits to a site by scouts, and the propensity of bees to search by themselves as opposed to being recruited by other scouts, are all found to be important. The potential for drawing inspiration from the bee nest site selection process in order to design general-purpose optimisation algorithms was indicated by [162]. This framework was further developed and applied in [163] with the creation of the *bee nest site selection optimisation algorithm* (BNSO). The next subsection outlines this algorithm.

Algorithm 10.3: Bee Nest Algorithm (from [163])

```
Place swarm on random location p (p_swarm = p);
Let counter = 0;
Set value for d_scout, d_follower, and number of scouts and followers;
repeat
    Set value of f_range using (10.4);
    for each scout in turn do
        Choose new location p_s with a maximum distance of d_scout × f_range to
        the nest;
        fit_s = max{0, (F(p_swarm) × f_q) − F(p_s)};
    end
    for each follower in turn do
        Choose a scout s according to (10.5);
        Choose a new location p_follower with a maximum distance of d_follower
        to chosen scout's position p_s;
        Sample search space between p_s and p_follower in m equally-spaced
        flight steps;
    end
    if a new location p was found which is better than p_swarm then
        Relocate swarm to p (p_swarm = p);
    else
        if counter ≥ maxcount then
            Place swarm on new random location p (p_swarm = p);
            counter = 0;
        else
            counter = counter + 1;
        end
    end
until termination condition is satisfied;
Return location of nest (best solution found);
```

10.4.1 Bee Nest Site Selection Optimisation Algorithm

The essence of the algorithm is that the 'swarm' of bees searching for a new nest seeks to iteratively improve the quality of its location. The swarm is comprised of two types of bees, namely scouts and followers. The scouts seek new potential nest sites in the vicinity of the swarm's current location and if a scout succeeds, it recruits a number of followers based on the fitness of the location that it has found. The followers then go to the location found by the scout and then search locally around that location. If a better site is found than the swarm's current location, the swarm moves to the new location and the search process begins anew. See Algorithm 10.3.

In the algorithm, there is a swarm of n bees comprised of n_{scout} and n_{follower} bees ($n = n_{\text{scout}} + n_{\text{follower}}$). These are searching in a real-valued space and

the location p_{swarm} of the swarm in this space corresponds to a solution to the maximisation or minimisation problem of interest. During the algorithm, the swarm is initially placed at a random location, and each scout s chooses a random location p_s which is within $d_{\text{scout}} \times f_{\text{range}}$ of the swarm's current location (hence, $\mid p_{\text{swarm}} - p_s \mid \leq d_{\text{scout}} \times f_{\text{range}}$ for all scouts). This simulates the real-world phenomenon that scouts are more likely to search within a few minutes' flight time of the swarm's current location.

In order to encourage convergence of the search process, $f_{\text{range}} \in [0, 1]$ is decreased as the algorithm iterates. A simple mechanism for implementing this is presented in (10.4) whereby $\text{iter}_{\max}$ is the maximum number of iterations that the algorithm will run and iter is the current iteration number. If the new location found by a scout is of sufficient quality (i.e., assuming a minimisation problem, $F(p_s) \leq F(p_{\text{swarm}}) \times f_q$, where $f_q \in [0, 1]$ is a quality 'improvement' factor), then the location is a candidate for recruitment of follower bees.

$$f_{\text{range}} = 1 - \frac{\text{iter}}{\text{iter}_{\max}} \tag{10.4}$$

The relative fitness of each scout is calculated using $\text{fit}_s = \max\{0, (F(p_{\text{swarm}}) \times f_q) - F(p_s)\}$. Each follower then selects one scout to follow using (10.5); hence the fitter locations uncovered by the scouts will tend to recruit a greater number of followers. Each follower then proceeds to choose a random location p_f in the neighbourhood of the scout's location (p_s) subject to the constraint that the selected location is not greater than d_{follower} away from p_s so that $\mid p_s - p_f \mid \leq d_{\text{follower}}$. As for d_{scout}, the value of the parameter d_{follower} is set by the user of the algorithm. The follower bee then samples the search space between p_s and p_f in m equal-sized steps (calculating the fitness at each of the m steps) on a straight-line flight between the two locations.

During the scout and follower search phase, a record of the location of the fittest point found (p_{best}) is maintained and if this is better than the fitness of the current location of the swarm (p_{swarm}), then the swarm moves to the new location and restarts the process. Otherwise the nest search process is restarted from the swarm's current location. If the swarm cannot improve its location after maxcount attempts, it is moved to a new random location in the search space and the search process recommences from that location.

$$P_s = \frac{\text{fit}_s}{\sum_{k=1}^{n_{\text{scout}}} \text{fit}_k} \tag{10.5}$$

The Bee Nest algorithm as described above was applied in [163] for the molecular docking problem. The paper compared the results from the algorithm against those of PSO and random search on the same problem, finding that the Bee Nest algorithm was reasonably competitive. Further development of the algorithm and further testing will be required before its utility can be fully assessed. However the algorithm does represent an interesting new avenue for honeybee inspired algorithmic research.

10.5 Honeybee Mating Optimisation Algorithm

The mating flight behaviour of honeybee queens has also been used as the inspiration for the design of an optimisation algorithm [1, 2]. This algorithm is also referred to as the *marriage in honeybees optimisation algorithm*. Before outlining the algorithm a brief introduction to the honeybee mating process is provided.

A normal honeybee colony consists of a queen bee, drones and workers. The queen bee is the only egg-laying female in the colony. Drones are male bees whose primary task is to mate with the queen. Worker bees (which are all female) undertake all of the day-to-day activities of the colony including food foraging and storage, cleaning of the colony, guarding of the colony and feeding the queen, drones and larvae. The mating process commences when the queen performs a dance in the hive after which a mating flight occurs. During this flight, the queen departs from the hive followed by drones who attempt to mate with the queen in mid-air. During each mating, sperm from the drone is accumulated in the queen's spermatheca. When the queen subsequently lays eggs in the hive these are fertilised using randomly drawn sperm from her spermatheca. In the adaptation of this process to create an optimisation algorithm, the genome of each bee (queen and drone) corresponds to a location in the search space.

An overview of a general honeybee optimisation algorithm is provided in Algorithm 10.4. In this algorithm the queen is initialised with a random genotype (or 'location'), an energy level and a speed. The drones are also initialised in random locations. The probability that a specific drone mates with the queen is governed by (10.6), where $P(Q, D)$ is the probability of adding the sperm of drone D to the spermatheca of queen Q, in other words the probability of successful mating. As the queen mates with several drones in turn, her spermatheca fills up with the genotypes of the drones with whom she has mated. The parameter d is the absolute difference between the fitness of the drone and that of the queen, and $s(t)$ is the speed of the queen at time t. From the structure of (10.6), a form of annealing function (Sect. 23.1), it can be observed that the mating probability is higher in the earlier stages of the queen's mating flight (when her speed is higher), and when the drone has a good level of fitness (vs. that of the queen). During each iteration of the algorithm, the energy level E and speed of the queen are decreased as given by (10.7) and (10.8), where $\alpha \in [0, 1]$ and $\beta \in [0, 1]$ represent step sizes to control the reduction of each property.

$$P(Q, D) = \mathrm{e}^{-d/s(t)} \tag{10.6}$$

$$s(t+1) = \alpha s(t) \tag{10.7}$$

$$E(t+1) = \beta E(t) \tag{10.8}$$

Algorithm 10.4: Honeybee Mating Optimisation Algorithm

```
Generate the initial population of bees at random;
Evaluate the fitness of the location of each bee;
Select the best of these bees to be the queen;
Select the number of mating flights (M);
Let i = 1;

repeat
    Initalise energy E, speed s and spermatheca size of queen;
    Set value for α ∈ [0, 1] and β ∈ [0, 1];
    while E > 0 and queen's spermatheca is not full do
        Select a drone at random;
        if drone passes the probabilistic condition for mating (10.6) then
            Add sperm of drone to queen's spermatheca;
        end
        Let s(t + 1) = αs(t);
        Let E(t + 1) = βE(t);
    end
    Generate broods using crossover and mutation;
    For each brood member, randomly select a worker from the population
    W of workers who will apply a local search heuristic in an attempt to
    improve that brood member's fitness;
    if the fitness of the best brood member is greater than that of the queen
    then
        Replace the queen by that brood member and remove the brood
        member from the brood list;
        Generate an additional member for the brood population;
    end
    i = i + 1;
until i = M;
Replace population of drones by the current brood (these will be the drones
in the next iteration of the algorithm);
Return location of queen (best solution found);
```

When the mating flight is complete the breeding process commences. In this process a sperm is randomly selected from the queen's spermatheca and this is crossed over with the queen's genome to produce a 'brood'. This crossover need not be restricted to take place between the queen's genotype and that of a single drone as the process in real life can utilise genetic material from multiple drones in creating the new brood. A mutation process is then applied to the new genome. In an analogue of the role of worker bees in the colony which look after the brood as they mature, a total of W 'worker bees' exist in the algorithm. Each of these worker bees corresponds to a *local search heuristic* rather than to a genotype. In the algorithm they are used in an attempt to further improve the quality of each brood or protosolution, by

selecting a worker bee and applying its corresponding local search heuristic to the individual brood solution. This step can be loosely considered as corresponding to the process of brood care by the workers in a hive, with the time devoted to brood care forming an analogue with the 'depth' of a local search process [41]. The choice of local search heuristic to be applied to an individual brood member is made stochastically.

After all the brood members have been locally improved, their fitnesses are compared and the best of them is compared with the fitness of the queen. If the best resulting brood is better than the queen, the best brood member replaces the queen and a new mating flight commences using the new queen. Depending on the variant of the algorithm the drones in the new mating flight may be chosen randomly [1], or may consist of the members of the brood in the previous iteration of the algorithm, corresponding to a generational replacement strategy in a GA.

As can be seen from the above discussion, the algorithm can be implemented in a multiplicity of ways depending on the user's choice of the various parameter settings (including the number of queens, the number of drones, the number of mating flights and the size of the spermatheca for each queen), the mechanism for choosing which drones get to mate with the queen, the precise mechanisms for undertaking crossover and mutation, and the form of the 'local improvement' step. A significant body of literature has developed wherein a wide variety of design choices for this algorithm have been tested, and the reader is referred to [315]. Applications of the algorithm have also been extended beyond real-valued optimisation to encompass combinatorial optimisation. Whilst good quality parameter settings will be problem-specific, parameters from [389], including one queen, 200 drones, number of mating flights = 1500, size of spermatheca = 50, number of broods = 50 and $\alpha = 0.9$, may provide a starting point. The algorithm bears some similarity with the GA, where the mating system embeds polyandry (the queen mates with several males) along with a local search phase.

10.6 Summary

Honeybee behaviours are a rich source of inspiration for the design of computational algorithms and this area is a rapidly growing subfield of natural computing. Thus far, most algorithms incorporate a limited number of features from the full repertoire of behaviours of honeybees. Scope exists to refine these algorithms further and to more fully test their utility on real-world applications. In addition to real-valued optimisation, there have been applications of honeybee-inspired algorithms for binary-valued optimisation problems [492], for combinatorial optimisation and for clustering purposes. Readers are directed to [41] and [315] for relevant references.

11

Bacterial Foraging Algorithms

Bacteria are amongst the oldest and most populous forms of life on earth (a human typically has about 10^{14} bacteria in the gastrointestinal tract [499]). Despite possessing a relatively simple physical structure in comparison with mammals or insects, bacteria are capable of sophisticated interactions with their environment and with each other. Even though an individual bacterium has limited information processing power, colonies of bacteria can solve complex problems such as those of survival in a dynamic environment. This capability arises in part as bacteria are metaphorically capable of 'communication' with each other by means of chemical molecules, a process which has parallels with the use of pheromone trails by ants. While no claim is made that bacteria are *consciously* communicating with each other, or that they are intentionally guiding their behaviours, the chemical signalling mechanisms produce emergent outcomes which are functionally equivalent to problem-solving strategies. In this chapter we describe a number of bacterial behaviours and demonstrate how these can be used to design optimisation algorithms.

11.1 Bacterial Behaviours

In this chapter we concentrate on three bacterial behaviours, namely:

i. quorum sensing,
ii. sporulation, and
iii. mobility.

The first two of these behaviours provide illustrative examples of chemical communication between bacteria. The third behaviour (mobility) is metaphorically used to develop an optimisation algorithm in Sect. 11.2.

11.1.1 Quorum Sensing

Commonly, bacteria can emit and detect chemicals known as *autoinducers* which allow them to sense whether more bacteria of their own (or other)

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_11

kind are around [37]. The greater the concentration of peer bacteria in close proximity, the greater the concentration of autoinducer chemical. If a sufficient density of bacteria is present (a *quorum*), some phenomenon may occur, such as the formation of a biofilm (an assembly of bacteria that is attached to a surface) or bioluminescence. An interesting example of quorum sensing behaviour is that a bacterial colony, for example the culprit in many food poisoning cases, salmonella, may wait until a critical mass of bacteria is present before releasing a toxin to poison its host. This makes the bacterial colony more resistant to attack from the host's immune system.

11.1.2 Sporulation

Sporulation of bacteria occurs in response to starvation. If environmental conditions become too stressful, individuals in some bacterial species can transform themselves into inert, enduring spores (thick-coated cells which are resistant to heat, desiccation, and long-term starvation). These spores can then be dispersed (for example, via the wind) to a more benign environment. The sporulation process commences when starving bacteria emit chemical messages which act metaphorically to convey their stress to their peers. On receiving these messages, neighbouring bacteria compare the strength of these messages with their own state and vote (chemically) for or against sporulation. If a sufficiently strong concentration of the voting chemical is emitted by the members of the bacterial colony, the individual bacteria enter a dormant, or spore, state. When more abundant environmental conditions occur the cells emerge from their dormant state and reactivate.

11.1.3 Mobility

Many species of bacteria are able to move in response to external stimuli (this phenomenon is called *taxis*) and this provides them with a powerful adaptive capability. Examples of such movement include chemotaxis, where bacteria move towards or away from certain chemical concentrations; phototaxis, where they move in response to light; thermotaxis, where they move in response to different temperatures; aerotaxis, where they move in response to different levels of oxygen concentration; and magnetotaxis, where they align themselves in response to magnetic lines of flux.[1] Depending on the environment occupied by a bacterium, movement can be through a medium (for example, movement through a fluid) or along a surface.

---

[1] Magnetotactic bacteria contain nanoparticles of magnetite or greigite which are assembled into linear arrays. These arrays create a magnetic dipole in the bacterium that forces orientation, and therefore travel, along geomagnetic field lines [580].

11.2 Chemotaxis in E. Coli Bacteria

Escherichia (*E.*) *coli* are amongst the most-studied species of bacteria, partly because of their prolific ability to reproduce. An E. coli cell can reproduce in approximately 20 minutes under ideal conditions. Given sufficient food and ideal environmental conditions, a single E. coli bacterium could give rise to a bacterial colony with a total population exceeding the number of humans on earth within 11 hours!

A simplified diagram of an E. coli cell is shown in Fig. 11.1. Each E. coli cell is approximately 1-2 μm in length, with approximately eight flagella (only one is shown in Fig. 11.1) which allow it to swim through a liquid medium. Each flagellum is approximately 10 μm long, has a rigid left-handed corkscrew shape, and forms a propeller. The propeller is powered by a tiny biological 'electric' rotary motor. The power source for the motor is the electromotive gradient arising from proton-sodium ion flows across the cell's membrane [153]. The motor is highly efficient and in an E. coli cell can reach approximately 18,000 RPM [153], or around the same number of RPM as a Formula One racing car engine.

The motor and hence the propeller shaft is capable of turning both clockwise (CW) and counter-clockwise (CCW). If the flagella are rotated CCW in tandem they produce a force against the E. coli cell and it moves (swims) forward (termed *a run*). If the flagella switch from a CCW to a CW rotation, the cell tends to rotate randomly (termed *a tumble*) to point in a new direction.

E. coli cells are rarely still but instead are engaged in continual run-and-tumble behaviours [52]. The two modes of movement alternate. Runs tend to last for about one or two seconds, whereas tumbles take less time to complete (circa 0.1 to 0.2 s). Even this brief time span is sufficient to orient the bacterium in a random direction, causing successive runs to occur in nearly random directions (there is a slight bias in the tumbling process towards continuing in the same direction as before). Therefore, a bacterium runs, tumbles to face a new direction, and runs again (Fig. 11.2). The cell is also subject to Brownian movement as it is moving through a liquid medium and it has very little mass. This causes the cell to tend to drift off course every 10 seconds or so, even when it is following a strong chemical trail.

In spite of the tumbling process and the effects of Brownian movement, the movement of a bacterium is not completely random. When a bacterium finds stronger concentrations of a chemically attracting item, the mean length of its runs increases. This effect biases its movement towards the climbing of attractive chemical gradients, and therefore the process can be considered as a stochastic gradient search for chemical attractants.

The running and tumbling behaviours of bacteria result from the stimulation of receptors on the cell for the chemicals of interest to it. Increasing concentrations of chemical attractants increase CCW rotation. Repellant stimuli have the opposite effect. The behavioural response of the cell is temporal rather than spatial as E. coli cells are too short to enable differences in chem-

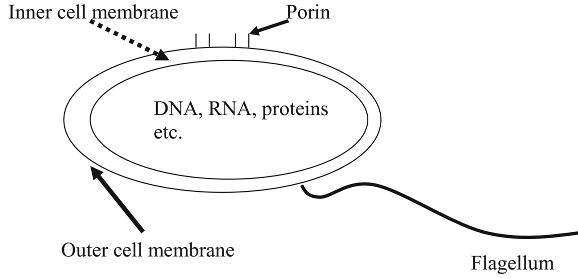


Fig. 11.1. A simplified diagram of an E. coli cell, showing one flagellum (not to scale) and two porins (protein channels which allow the entry of water-soluble nutrients). Typically a cell will have hundreds of porins

ical concentrations between receptors at each end of the cell to be significant. Hence, a cell cannot directly sense and use (chemical) gradient information from its environment. Instead, it has an implicit short-term memory [78, 383] lasting up to about 3s which allows it to detect whether the concentration of a chemical is changing over time.

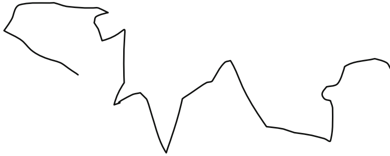


Fig. 11.2. A sample movement track for an E. coli cell. The track consists of a series of runs punctuated by tumbles (directional changes)

11.3 Bacterial Foraging Optimisation Algorithm

The idea that chemotactic behaviour in bacteria could be considered as an optimisation process was originally proposed by Bremermann in 1974 [76]. However, the idea did not attract substantial attention when published and it is only in recent times that it has been revisited. In this section we describe one genre of real-valued bacterial foraging optimisation algorithms (BFOA) which are loosely derived from the foraging strategy of E. coli bacteria. This genre of algorithm was described by Passino [496, 497] and has been applied in a series of papers including [377, 415] and [416].

This family of BFOAs draws its inspiration from four aspects of E. coli behaviour, namely:

i. chemotaxis,
ii. swarming,
iii. reproduction, and
iv. elimination-dispersal.

Chemotaxis refers to the tumble and run behaviour of an individual bacterium in response to chemical gradients. Swarming refers to the capability of E. coli bacteria to emit the chemical attractant aspartate when they uncover nutrient-rich environments. In turn, other E. coli bacteria respond by moving (swarming) towards bountiful regions which are marked with attractant. The emission of attractant produces an indirect social communication mechanism between bacteria. Under selection for reproduction, the healthiest bacteria are more likely to survive and divide, thereby creating more bacteria which are similar to themselves and which are colocated in nutrient-rich environments. Bacterial elimination and dispersal events, a form of randomisation and movement, occur frequently in the real world. The former occur when a local environmental event kills a bacterial colony, and the latter occur when bacteria are transported to a new location, for example via wind dispersal.

A wide variety of BFOAs can be created from the above highly stylised aspects of bacterial foraging. For example, an algorithm could be developed solely based on a metaphor of chemotatic mobility or an algorithm could be developed by combining all of the above elements. In addition, the individual characteristics such as reproduction or swarming could be operationalised in many ways.

11.3.1 Basic Chemotaxis Model

The pseudocode in Algorithm 11.1 describes a search process which solely employs the chemotaxis concept. A population of S bacteria is created and distributed randomly in the search space, assumed to be a domain in $\mathbb{R}^D$. The position (vector) of each bacterium i is stored as $\theta^i \in \mathbb{R}^D$. We assume that there is a cost $J^i = J(\theta^i)$ associated with each location that a bacterium can occupy and that the intention is to find the location in the search space with minimum cost.

The fitness of each bacterium is calculated, after which each bacterium seeks to move. The bacterium tumbles to face a random direction and then continues to take equal-sized steps in this direction while this improves its fitness, up to a maximum of N_s steps. The best location found by any bacterium is stored and is returned at the algorithm's termination.

Although this algorithm does produce a search process, it is not particularly effective or efficient as it amounts to a population of bacteria engaging in a series of biased random walks across the environment. Each bacterium

Algorithm 11.1: Basic Chemotactic Bacterial Foraging Algorithm

```
Randomly distribute initial values for the position vectors θ^i, i = 1, 2, ..., S
across the optimisation domain;
Compute the initial cost function value J^i = J(θ^i) for each bacterium i;
repeat
    for each bacterium i do
        Tumble: Apply random tumble to bacterium to face it in a new
        direction;
        Take a step in this direction;
        Measure fitness of new location;
        while number of swim steps < N_s do
            if fitness of new position > fitness of previous position then
                Take another step in current direction and calculate fitness of
                new location;
            else
                Let number of swim steps = N_s − 1;
            end
            Increment number of swim steps;
        end
    end
until terminating condition;
```

searches individually and there is no social communication between bacteria. Hence, information about nutrient-rich or nutrient-poor regions is not passed amongst the members of the population.

11.3.2 Chemotaxis Model with Social Communication

Passino [496, 497] describes a more complex BFOA which includes social communication (swarming and reproduction) along with an elimination dispersal process which promotes diversity. Initially, an overview of the algorithm (Algorithm 11.2) is provided, followed by a discussion of how it can be operationalised.

Initialisation of the Algorithm

As above, each bacterium i is initially randomly located in the search space with its position being stored as $\theta^i \in \mathbb{R}^D$. Let $J^i = J(\theta^i)$ be the cost associated with the bacterium's location. Assume that the intention is to find the location in the search space with minimum cost.

Algorithm 11.2: BFO Algorithm with Social Communication

Randomly distribute initial values for θ^i, $i = 1, 2, ..., S$ across the optimisation domain;
Compute the initial cost function value for each bacterium i as $J^i = J(\theta^i)$, and the initial total cost with swarming effect as J^i_{sw};

```
for Elimination-dispersal loop do
    for Reproduction loop do
        for Chemotaxis loop do
            for Bacterium i do
                Tumble: Generate a unit length vector φ ∈ ℝ^D in a random
                direction;
                Move: Let θ^new = θ^i + cφ and compute corresponding J^new;
                Let J_sw^new = J^new + J_cc(θ^new, P);
                Swim: Let m = 0;
                while m < N_s do
                    Set m = m + 1;
                    if J_sw^new < J_sw^i then
                        Let θ^i = θ^new and compute corresponding J^i and J_sw^i;
                        Let θ^new = θ^i + cφ and compute corresponding J^new;
                        Let J_sw^new = J^new + J_cc(θ^new, P);
                    else
                        Let m = N_s;
                    end
                end
            end
        end
        Sort bacteria in order of ascending cost J_sw;
        The S_r = S/2 bacteria with the highest J value (the 'least healthy')
        die and the remaining S_r bacteria split;
        Update value of J and J_sw accordingly;
    end
    Eliminate and disperse individual bacteria to random locations on the
    optimisation domain with probability p_ed;
    Update corresponding values for J and J_sw;
end
```

Select highest fitness location from final population (or the best location found during the algorithm);

Notation Used

The ordered S-tuple of positions of the entire population of S bacteria at the jth chemotactic step, the kth reproduction step and the lth elimination-dispersal event is denoted by $P(j,k,l) = (\theta^i(j,k,l)|i = 1,2,\ldots,S)$. As the algorithm executes, $P(j,k,l)$ is updated immediately once any bacterium moves to a different location. The cost associated with location $\theta^i(j,k,l)$ is denoted as $J^i(j,k,l)$. Each bacterium has a lifetime N_c, measured as the maximum number of chemotactic cycles it can undertake.

Chemotaxis Loop

At the start of each chemotactic loop, the 'swarm-effect inclusive' (SEI) cost corresponding to each bacterium's current location is calculated as follows:

$$J^i_{\text{sei}}(j,k,l) = J^i(j,k,l) + J^i_{\text{cc}}(\theta^i(j,k,l), P(j,k,l)) \tag{11.1}$$

Hence, the SEI cost is comprised of both the underlying cost of the bacterium's location (as given by $J^i(j,k,l)$) and a value for the cell-to-cell attraction and repelling (swarming) term $J_{\text{cc}}(\theta(j,k,l), P(j,k,l))$. In the swarming behaviour of the bacteria, each individual is trying to minimise $J^i_{\text{sei}}(j,k,l)$, so they will try to find low-cost locations and move closer (but not too close) to other bacteria.

The effect of the cell-to-cell attraction and repelling term is to create a time-varying SEI cost function which is used in the chemotaxis loop. As each bacterium moves about on the landscape, its $J^i_{\text{sei}}(j,k,l)$ alters, as it depends not just on the bacterium's own current location but also on the simultaneous locations of all other bacteria in the population. Another way of thinking about this is that the landscape being searched is dynamic and it deforms as the bacteria traverse it.

Once $J^i_{\text{sei}}(j,k,l)$ is calculated, it is stored in J^i_{curr} for each bacterium. This value is used later in the chemotaxis loop to determine whether a bacterium's movement is improving its fitness.

The calculation of $J^i_{\text{cc}}(\theta^i(j,k,l), P(j,k,l))$ depends on the proximity of each bacterium to its peers. Each bacterium is attracted to its peers, loosely mimicking the effect of the chemical attractant aspartate. Bacteria are also repelled from one another, mimicking the real-world problems of too-close location, as the bacteria would then compete for the same nutrients. Each of these mechanisms is included in (11.3).

$$J_{\mathrm{cc}}(\theta, P(j,k,l)) = \sum_{i=1}^{S} J_{\mathrm{cc}}^{i}(\theta, \theta^{i}(j,k,l) \tag{11.2}$$

$$= \sum_{i=1}^{S} \left[-d_{\mathrm{attract}} \exp\left(-w_{\mathrm{attract}} \sum_{b=1}^{p} (\theta_b - \theta_b^i)^2 \right) \right] \tag{11.3}$$

$$+ \sum_{i=1}^{S} \left[h_{\mathrm{repel}} \exp\left(-w_{\mathrm{repel}} \sum_{b=1}^{p} (\theta_b - \theta_b^i)^2 \right) \right]$$

In (11.3), the parameter p is the dimensionality of the real vector space $\mathbb{R}^p$ being searched by the bacteria (here $p = 2$ is assumed). The first additive component of (11.3) acts to reduce the SEI cost of each bacterium, as it is restricted to returning a nonpositive value. In the limit, if all bacteria converge to the same location, the exponential term in this component will tend towards its maximum value of 1 and the SEI costs of all bacteria will be reduced by $-d_{\mathrm{attract}}S$. Metaphorically, the terms represent the depth of the attractant released by a bacterium and the width of the attractant signal respectively. The second additive component of (11.3) repels the bacteria from one another. If the bacteria swarm to such a degree that they colocate, the SEI costs of all bacteria will be increased by $h_{\mathrm{repel}}S$.

The values of the parameters d_{attract}, w_{attract}, h_{repel} and $-w_{\mathrm{repel}}$ therefore control the strength of the swarming effect and repulsion effect relative to each other, and relative to the impact of the cost function $J^i(j,k,l)$.

In order to move a bacterium in the search space, a tumble followed by run behaviour is simulated. The tumble acts to orientate the bacterium in a random direction, with the bacterium then continuing to move in that direction until either it has taken the maximum number of chemotactic steps (N_s) or its cost function stops declining. In generating a tumble, a vector of unit length and random direction $\phi^i(j)$ are used. The bacterium is then moved a step ($C(i) > 0$) in this direction:

$$\theta^i(j+1,k,l) = \theta^i(j,k,l) + C(i)\phi^i(j) \tag{11.4}$$

The term $\phi^i(j)$ in (11.4) is obtained by generating a vector $\Delta(i) \in \mathbb{R}^2$ (assuming the search space is of two dimensions), where each element of the vector is randomly drawn from the interval $[-1, 1]$, using:

$$\frac{\Delta(i)}{\sqrt{\Delta^T(i)\Delta(i)}} \tag{11.5}$$

For example, if (0.6, 0.3) were drawn randomly to be $\Delta(i)$, the resulting unit vector would be (0.894, 0.447), calculated as $\frac{0.6}{\sqrt{0.6^2+0.3^2}}$ and $\frac{0.3}{\sqrt{0.6^2+0.3^2}}$. Once the bacterium has moved in a random direction, its SEI cost is updated:

$$J_{\mathrm{sei}}^i(j+1,k,l) = J^i(j+1,k,l) + J_{\mathrm{cc}}(\theta^i(j+1,k,l), P(j+1,k,l)) \tag{11.6}$$

Then, if $J^i_{\text{curr}} < J^i_{\text{sei}}(j+1,k,l)$ (the location after the tumble and move has lower SEI cost than the bacterium's location before the tumble), run behaviour is simulated. The bacterium iteratively takes a further step of the same size in the same direction as the tumble, checks whether this has lowered its cost value further, and if so, continues to move in this direction until it has taken its maximum number of chemotactic steps, N_s.

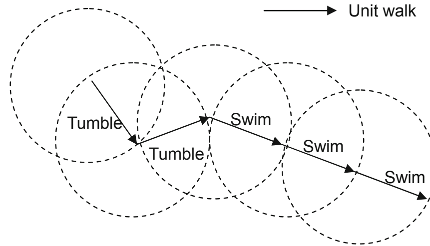


Fig. 11.3. Chemotactic step

Reproduction Cycle

After N_c chemotactic steps have been undertaken for the complete population of bacteria, a reproduction cycle is undertaken. In each reproduction cycle the 'health' of each bacterium is calculated as the sum total of its location costs during its life (over all its chemotactic steps), $J^i_{\text{health}} = \sum_{j=1}^{N_c} J^i_{\text{sei}}(j,k,l)$. All the bacteria in the population are then ranked in order of their fitness with higher costs corresponding to lower health. The $x\%$ (where x is a user selected parameter) healthiest bacteria split in two (reproduce) at their current location and a corresponding number of less healthy bacteria are eliminated, keeping the total population of bacteria constant. The reproduction cycle is repeated N_{re} times during the algorithm's execution.

Elimination-Dispersal Events

The entire population is subject to a total of N_{ed} elimination-dispersal events. In each of these events, individual bacteria in the population are killed with a probability of p_{ed}. If a bacterium is killed, a new bacterium is randomly located in the search space. If a bacterium is not selected for elimination (probability of $1 - p_{\text{ed}}$), it remains intact at its current location.

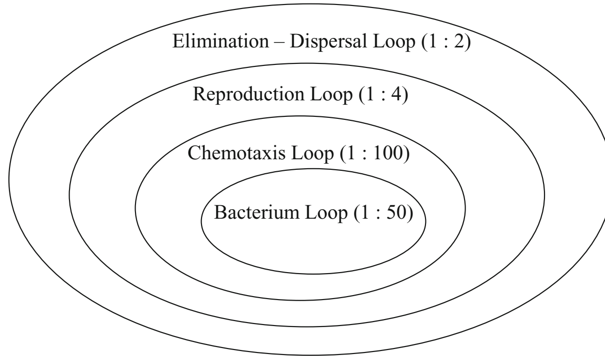


Fig. 11.4. The key loops in the BFOA algorithm

Parameter Values for the BFOA

The BFOA described above has a number of parameters which the modeller must select. While good choices of parameters will vary depending on the problem to which the algorithm is being applied, Passino [496] used the following in a sample application: $S = 50$ (number of bacteria), $N_c = 100$ (chemotactic cycles per generation), $N_s = 4$ (number of steps per run), $N_{\text{re}} = 4$ (number of generations), $N_{\text{ed}} = 2$ (number of elimination-dispersal cycles), $p_{\text{ed}} = 0.25$ (dispersal probability per bacterium in an elimination-dispersal cycle) and $C(i) = 0.1 : \; i = 1, \ldots, S$ (step size).

In selecting parameter values a few issues should be borne in mind. As values of S, N_c, N_{re}, etc. increase, the computational cost of the algorithm increases proportionately. The choice of $C(i)$ will be domain-specific, depending on the scaling of the solution space that the algorithm is searching. Decreasing the value of $C(i)$ during the run will encourage convergence (see [133] for a discussion of a number of schemes for chemotactic step adaptation). The value for N_s determines the maximum number of steps that a bacterium can take in a single run. If $N_s = 0$, the search process becomes a random walk, and as N_s increases, the search tends towards a gradient-descent. Intermediate values trade off the exploration-exploitation balance of the algorithm. The value of N_{re} and the way that bacterial eliminations are implemented impacts on the convergence properties of the algorithm. If a heavy-handed selection process occurs in each reproduction-elimination cycle, the algorithm will tend to converge more quickly but may get trapped in a local minimum. Similarly, if a low value of N_c is chosen, the algorithm will be more prone to getting stuck in a local optimum. The values of N_{ed} and p_{ed} also impact on the exploration-exploitation balance. If both values are large, the algorithm tends towards exploration (random search). The values selected for w_{attract}, h_{repel}, w_{repel} and d_{attract}, impact directly on the value of $J_{\text{cc}}(\theta, P(j,k,l))$ and therefore define the tendency of the bacteria to swarm. If the attractant width

and depth parameters are set to high values, the bacteria will tend to swarm easily, and if extreme values are set for these parameters, bacteria may prefer to swarm rather than to search for nutrients. On the other hand, if very low values are set for these parameters bacteria will search individually and will ignore communications from other bacteria.

11.4 Dynamic Environments

Real-world bacteria inhabit a dynamic environment and it is plausible that their survival strategies, including their foraging mechanisms, have evolved in order to cope with the difficulties of surviving in such environments. In applying the BFOA to static problems the rate of the algorithm's convergence, like that of all search algorithms, depends in large part on the choice of parameter settings. In dynamic problems, there is obvious utility to maintaining diversity in the population; hence these settings can be chosen in order to promote this. The objective of slowing down the rate of convergence is to promote the continual exploration of new regions by the population of bacteria. As for other natural computing optimisation algorithms, multiple strategies for dealing with dynamic environments could be added to the canonical BFOA. These could include the use of multiple species of bacteria (multiple populations), the maintenance of a memory of good past solutions and the use of sentries to flag the occurrence of environmental change.

Another approach to maintaining diversity in the population of bacteria would be to implement hybrid versions of the BFOA. For example, Tang et al. [615] describe a hybrid GA-BFOA. In this hybrid the dispersion and elimination loop of the canonical BFOA is removed and a more complex reproduction mechanism is implemented at the end of each chemotactic loop. In this reproduction mechanism, the bacteria which will make up the next generation of the population are determined using a selection process which is based on a weighted ranking of the bacterial fitnesses. The key element of this process is that all bacteria have some possibility of survival into the next generation. This contrasts with the hard threshold selection process in the canonical BFOA where the least fit $y\%$ of the population is automatically eliminated in the reproductive step.

11.5 Classification Using a Bacterial Foraging Metaphor

Another application of bacteria inspired algorithms which has not yet received much attention in the literature is the design of classification systems. A BFOA could be used to uncover good coefficients for a classification model as well as to uncover features for inclusion in the model.

More sophisticated model construction possibilities exist using multiple species of bacteria, each attracted to a different 'food' corresponding to a

different class of data item. For example, by allowing bacteria of a single species to swarm to a region of feature space which corresponds to a particular class, a net of detectors corresponding to the individual bacteria could be created. Each species of bacteria would be trained using a sample of data from a specific class. Once the swarm of bacteria is trained, out-of-sample data could be classified by determining which species' detection range it fell into. The data item would be assigned the class label corresponding to that species of bacteria.

Another possibility is to hybridise the BFOA with concepts from the negative selection algorithm in artificial immune systems (Chap. 16.3). In a two-class case, *self* examples could be designated as toxins and *nonself* examples as food. The bacteria would then move around avoiding the toxins and seeking the food. Just as for the negative-selection algorithm, the bacteria would act as detectors of nonself.

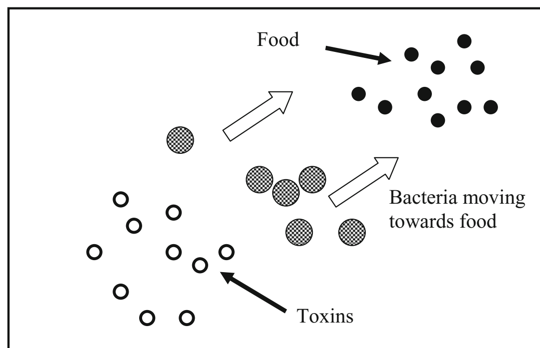


Fig. 11.5. Bacteria moving towards examples of nonself (denoted as food), away from examples of self (denoted as toxins)

11.6 Summary

This chapter has described the BFOA, an optimisation algorithm based on the foraging strategy of a single species of bacteria, E. coli. Clearly, considerable scope exists to design a wide array of related algorithms. In common with many other biologically inspired algorithms, bacterial foraging algorithms are an example of a populational, nongradient based optimisation method. The BFOA embeds concepts such as selection, reproduction and social communication which are also found in various other algorithms.

12

Other Social Algorithms

In the final chapter in this part of the book, we introduce four emerging families of algorithms which have been inspired by various processes of social communication, in insects, bats and fish. Although these algorithms are not as fully developed or explored as ant or honey bee algorithms, they provide interesting examples of the diversity of natural computing algorithms that can be developed from real-world social behaviours.

12.1 Glow Worm Algorithm

The glow worm belongs to the *Lampyridae* family of beetles. These beetles can produce light by means of bioluminescence. This light is typically used to attract resources, either mates or prey. In the first instance, bioluminescence acts as a communication mechanism alerting glow worms to the location of potential mates. In the second case, curious prey are attracted to the light source — and to their doom. The *glow worm swarm algorithm* (GWSA), loosely inspired by the phenomenon of bioluminescence, was introduced by Krishnanand and Ghose [351, 352, 353] as a general optimiser which may have particular potential for multimodal environments. In the algorithm, each agent (glow worm) has an associated luminescence which indicates the quality of its current location. It also has a sensor range within which it can detect the location of other light-emitting glow worms. The essence of the algorithm is that glow worms in the best locations shine most brightly and thereby attract other glow worms towards them. This leads to more intensive searching of regions around the brighter glow worms. Algorithm 12.1 describes the GWSA presented in [353].

Sensor Range

Each glow worm i has two ranges defined around its current location, its *local decision range* r_d^i and its *sensor range* r_s^i, where $0 < r_d^i \leq r_s^i$. The size of

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_12

Algorithm 12.1: Glow Worm Algorithm

```
Select number of glow worms and radius of sensor range;
for each glow worm do
    Place glow worm randomly in search space;
    Initialise each glow worm's luminescence level and the radius of its
    local-decision domain;
end

repeat
    for each glow worm i do
        Update glow worm's luminescence value;
        Find set of each glow worm's brighter neighbours;
        Calculate probability of glow worm moving towards each of these
        neighbours;
        Stochastically select which peer glow worm to move towards;
        Update glow worm's position;
        Update glow worm's local decision range;
    end
until terminating condition;
```

the sensor range remains fixed during the algorithm but the size of each glow worm's local decision range varies as the algorithm executes. Each of these ranges is illustrated in Fig. 12.1.

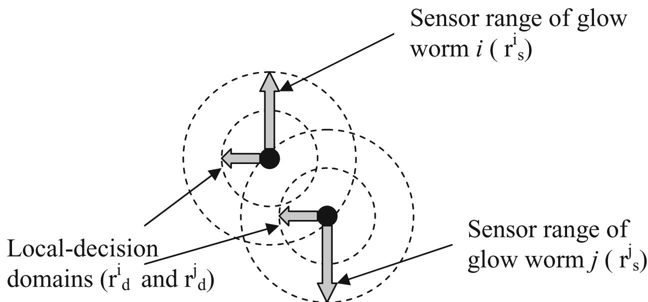


Fig. 12.1. Glow worms i and j have a sensor range with radii of (r_s^i, r_s^j) and each have their own local decision domains defined with radii of (r_d^i, r_d^j). The size of the local decision domain for each glow worm adapts during the course of the algorithm

The local decision range defines a neighbourhood around each glow worm which is used to control how the glow worm moves through the search space. In each iteration of the algorithm every glow worm stochastically selects one

of its neighbours which is brighter than it and moves towards that neighbour. One particular feature of using a local decision range is that the size of the neighbourhoods can be tuned in order to facilitate the use of GWSA for the detection of multiple local optima.

At the start of the algorithm the individual glow worms are initially located randomly in the search space, each is given an equal quantity of luminescence, $\tau_i(0)$, and each is given the same initial local decision range $r_d^i(0)$. Subsequently, in each iteration of the algorithm, the location and the luminescence value of each glow worm are updated. This location update rule implies that the brightest glow worm in the population in a given iteration of the algorithm will not move; hence the algorithm embeds elitism as the best-so-far solution cannot be lost between iterations of the algorithm.

Luminescence Update

The luminescence update depends on the value of the objective function at the current location of the glow worm. The previous luminescence value of the glow worm is updated by adding a component which is proportional to the objective function value at the glow worm's current position. Just as in the case of ant foraging algorithms, an evaporation mechanism is also implemented such that a portion of the luminescence value at the end of the previous iteration is eliminated:

$$\tau_j(t+1) = \max\{0, (1-p)\tau_j(t) + \gamma J_j(t+1)\} \tag{12.1}$$

where p is the evaporation or decay constant $(0 < p < 1)$ and γ is the fixed portion of the value of the objective function $J_j(t)$ at glow worm j's current position which is added to its luminescence value. If the glow worm moves towards ever-improving regions of the search space, its luminescence value will tend to increase (assuming that the value of p is not excessive). On the other hand, if it moves away from a good region, its objective function value will decrease, as will its luminescence.

The term $\tau_j(t)(1-p)$ can be considered as playing a smoothing role, as once $p < 1$ the luminescence value at $t+1$ is influenced by its previous value, embedding an implicit memory of the quality of past locations visited by the glow worm. If $p = 1$ this memory is 'turned off'. The term γ determines the contribution the quality of the glow worm's current location makes to its luminescence update. Once again, setting $\gamma = 0$ turns off this component of the update. Hence, the values of p and γ control the balance between current and past information about the glow worm's trajectory in influencing the search process.

Location Update

The location update is driven by the relative luminescence of nearby peers. A glow worm can only move in the direction of peers which are within its

local-decision radius r_d^i and which are brighter than it. The choice as to which of these neighbours a glow worm tries to move towards is made stochastically. The probability that glow worm i moves towards a brighter neighbour j is given by:

$$P_j(t) = \frac{\tau_j(t) - \tau_i(t)}{\sum_{k \in N_i(t)} (\tau_k(t) - \tau_i(t))} \tag{12.2}$$

where $j \in N_i(t), N_i(t) = \{j : d_{i,j}(t) < r_d^i(t) \text{ and } \tau_i(t) < \tau_j(t)\}$. The parameter t denotes the iteration number, $\tau_i(t)$ is the luminescence of glow worm i at iteration t, and $N_i(t)$ denotes the neighbourhood set of all glow worms which are brighter than i and which are within a threshold distance r_d^i of i (measured using Euclidean distance). By generating a random number and associating this with the calculated probability ranges, a decision is made as to which of its brighter peers to move towards.

For example, suppose glow worm i has three neighbours, a, b and c, all of which are brighter than it (Fig. 12.2) and that $\tau_a(t) = 30$, $\tau_b(t) = 70$, $\tau_c(t) = 30$ and $\tau_i(t) = 10$. Plugging these values into (12.2) produces $P_a(t) = 0.2$, $P_b(t) = 0.6$ and $P_c(t) = 0.2$. If a random variable of 0.31 is drawn from $U(0, 1)$, this falls into the range $0.20 \rightarrow 0.79$; hence, the glow worm i moves towards neighbour b.

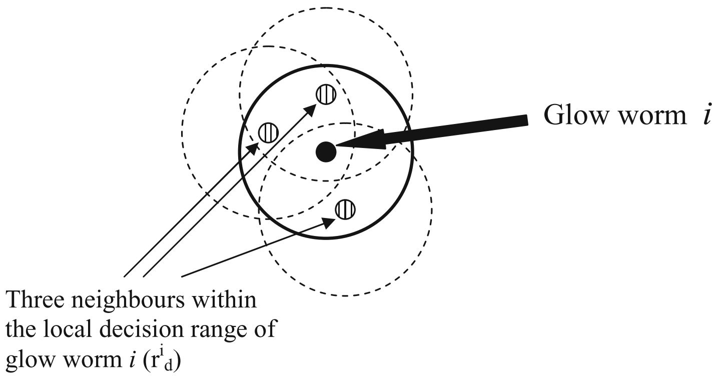


Fig. 12.2. Glow worm i has three neighbours which are brighter than it within its local decision range

The glow worm takes a step of size s from its current location $X_i(t)$ towards the location of this peer $X_j(t)$, leading to a 'line of sight' move towards a neighbouring glow worm,

$$X_i(t+1) := X_i(t) + s d_{ij}(t) \tag{12.3}$$

where $d_{ij}(t)$ is the distance between glow worms i and j, calculated as:

$$d_{ij}(t) = \frac{X_j(t) - X_i(t)}{\|X_j(t) - X_i(t)\|} \tag{12.4}$$

If a glow worm uncovers a very good location it may remain at that location for several iterations of the algorithm. As other neighbouring glow worms converge on that location, the fixed step size ensures that there will be local search around the location of the best neighbourhood glow worm. Although not implemented in the canonical form of the GWSA, the step size could be altered dynamically during the algorithm in order to bias the exploration-exploitation balance of the search process.

Local Decision Range Update

The local decision range, in effect the visual range, for each individual glow worm i does not stay constant during the algorithm, rather it adapts:

$$r_d^i(t+1) = \min\{r_s, \max\{0, r_d^i(t) + \beta(n_t - |N_i(t)|)\}\} \tag{12.5}$$

Looking at (12.5), r_s is the fixed range of a luminescence sensor and this acts as a hard limit on the possible size of the local-decision domain. The parameter β is a constant which parameterises the relationship between the size of $r_d^i(t+1)$ and the number of neighbours a glow worm has, and n_t (a user-chosen parameter) is used to control the number of neighbours. The higher the value of n_t, the larger the local decision range.

Considering specific values of the terms in (12.5), setting $\beta = 0$ implies that the radius of the local-decision domain cannot change over time. If $\beta > 0$ and there are many neighbouring glow worms ($n_t < N_i(t)$), then the value of $r_d^i(t+1)$ is reduced, tending to reduce the number of close-by neighbours in subsequent iterations of the algorithm. On the other hand, if $\beta > 0$ and there are few neighbouring glow worms ($n_t > N_i(t)$), then the value of $r_d^i(t+1)$ is increased, typically leading to an increase in the number of neighbours.

Comparison with Other Swarm Algorithms

A number of parallels can be drawn between the GWSA and other swarm algorithms such as ACO (Chap. 9) and PSO (Chap. 8). All the algorithms are based on population search and they are critically dependent on interagent communication.

Both the GWSA and the particle swarm algorithm have been primarily applied to real-valued problems. The neighbourhood concept in the glow worm algorithm is somewhat similar to that of l^{best} in PSO in that a good location, once uncovered, acts as an attractor for neighbouring individuals. Each glow worm uses its luminescence to communicate information about the objective function's value at its current location to its neighbours. However, l^{best} is implemented using a memory in PSO whereas individual glow worms do not maintain a memory of previously visited locations.

Another distinction between the neighbourhood concepts in both algorithms is that the neighbourhood topology is usually fixed in PSO, whereas it adapts dynamically during the GWSA. This feature makes the GWSA potentially useful for detection of multiple local optima, as well as for general optimisation purposes.

In comparison with ant foraging algorithms, the GWSA also uses the idea of depositing and evaporating an attractant. In the case of ants, the attractant is chemical (pheromones); in the case of glow worms the attractant is light. Despite this similarity, there is a clear distinction in the manner in which the mechanisms operate in each algorithm. In ant foraging algorithms, pheromone values are deposited on a construction graph rather than being associated with particular ants. In contrast, luminescence is a property of each individual glow worm.

Recently, a variation on the glow worm algorithm has appeared called the Firefly algorithm [567, 673], which also adopts bioluminescence.

12.2 Bat Algorithm

A recent addition to the family of foraging algorithms in the natural computing literature is the *bat algorithm* developed by Yang (2010) [674]. As its name suggests it draws inspiration from elements of the foraging processes of bats in order to design an optimisation algorithm.

Although it was documented as long ago as 1793 by the Italian scientist Lazzaro Spallanzani that bats could avoid obstacles whilst flying in the dark, it was only in relatively recent times that the underlying mechanism of *echolocation*, or active *biosonar*, was identified (Griffin 1944 and 1958) [224, 225]. In echolocation, bats transmit pulses of acoustic energy (a bat 'call') and resolve the resulting echoes into an acoustic 'image' of their environment. This is used to detect objects and to locate food resources such as flying insects. In essence, the brain of echolocating bats produces 'images' of their surroundings by comparing the outgoing pulse with the returning echo. In contrast to popular belief, no species of bat is blind and many have good vision [188]. It is speculated that echolocation arose as a result of an evolutionary adaptation to hunt at night rather than compete for food during the day [188, 308].

12.2.1 Bat Vocalisations

Different species of bat produce echolocation calls in various ways, but broadly, they can be split into two groups, *Microchiroptera* (bat species that produce echolocation calls using vocal chords in their larynx) and *Megachiroptera* (bat species which produce echolocation calls by other means such as tongue clicking).

The nature of the echolocation calls produced by bats vary with some being broadband signals (typically of short duration but having a wide range

of frequencies with bandwidths of up to 100 kHz) and others being narrowband signals (typically of relatively long duration and consisting of a narrow range of frequencies with a bandwidth of circa 5 kHz) [18, 306]. Narrowband signals are good for ranging distant items (or prey) and broadband signals are well adapted for the fined-grained localisation of items. This leads to a phenomenon whereby as insectivorous bats home in on their aerial prey, they switch from narrowband to broadband signals. The broadband signals are then emitted at an increasingly rapid rate as the bat approaches their prey target, resulting in what is known in the bat literature as the 'feeding buzz'.

As bat call echoes (reflections from objects including prey) are strongly attenuated while travelling in air, bats can hear the calls produced by other bats from much further away that they can detect echoes from their own calls. Individuals approaching feeding groups, and eavesdropping on their calls, can therefore increase their effective prey detection range between 10 to 50 times, depending on species, over that provided by their own echolocation ability. Eavesdropping therefore creates a mechanism akin to a *recruitment* process wherein successful foragers implicitly communicate information which allows other bats to home in on resource rich locations.

12.2.2 Algorithm

Algorithm 12.2 describes a variant on the canonical bat algorithm as presented in [674]. In essence, the virtual bats in the algorithm commence by flying randomly in the search space, and each bat has a different initial call frequency, loudness and rate of pulse emission. The bats move around the search space, using a mix of social information and random search. Search is also intensified around good solutions using a local search step.

The algorithm consists of a number of discrete elements, namely: initialisation; generation of new solutions; stochastic local search around existing good solutions combined with the stochastic generation of randomly-created solutions; and finally, an updating of the location of the current best solution.

The objective function is denoted by $F : \mathbb{R}^d \longrightarrow \mathbb{R}$, which we seek to minimise. At each iteration (timestep) t, each bat i has a location vector $x^i(t) = (x^i_1(t), \ldots, x^i_d(t))^T$ and a velocity vector $v^i(t) = (v^i_1(t), \ldots, v^i_d(t))^T$ in the d-dimensional search space $\mathbb{R}^d$. The current best location of all bats is denoted by x^*. The general relationship between wavelength λ and frequency f for echolocation calls in air is $\lambda = v/f$, where the speed of sound v is approximately 340 metres per second. As noted above, bats can adjust the wavelength or equivalently, the frequency of their calls. In the algorithm the frequency is varied.

Generate New Solution

As the bat searches, the frequency of its echolocation calls at time step t is generated using:

Algorithm 12.2: Bat Algorithm

Define an objective function $F(x)$ where $x = (x_1, \ldots, x_d)^T \in \mathbb{R}^d$;
Set $t := 0$;
for *each bat* $i = 1, 2, \ldots, n$ **do**
  Randomly initialise the location $x^i(0)$ and velocity $v^i(0)$;
  Define pulse frequency $f^i(0) \in [f_{\min}, f_{\max}]$ at $x^i(0)$;
  Initialise pulse rate $r^i(0)$;
  Initialise loudness $A^i(0)$;
end
Let $x^* :=$ the x^i with best fitness;
while $t <$ *maximum number of iterations* **do**
  for *each bat* $i = 1, 2, \ldots, n$ **do**
    Adjust frequency to $f^i(t+1)$ using (12.6);
    Update velocity to $v^i(t+1)$ using (12.7);
    Generate new solution $x^i_{\text{new}}(t+1)$ for bat i by updating location using (12.8);
    if rand $> r^i(t)$ **then**
      Generate a local solution around x^* and store it in $x^i_{\text{new}}(t+1)$;
    end
    Generate a random solution in a bounded range about $x^i(t)$ and store it in $x^i(t+1)$;
    if (rand $< A^i(t)$ **and** $F(x^i_{\text{new}}(t+1)) < F(x^i)$) **then**
      Set the location of bat i, $x^i(t+1) := x^i_{\text{new}}(t+1)$;
      Increase $r^i(t)$ and reduce $A^i(t)$;
    end
  end
  Rank the bats in order of fitness and update the location of the best solution found by the algorithm so far (x^*) if necessary;
  Set $t := t + 1$;
end
Output best location found by the bats;

$$f^i(t) = f_{\min} + (f_{\max} - f_{\min})\beta \tag{12.6}$$

where $f_{\min}$ and $f_{\max}$ are the minimum and maximum frequencies of bat calls respectively, and β is randomly drawn from a uniform distribution on $[0, 1]$. Initially, each bat i is assigned a random frequency $f^i(0)$ drawn uniformly from $[f_{\min}, f_{\max}]$.
The velocity update of each bat in each iteration of the algorithm is given by

$$v^i(t+1) = v^i(t) + (x^i(t) - x^*) f^i(t+1) \tag{12.7}$$

and the position update is given by:

$$x^i_{\text{new}}(t+1) = x^i(t) + v^i(t+1). \tag{12.8}$$

In essence therefore, the value of $f^i(t)$ controls the pace and range of the movement of bat i in each iteration and the precise value of $f^i(t)$ is subject to a random influence due to β.

Local Search

The local search component of the algorithm is operationalised as follows. If the condition rand $> r^i(t)$ is met for an individual bat i (note that rand is drawn from a uniform distribution whose range depends on the scaling of r as discussed below), the current best solution (or a solution from the set of the better solutions in the population) is selected, and a random walk is applied to generate a new solution. The random walk is produced using:

$$x^i_{\text{new}} = x^* + A(t)\epsilon \tag{12.9}$$

where x^* is the location of the best solution found so far, ϵ is a vector where each component results from a random draw on $[-1, 1]$, and $A(t)$ is the average loudness of all bats in the population at time step t.

The rate of local search during the algorithm depends on the values of $r^i(t)$ (the rate of pulse emission) across the population of bats. If this rate increases, the degree of local search activity will decrease. The average loudness $A(t)$ will tend to decrease as the algorithm iterates (discussed below), hence the step sizes in the local search will reduce to become finer-grained. In order to enhance the explorative capability of the algorithm, the local search step is complemented by a random search process.

In real world bat foraging, the loudness of calls reduces when a bat approaches a prey target, while the rate of pulse emission from the bat increases. This can be modelled using:

$$A^i(t+1) = \alpha A^i(t) \text{ and } r^i(t) = (1 - \mathrm{e}^{-\gamma t})r^i(0) \tag{12.10}$$

where α (similar in concept to a cooling coefficient in simulated annealing) and γ are constants. For any value $0 < \alpha < 1, \gamma > 0$, the following is obtained:

$$A^i(t) \to 0, \;\; r^i(t) \to r^i(0), \;\; \text{as } t \to \infty. \tag{12.11}$$

The loudness and pulse emission rate of individual bats are only updated if a solution is found by a bat which is better than its previous solution. The update process is stochastic as it only takes place if a random draw from a uniform distribution, bounded by the maximum and minimum allowable values for loudness, is less than $A^i(t)$. As the algorithm iterates, the values for loudness will tend to decay to their minimum, hence reducing the probability that an update will occur. A side effect of this process is that it makes the algorithm less ‘greedy’.

Parameters

In setting parameters for the algorithm, [675] suggests values of $\alpha = \gamma = 0.9$, with each bat having a different initial random value for loudness $A^i(0) \in [1, 2]$, with $A_{\min} = 0$. A value of $A_{\min} = 0$ indicates that a bat has just found prey and therefore stops emitting sound. Initial values for pulse rate $r^i(0) \in [0, 1]$, if using (12.10), and values of $f_{\min} = 0$ and $f_{\max} = 100$ (these values are domain specific and each bat is assigned a random value in this range at the start of the algorithm) are also suggested. The performance of the algorithm will be critically impacted by these parameters, and trial and error will be required in order to adapt the algorithmic framework to a specific problem.

Yang and Gandomi (2012) [675] point out that there is some similarity between the bat algorithm as outlined above and PSO as, if the loudness is set to 0 ($A^i = 0$) and the pulse rate is set to 1 ($r^i = 1$), the bat algorithm becomes similar to a variant of PSO algorithm (without p_i^{best}), since the velocity updates are governed by prior-period velocity and a noisy attraction to g^{best}.

12.2.3 Discussion

The bat algorithm has produced competitive results on a series of benchmark optimisation problems. Despite the relatively recent development of the algorithm it has also been successfully used in a variety of applications, encompassing constrained optimisation [207, 675], multiobjective optimisation [69, 451], binary-valued representations [431], and clustering [544]. A detailed review of applications of the bat algorithm is provided in [676].

There are multiple open areas of research concerning bat inspired algorithms. The canonical version of the algorithm does not explicitly include a personal detection mechanism (i.e., a bat can 'see' any prey in an arc around its head) or a personal memory as to good past foraging locations. These mechanisms could be included in a variation of the canonical algorithm. Another area of potential research is to embed a more complex processing of social influences (feeding buzzes) in the bat algorithm. Plausibly, a bat will be more influenced on hearing many feeding buzzes coming from a small area (indicating a heavy concentration of prey in that area) than a solitary feeding buzz coming from elsewhere.

Apart from the processes of echolocation in flight, it is also noted that there is research in the foraging literature which claims that information transfer between bats can occur at roost sites [658]. This study suggested that evening bat species at nursery colonies transfer information by following each other to feeding sites, with unsuccessful foragers leaving a roost and following previously successful foragers. Similar findings were reported for *Phyllostomus hastatus*, a frugivore bat species [143, 659]. Such information transfer at roosts bears parallel to the colony-based information transfer of other central place

foragers such as honey bees, and the mechanisms of this process could inspire the design of an optimisation algorithm.

12.3 Fish School Algorithm

Another example of grouping or swarming behaviour is provided by fish shoaling and schooling. Shoaling occurs when fish group together, with schooling arising when these aggregations move in a coordinated manner. These behaviours are common with about half of all fish displaying shoaling behaviour at some stage in their life cycle, and some species such as tuna, herrings and anchovy are obligate shoalers and shoal for their entire life cycle. Fish shoals can be very large with group sizes of up to a billion herring being reported [495].

Shoaling behaviour carries certain costs including oxygen and food depletion in the vicinity of the shoal, so the behaviour must also offer some benefits to fish as gregarious behaviour would only have evolved if it enhanced survivability. These benefits include enhanced defence against predators (the multiple eyes in a shoal provide a higher vigilance and the many moving targets overload the visual channel of predators), increased hydrodynamic efficiency in moving through water and enhanced foraging success [598]. This latter benefit has inspired the development of a series of search algorithms, drawing on a fish school metaphor.

In common with the population of 'agents' in other swarm inspired algorithms, shoals of fish have no leader and their aggregate behaviour is the result of individuals acting on local information (including information gleaned from the behaviour of their neighbouring conspecifics), leading to emergent self-organisation and problem-solving capabilities [598].

One of the better-known fish school algorithms, *Fish School Search* (FSS) was developed by [38]. In this algorithm, fish swim (search) to find food (candidate solutions) in an aquarium (search space). Unlike PSO, no explicit memory of the best locations found to date in the search process is maintained; rather the weight of each fish acts as a memory of its individual success to date during the search process, and promising areas in the search space can be inferred from regions where larger ensembles of fish gather. The 'barycentre' (or the location of the 'centre of gravity') of the whole school of fish is considered to provide a proxy for this. In designing the algorithm, the authors considered six design principles to be important, namely:

i. simple computation at each agent,
ii. creative yet simple means of storing distributed memory of past computations in the fish weights,
iii. local computations (preferably in small radiuses centred on each fish),
iv. simple communication between neighbouring individuals,
v. minimum centralised control (preferably only the barycentre information is exchanged), and

vi. simple diversity-generating mechanisms among individuals.

Considering the impact of each of the above on a resulting algorithm, item (i) reduces overall computation cost, item (ii) allows adaptive learning, items (iii), (iv) and (v) keep computation costs low and allow some local knowledge-sharing promoting convergence, and item (vi) speeds up search and is a useful mechanism in a dynamic setting.

12.3.1 Fish School Search

Fish School Search (FSS) [38, 39] has two primary operators which are inspired by fish behaviour. These are feeding, inspired by natural instinct of fishes to feed (food here is a metaphor for the evaluation of candidate solutions in the search space), and swimming, which aims at mimicking the coordinated movement of fish in a school and guiding the actual search process. The operationalisation of each of these behaviours in the algorithm is discussed below.

Individual Movement

A swim direction is randomly chosen. Along each dimension of the search space, a variable r randomly drawn from a uniform distribution $U(-1,1)$ is multiplied by a fixed step size ($\mathrm{step_{ind}}$). If the food density (fitness) at the resulting location is greater than at the fish's current location, and the new location is within the feasible search space, then the fish moves to the new location (12.12); otherwise no move is made.

$$x_j(t+1) = x_j(t) + r\mathrm{step_{ind}} \tag{12.12}$$

In order to promote exploitation as the algorithm progresses, the parameter $\mathrm{step_{ind}}$ decreases linearly as the algorithm iterates (up to max iter).

$$\mathrm{step_{ind}}(t+1) = \mathrm{step_{ind}}(t) - \frac{(\mathrm{step_{ind}^{initial}} - \mathrm{step_{ind}^{final}})}{\max \mathrm{iter}} \tag{12.13}$$

Feeding

As a fish moves in the search space, it gains 'weight' in proportion to its success in finding food (fitness). Weight is gained if the fitness at the current location of a fish is better than the fitness at its previous location; weight is lost otherwise. The value of weight is constrained to lie in the range 1 to max *value* and all fish are initialised to a weight of $\frac{\max\ value}{2}$ at the start of the algorithm. The starting positions for all fish are chosen randomly at the start of the algorithm:

$$w_i(t+1) = w_i(t) + \frac{\Delta f_i}{\max(\Delta f)} \tag{12.14}$$

where $w_i(t)$ is the weight of the fish i, Δf_i is the difference of the fitness between the previous and the new location, and $\max(\Delta f)$ is the maximum fitness gain across all the fish. As discussed above, $\Delta f_i = 0$ for any fish which do not undertake individual movement in that iteration.

Collective-Instinctive Movement

After the individual movement step is completed for all fish (as above, not all fish will actually move), a weighted average of all movements in the school is computed. Fish with successful moves influence the resulting direction of movement of the entire school and fish which did not update their position under the individual movement step are ignored.

$$m(t) = \frac{\sum_{i=1}^{N} \Delta x_i \Delta f_i}{\sum_{i=1}^{N} \Delta f_i} \tag{12.15}$$

When the overall direction is computed, all fish are repositioned, including those which did not undertake an individual movement:

$$x_i(t+1) = x_i(t) + m(t) \tag{12.16}$$

Collective-Volitive Movement

After the individual and collective-instinctive movements, a final positional adjustment is made based on the overall weight variation of the school as a whole. If the school is putting on weight collectively (it is engaging in successful search) then the radius of the school is contracted in order to concentrate the search. Otherwise the radius of the school of fish is increased in order to enhance the diversity of the school. The expansion or contraction occurs by applying a small step drift to the position of every fish in the school with respect to the location of the barycentre for the entire school.

First, the barycentre b (centre of mass) needs to be calculated:

$$b(t) = \frac{\sum_{i=1}^{N} x_i w_i(t)}{\sum_{i=1}^{N} w_i(t)} \tag{12.17}$$

If the total weight of the school has increased during the current iteration, all fish update their position (including those which did not undertake an individual movement) using:

$$x_i(t+1) = x_i(t) - \text{step}_{\text{vol}} \text{rand}(0,1) \frac{(x(t) - b(t))}{d(x(t), b(t))} \tag{12.18}$$

or, if total weight has decreased:

$$x_i(t+1) = x_i(t) + \text{step}_{\text{vol}} \text{rand}(0,1) \frac{(x(t) - b(t))}{d(x(t), b(t))} \tag{12.19}$$

where $d()$ is a function which returns the Euclidean distance between x_i and b. The parameter step_{vol} is a predetermined step size used to control the displacement from/to the barycentre.

Looking at each of the swimming mechanisms above, individual movement is a mix of a stochastic search and personal cognition (as the fish needs to be able to assess the worth of each location). Collective-instinctive movement embeds an element of social learning, as fish are assumed to move based on the success of their conspecifics in finding new high-quality feeding locations. Collective-volitive movement embeds an exploration-exploitation adaptation based on the collective success of the school. Therefore this also embeds an element of social learning.

Algorithm 12.3: Fish School Search Algorithm

```
Initialise all fish randomly;
repeat
    for each fish in turn do
        Evaluate fitness function;
        Implement individual movement;
        Implement feeding;
        Evaluate fitness function;
    end
    for each fish in turn do
        Implement instinctive movement;
    end
    Calculate barycentre;
    for each fish in turn do
        Implement volitive movement;
    end
    Update step size;
until termination condition is satisfied;
Return the location of the best solution found;
```

12.3.2 Summary

The FSS can be considered as a swarm algorithm as the search process embeds bottom-up learning via information flow between searching agents. The results from the algorithm have been found to be competitive with those from other swarm algorithms such as PSO. Other work which has developed FSS includes [39], which analyses the importance of the swimming operators and shows that all the operators have important impact on the results obtained; [295], which examines a variety of alternative weight update strategies for FSS; [96],

which develops a PSO-FSS hybrid called 'volitive clan PSO'; and [373], which describes a parallel GPU implementation of FSS.

Other approaches to the design of fish school algorithms have been taken. Another related strand of literature concerns the Artificial Fish Swarm Algorithm (AFSA) [367, 259]. This embeds a number of fish behaviours including preying, swarming, and following so that the behaviour of an artificial fish depends on its its current state, its local environmental state (including the quality of its current location), and the states of nearby companions. A good review of the recent literature on AFSA, including a listing of some applications, is provided in [436] and the reader is referred there for further information.

12.4 Locusts

Another interesting example of foraging behaviour is exhibited by some species of locust. Locusts are a member of the grasshopper family of insects. Perhaps the best-known, and indeed infamous, species of locust is the desert locust (*Schistocerca gregaria*), which inhabits some 60 countries encompassing Africa, Asia and the Middle East. Given that a locust can eat its own body weight of vegetation in a single day, a large swarm of locusts, which in extreme cases can include billions of individuals, can be highly destructive. Locust plagues have even resulted in famines.

A desert locust lives for three to five months, going through three primary life cycle stages. Initially, female locusts lay eggs which take a few weeks to hatch into wingless larvae or nymphs (also called 'hoppers'). These in turn mature over about a month via five or six moults, eventually resulting in mature adults which are capable of flight. Adult locusts can live for several months, depending on environmental conditions.

One curious aspect of desert locust behaviour is that it exhibits two distinct social states, being either solitary or gregarious depending on external stimuli. Typically, locusts live in a solitary fashion but certain environmental conditions can encourage them to congregate en masse and become gregarious. For example, desert locust populations tend to flourish after rainstorms cause plant growth and create good conditions for egg laying. As drought returns and food resources become scarce for the now enlarged population, locusts congregate in the remaining food patches. As they crowd together and the locust population becomes densely packed, sensory stimuli including the touch, smell and sight of other locusts cause the serotonin levels of individual locusts to rise [17], in turn resulting in the triggering of a gregarious state. In this state the locusts give off a pheromone that causes them to be attracted to each other and their colouring also changes. As smaller groups coalesce, huge swarms can form.

Adult locusts are highly mobile and can travel long distances in search of food resources. In a daily cycle, locusts will roost overnight on vegetation,

before moving down to ground level to bask and warm up in the morning. By mid-morning, the locust swarm takes to the air and flies for several hours, typically landing and settling an hour before sunset. Locusts tend to fly with the wind and can cover over 100 km in a day, sometimes reaching heights of up to 1,800 metres [613]. Some notable long-distance locust migrations have occurred, including, for example, from West Africa to the Caribbean, a distance of 5,000 km, in about ten days in 1988.

12.4.1 Locust Swarm Algorithm

Several aspects of locust behaviour could potentially inspire the design of optimisation algorithms, but thus far, attention has only been focussed on the 'devour and move on' behaviour which characterises locust plagues. The *locust swarm algorithm* which was developed by Chen (2009) [103, 104] is loosely inspired by this idea and also draws inspiration from a variant of particle swarm optimisation (PSO) called 'Waves of Swarm Particles (WoSP)' [269].

The locust swarm algorithm is specifically designed for application to multimodal problems as it uses multiple swarms in order to explore wide expanses of the search space. The essence of the algorithm is that it explores multiple optima using a 'devour and move on strategy' [103]. The algorithm is non-convergent, making it useful for the exploration of non-globally-convex search spaces (i.e., search spaces which are *deceptive*).

The algorithm blends a coarse search phase with a greedy search phase. The coarse search is operationalised using a PSO variant and this is intended to generate good starting points for the greedy search process. After the greedy search process has 'devoured' a region around these starting points and found the local optimum, scouts are then deployed in order to find new promising locations from which to initiate a new search. The process is iterated over a series of searches and the best result found is returned as the final result.

Pseudocode for the algorithm, drawn from [103], is provided in Algorithm 12.4. The workings of the algorithm can be divided into two stages. In the first stage, a large number R of random points are generated in the search space and the best S of these are selected to form an initial swarm. Next, a variant on the standard PSO algorithm is run for a set number n of iterations, and the resulting g^{best} is obtained. In this phase of the algorithm, the objective is to generate a coarse-grained search process, with little convergence of the swarm of particles (locusts). In order to achieve this, the particle position and velocity update processes of the canonical PSO (Chap. 8.2) are altered [103]. The position update is generated from:

$$x_{id}(t+1) = x_{id}(t) + 0.95 v_{id}(t+1) \tag{12.20}$$

and the velocity update is given by:

$$v_{id}(t+1) = M v_{id}(t) + G(g_d^{\text{best}}(t) - x_{id}(t)) \tag{12.21}$$

Algorithm 12.4: Locust Swarm Algorithm

Generate R points in the search space at random;
Select the best S of these to form the initial particle swarm;
Assign a random velocity to each of these particles;
Run a particle swarm algorithm for n iterations using update equations (12.20) and (12.21);
Locally optimise the resulting g^{best} using a local search algorithm to get x^{opt};

for *swarms* $2, \ldots, N$ **do**
  Stochastically generate r points around the last x^{opt};
  Select the best S of these to form a particle swarm;
  Assign initial outward velocity to each particle using (12.22);
  Run the particle swarm algorithm for n iterations using update equations (12.20) and (12.21);
  Optimise the resulting g^{best} found by this swarm using a local search algorithm in order to produce a new x^{opt};
end
Return the location of the best solution found;

Hence, in each iteration, the location of particle x_i is updated along each of its d dimensions by applying a velocity vector to it (12.20), where the magnitude and direction of the velocity vector (generated using 12.21) is a function of the prior period velocity of that particle and the particle's position relative to the g^{best} of the swarm. Unlike the canonical version of PSO, the velocity update equation (12.21) does not include a p^{best} term and the momentum coefficient is given much greater weight than that assigned to g^{best} (values of $M = 0.95$ and $G = 0.05$ are used in [103]). This formulation of the velocity update promotes continued exploration of the search space rather than encouraging exploitation of information gained thus far during the search process, thereby producing a coarse-grained search. At the end of the PSO phase, the g^{best} solution is (locally) optimised using a local search method such as gradient descent.

In the second stage of the algorithm, the best location found in the first stage acts as the launch point for the second swarm. For each of the $2, \ldots, N$ swarms in turn, a total of r points (or scouts) are generated from the optimum point found by the previous swarm by applying a mutation operator to that point. In [103] the mutation step is operationalised so that newly generated points must be at least a threshold distance from the previous optimum point in order to ensure that there is a suitable balance between exploitation of the information contained in the previous optimum and exploration of other regions of the search space.

Unlike the first stage of the algorithm, the initial velocities of the particles in each swarm are not assigned randomly. Instead the velocity is set in order

to promote exploration of the search space by launching the particles away from the optimum point found by the last swarm. Hence, in (12.22) the initial ($t = 0$) velocity for each particle i on dimension d (i.e., $v_{id}(0)$) is set using parameter $p = 0.8$, the location of particle i relative to the position of the optimal point found by the previous swarm, $x_{id}(0) - x_d^{\text{opt}}(0)$, and a stochastic component r_d which is chosen depending on the scaling of the search space on dimension d [103].

$$v_{id}(0) = p(x_{id}(0) - x_d^{\text{opt}}(0)) + r_d \tag{12.22}$$

After the velocity and location of each particle in the swarm are initialised, the update equations (12.20) and (12.21) are applied. At the conclusion of n iterations of the PSO for that swarm, the resulting x^{opt} is locally improved and becomes the optimum point for the next swarm launch. At the conclusion of all $N - 1$ swarms, the location of the best result found by the algorithm is returned.

The foraging behaviour of locusts offers another interesting example from the biological world of a phenomenon which can be used to inspire the design of optimisation algorithms, and the work of Chen [103, 104] opens the door to this area. Whilst the locust swarm algorithm relies heavily on a modified particle swarm framework, it should be noted that the modifications result in a blend of nonconvergent and stochastic search which differs from a canonical PSO or indeed pure random search. Further work is required in order to fully determine the utility and computational efficiency of the resulting algorithm. It is also evident that scope exists to design other locust behaviour inspired algorithms which embed more complex models of swarm communication and decision making.

12.5 Summary

The core of most swarm-based algorithms lies in the mechanisms of social communication and the resulting diffusion of information between the individual agents in the swarm. In this chapter we introduced four emerging families of algorithms which are not yet as fully developed or explored as ant or honey bee algorithms, providing interesting examples of the diversity of natural computing algorithms that can be developed from real-world social behaviours.

Part III

Neurocomputing

13

Neural Networks for Supervised Learning

Quite commonly, we are faced with the problem of taking a vector $x = (x_1, \ldots, x_n)$ of inputs and producing a vector $y = (y_1, \ldots, y_m)$ of outputs. For example, in a classification problem, the $x_1, \ldots, x_n$ may be characteristics of an item to be classified, and the corresponding output could be a single y, the class label for that item. Hence, the task is to uncover a function g such that $y = g(x)$. Of course, the mapping g may be nonlinear. Generally, we are satisfied if we can approximate the 'true' function g sufficiently accurately by a function f of some particular form, e.g., polynomial in several variables, where f has coefficients or parameters whose values we need to determine. This is known as the *function approximation problem.*

Different approaches to finding f (e.g., regression modelling) amount to assuming particular functional forms for f. Typically, we need a dataset of examples, called *training data*, to allow us to find a suitable mapping f. This problem is complicated for at least two reasons:

i. we try to derive f from a *finite* dataset; this generalisation is an inherently difficult problem;
ii. the $y_1, \ldots, y_m$ or $x_1, \ldots, x_n$ may be noisy.

In this chapter we introduce a family of algorithms, artificial neural networks, which can be used for function approximation.

13.1 Biological Inspiration for Neural Networks

The human brain may be considered as a vast, interconnected parallel-processing system. It receives inputs from its environment, can encode and recall memories, and can integrate inputs to produce a thought or an action (an output). The brain has the capability to recognise patterns and to predict the likely outcome of an event based on past learning. The brain consists of about 100 billion nerve cells or *neurons*. Each of these is connected to

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_13

a few thousand other neurons and is constantly receiving electrical signals from them along fibres, called dendrites, that emanate from the cell body (Fig. 13.1). If the total signal coming into an individual neuron along all its

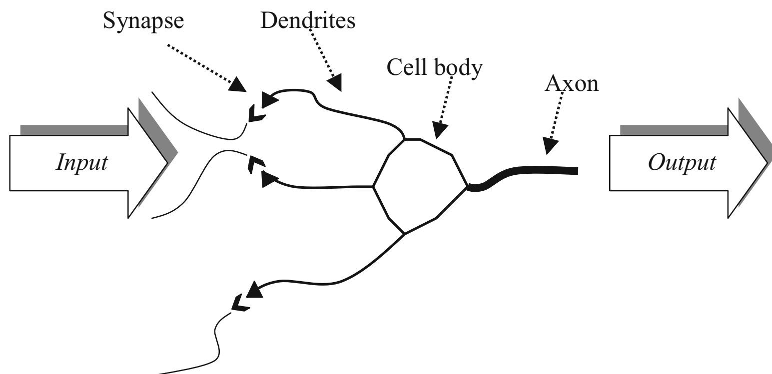


Fig. 13.1. A simplified diagram of a nerve cell

dendrites exceeds a threshold value, the neuron fires and produces an outgoing signal along its axon which is in turn transmitted to other neurons.

Connections between neurons occur at synapses and signals cross the synaptic gap by means of a complex electrochemical process. When a synapse's electrical potential is raised sufficiently due to signals from the axon, it releases chemicals known as neurotransmitters, which in turn chemically activate gates on the dendrite, which allow charged ions to flow. Each dendrite can have multiple synapses acting on it, with some of these serving to amplify signals along the dendrite and others serving to inhibit (weaken) them.

13.2 Artificial Neural Networks

Artificial neural networks (ANNs), usually shortened to Neural Networks (NNs), are a family of computational methodologies whose design is inspired by stylised models of the workings of the human brain and central nervous system. NNs can be used for a wide variety of tasks including the construction of models for the purposes of prediction, clustering and classification.

13.2.1 Neural Network Architectures

At a basic level, a NN is just a network of simple processing units called *nodes* or *neurons*. Signals or influences can only pass in one direction along a

given connection (also called *arc* or *edge*). Furthermore, the effect of the signal along a connection may be adjusted by a weight on that edge. This means that NNs are *weighted directed* graphs. Each node processes the combination of weighted signals presented to it, in a manner that varies according to the type of NN.

NNs can exhibit complex emergent global behaviour, determined by the connections (arcs) between the processing units (nodes) and network parameters (weights). NNs are inductive, *data-driven* modelling tools which do not require an explicit a priori specification of the relationship between model inputs and outputs. They have the ability to *learn* in the sense that they use a set of data (observations), so as to find a function from a predetermined class $\mathcal{C}$ of functions which solves the problem at hand in an optimal — or at least feasible — way. The class $\mathcal{C}$ is restricted to functions which may be expressed in terms of a *basis* of certain building block functions: the choice of basis function is one of the distinguishing characteristics of a NN. NNs provide a very general and powerful framework for representing nonlinear mappings from several input variables to several output variables, where the form of the mapping is controlled by a number of parameters (whose values may be adjusted). In NN terms, the unknown function parameters sought are usually called *weights*, since they are weights on graph edges. Learning the weights is also called *training* the NN.

The capabilities of NNs stem from their connection architectures, the processing that takes place at each node, and the way that the network learns from data. NNs can be differentiated from each other along four main axes:

i. connection topology,
ii. basis function,
iii. training method, and
iv. learning algorithm.

The *connection topology* defines how the processing units or nodes are connected to each other.

The *basis function* defines what processing each node carries out on the combination of all its inputs, in order to generate its output value.

The *training method* is concerned with how the NN learns. In *supervised learning*, the NN is provided with training data (input data for which the output is already known); over multiple iterations, the learning algorithm discovers how to link the inputs to the associated known outputs, and how to predict the correct output for inputs not given in the training data. Here, the aim is to find a function that matches or *fits* the training data (or at least minimises an error measure such as least squared error). Examples of supervised learning tasks include classification and regression. In contrast, *unsupervised learning* occurs when the NN is not provided with outputs, but rather is left to uncover patterns in the input data without a priori information as to what these patterns may be. An example of unsupervised learning would

be the uncovering of previously unknown patterns in databases of customer information, such as clustering, segmentation or density estimation.

The *learning algorithm* defines how error is measured during the training process, and how the NN model is updated during training in order to reduce this error.

Many forms of NNs can be developed by making different choices for the above items.

In this chapter, we concentrate on the two most common forms of supervised NN, the multilayer perceptron (MLP) and the radial basis function network (RBFN). We also introduce support vector machines, which, although not derived from a neurocomputing metaphor, bear similarities to radial basis function networks. In Chap. 14, we examine self-organising maps, the most popular unsupervised NN algorithm, and adaptive resonance theory, a family of NNs most of which are unsupervised but some of which may be used in supervised learning. We conclude this part of the book with a chapter on neuroevolutionary hybrids, illustrating how evolutionary processes can be applied to generate NNs (Chap. 15).

13.3 Structure of Supervised Neural Networks

A supervised NN aims to address the function approximation problem by building up an internal model of a function which is a good fit to the training data provided. The model has the form of a weighted directed graph, where each node constructs a weighted sum (linear combination) of building block (or basis) functions (from the nodes of the previous level, which feed into this node). The NN parameters are the arc weights, which are learned from provided training data.

We discuss in detail below the two most common supervised NNs, Multilayer Perceptrons (MLPs) and Radial Basis Function (RBF) networks (RBFNs). In this section, we note that they have much more in common than might be gleaned from a quick review of the NN literature. The MLP and RBFN architectures are actually quite closely related, in that each is a *basis function network*, that is, a network of nodes, all nodes having identical basis functions which convert the input data to an output. In fact, all NNs may be viewed as *basis function networks* [58]. Recall that a *basis* of a vector space V is a set B which is both linearly independent and spans V. If V is a vector space of functions, typically infinite-dimensional, the definition of basis is more complicated (because infinite sums need not converge) but under certain conditions can be shown to work. Then a basis of V consists of basis functions, i.e., building blocks: we can write other functions as linear combinations (that is, weighted sums) of the basis. Basis functions should provide maximum flexibility in the contours they can represent, together with a small number of tunable parameters. Large NNs with simple nodes may have the same power as small networks with more complex nodes.

Figure 13.2 shows a standard *feedforward* network architecture typical of both MLP and RBFN. There are multiple layers or levels of nodes, including

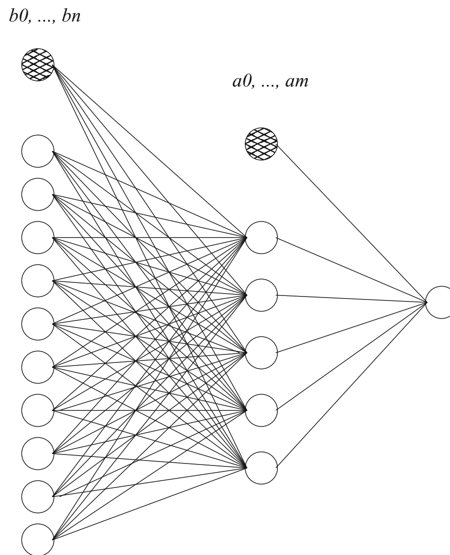


Fig. 13.2. An example of a three-layer ANN (a tripartite graph), e.g., a feedforward MLP or RBFN. All arcs go from left to right. The leftmost layer, $b_0, b_1, \ldots, b_n$, of nodes is the input layer, the middle layer, $a_0, a_1, \ldots, a_m$, is a hidden layer, and the rightmost node is the output node. The shaded nodes are bias nodes

an *input layer*, an *output layer* and one or more *hidden layers* that are not directly visible or modifiable from outside the NN. The input layer serves as a holding layer for the data being input to the NN. Nodes in the final hidden layer are connected in turn to an output layer which represents the processed output from the model. Hidden and output nodes process the data using activation and transfer functions (Sect. 13.3.1). Input layer nodes do not carry out real calculations, since all they do is distribute the inputs to hidden layer neurons. Also, offsets or biases can be fed into each hidden or output neuron. The inclusion of a bias node serves a similar purpose as the inclusion of a constant term in a regression equation and also has a parallel with the idea of a threshold value in biological neuron firing processes. The input value of the bias node is usually held constant at 1, and is automatically rescaled as necessary as the weights on its outgoing connections change.

NNs provide an example of parallel, distributed computing, since each hidden layer node acts as a local processor of information yet also acts concurrently and in parallel with the other nodes in its layer. Although the processing which takes place at individual nodes is relatively simple, the linkage of these

nodes gives rise to emergent global capabilities for the network, permitting complex nonlinear mappings from input to output.

13.3.1 Activation and Transfer Functions

In the NN context, each basis function $h : \mathbb{R}^n \longrightarrow \mathbb{R}$ at a node is actually a composition of two functions:

i. an *activation*[1] (or *likeness* or *similarity*) function $a : \mathbb{R}^n \longrightarrow \mathbb{R}$, which measures similarity between the current input and the NN weight parameters, composed with
ii. a *transfer* function $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$, of saturation type in the MLP case, of radial symmetry type in the RBFN case.

That is, $h = \sigma \circ a$, with the output of the activation function being the input to the transfer function. Every NN transfer function is a function of one argument (a scalar), that argument being constructed by the activation function out of several input variables. However, MLPs, RBFNs, etc., have different particular forms of the transfer and activation functions.

Typically, the transfer function is a squashing function, so as to keep the output values within specified bounds. The three major possibilities are:

i. a sigmoidal nonlocal transfer function (saturation type, used in MLPs: see Sect. 13.4.1);
ii. a transfer function localised around a single centre (radially symmetric type, used in RBFNs: see Sect 13.5.2);
iii. a semicentralised transfer function that has either many centres or hard-to-define centres.

Examples of activation function include the following. Each of these measures how 'alike' are the input vector and the vector of weights on the arcs into the current node.

- A weighted combination of the input signals:[2] this is the most common activation function, sometimes called a *fan-in* function; it is used in MLPs (Sect. 13.4). Suppose that a given node j receives input vector $x = x_0, \ldots, x_n$ along arcs whose weights make up the *weight vector* $w_j = (w_{0j}, w_{1j}, \ldots, w_{nj})$. Then the activation is

$$a(x, w_j) := \sum_{i=0}^{n} x_i w_{ij} = x^t w_j = x \cdot w_j \text{ (inner or dot product).} \qquad (13.1)$$

[1]There appears to be no standard term in the NN literature for the input to the transfer function; in the MLP context, [170] calls it the *net activation*, net_j, while [58] simply calls it the 'sum' a_j. Furthermore, some texts give the name 'activation function' to the transfer function. The activation functions are linear in the case of *linear* NN models.

[2]This approach is biologically inspired, mimicking the inputs from multiple dendrites to a neuron.

For fixed w_j, this gives hyperplanes as the contours of $a(x, w) = \text{const}$, illustrated in Fig. 13.6.

- A distance (norm[3]) based activation function: $a(x, c) := \|x - c\|$, used to calculate distance from x to a 'centre' vector c. Here the weights are viewed as the components of the centre c. This is used in RBFNs.
- A linear combination of the above two approaches: $a(x, w_j) := \alpha x \cdot w_j + \beta \|x - w_j\|$.

The fundamental difference between MLPs and RBFNs is the activation function, i.e., the way in which hidden units combine values coming from preceding layers in the network:

- an MLP uses a dot product, a weighted combination of the input signals, giving a projection; this means it considers how alike in direction are the input vector and weight vector; while
- an RBFN uses a Euclidean or other distance function (metric); the RBFN measures how far apart are the input and weight (centre) vectors and responds less strongly to input vectors further away.

Arising from this fundamental difference come the commonly known differences:

- the MLP is a *global* network where every input data vector can influence potentially every parameter in the network;
- the RBFN by contrast is *local*, with a given node only reacting to input data from a small local region of the input data space.

The local nature of the RBFN means it is quicker to train but suffers from the curse of dimensionality: the number of hidden nodes required grows exponentially with the number of inputs. This happens because in an RBFN there must be a certain number of receptors per unit length *in every dimension*, just to give reasonable coverage of the input data set. This is the case no matter how many input data are relevant. This effect may be mitigated somewhat by the training strategy used and/or by clustering. The main ways to address the effect are to incorporate domain knowledge about the function to be approximated, and/or to ensure the function is smooth by careful network design. The global nature of the MLP means it can use fewer nodes internally to achieve equal quality of approximation at a faster running speed by avoiding the curse of dimensionality.

The standard methods for training MLPs and RBFNs differ too, although most MLP training methods can also be used on RBFNs (Sect. 13.4.5).

[3] The notation $\|v\|$ denotes the *norm* or *length* of a vector $v = (v_1, \ldots, v_n)$. There are many possible definitions of norm; one of the most common is the *Euclidean* norm $\|v\| = \sqrt{v_1^2 + \cdots + v_n^2}$.

13.3.2 Universal Approximators

It can be shown [58, 125, 285, 556] that MLPs have *universal approximator* capabilities, in that under general conditions (sigmoidal transfer function, one or more hidden layers) they are capable of approximating to arbitrary accuracy any continuous function on a compact (closed and bounded) domain, and thus any given classification or decision boundary. Similarly [58], RBFNs are universal approximators. These universal approximator results follow from Kolmogorov's theorem (Theorem 13.1) and related results in approximation theory.

Theorem 13.1 (Kolmogorov, 1957). *Let* U *be a compact subset of* $\mathbb{R}^n$ *and let* $f : \mathbb{R}^n \longrightarrow \mathbb{R}$*. Then* f *may be represented as a superposition (i.e., linear combination) of a 'small' number of functions of one variable.*

That is, f may be represented as a superposition of relatively few transfer functions. Although Kolmogorov's theorem shows NNs can work perfectly in theory, it is necessary to use modifications to show we can approximate as closely as desired if we restrict consideration to particular kinds of basis function.

In NN terms, developments of these results say that any continuous function $f : \mathbb{R}^n \longrightarrow \mathbb{R} : x = (x_1, \ldots, x_n) \longmapsto f(x)$ can be represented exactly by a four-layer NN having $n(2n + 1)$ nodes in the first hidden layer and $2n + 1$ nodes in the second hidden layer.

13.4 The Multilayer Perceptron

The canonical (artificial) multilayer perceptron (MLP) draws metaphorical inspiration from the brain process described in Sect. 13.1. MLPs are the standard, and most used, neural networks. Since the MLP is often trained using the *(error) backpropagation* training algorithm, it is sometimes (incorrectly) referred to as a 'backpropagation network'.

The MLP consists of three (typically) or more layers of interconnected nodes (Fig. 13.2). It has one or more hidden layers, in addition to the input and output layers. The optimal size(s) of the hidden layer(s) is (are) not known a priori and is (are) usually determined heuristically by the modeller. Topologically, MLPs are directed multipartite graphs, tripartite in the case of three layers. The standard architecture is *feedforward* in that the pattern of activation of the network flows in one direction only, from the input to the output layer. That is, the standard MLP is a directed acyclic graph. *Recurrent* MLPs (Sect 13.4.9) have feedback loops and so are not acyclic.

In an MLP, the signal or value passing along an arc is modified by multiplying it by the weight on that arc before it reaches the next node. The weight therefore serves to amplify or dampen the strength of a signal along that arc. An MLP weight is similar in concept to a regression coefficient. The

processing carried out at each node in the hidden and output layers consists of passing the weighted sum of inputs to that node (its *activation*, a scalar) through a nonlinear transfer function (Fig. 13.3).

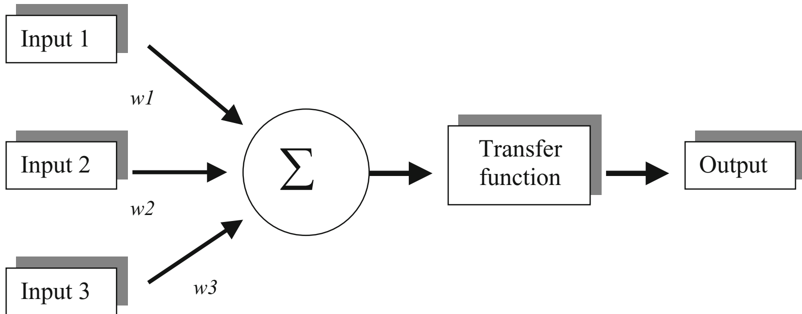


Fig. 13.3. A single processing node in an MLP. The arc weights are denoted by w_1, w_2 and w_3 (if this is node j, we would more correctly denote them by w_{1j}, w_{2j} and w_{3j}, as below). The weighted sum of the inputs to the node, or *activation*, shown as $\sum$, is passed through a transfer function, to produce the node's output

The general form of a single output y from a three-layer MLP (that is, a single hidden layer) is:

$$y = \sigma\left(w'_0 + \sum_{j=1}^{m} w'_j \sigma\left(\sum_{i=0}^{n} x_i w_{ij}\right)\right) \tag{13.2}$$

where:

- x_i represents input i (x_0 is a *bias* node, the leftmost shaded node in Fig. 13.2);
- w_{ij} represents the weight between input node i and hidden node j, with w_{0j} being the bias input to hidden node j: typically, $w_{0j} = 1$. We say that hidden node j has *weight vector* $w_j = (w_{0j}, w_{1j}, \ldots, w_{nj})$;
- w'_0 is the bias node (rightmost shaded node in Fig. 13.2) weight fed to the output layer;
- w'_j, $j = 1, \ldots, m$, represents the weight between hidden node j and the output node;
- y denotes the output produced by the network for input data vector $x = (x_1, \ldots, x_n)$; and
- σ represents a nonlinear transfer function.

It is possible to have different transfer functions σ_1 and σ_2 at different layers, or indeed for different nodes in the same layer, but this is rarely done.

13.4.1 MLP Transfer Function

The MLP transfer function is usually of *saturation* type, sometimes called a *squashing function*. A saturation function is one with finite upper and lower bounds.

The usual choice of transfer function is a sigmoidal (S-shaped) response function, such as the logistic and hyperbolic tan functions, which transform an input in the range $(-\infty, +\infty)$ to the range $(0, 1)$ and $(-1, 1)$ respectively. They thus keep the response of the MLP bounded. Alternatives are the Heaviside step function,[4] and piecewise linear (ramp) function but the former is not continuous and the latter not differentiable, so each gives rise to mathematical difficulties when training the network using gradient-based algorithms such as the standard backpropagation algorithm (Sect. 13.4.5).

The logistic function, lgt, and hyperbolic tangent, tanh, have the forms:

$$\mathrm{lgt}(x) = \frac{1}{1+\mathrm{e}^{-x}} \qquad \text{and} \qquad \tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}} = \frac{1-\mathrm{e}^{-2x}}{1+\mathrm{e}^{-2x}}. \tag{13.3}$$

Each is related to its derivative, which is useful in the optimisation required for training by backpropagation (Sect. 13.4.5):

$$\frac{\mathrm{d}}{\mathrm{d}x}\,\mathrm{lgt}(x) = \mathrm{lgt}(x)(1-\mathrm{lgt}(x)), \quad \text{while} \quad \frac{\mathrm{d}}{\mathrm{d}x}\tanh(x) = 1-\tanh^2(x). \tag{13.4}$$

Geometrically, all of these MLP transfer functions (step, ramp, sigmoid) are threshold response functions, which change from min to max value over a narrow interval. Outside this interval, the output is saturated at either the max or min value. As can be seen in Fig. 13.4, the sigmoidal functions saturate as their input values go to $-\infty$ or $+\infty$ and the functions are most sensitive at intermediate values. A piecewise linear ramp saturates at all values outside a finite interval (Fig. 13.5, left). The step function saturates at all input values (Fig. 13.5, right) and so has limited use.

13.4.2 MLP Activation Function

In an MLP, by (13.2), the transfer function at hidden node j receives as input a dot product activation, the weighted sum (a scalar)

$$w_j \cdot x = w_{0j}x_0 + w_{1j}x_1 + w_{2j}x_2 + \cdots + w_{nj}x_n. \tag{13.5}$$

It can be shown that $w_j \cdot x = \|w_j\|\|x\| \cos\theta$ where θ is the angle between x and w_j. Thus, this dot product gives the length of the projection of the vector x along the direction w_j. It has a maximum with respect to θ when x and w_j are parallel and is 0 when they are orthogonal. Thus, the weighted sum $\sum_{i=1}^{n} x_i w_{ij} = x \cdot w_j$ fed to the transfer function may be viewed as a measure

[4] Rosenblatt's original perceptron [539] used a step transfer function.

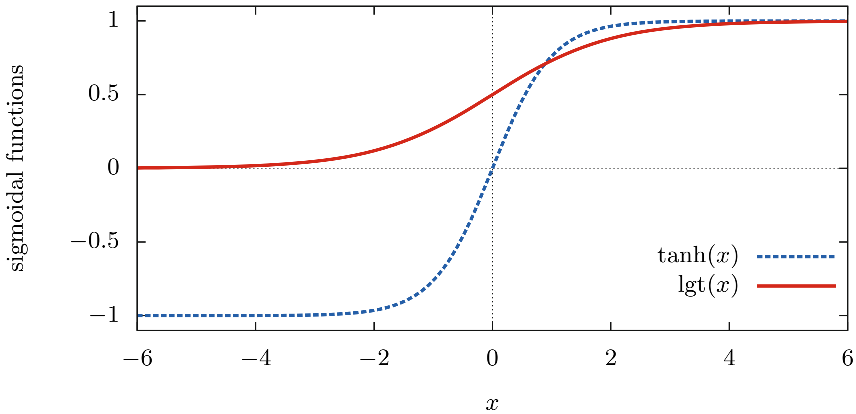


Fig. 13.4. tanh (dotted) and logistic (continuous) sigmoidal functions

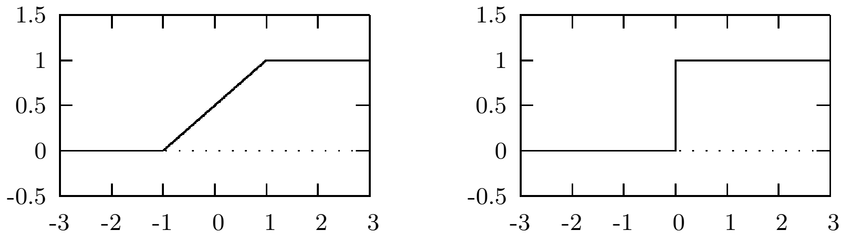


Fig. 13.5. Piecewise linear ramp function (left); Heaviside step function (right)

of similarity between x and w_j. Hence, our use of the term 'likeness function' for activation.

The dot product $w_j \cdot x$ is a *bilinear form*, that is, it is linear in each argument separately when the other is held constant. Thus, in the case of MLPs, the activation function at neuron j is linear when viewed as a function of the weights, $\mathbb{R}^{n+1} \longrightarrow \mathbb{R} : w_j \longmapsto w_j \cdot x$. This linear functional form, together with the saturation type sigmoidal transfer functions, distinguish the MLP from other NNs.

13.4.3 The MLP Projection Construction and Response Regions

Since $w_j \cdot x$ is the length of the projection of x along the direction w_j, the weight vector w_j controls the direction of a 'plateau' in $\mathbb{R}^{n+1}$. Then (Fig. 13.6) the transfer function responds to how well x points along this direction. The transfer function does not respond to the component of x in any direction

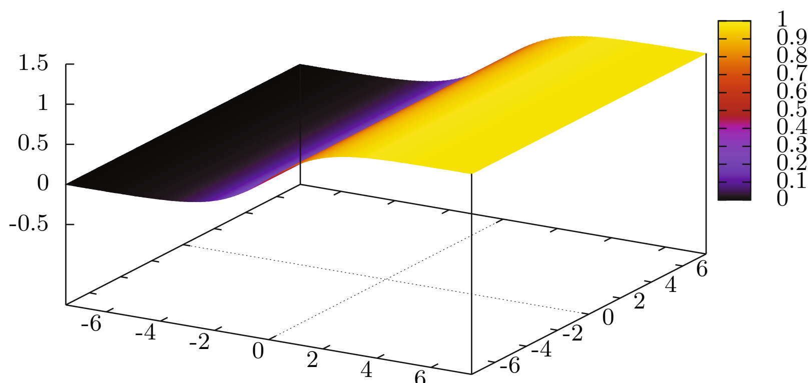


Fig. 13.6. The output from a single sigmoidal basis function is a plateau. Here, $f(x, y) = 1/(1 + \mathrm{e}^{-x})$.

orthogonal to w_j; that is, there in only one direction in the n-dimensional input data space along which the transfer function value changes.

A combination (superposition) of two parallel sigmoids acting on n inputs, which overlap but slope oppositely, gives a ridge-shaped activation area (Fig. 13.7); the combination of two such ridges which intersect but are not parallel, gives a localised 'hillock' (Fig. 13.8). This means that an MLP with sufficiently many neurons is able to detect local variations, and so approximate any smooth function. For further details on the theory and geometrical intuition behind MLPs, see [58, 435].

The aim of this projection construction is to project the input data onto some smaller number of dimensions that still captures much of the important information in the data and so avoid the curse of dimensionality. If the weights are well chosen, we can preserve much of the information in the vector, despite collapsing it to a single dimension. Other statistical methods, such as Principal Components Analysis, and NN approaches such as Self-organising Maps (Sect. 14.1), also attempt to compress data in this way.

As sigmoidal functions are monotonic, the MLP tends to interpolate monotonically. This leads to a very smooth output function. Because sigmoidal functions saturate, the MLP tends to extrapolate to a constant value over the long range, but can extrapolate linearly over short ranges; however, it is difficult to predict since it may be unclear in which region of the sigmoidal

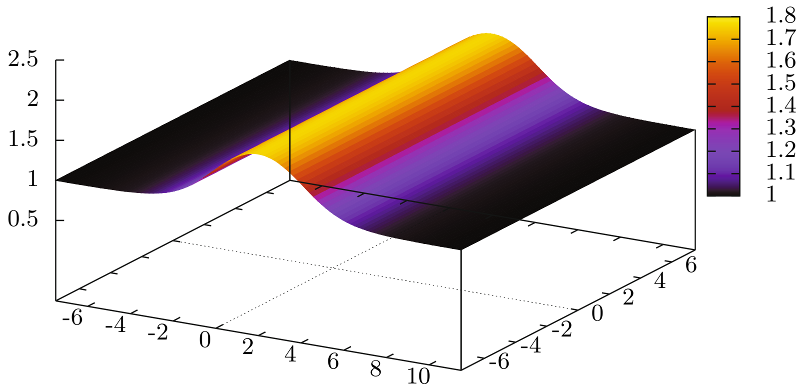


Fig. 13.7. The output from two overlapping parallel sigmoidal basis functions is a ridge. Here, $f(x, y) = 1/(1 + e^{-x}) + 1/(1 + e^{x-4})$.

response the MLP is working for a given input. Locality is very limited, since a change in one weight can significantly influence a model output for a large region of the space.

13.4.4 Relationship of MLPs to Regression Models

The MLP is an elaboration of the original single layer perceptron (no hidden layer) [539], which was shown to be sufficient for linearly separable patterns but not for nonlinear ones [414]. It is easy to show that a simple two layer MLP with no hidden layer and a single output node with a linear transfer function is equivalent to a linear regression model, where the arc weights correspond to regression coefficients. For example, the regression equation $y = a + bx_1 + cx_2$ can be represented as in Fig. 13.9. Similarly, a logistic regression model can be recast as a two-layer MLP with a sigmoidal transfer function at the output node. A multilayer MLP with nonlinear transfer functions can therefore be considered as a nonparametric, nonlinear regression model. An MLP of three or more layers using exclusively linear transfer functions can always be recast, using linear algebra, as a two-layer MLP with linear transfer functions.

In contrast to ordinary linear least squares regression models, which produce a line, plane or hyperplane depending on the number of independent variables, MLPs which use nonlinear transfer functions can produce com-

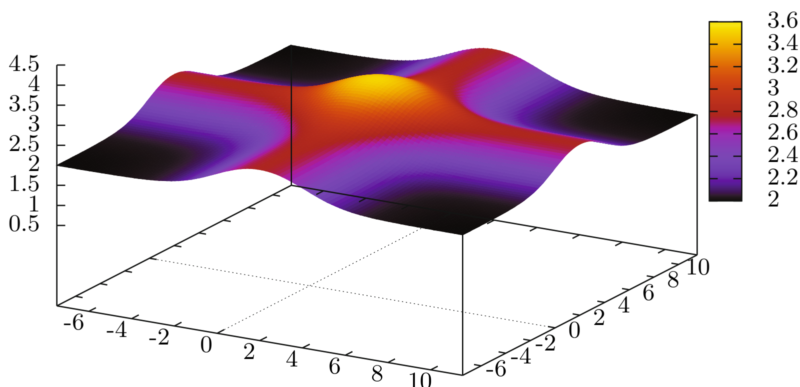


Fig. 13.8. The output from two overlapping nonparallel ridges is a hillock. Here, $f(x, y) = 1/(1 + \mathrm{e}^{-x}) + 1/(1 + \mathrm{e}^{x-4}) + 1/(1 + \mathrm{e}^{-y}) + 1/(1 + \mathrm{e}^{y-4})$.

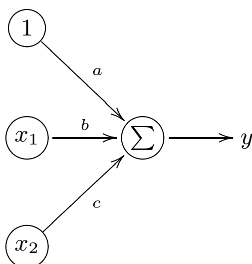


Fig. 13.9. Linear equation as a node-arc structure, $y = a + bx_1 + cx_2$.

plex (but smooth) response surfaces with peaks and troughs in n dimensions. Changing the weights during the learning process tunes this response surface to more closely fit the training data.

13.4.5 Training an MLP

MLPs are trained using a *supervised* learning paradigm. In supervised learning a set of input data vectors for which the output is already known are presented

to the MLP. The MLP predicts an output for each input vector and the error between the predicted and the actual value of the output is calculated. The weights on each connection in the network are then adjusted in order to reduce this error. By altering the weights, the network can place different emphasis on each input and differing emphasis on the output of each hidden layer node in determining the final output of the network. The *knowledge* of the network is therefore embedded in its connection weights. Once the network has been trained, it can be used to predict an output for an input data vector which it has not previously seen.

The hidden layer weights may be learned by several methods, the most common being (error) backpropagation, as described below. It is important to distinguish between the NN *model* — in this case an MLP — and the training method used for that model — typically, but not necessarily, backpropagation.

Because of the high dimensionality and projection construction, hidden layer parameters are not easily visualised or interpreted, and thus are difficult to initialise using prior knowledge. The simplest approach is random choice of weights. However, care must be taken to avoid having neurons in a saturated state, as these will give effectively the same response for all inputs. Thus, a reasonable initialisation should try to have neuron activation values lying in an interval around 0.

The Backpropagation Algorithm

The most common way of altering the weights in response to an error in the network's prediction is through use of the *backpropagation algorithm* [81, 545, 655]. It is applicable when the transfer functions are twice differentiable, as is the case with the standard sigmoidal transfer functions (tanh and the logistic function).

At the start of the learning process the weights on all arcs are initialised to small random values. The network is presented with an input data vector and proceeds to predict a value for the output. Total squared error is defined as:

$$E = \sum_{p=1}^{P} \sum_{q=1}^{S} (y_q^p - z_q^p)^2 \tag{13.6}$$

where P is the number of input–output vectors, S is the number of output neurons, and y_q^p and z_q^p are respectively the target and predicted values of component q of the pth output vector. The aim is to minimise this 'sum of squares' (quadratic) error function.

The backpropagation algorithm seeks to reduce the total error by calculating the gradient of the error surface at its current point (corresponding to the current weight vector for the network) and adjusting the weights in the network in order to descend the error surface. This is achieved by making a backward pass through the network, from the output to the input layers, in which weight changes are propagated back through the arcs of the network, so

as to make the prediction of the network better agree with the actual output value(s). The bigger the error, the more the arc weights are adjusted. The total error of the network is a function of the values of all of its weights. From (13.6), we can minimise total error by sequentially minimising $E_p := \sum_{q=1}^{S}(y_q^p - z_q^p)^2$ for $p = 1, \dots, P$.

Consider the arc ij from node i to node j. Let $w_{ij}(t)$ denote the weight on this arc at iteration t. Let u_i be the amplitude output by i (for example, u_i is just x_i if i is an input node). Then the activation function fed into node j is $a_j = \sum_{i=0}^{n} w_{ij}u_i$; and the output of node j is $\sigma(a_j)$ where σ is the transfer function. When altering the individual weights during the backpropagation step, in order to minimise the error E_p, we consider the partial derivative $\frac{\partial E_p}{\partial w_{ij}(t)}$ of E_p with respect to each individual weight $w_{ij}(t)$: this is just the contribution of that weight to the total network error for input data vector p.

A standard gradient descent gives the correction to weight w_{ij} at iteration t as

$$\Delta w_{ij}(t) := w_{ij}(t+1) - w_{ij}(t) = -\alpha\left(\frac{\partial E_p}{\partial w_{ij}(t)}\right). \tag{13.7}$$

Here, α is the *learning rate* ($\alpha > 0$) which controls the strength of response to errors. By the chain rule for partial derivatives,

$$\frac{\partial E_p}{\partial w_{ij}(t)} = \frac{\partial E_p}{\partial a_j(t)}\frac{\partial a_j}{\partial w_{ij}(t)}. \tag{13.8}$$

Since each u_i is fixed, and $a_j = \sum_{i=0}^{n} w_{ij}u_i$, we have that

$$\frac{\partial a_j}{\partial w_{ij}(t)} = u_i. \tag{13.9}$$

If we denote $\frac{\partial E_p}{\partial a_j(t)}$ by δ_j (the 'error' at j), (13.8) gives

$$\frac{\partial E_p}{\partial w_{ij}(t)} = \delta_j u_i. \tag{13.10}$$

Thus the required partial derivative is got by multiplying the amplitude fed into arc ij by the δ value at the output of arc ij. It is only necessary to calculate the value of δ for each output node, then 'propagate' these 'back' to get the δ values at the hidden layer nodes, then backpropagate these to the input layer. Specifically, at an output node q, the error for data item p is

$$\delta_q = y_q^p - z_q^p. \tag{13.11}$$

At a hidden node j, using the chain rule again,

$$\delta_j = \frac{\partial E_p}{\partial a_j(t)} = \sum_{q=1}^{S}\frac{\partial E_p}{\partial a_q(t)}\frac{\partial a_q(t)}{\partial a_j(t)}. \tag{13.12}$$

Then for any nonoutput node j, (13.12) and the fact that $a_j = \sum_{i=0}^{n} w_{ij}\sigma(a_i)$ gives the backpropagation formula

$$\delta_j = \sigma'(a_j(t)) \sum_{i=0}^{n} w_{ij}(t)\delta_i, \qquad (13.13)$$

which allows efficient calculation of each $\frac{\partial E_p}{\partial w_{ij}(t)}$ since the derivative σ' of a sigmoidal transfer function σ is easily expressed in terms of σ itself (see (13.3) and subsequent text).

This step is performed repetitively over the entire training dataset (that is, update weights after each individual training vector is presented to the network), until the network reaches a stable minimum error. It is also possible to use a *batch mode* method, in which case the results are based on the total error accumulated over the entire training set. The gradient descent may be replaced by the Conjugate Gradients method, which is well suited to optimising quadratic functions such as the error E.

A basic algorithm for training a single-hidden-layer MLP is presented in Algorithm 13.1. In this algorithm,

- t is the current time step;
- there are m hidden nodes and we denote their outputs by $s_1, \ldots, s_m$; the $s_1, \ldots, s_m$ are fed to the output node(s)' activation(s);
- Equation (13.16) is first used on the hidden to output layer weights w_{jq}, and then on the input to hidden layer weights w_{ij}.

13.4.6 Overtraining

Any sufficiently powerful machine learning approach is, if given enough time to learn, capable of explaining everything it sees in the training dataset — including noise. If the MLP training algorithm is applied as is, there is a good chance that *overtraining* (or *overfitting*) will occur, giving rise to poor predictive performance when applied to new data (Fig. 13.10). That is, the network fails to capture the true statistical process generating the data and will not *generalise* well. For example, if there were too few training examples, or learning was carried out for too long, the network may incorporate particular random features of the training data that are not related to the function f to be learned. Overtraining usually occurs when a network is overly complex relative to the quantity of data, e.g., having too many degrees of freedom. On the other hand, if there are too few degrees of freedom, the MLP will not be able to adequately learn from the training data. In the MLP context, the number of degrees of freedom is essentially the number of weights. As a rough rule of thumb from statistics, there should be at least five to ten data vectors for each weight estimated, to reduce the chance of overfitting the training data. A more theoretically justifiable approach is to modify the NN complexity according to the training data: for example, by beginning with a large

Algorithm 13.1: MLP Training Algorithm

Let n be the number of input nodes;
Initialise each connection weight to a small random value in the range $[0, 1]$;

repeat

Present an input vector $x = (x_1, \ldots, x_n)$ and its associated target output z;
Calculate the output from each hidden layer node j using

$$s_j = \sigma \left(\sum_{i=0}^{n} w_{ij} x_i \right) , \tag{13.14}$$

and then from each output layer node q using

$$y_q = \sigma \left(\sum_{j=0}^{m} w_{jq} s_j \right) . \tag{13.15}$$

Let the total error be $E = \sum_{p=1}^{P} \sum_{q=1}^{S} (y_q^p - z_q^p)^2$;
Adjust the weights on the connections between the nodes, by setting

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \left(\frac{\partial E}{\partial w_{ij}(t)} \right) , \tag{13.16}$$

commencing with the output layer and working back to the input layer;
Update weights in batch mode *or* after each individual training vector is presented to the network;

until *training error reaches a steady state or an acceptable minimum*;

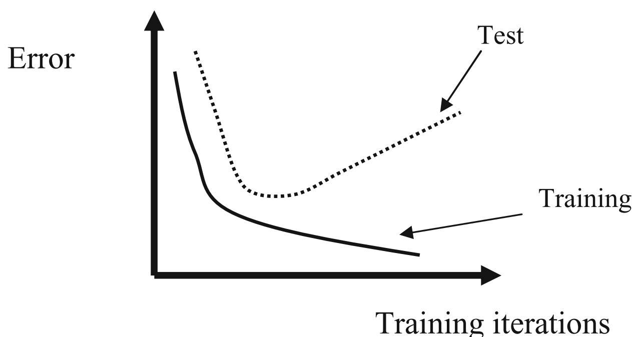


Fig. 13.10. As training continues on the same set of training data, the MLP performs better on that data set. However, it eventually starts to perform more poorly on out-of-(training) sample test data, which indicates possible overfitting

number of hidden layer nodes and 'pruning' or eliminating weights methodically [170, Sect. 6.11 and Chap. 9]. In addition, the training process should be controlled using, for example, the method of *early-stopping*. In this method, the dataset is divided into three components: *training* data, *validation* data, and *out-of-sample* (test) data. The MLP is constructed using the training dataset; but, periodically during this process, the performance of the network is tested against the validation dataset. The network's performance on the validation dataset is used to determine when the learning process is stopped, and the best network is defined as that which produces the minimum error on the validation dataset. Once the best network is found, the weights are fixed and the network is ready for application to out-of-sample data. Early-stopping enhances the *robustness* (against fitting noise) of the MLP. Another approach which reduces overtraining is regularisation of the error measure (see (13.18)).

13.4.7 Practical Issues in Modelling with and Training MLPs

A number of practical problems arise in using MLPs for modelling:

i. what measure of error should be used?
ii. what parameters should be chosen for the backpropagation algorithm?
iii. how many hidden layers (or nodes in each hidden layer) should there be?
iv. is the data of sufficient quality to build a good model?

Measure of Error

Many different error criteria can be applied in determining the quality of fit of a NN model. Most applications use traditional criteria drawn from statistics such as the sum of the squared errors, or mean squared error (MSE):

$$\mathrm{MSE} = \frac{1}{P}\sum_{p=1}^{P}\sum_{q=1}^{S}(y_q^p - z_q^p)^2 \tag{13.17}$$

in the notation of (13.6), where: y_q^p is the value predicted by the NN model for component q of the output vector, given input vector p; z_q^p is component q of the actual output vector, given input vector p; and there are P output vectors, each Q-dimensional and each corresponding to an input vector. Although this is a common error metric it can lead to poor generalisation, as one way of reducing MSE is to build a large NN which learns the noise in the training dataset. This can be discouraged by using a *regularised* performance function (error measure), where the performance function is extended to include a penalty term which gets larger as network size grows. As an example, the MSE error term could be adjusted to give:

$$\text{Regularised Error Measure} = \gamma \cdot \mathrm{MSE} + (1-\gamma)\cdot \mathrm{MSW} \tag{13.18}$$

where MSW is a penalty term, calculated as the mean sum of the squared weights in the network. The values of γ and $1 - \gamma$ represent the relative importance that is placed on the MSE and the penalty term respectively. The penalty term will tend to discourage the use of large weights in the network, and will tend to smooth the response of the network.

Parameters for the Backpropagation Algorithm

The essence of training an MLP is the determination of good values for the individual weights in the network. If there are N weights in the network, the task of uncovering good weights amounts to a nonlinear optimisation problem where an error surface exists in $(N + 1)$-dimensional space. Unfortunately, no general techniques exist to optimally solve this problem. The backpropagation training algorithm is a gradient-descent, local search algorithm, and so is prone to becoming trapped in local optima on the error surface. A number of steps can be taken to lessen the chance of this happening.

Typically, during the training process, the network weights are altered, based on the current model error and a modeller-tunable parameter (the *learning rate*) which governs the size of weight change in response to a given size of error. Usually the value of the learning rate will decay, from a higher to a lower value during the training run with fairly rapid learning in the initial training stages and smaller weight adjustments later in the training run. The object in varying the learning rate during the training process is to enable the MLP to quickly identify a promising region on the error surface and later to allow the backpropagation algorithm to approach the minimum error point in that region of the error surface.

However, there is no easy way to determine a priori what learning rates will produce the best results. The learning process will typically have an element of *momentum* built in, whereby the direction and size of weight change at each step is influenced by the weight changes in previous iterations of the training algorithm. Therefore the weight change on iteration $t + 1$ is given by:

$$w_{ij}(t+1) = \lambda w_{ij}(t) - (1-\lambda)\alpha \left(\frac{\partial E}{\partial w_{ij}(t)} \right) \tag{13.19}$$

where λw_{ij} is the momentum term and α is the learning rate. By varying the value of the momentum coefficient λ in the range 0 to 1, the importance of the momentum coefficient is altered. Under the concept of momentum, if the MLP comes across several weight updates of the same sign, indicating a uniform slope on the error surface, the weight update process will gather momentum in that direction. If later weight updates are of different signs, the effect of the momentum term will be to reduce the size of the weight updates to below those which would occur in the absence of the momentum component of the weight update formula. The practical effect of momentum is to implement *adaptive learning*, by speeding up the learning process over uniform or shallow gradient regions of the error surface.

The backpropagation learning algorithm can be likened to jumping around an error surface on a pogo stick. If the jumps are too small (corresponding to a low learning rate) the pogo stick jumper could easily get stuck in a local minimum, if the jumps are too large, the pogo stick jumper could overshoot the global minimum error. This analogy also underlines the importance of the initial weight vector. The initial weight vector determines the starting point on the weight-error surface for the backpropagation algorithm. If a poor starting point is chosen, particularly if the learning rate is low, the algorithm could descend into an inescapable local minimum (Fig. 13.11). To reduce the chance

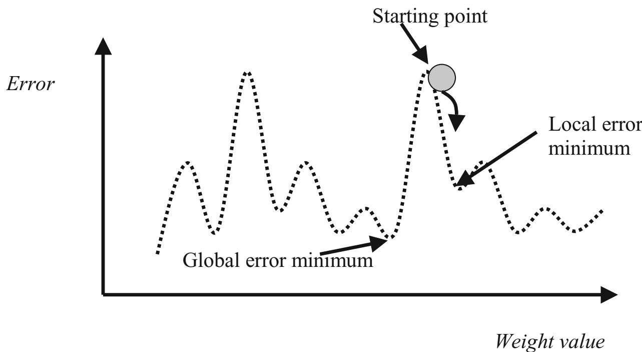


Fig. 13.11. Given this starting point on the weight surface, a gradient-descent algorithm will only find a local error minimum

that a bad *initialisation* of the weight vectors will lead to poor performance of an MLP, performance should be assessed across several training runs using different initialisations of the connection weights (Fig. 13.12).

Selecting Network Structure

Although a three-layer MLP is theoretically capable of approximating any continuous function to any desired degree of accuracy, there is no theory to decide how large the hidden layer needs to be in order to achieve this [125]. Typically the size of this layer is determined heuristically by the modeller. However, as the hidden layer gets larger, the number of degrees of freedom (weights) consumed by the MLP rises. Thus, the amount of data needed to train it increases by five to ten data vectors per weight (Sect. 13.4.6). For example, a fully connected 20–10–1 MLP (input layer nodes–hidden layer nodes–output node) contains $(20 \times 10) + (10 \times 1) = 210$ weights. Therefore the above network will require a fairly large dataset of at least 1,000 to 2,000 data vectors for training purposes. The selection of the size of the hidden layer

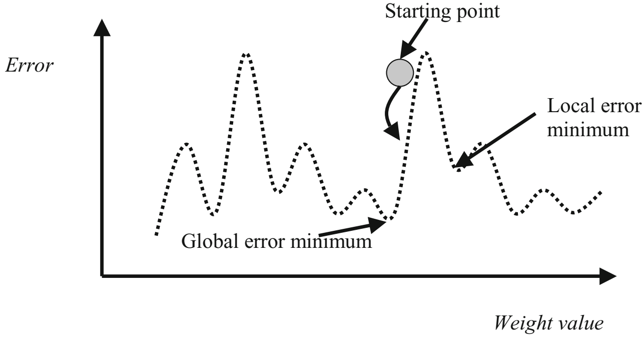


Fig. 13.12. Altering the initial weights moves the starting point on the weight surface, making the global error minimum point accessible

entails a trade-off between increasing the power of the MLP (more nodes) and avoiding overfitting (fewer nodes).

In designing MLPs, there is no restriction that they must have a fully connected (or *complete*) feedforward connection structure. Each input need not be connected to each hidden layer node, and nodes can be connected to nodes which are more than one layer ahead in the network (a *jump connection network*) (Fig. 13.13).

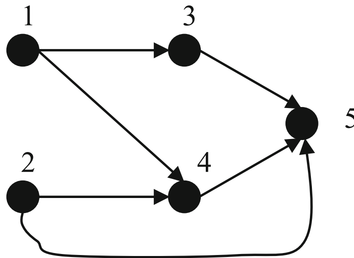


Fig. 13.13. Input 2 is connected to only one hidden layer node, and also has a jump connection directly to the output node

Multiple Hidden Layers

The use of several hidden layers makes the network more complex and is an alternative to having more neurons in one hidden layer. While single hidden

layer networks are most common, very occasionally two (or more) hidden layers are used. Use of more hidden layers makes training more difficult, because of stronger nonlinearity and more complicated gradients. However, fewer weights may be required to achieve the same approximation capability.

One case where a second hidden layer is useful is when trying to learn complicated target functions g, particularly multimodal functions (those with multiple local maxima (peaks) and local minima (valleys)). The first hidden layer learns local features: some nodes in this layer divide the function into regions while others learn the small-scale features of that region. The second hidden layer learns global features: each node here allows the MLP to fit a separate peak or valley by combining the outputs of the first hidden layer nodes corresponding to that region, while ignoring other nodes in the first hidden layer. The result is that a two hidden layer MLP can use fewer weights than a single hidden layer MLP, while giving an equally good approximation to g [106]. However, the use of two hidden layers may lead to a very spiky landscape, making the problem of local optima in the total squared error surface (13.6) worse, even when the total number of weights is much less than the size of the training dataset. De Villiers et al. [156] conclude that 'there seems to be no reason to use four layer networks in preference to three layer nets in all but the most esoteric applications'.

Data Quality and Predictive Ability

The quality of the dataset also plays a key role in determining the quality of the MLP. Obviously if important data is not included, perhaps because it is not available, the results from the MLP are likely to be poor.

Another data-related issue is how representative the training data is of the whole dataset. If the training data is not fully representative of the behaviour of the system being modelled, out-of-sample results are likely to be poor. The dataset should be recut several times to produce different training and out-of-sample datasets and the stability of the results of the developed MLPs across all of the recuts should be considered.

Furthermore, in any dynamic environment, the predictive ability of the MLP (or indeed any predictive model) may be compromised by the 'shelf life' effect: every model is trained on past data but in some cases the patterns learned during training may go out of date (in other cases, behaviour patterns may be stable over time). This means regular retraining may be necessary.

13.4.8 Stacking MLPs

The predictive ability of an individual MLP is critically affected by the choice of network weights. Since the canonical back-propagation algorithm is a local search technique, the initial weights can have a significant impact on the quality of the final MLP. A stacking process can be used to reduce this problem

and to combine the predictive abilities of individual models which may possess differing pattern-recognition capabilities.

In an idea similar to the combination of the outputs of multiple nonlinear processing elements in an individual MLP, the outputs of individual MLPs can be combined (*stacked*) to form a *committee decision*. The overall output of the stacked network of MLPs is a weighted combination of the individual network outputs:

$$F(x_{\text{in}}) = \sum_{i=1}^{r} w_i f_i(x_{\text{in}}) \tag{13.20}$$

where $F(x_{\text{in}})$ is the output from the stacked network for input vector x_{in}, $f_i(x_{\text{in}})$ represents the output of network i, and w_i represents the stacking weight. In this example, the outputs are combined on an equally weighted, linear basis. Figure 13.14 provides an illustration of a stacked MLP. More

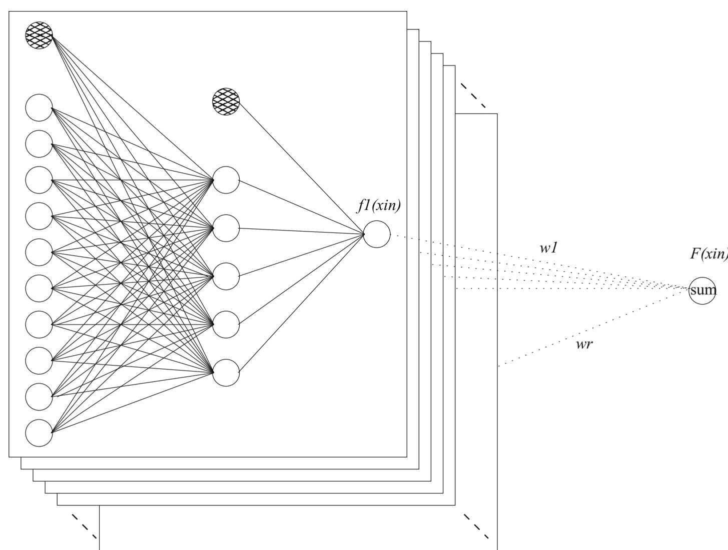


Fig. 13.14. The outputs of multiple MLPs being combined to produce a single stacked output

complex aggregation methodologies could be applied, including the implementation of another MLP at the stacking stage. The input data may also be split between each of the initial MLPs so that each MLP is trained using a different subset of the training data.

13.4.9 Recurrent Networks

The inspiration for recurrent networks, which allow feedback connections between the nodes, is the observation that the human brain is a recurrent net-

work. The activation of a particular neuron can initiate a flow of activations in other neurons which in turn feed back into the neuron which initially fired. The feedback connections in a recurrent network imply that the output from node b at time t can act as an input into node a at time $t + \ell$. Nodes b and a may be in the same layer, or node a may be in an earlier layer of the network. A node may also feed back into itself ($a = b$). Of course, positive feedback may lead to instability, and techniques from Control Theory may be required to bring the system back into a stable domain of operation.

Recurrent networks can be useful when modelling time series data, as the recurrent connections allow the network to store processing results (or raw input data) from previous time steps, and later feed it back into the network. In contrast, a standard feedforward network has a data window of a fixed size, and associations in the data that extend beyond this window cannot be uncovered by the network.

A practical benefit of recurrent network designs is that they can be compact. Consider the case where a modeller wishes to provide a neural network with information on the past N values of n input variables. If a canonical feedforward MLP were used, this would require Nn inputs, possibly a large number, leading to a large number of weights which require training. In contrast, as recurrent networks can embed a *memory*, their use can notably reduce the number of input nodes required.

An example of a simple recurrent network is an *Elman network* [176]. This includes three layers, with the addition of a set of *context* nodes which represent feedback connections from hidden layer nodes to themselves (Fig. 13.15). The connections to the hidden layer from these context nodes have trainable weights. The context nodes act to maintain a memory of the previous period's activation values of the hidden nodes. The output from the network depends therefore on both current and previous inputs. An implication of this is that recurrent networks operate on both an input space and an internal state space. Generalising the context layer concept, it is possible to implement more than one context layer, each with a different lag period. Time is represented implicitly as a result of the design of the network, rather than explicitly through the use of a large number of time-lagged inputs.

Several methods exist to train Elman networks. The original method proposed [176] was to treat each of the feedback inputs from the context layer as an additional input to the network at the next time step. A standard backpropagation algorithm was then used to train all the weights in the network. Training of recurrent networks using gradient-based methods can be time-consuming, and alternative methods using evolutionary, particle swarm or hybrid approaches exist [569].

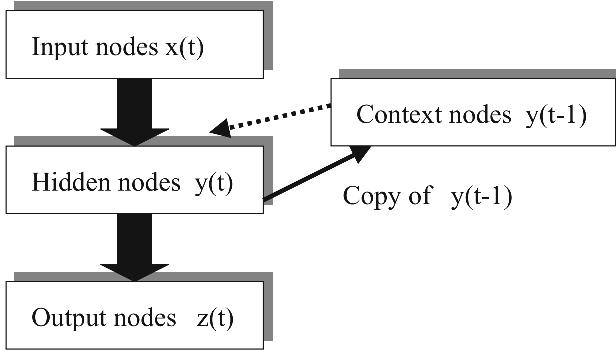


Fig. 13.15. An Elman network. The output of each of the hidden layer nodes at time $t-1$ is stored in individual context nodes, and each of these is fed back into all the hidden layer nodes as an input at time t. The context layer nodes are empty during the first training iteration.

13.5 Radial Basis Function Networks

Another common form of supervised neural network which can be used for classification and prediction is the radial basis function network (RBFN). While these networks can be used for the same purposes as an MLP and nominally have a three-layer feedforward structure, their internal workings and their training processes are quite different from those of MLPs. Another feature of RBFNs is that the training process is fast, typically only requiring a single pass of the training data through the network, in contrast to the multiple iterations of the training dataset which is required when training an MLP.

In RBFNs the processing at hidden layer nodes is carried out using as transfer function a *radial basis function* rather than a sigmoidal function. The radial basis function is usually chosen to be local in nature, and have a maximum at $x = c$, its centre. It is a *kernel function.* Typical choices include the Gaussian and inverse multiquadric functions.

13.5.1 Kernel Functions

Intuitively, we think of the *kernel* of a function f as the equivalence relation on the domain of f that says 'equivalent as far as f can tell'.

Definition 13.2. *Let* $f : X \longrightarrow Y$ *where* X *and* Y *are sets. Call elements* $x_1, x_2 \in X$ (f-)equivalent *if* $f(x_1) = f(x_2)$*. Then the kernel of* f *is the set of all equivalent pairs in* X*:* $\ker f = \{(x_1, x_2) \in X \times X : f(x_1) = f(x_2)\}$.

For example, the kernel of a linear map is the set of vectors the map sends to 0: as far as this map can tell, all of these vectors are the same (by linearity, this also works for preimages of any nonzero image vector). Thus, a kernel can help in measuring 'similarity' of two objects.

More commonly, we define a *kernel function* as a symmetric real-valued function k of two objects, with no f directly used. Such a kernel k is simply taken as a measure of similarity of two objects, with the real number it returns giving the degree of similarity. In this sense, a kernel may be regarded as a generalisation of a dot (or inner) product and in particular must be symmetric and positive definite. Kernels first arose in this way in the study of integral operators.

A critical aspect of kernel functions is that basic operations such as addition and multiplication (or linear combinations of kernel functions) preserve their properties.

One approach to giving a distance-based measure of similarity of two vectors v_1 and v_2 is to centre a (kernel) density function on one of them (by symmetry, the choice of v_1 or v_2 as centre does not matter), and examine the value of the density function on $v_1 - v_2$. This kind of kernel is called a *radial basis function*.

13.5.2 Radial Basis Functions

Radial basis functions [518] are radially symmetric functions concentrated about a 'centre'. That is, they have the same form no matter what direction is taken from the centre. Originally, radial basis functions came from the mathematical topics of approximation theory and interpolation.

In applications such as Radial Basis Function Networks, they are typically smooth functions and monotonically decreasing along any direction from the centre.

Definition 13.3. *Given a centre $c \in \mathbb{R}^n$, a function $\vartheta : \mathbb{R}^n \to \mathbb{R}$ is called* radial *if $\vartheta(x) = \varphi(\|x - c\|)$ for some $\varphi : \mathbb{R} \to \mathbb{R}$.*

Writing $r := \|x - c\|$ for simplicity, some examples of radial basis functions include:

- Gaussian: $\varphi(r) = \mathrm{e}^{-r^2/(2\pi\sigma^2)}$, where σ is a parameter which determines the *bandwidth*, or *effective width*, or *sensitivity*, of the radial basis function. It is analogous to the standard deviation of the normal distribution.
- Multiquadratic: $\varphi(r) = \sqrt{r^2 + a^2}$ for some $a > 0$
- Multiquadric: $\varphi(r) = (r^k + a^k)^{1/k}$ for some $a > 0$
- Inverse multiquadric: $\varphi(r) = (r^k + a^k)^{-1/k}$ for some $a > 0$
- Polyharmonic spline: $\varphi(r) = \begin{cases} r^k, & k = 1, 3, 5, \ldots, \\ r^k \ln(r), & k = 2, 4, 6, \ldots \end{cases}$

Among radial basis functions, the Gaussian function is unique in that for Euclidean (and some other) metrics it is *separable*, i.e., it can be written as a product of independent factors, one factor for each input component. The Gaussian radial basis function will produce an output value of 1 if the input and weight vectors are identical, falling towards 0 as the distance between the two vectors gets large. Such a smooth local response function is sometimes called a *bump function*. In the case of the multiquadratic function, its activation level increases as the distance between the vectors increases.

A radial basis function may be used to provide a distance-based measure of the 'similarity' of a vector x to another vector c, the centre of the function. Thus, the RBFs are kernel functions: the response is weighted according to the distance from the RBF centre, and measures similarity of the vectors in the sense of closeness.

13.5.3 Intuition Behind Radial Basis Function Networks

A radial basis function network is best motivated by viewing the design as a surface/curve-fitting (function approximation) problem in a high-dimensional space. Alternatively, this may be viewed as classification, with the surface being the interclass boundary. As with MLPs, learning with an RBFN means finding a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ that provides a best (or simply good) fit to the training data, with the criterion for 'best' measured in some appropriate statistical sense. This high-dimensional surface may then be used to interpolate the test data. As with other neural networks, the hidden nodes implement a set of functions that make up a basis for the distribution of input vectors (Fig. 13.16).

In RBFNs, each basis function is a radial basis function, typically a bump function. We think of an RBFN hidden layer neuron as representing a *detector*, which 'fires' to a greater or lesser degree, as the input vector is 'closer to' or 'further from' the centre of the detector. Its operation can be split into two parts. First, the distance r of the input vector x to the centre vector c with respect to a norm matrix[5] is calculated. Second, this scalar distance r is transformed by the radial transfer function $\varphi(r)$.

We view the 'detector' neurons as being distributed through the pattern (input) space — rather like weather balloons distributed through the atmosphere — and detecting conditions locally. If many input vectors are close to a given detector, it will fire more often: thus, the RBFN has the potential to 'learn' the distribution of the input. The function or distribution f to be learned can be thought of as being made up of a weighted sum of bump functions: these bump functions are the building blocks of f; they are a *basis* in terms of which we describe f.

[5]Given any positive definite $n \times n$ real matrix, we may define an inner product (positive definite symmetric bilinear form) on $\mathbb{R}^n$ by $\langle x, y \rangle := x^t A y$ for all $x, y \in \mathbb{R}^n$. Then $\|x\| := \sqrt{\langle x, x \rangle} = \sqrt{x^t A x}$ defines a norm on $\mathbb{R}^n$.

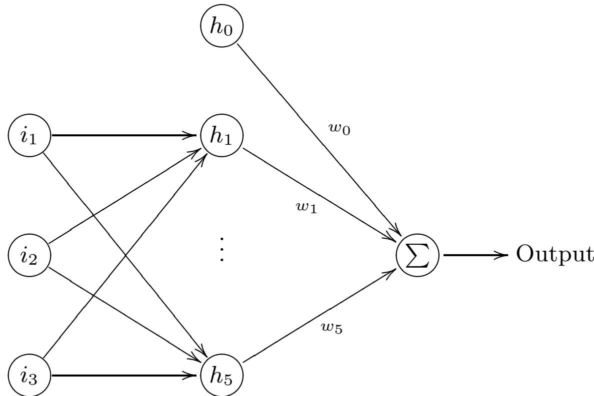


Fig. 13.16. A radial basis function network. The output from each hidden node (h_0 is a bias node, with a fixed input value of 1) is obtained by measuring the distance between each input pattern and the location of the centre represented by that hidden node, and applying the radial basis function to that distance. The final output from the network is obtained by taking the weighted sum (using $w_0, w_1, \ldots, w_5$) of the outputs from the hidden layer and from h_0.

The transformation from the input layer to the hidden layer is nonlinear, while the transformation from the hidden layer to the output layer is linear. This is motivated by Cover's Theorem on the separability of patterns [118]: *complex pattern classification problems are more likely to be linearly separable if the patterns are initially mapped nonlinearly (but injectively) into a higher dimensional space.* This leaves open the nontrivial question as to what mapping function should be used in a particular case. An alternative, simpler, approach which approximates this is to use a *kernel function* as described above. Such kernel functions allow the computation of dot products in high-dimensional spaces, without having to explicitly map the vectors into these spaces (this is known as the *kernel trick*). Thus, using a kernel function eliminates the need to uncover an explicit nonlinear mapping function for the input data vectors in order to make them linearly separable. RBFNs explicitly adopt a kernel function approach, as they usually use a radial local response Gaussian kernel as transfer function at each hidden node.

13.5.4 Properties of Radial Basis Function Networks

In a NN such as an RBFN, the components of a node's centre c are implemented as the weights on the arcs entering the node, so the centre is regarded as the 'weight vector'. Unlike the case of the MLP, multiple hidden layers do not make any sense for an RBFN, as the extra layer(s) cannot be interpreted and so their parameters cannot be chosen by prior knowledge.

Interpretation of the centres, bandwidths, and heights is possible if the basis functions are local and their widths are chosen small. However, the interpretability in high-dimensional spaces is limited. Incorporation of constraints and of prior knowledge is possible in RBFNs because the parameters can be interpreted, and the local nature allows one to drive the network toward a desired behaviour in a given operating regime. Again because of the locality of the basis functions, online adaptation in one operating regime does not appreciably influence the others.

An RBF Network consists of three types of parameter, all of which need to be determined or optimised during training.

i. Output layer weights are linear parameters. They determine the heights of the basis functions and the bias value.
ii. Centres are nonlinear parameters of the hidden layer neurons. They determine the positions of centres of the basis functions.
iii. Bandwidths (and possibly off-diagonal entries in the norm matrices) are nonlinear parameters of the hidden layer neurons. They determine the widths (and possibly rotations) of the basis functions.

The total number of parameters of an RBF Network is $2mn+m+1$ where m is the number of hidden layer neurons (centres) and n is the number of inputs. Since n is given by the problem, m allows the user to control the network complexity, that is, the number of parameters.

The interpolation behaviour may have dips (be too bumpy) if the bandwidths are too small, and overshoot if they are too large. The wider the basis functions are, the slower the extrapolation behaviour decays towards 0.

13.5.5 Training Radial Basis Function Networks

The radial construction approach gives the hidden layer parameters of RBFNs a better interpretation than for the MLP, and so allows faster training methods: typically, we only require a single pass of the training data through the network, in contrast to the multiple iterations of the training dataset which is required when training an MLP.

During the training process, each hidden layer centre is initially located at a fixed location in the input feature space and this location is designated as the weight vector for that hidden node. Therefore, unlike in MLPs, there are no trainable weights between the input and the hidden layer as the locations of the centres do not change as the RBFN algorithm runs.

The only trainable weights in an RBFN are those between the hidden and the output layers. In a basic RBFN implementation where there is only one output node, these weights can be trained using linear regression in a single pass through the training dataset. Linear regression can be used to determine the weights between the hidden layer and the output node; as the output from each hidden node will correspond to a set of x_is, the correct (known) output for each training data vector j is the set of y_js, and the connection weights,

which correspond to the regression coefficients, are the β_is. In essence, in developing an RBFN the object is to locate a set of detectors (centres) in the input pattern space and then during the training process determine what weight to attach to the output from each detector in order to compute the final predicted output from the RBFN. Algorithm 13.2 outlines the pseudocode for the canonical RBFN.

Algorithm 13.2: RBFN Training Algorithm

Select the initial number of centres (m) and the form of the radial basis function;
Select the initial location of each of the centres in the data space;

for *each pairing of input data vector x and centre c* **do**
  Calculate the activation value $\varphi(\|x - c\|)$, where φ is a radial basis function and $\|\ \|$ is a distance measure in the data space;
end
Calculate the weights for the connections between the hidden and output layers using linear regression;

13.5.6 Developing a Radial Basis Function Network

As with MLPs, the development process for RBFNs is iterative as the performance of an RBFN is influenced by a number of modeller choices, including:

i. the number of centres,
ii. the location of centres, and
iii. the choice of radial basis function and its parameters.

As the number of centres increases, the RBFN will tend to improve fit on the training data but this could occur at the expense of poor out-of-sample generalisation. Hence, the object is to choose a sufficient number of centres to capture the essential features in the training data, without overfitting that data. A simple approach is to use a validation dataset. Initially, the RBFN is developed using a small number of centres and its performance on the training and validation dataset is monitored. The number of centres is then increased gradually, with the RBFN being retrained each time the number of centres is changed, until validation set performance degrades.

In initially positioning the centres, they can be located by placing them on randomly selected members of the training dataset or by attempting to distribute them uniformly throughout the input space. The former approach has the advantage that if the training data is truly representative, the distribution of the centres will be determined by the distribution of the underlying data. Centres could also be located using more sophisticated approaches.

Unless the problem has special structure, the choice of radial basis function is not usually critical and most common applications use a Gaussian kernel. This function requires the choice of a bandwidth parameter and the results obtained from a network will be sensitive to this value. As the RBFN relies on the calculation of a distance metric, it is usual to standardise input data vectors before training.

If the unit's bandwidth parameter is learned, its 'footprint' — the region it responds to — can be grown or shrunk, and so can adapt to give fine-grained coverage of subsets of inputs which are locally very dense, or alternatively to cover much or all of the training data set.

The region of response of an individual node can be adjusted to be non-radial (an elliptical rather than circular footprint), with a chosen or trained long axis, by applying a linear transformation to the input data vectors before presenting them to the nonlinear hidden layer. This has the effect of rotating the response region and expanding it in certain directions (thus enhancing the response in those directions), while contracting it in other directions (so suppressing the response in those directions). A motivation for building this flexibility into an RBFN is if we suspect there are irrelevant input variables or directions. Such an RBFN is known as an *elliptical basis function network* or EBFN. This can be implemented by incorporating an extra linear hidden layer feeding into the radial hidden layer. If irrelevant data are considered very probable, the number of units in the linear hidden layer can be made smaller than the number in the nonlinear (RBF) layer.

The selection of the number of centres, their location, and the parameters for the RBF is a combinatorial problem and there is potential to automate the process by creating a hybrid algorithm. The idea of neuroevolution is discussed in Chap. 15.

13.6 Support Vector Machines

Support vector machines (SVMs) [71, 117, 642], although drawn from statistical learning theory rather than a biological inspiration, are another popular supervised learning methodology. They bear some similarity with RBFNs. Like RBFNs, SVMs can be used for classification, regression and prediction and are particularly suited to binary classification, that is, where there are two classes (though this is not essential). They aim to fit the training set data well, while avoiding overfitting, so that the solution generalises well to new instances of data. In this section, we briefly outline the application of SVMs for classification. More detailed introductions to SVMs, including their application for prediction purposes, are provided in [121, 559, 647].

Classifiers are simpler to deal with and have better properties if the interclass boundaries are linear (Fig. 13.17). Thus, linear boundaries between classes are preferable. In general, these are called *hyperplanes*: linear or affine

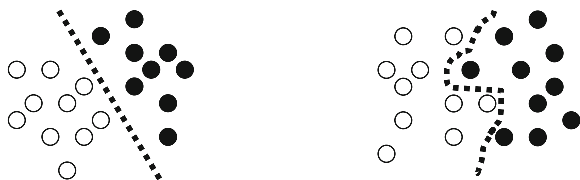


Fig. 13.17. A linearly separable two-class dataset (left) compared with a nonlinearly separable two-class dataset (right).

subspaces of dimension $n-1$, where n is the dimension of the space of linearly separable classes. Examples of hyperplanes include a line in 2-D space, a plane in 3-D space, or a 3-D space in 4-D space. Thus a hyperplane splits the n-dimensional space into two parts, in the positive and negative directions along the nth dimension either side of the hyperplane. The *separating hyperplane theorem* (a special case of the famous Hahn-Banach theorem for Banach spaces) says:

Theorem 13.4 (Hahn-Banach). *Let V be a finite-dimensional Banach space and let A and B be disjoint closed convex subsets of V. Then there exists a hyperplane H in V such that A lies on one side of H and B lies on the other, and H does not intersect either A or B.*

Separating hyperplanes may exist even if one or both subsets are nonconvex.

In this context, the input data items are often called *patterns*. Denote the input data space or *pattern space* by X. In X, the class boundaries may be nonlinear. Cover's theorem [118] states that under certain assumptions, nonlinearly separable patterns in X can be mapped nonlinearly to another, possibly high-dimensional, vector space F, in which their images are linearly separable. F is called a *feature space*. However, finding an explicit nonlinear map $\varphi : X \longrightarrow F$ is computationally expensive. Assuming we have such a map φ, the dot product in X is now transformed as $x_i \cdot x_j \longmapsto \varphi(x_i) \cdot \varphi(x_j)$. This mapping to F is carried out in order to make input data corresponding to two classes separable using a hyperplane (Fig. 13.18).

Figure 13.19 shows a simple example of a dataset of items with two characteristics divided into two classes in $\mathbb{R}^2$ (with class labels $\times$ and $+$) which are not linearly separable. However, when mapped into the higher dimensional 'feature space' $\mathbb{R}^3$ via the mapping $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^3 : (x, y) \longmapsto (x, y, x^2 + y^2)$, the two classes are now linearly separable; the separating plane is $z = 12.5$ (Fig 13.20).

Although many possible separating hyperplanes exist, a SVM aims to find the *maximally separating hyperplane* in the feature space in order to generate a classifier which will generalise to out-of-sample data. The maximally separating hyperplane is defined as the hyperplane which separates the nearest

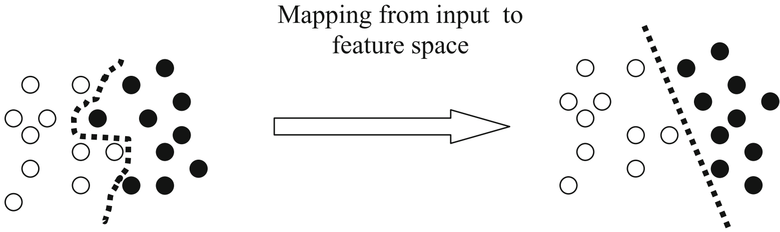


Fig. 13.18. In constructing a SVM classifier the object is to find a mapping, using a suitable kernel function, from the input space to the higher-dimensional feature space so that the resulting injection of the data is (ideally) linearly separable. For ease of illustration, a stylised mapping is shown here from a 2-D input space to a 2-D (rather than to a 2-D+) feature space.

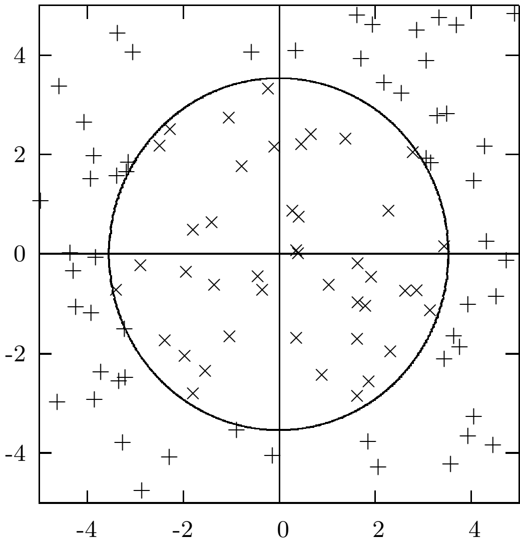


Fig. 13.19. A two-class dataset which is not linearly separable: the class labels are $\times$ and $+$

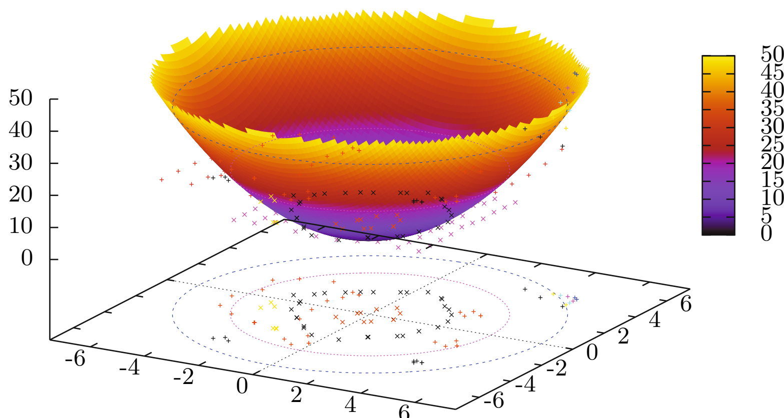


Fig. 13.20. The two-class dataset from Fig. 13.19 when transformed to higher dimension is now linearly separable: the separating plane is the horizontal plane $z = 12.5$, and the class labels are $\times$ and $+$. The $\times$ and $+$ points are actually on the paraboloid surface but for clarity are shown slightly away from it.

transformed data points in each distinct class as much as possible. In defining this hyperplane, not all of the transformed input data vectors are equally important. Some vectors will be critical in defining the class boundaries and these are termed *support vectors*. Support vectors are critical in that their removal from the dataset would alter the location of the maximally separating hyperplane. Figure 13.21 illustrates four support vectors and the corresponding maximally separating hyperplane. The distance between the support vectors of each class is known as the *margin* and consequently SVMs are also known as *maximum margin classifiers*. One interesting aspect of SVMs is that the complexity of the resulting classifier is characterised by the number of support vectors rather than by the dimensionality of the transformed space. This tends to make SVMs robust with respect to overfitting.

Maximising the margin gives rise to a constrained optimisation problem with a quadratic objective function: a quadratic programming problem. Its dual problem is phrased using Lagrange multipliers. It turns out that in this dual problem, the objective function and constraints are built *entirely* from inner products of unknown feature space vectors.

Thus, the formalism of SVMs only requires that the inner products of vectors in the higher dimensional feature space be computable. It is not actually

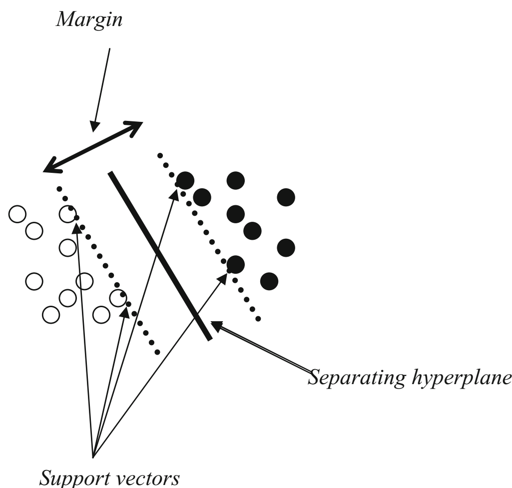


Fig. 13.21. Illustration of a maximally separating hyperplane. This maximises the distance between the support vectors associated with each class. In this case, the maximally separating hyperplane neatly splits the two data classes. This will not always occur, and more generally, the object is to trade off margin separation and misclassified training vectors.

necessary to have an explicit map $\varphi : X \longrightarrow F$: we can do the same job if we can find a function k which satisfies $k(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j)$ for all $x_i, x_j \in X$. Then the required computations will only use k. The core idea behind this and other kernel methods such as RBFNs is that a lot of computation can be avoided if we can find a proxy function k of the original input data patterns that does the same work as transforming to F composed with the inner product on F. This proxy function k is in fact a *kernel function*. It generally has a far lower computational cost than that of explicitly mapping vectors to the feature space followed by computing inner products there (the kernel trick) (Sect. 13.5.2). The kernel is the magic in SVM: implicitly, it transforms the messy *pattern* or *input* vector space to the higher dimensional *feature* space by a nonlinear mapping and saves computation; the classes in the feature space may be linearly separable even if they were not so in the original pattern space.

To be useful, a kernel must satisfy certain criteria, such as symmetry ($k(x_i, x_j) = k(x_j, x_i)$ for all $x_i, x_j \in X$) and positive semidefiniteness.[6] For positive semidefinite kernels, Mercer's Theorem gives an absolutely uniformly convergent representation of k in terms of an orthonormal basis, which can

[6] A kernel k is called positive semidefinite if for any positive integer p, and any $x_1, \ldots, x_p \in X$, the $n \times n$ (Gram) matrix K with $k(x_i, x_j)$ as the (i, j) entry is a positive semidefinite matrix, that is, $v^t K v \geq 0$ for all $v \in \mathbb{R}^n$.

provide information about the feature space. Also, new kernels can be built out of existing kernels in defined ways. Determining the 'best' kernel function, according to some measure of quality of fit to the input pattern dataset, is a difficult problem and the subject of ongoing research. The most commonly used kernels are:

i. Dot (inner) product kernel: $k(x_i, x_j) = x_i \cdot x_j = x_i^T x_j$;
ii. Polynomial kernel: $k(x_i, x_j) = (1 + x_i \cdot x_j)^p$;
iii. Gaussian kernel: $k(x_i, x_j) = \mathrm{e}^{-(\|x_i - x_j\|)^2/\sigma^2}$, also called radial basis kernel because of its use in RBFNs;
iv. Sigmoidal kernel: $k(x_i, x_j) = \tanh(kx_i \cdot x_j - \delta)$; however, this satisfies Mercer's theorem only for some values of k and δ.

Other standard kernels include multiquadric and inverse multiquadric kernels. Depending on the choice of kernel, the value of a number of associated parameters such as σ in the case of Gaussian kernel, is required. The choice of a kernel may be viewed as any or all of:

- choosing a linear 'feature space' representation of the data;
- choosing a measure of similarity for the data;
- choosing a basis function space for learning: the kernel determines the functional form of the solutions;
- choosing a covariance measure;
- choosing a distribution capturing prior probabilities of functions.

Thus, the choice of kernel should reflect domain knowledge about the problem.

In applications of SVMs there may be no hyperplane that cleanly splits the various class-labelled items, even in the feature space. For example, the training data may be noisy and/or may contain incorrectly labelled items. The algorithm's effectiveness would be decreased if an outlier such as an individual mislabelled input datum were able to significantly affect the hyperplane. It is preferable to be able to tolerate or ignore a proportion of outliers. Hence, in applying a SVM methodology, the objective function is altered to allow a trade-off between margin maximisation and a penalty for misclassifying some training data vectors. In calculating the penalty, the further a data item is away from a support vector for that class (on the wrong side), the greater the penalty. This is implemented by introducing slack variables which relax separation constraints in the quadratic programme, and then penalising these slacks in the objective function. The result is called a *soft margin classifier*, and can be shown to also depend only on inner products in the feature space, and so works with kernels. Soft margin methods are generally used in preference to the original maximum margin approach.

The next step is to determine the location of the optimally separating hyperplane. This is done using an iterative training algorithm which seeks to minimise an error function which combines margin separation and a penalty for misclassifications.

13.6.1 SVM Method

Algorithm 13.3 provides a generic overview of the (soft margin) SVM method.

Algorithm 13.3: Support Vector Machine Algorithm

Select a value for C. This is the penalty function applied when training data vectors are misclassified;
Select the kernel function and any associated parameters, using domain knowledge;
Solve the dual quadratic program derived the objective function;
Use the support vectors to determine the value of the primal threshold variable b;
Classify a new data vector x using $f(x) = \text{sign}(\sum_i y_i \alpha_i k(x, x_i) - b)$, where each y_i is the label for pattern i (a coefficient in the quadratic program), and each α_i is a Lagrange multiplier;

13.6.2 Issues in Applications of SVM

SVMs have become widely used classification tools and have produced excellent results on many real world problems. With improvements in the optimisation algorithms used, they also exhibit good scaling properties. SVMs are also robust with respect to noisy training data. As with all methods, however, SVMs have acknowledged drawbacks.

i. It is difficult to incorporate domain knowledge into an SVM, other than in the data preprocessing or kernel selection steps.
ii. The rationale for the resulting classification decisions can be hard to reverse-engineer, as the support vectors provide limited information to the modeller.
iii. SVMs were originally designed to work with real-valued vectors, so there is no unique way to incorporate noncontinuous data (for example, categorical data) into an SVM.
iv. The SVM methodology also requires that data vectors be scaled, and different methods of scaling can produce different results.

However, many of these drawbacks also apply to other methodologies, including MLPs and RBFNs, and hence are not unique to SVMs.

An important issue in implementing SVMs is choosing an appropriate kernel function and choosing good values for its associated parameters. This can be undertaken using a combination of domain knowledge and trial and error. However, an interesting line of recent research concerns the hybridisation of EC and SVM whereby EC methodologies have been used to uncover good parameters for a specific kernel function. More generally, genetic programming (Chap. 7) has been used to evolve problem-specific kernel functions [286, 605].

13.7 Summary

NNs consist of a family of robust, data-driven modelling methodologies. However, the earlier comments regarding the clarity of NN models should be borne in mind. A charge which is sometimes levelled against NN techniques is that they result in a *black box model* as it can be difficult to interpret their internal workings and understand why the model is producing its output. However, this criticism generally fails to consider that any truly complex, nonlinear system is unlikely to be amenable to simple explanation. Despite the powerful modelling capabilities of NNs, they do suffer from a number of practical drawbacks:

i. it is difficult to embed existing knowledge in the model, particularly non-quantitative knowledge,
ii. care must be taken to ensure that the developed models generalise beyond their training data,
iii. results from the commonly used MLP methodology are sensitive to the choice of initial connection weights, and
iv. the NN model development process entails substantial modeller intervention, and can be time-consuming.

The last two of these concerns can be mitigated by melding the methodology with an evolutionary algorithm. The resulting hybrid models are discussed in Chap. 15.

14

Neural Networks for Unsupervised Learning

In Chap. 13, a series of NN models were described which can be used for *supervised learning*. In supervised learning the output for an associated input vector is already known and is used to guide the learning process. For example, in training a multilayer perceptron (MLP) the weights on arcs are adjusted in response to the difference which arises at the output node(s) between the MLP's output and the correct, known, output for a given training vector. The network is therefore trained using a feedback mechanism.

An alternative form of learning is *unsupervised learning*. This occurs when there are no clear outputs available to provide feedback for the training process. A common example of this is data clustering in poorly understood datasets, where the number of clusters and their structure is not known in advance. The underlying assumption in clustering is that 'similar' objects are defined as data items that share common features. Hence, objects which are close together in the input, or feature, space should be grouped into the same clusters, with dissimilar items being grouped into different clusters. The number of clusters may arise naturally from the data. If it is too large — more than about ten — then the ability of a human being to understand the clustering may be diminished. In this case, an additional segmentation step may be applied, where 'similar' clusters are combined into a segment, e.g., using decision trees, until there are 10 or fewer segments.

Real-world applications of clustering include the mining of customer databases, the classification of plants and animals, gene clustering, fraud detection, data compression and image analysis.

Clustering presupposes that there is a measure of the 'similarity' of inputs, effectively, a distance measure or metric on the space of inputs. As clustering methods aim to identify which objects are most similar to each other, a critical issue is how similarity or distance between items is measured. Clustering methods therefore seek to assign cluster labels to data vectors so that intracluster distances (those between members of the same cluster) are small and intercluster distances (those between distinct clusters) are large. For example, in Fig. 14.1, the data form three natural clusters.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_14

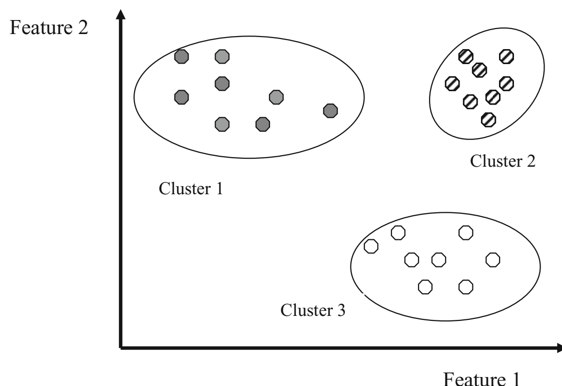


Fig. 14.1. Example where data splits neatly into three clusters, based on the two input features of each item

A wide variety of algorithms have been developed for clustering purposes including k-means [384], fuzzy C-means [172], fierarchical clustering [304], and mixture of Gaussians [151]. With the growth of the field of natural computing, a multitude of naturally inspired clustering algorithms have been developed, including algorithms based on evolutionary processes and brood-sorting behaviours (Chap. 9.7). Perhaps the best-known family of naturally inspired clustering algorithms is the self-organising map (SOM), first introduced by Teuvo Kohonen [332, 333]. They are also called Kohonen Maps or self-organising feature maps (SOFMs).

14.1 Self-organising Maps

Self-organising maps are loosely inspired by the self-organising capability of neurons in the cortex. Experimental evidence has shown that certain parts of the brain perform specific tasks such as processing touch, sound and visual stimuli. In these regions neurons spatially self-organise, or cluster, depending on their function. Inspired by these processes of self-organisation, SOMs are artificial neural nets which use unsupervised learning to adapt (organise) themselves in response to signal inputs. SOMs have been utilised for a large variety of clustering and classification problems in domains as diverse as speech recognition, medical diagnosis and finance [138, 238, 334]. Kohonen describes the main principle of SOMs [332] as follows

> ... in a simple network of adaptive physical elements which receives signals from a primary event space, the signal representations are automatically mapped onto a set of output responses in such a way that

the responses acquire the same topological order as that of the primary events.

A SOM consists of two layers,

(a) an input layer which serves as a holding point for the input data; this layer has as many nodes as there are input variables;
(b) a *mapping* layer which makes up a low-dimensional grid, typically two-dimensional, of *models* of the data.

The two layers are fully connected to each other; each mapping layer node has an associated weight vector, with one weight for each connection with the input layer, as well as a position in the map space (Fig. 14.2).

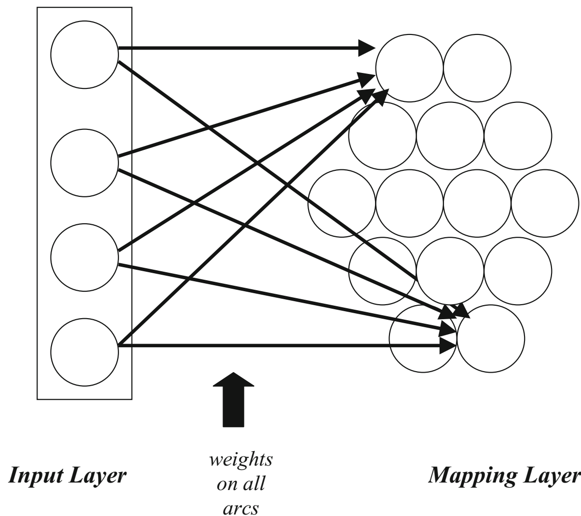


Fig. 14.2. A SOM with a 2-D mapping layer. On grounds of visual clarity, only the connections between the input layer and two of the mapping layer nodes are shown

We can think of a mapping layer node as being a form of detector, with a weight vector which is that node's model of the input data: the weight vector 'points to' a location in the space (usually $\mathbb{R}^n$) of input data. A SOM acts to project (compress) the input data vectors onto the mapping layer. The aim of the SOM is to group 'like' input data vectors close together on the mapping layer. The method is therefore *topology preserving* or *locality preserving*, since items which are close in the input space are also close in the mapping space.

An input vector will be mapped to the mapping layer node whose weights are most similar to those of the input vector. Thus, the SOM is a similarity graph and also a clustering diagram. This compression onto two dimensions has the extra advantage of aiding visualisation and interpretation and this is one of the main uses of SOMs. Of course, as with any projection, there is some information loss but the SOM acts as a nonlinear Principal Components Analysis (PCA). It picks out the directions of greatest dispersion in the data, minimising the information lost by reducing the number of dimensions.

14.2 SOM Algorithm

Algorithm 14.1 outlines the general training algorithm for the SOM. In this algorithm:

Algorithm 14.1: Self-organising Map Algorithm

```
Choose the number m of mapping layer nodes;
Initialise the weight vectors for each of these nodes;
repeat
    for each vector x = (x_0, x_1, ..., x_{n-1}) in the training dataset do
        for each mapping layer node i ∈ {1, ..., m} do
            Calculate the distance between the training vector x and the
            weight vector w_i using
                d_i = sum_{j=0}^{n-1} (x_j - w_ij(t))^2            (14.1)
        end
        Select the mapping node i* that has the minimum value of d_i;
        for each neighbouring mapping node k of i*, including i* itself do
            Update the weight vector for node k using
                w_k(t+1) := w_k(t) + α(t) h_{i*,k}(t) (x - w_k(t))   (14.2)
        end
    end
until weight vectors stabilise;
```

$$d_i = \sum_{j=0}^{n-1} (x_j - w_{ij}(t))^2 \quad (14.1)$$

$$w_k(t+1) := w_k(t) + \alpha(t) h_{i^*,k}(t)(x - w_k(t)) \quad (14.2)$$

- n is the dimension of the input data vectors: $x = (x_0, x_1, \ldots, x_{n-1})$, so there are n input nodes;
- there are m mapping layer nodes: m is determined during the initialisation phase;
- each node $i \in \{1, \ldots, m\}$ in the mapping layer has a weight vector $w_i = (w_{i0}, w_{i1}, \ldots, w_{i,n-1})$, so there are $n \times m$ weights in total;

- α is the learning rate of the map; and
- $h_{i^*,k}$ defines a neighbourhood function from mapping layer centre i^* to neighbour k, e.g., a radially symmetric Gaussian $h_{i^*,k}(t) = \exp\left(\frac{-\|r_k - r_{i^*}\|^2}{2\sigma(t)^2}\right)$ where $\|r_k - r_{i^*}\|$ is the distance between node k and the BMU i^* in the two-dimensional grid (r_k is the position vector of mapping node k in $\mathbb{R}^2$).

Note that (14.2) is a vector equation: all n components of w_k are updated. Both the neighbourhood size and the learning rate decay during the training run, in order to fine-tune the developing SOM. The neighbourhood size controls how many mapping layer nodes are adjusted (learn) for each data vector.

Initially most mapping layer nodes respond to every data vector, enhancing exploration and reducing the chance of getting stuck in a local minimum; as the neighbourhood size decreases, fewer mapping layer nodes respond (but may respond more strongly, depending on the rate of decrease of α), enhancing exploitation. In the early stages of the algorithm, a broad brushstroke picture of the input data distribution is learned with finer details being filled in as the neighbourhood size shrinks in later iterations of the algorithm. A slow rate of neighbourhood shrinkage reduces the danger of premature convergence to a locally optimal representation.

As the calculation of a distance metric is required in SOM training, input data vectors are typically standardised. Methods of standardisation include dividing each column of input variables by its standard deviation, or the standardisation of each column of inputs based on their range (e.g., $x^* = \frac{x - \min_{y \in X}\{y\}}{\max_{y \in X}\{y\} - \min_{y \in X}\{y\}}$). During training, a sample vector is drawn randomly from the input data set. The nodes in the mapping layer *compete* for the input data vector and the winner is the mapping node whose vector of incoming connection weights most closely resembles (is nearest to) the input data vector. The winner, or *best-matching unit* (BMU), has the values of its weight vector adjusted to move them towards the values of the input data vector, thereby moving the location of the BMU towards the location of the input data item. The training process is unsupervised as it does not use any explicit outputs. The process is based solely on measures of similarity between the input data vectors and weight vectors associated with each of the nodes on the SOM's mapping layer.

An important component of the training process is that not only the BMU, but also its neighbouring nodes on the mapping layer are adjusted in each training iteration. These neighbouring nodes also have their weight vectors altered to become more like the input data vector, resulting in a form of *cooperation* between these nodes. This is the local learning/plasticity referred to earlier.

As more input data vectors are passed through the network, the weight vectors of the mapping layer nodes will self-organise. By the end of the training process, different parts of the mapping layer will respond strongly to specific

regions of input space. The self-organisation process also encourages the mapping layer weight vectors to congregate to regions of the input space where the training data is concentrated, with relatively few (if any) weight vectors being located in sparsely populated regions of the input space. The self-organising map therefore tends to approximate the probability density function of the input data.

Figure 14.3 provides a stylised illustration of a trained SOM for the three data clusters in Fig. 14.1. The 12 (in this simple example) mapping layer nodes have self-organised so that the weight vectors for four nodes have moved towards each cluster of data in the feature space (only the weight vectors have changed, not the nodes' locations in the two-dimensional map).

Once training of the network is complete, the clusters obtained can be examined in order to gain insight into the underlying data. Although the original dataset may have been of high dimension, with complex nonlinear relationships between individual variables, its compression into a two-dimensional visual map allows the user to consider, for example, what data items have been grouped together and what are the typical values for each input in a specific cluster.

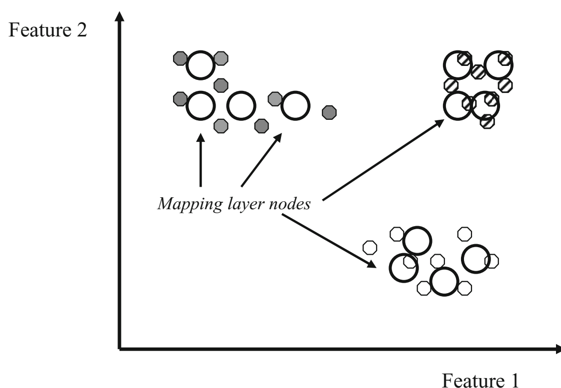


Fig. 14.3. A stylised illustration of a trained net (in the data space, not the mapping layer). Four mapping nodes have migrated to each data cluster during the training process

14.3 Implementing a SOM Algorithm

When implementing the general algorithm outlined above the modeller faces a number of choices, including the choice of method of weight initialisation

for mapping layer nodes, the choice of topology of the mapping layer, the choice of neighbourhood function, the choice of distance measure used when determining which mapping node is closest to a training vector, and the choice of learning method used to update the weight vectors of the mapping layer once a winning node is determined. Each of these is discussed in following subsections.

Initialisation of Weight Vectors

Three common methods exist for initialisation of the weight (also known as 'codebook' or 'model') vectors:

i. random,
ii. grid, and
iii. input data.

Under random initialisation, each element of each weight vector is assigned a random value, typically in the range $[-1, 1]$ or $[0, 1]$ (depending on the range of normalised input variables). While this is a simple method of weight initialisation it will tend to slow down the training of the SOM if in fact the input data is highly clustered in the input space. In an extreme case, if the data is highly clustered in small regions of input space and nodes in the mapping layer are initialised randomly, many of the mapping nodes may remain unused in the forming of the topological map (as they, and their neighbours, are never chosen as the BMU).

A grid approach can also be used whereby the weights are assigned in a grid pattern which is designed to give a uniform coverage of the input space. However, this can result in problems similar to those of random initialisation if the training data is densely clustered in certain regions of input space.

An input data-based approach is to assign each mapping layer node an initial weight by randomly sampling from the training vectors. Assuming that the random sample is representative, this method has the advantage that weight vectors will be located in the denser areas of the input space when training begins, which should help speed up the training process.

Other methods of initialisation use statistical properties of the input data to get better initial values for the weight vectors. One such approach is to use *Principal Component Analysis* (PCA).

PCA is the application of an orthogonal (i.e., length-preserving) change of basis to the vector space $\mathbb{R}^n$ of input data, so that in the new basis the data's greatest variance lies along the first axis (called the first principal component), its second greatest variance lies on the second coordinate, and so on. Since we are changing basis so that the new basis vectors point along the directions of greatest dispersion, we must first centre the data on the zero vector, that is, subtract off the sample's vector mean to make the zero vector the centre of gravity of the sample.

PCA amounts to diagonalising the $n \times n$ covariance matrix of the data, with the transformed variances placed on the main diagonal in descending order. The variances in this new basis are the eigenvalues of the original covariance matrix and the principal components are the corresponding eigenvectors. PCA is a kind of factor analysis. The first few principal components capture most of the variance of the sample and often the later ones are just ignored as not contributing much to dispersion. Figure 14.4 shows an example of the first two principal components of a dataset.

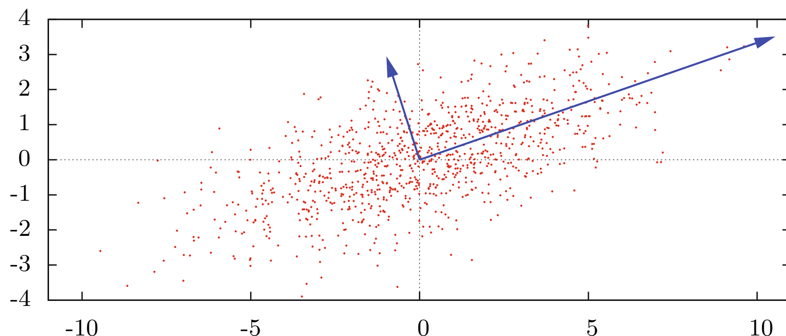


Fig. 14.4. First two principal components of a dataset.

Under the PCA approach to mapping layer weight initialisation,

- find the first two principal components of the input data, that is, the two directions in $\mathbb{R}^n$ which account for most variance in the training sample;
- then use a grid approach on the two-dimensional subspace spanned by these principal components, that is, choose the initial weight vectors to lie in the 'fattest' two-dimensional slice of the data.

This approach of weight initialisation has two advantages:

i. the SOM is partly organised to start with;
ii. the algorithm can begin with a narrower neighbourhood function h and a smaller learning rate α.

These can lead to an order-of-magnitude speedup in the SOM runtime.

Topology of the Mapping Layer

In selecting the topology of the mapping layer, three key choices are faced:

i. how many dimensions should the mapping layer have?
ii. how many mapping nodes should there be?

iii. what neighbourhood structure should mapping layer nodes have?

Typically SOM mapping layers are two-dimensional, as this assists in the visualisation of the resulting map. The selection of the number of nodes trades off the granularity of the resulting map against its generalisation capability. Larger maps will produce finer-grained clustering detail but they tend to generalise less well because of this. Larger maps also take longer to train as a greater number of mapping nodes must be examined in order to find the BMU in each training iteration. The number of nodes is also constrained by the number of data observations available for training purposes, and typically for clustering applications, the number of mapping nodes will only be a fraction of the number of training data samples.

In the organisation of the mapping layer, the most common approach when the map is two-dimensional, is to use a rectangular or a hexagonal topology (Fig. 14.5). A hexagonal grid of nodes is generally preferable to the naked

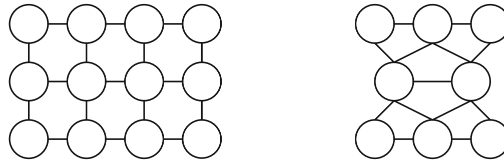


Fig. 14.5. Two forms of SOM topology, rectangular (left) and hexagonal (right)

eye, for ergonomic reasons. The grid of map nodes must be oriented along the distribution of the inputs x. Since the models are to approximate the distribution of the inputs, the width and height of the grid should more or less correspond to the major dimensions (e.g., the eigenvalues [variances] of the two principal components) of the distribution of input data.

Distance Measure

The distance measure is used when determining which mapping node the BMU is. The simplest metric to use is the Euclidean distance, with the BMU being the mapping node whose weight vector is closest to the input vector under this distance metric. The formula for calculating Euclidean distance in a two-dimensional case, for vectors x and y, is:

$$\mathrm{dist}(a, b) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{14.3}$$

In earlier chapters, some algorithms including the GA employed a binary string representation. For these representations, a natural distance metric between two binary strings is the Hamming distance, or the number of bit-flips required to make two binary strings equivalent. The two distance metrics are

of course linked if the input vectors have binary form, as in this case the Euclidean distance metric is equivalent to the square root of the Hamming distance metric.

Neighbourhood Function

The size of the neighbourhood and the form of the neighbourhood function impact on the computational effort required in training the SOM. The larger the neighbourhood, the greater the number of nodes that must be updated in each iteration; and the more complex the neighbourhood function, the greater the computational effort in performing each update.

A basic, and often effective, approach is to define a *bubble* neighbourhood function around each mapping node. If a particular mapping node is the BMU, the weight vectors of all its neighbours within this bubble are updated using the same update equation as the BMU. Weight vectors for nodes outside the bubble are not updated.

The size of the neighbourhood need not be kept constant during the training process and is usually reduced as the algorithm runs in order to produce a finer-grained map. If the neighbourhood is large, many mapping nodes will have their weight vectors altered on each training iteration: the self-organising occurs on a global scale. The effect of this, if all neighbours receive the same update, is that each element of the mapping node weight vectors will initially tend towards the average values of that element in the training data. Hence, viewed as models of the data, the mapping nodes will tend to cluster together in the data space. As the neighbourhood shrinks over time, fewer mapping nodes are involved in each weight update, and the mapping nodes will tend to separate in order to approximate the distribution of the training data: the weights converge to local estimates.

A more complex approach is to define a neighbourhood function using a Gaussian kernel, whereby the weight vectors of the closest neighbours to the BMU receive a greater update than those of neighbours which are further away. This can be formulated as:

$$h_{\mathrm{BMU},k}(t) = \alpha(t) \exp\left(\frac{-\|r_k - r_{\mathrm{BMU}}\|^2}{2\sigma(t)^2}\right) \tag{14.4}$$

where k is a neighbour of a specific BMU and $\|r_k - r_{\mathrm{BMU}}\|$ is the distance between node k and the BMU in the two dimensional grid (r_k is the position vector of mapping node k in $\mathbb{R}^2$).

Learning Rate

The learning rate is usually formulated as a decreasing function of the number of iterations of the SOM algorithm. A simple formulation is to apply a linear function such as:

$$\alpha(t) = \alpha(0)\left(1.0 - \frac{t}{\text{iter_max}}\right) \tag{14.5}$$

where t is the current iteration of the algorithm and maxits is the maximum number of iterations that the algorithm will run for. The parameter $\alpha(0)$ is the initial learning rate, and this is usually set above 0.5. The object in adapting the learning rate as the algorithm runs is to allow larger weight changes in the early training of the nodes in the mapping layer in order to form a reasonable quality course-grained mapping from the inputs to the mapping layer. This initial mapping is then fine-tuned using smaller weight vector adaptations.

In some applications, we may not have a large set of training data. However, for good statistical accuracy, the learning process may need a significant number of training steps. The solution is that if the number of available training data is small, they can be reused during training. The data may be used cyclically or in a random order; in practice it is typically found that ordered cyclic use of the available data is not noticeably worse than statistically robust sampling approaches.

Forming Clusters

In large SOMs, we may perform cluster operations on the map itself. After the training of the map is complete, the mapping layer nodes will have self-organised so as to approximate the probability density function of the input data. In most data mining applications, the number of useful clusters in the data is not known in advance; hence a clustering algorithm must be applied to the map in order to uncover these. A common approach is to use a hierarchical agglomeration clustering algorithm such as Ward's method [649]. In this approach, each mapping node is initially considered to be a discrete cluster. At each step in the clustering algorithm, the union of every possible cluster pair is considered, and the two clusters whose merging results in the smallest increase in 'information loss' (defined as the sum of errors squared between each node and its cluster centroid) are combined. This will iteratively tend to produce larger cluster groups and therefore a smaller number of clusters in the mapping layer. Additional information on the quality of the clusters produced by the algorithm is used to fine-tune the clusters and to decide when to terminate the clustering process.

14.4 Classification with SOMs

Although SOMs are clustering algorithms, they can be used to create classifiers. Before this can be done, each mapping node needs to be labelled once the clustering process has been completed, using training data for which the class outputs are already known or by using human expertise to assign class labels to the generated clusters.

If the first approach is taken, the map is initially created using only the input component of the training data vectors. Once this is done, each training data vector is shown to the map again, and the winning node is assigned the known label of that training vector. Once all the training data have been shown to the map, each mapping node will have zero, one, or more (possibly conflicting) labels associated with it. In the former case, if the user wishes to ensure that all nodes are labelled, a k nearest neighbours approach can be applied so that each unlabelled node is assigned the class label of its k nearest labelled node neighbours. In the case where a mapping node has conflicting labels associated with it, a majority voting decision could be applied in order to decide the final class label for that node. Once the mapping nodes are labelled, out-of-sample data vectors for which the classification is not known can be classified by presenting them to the map and determining the class label of the mapping node to which the input vector is closest.

Learning Vector Quantisation

Once the initial assignment of class labels has been made, it is possible to fine-tune the labelling using learning vector quantisation (LVQ), a supervised learning methodology [333]. Under LVQ, the weight vectors for mapping nodes are iteratively updated using:

$$\Delta w_j = \begin{cases} \eta(x_i - w_j) & \text{if } x_i \text{ is classified correctly,} \\ -\eta(x_i - w_j) & \text{if } x_i \text{ is classified incorrectly} \end{cases} \tag{14.6}$$

where x_i is each input data vector in turn, and w_j is the weight vector for the mapping node j which is closest to x_i. Typically a small value is set for η in the range $(0.01 \rightarrow 0.02)$, and it reduces to 0 as the algorithm runs. The object of the fine-tuning step is to pull weight vectors of mapping nodes in separate classes away from each other in order to improve the delineation of the class boundaries on the map.

14.5 Self-organising Swarm

The design and implementation of algorithms for unsupervised learning is not limited to a neural metaphor. A recent extension of the particle swarm algorithm (PSO) (Chap. 8) has produced another methodology for self-organisation, and, like a SOM, this can be applied for unsupervised learning and classification purposes. This methodology is known as self-organising swarm (SOSwarm) [464, 466].

The self-organising swarm bears some similarity to a SOM with the adoption of a two-dimensional mapping layer and an unsupervised learning methodology. However, in SOSwarm, the nodes in the mapping layer are considered to be particles which possess position and velocity components. The

particles adapt over time using variants on the velocity (14.7) and position update (14.8) equations governing the canonical particle swarm algorithm (see Chap. 8.2 for a discussion of these equations):

$$v_i(t+1) = v_i(t) + c_1 r_1 (y_i - x_i(t)) + c_2 r_2 (\hat{y} - x_i(t)) \tag{14.7}$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{14.8}$$

The canonical form of the PSA update embeds two key elements:

i. a history, and
ii. a social influence.

Recall that history is embedded in the PSA via the momentum term and the p^{best} components of the velocity update equation. The social influence is embedded via the influence of either g^{best} or l^{best} in the velocity update (g^{best} also embeds a swarm 'history'). These two elements can be embedded in SOSwarm in a number of ways.

A simple initial approach in adapting (14.7) for SOSwarm is to embed particle history using the momentum term only, omitting the explicit social influence term g^{best}. In its place, drawing on the SOM, a neighbourhood topology is defined for all the particles in the mapping layer, such that each particle is affected if any of its neighbouring particles moves. This results in a form of social learning in that neighbouring particles influence each other. During the training of the map in each iteration of the particle swarm algorithm, the swarm is perturbed using the current training data vector as l^{best}. Just as for the SOM, the training process is unsupervised. By labelling the mapping nodes after the unsupervised learning process is complete, the final mapping layer can be used for classification purposes. Algorithm 14.2 provides an outline of the SOSwarm algorithm for a classification application.

In order to determine the firing particle (the particle that is the closest match to an input vector) a distance measure is calculated. A number of alternative error functions could be adopted, such as Euclidean distance:

$$\text{Firing particle} = \text{argmin}_i \left\{ \sqrt{\sum_{j=1}^{d} (x_j - r_j^i)^2} \right\} \tag{14.9}$$

where x corresponds to the input vector, r^i is the ith particle's position vector (of which r_j^i is the jth component), m is the number of particles in the swarm, and d corresponds to the dimension of the vector or particle.

A visual representation of SOSwarm is presented in Fig. 14.6. The mapping layer is two dimensional and the nodes (particles) in the mapping layer are arranged a priori into a fixed grid neighbourhood topology. The firing particle, that is the particle whose position vector is closest to the current input vector (designated as g^{best}), updates its position vector according to the canonical PSO velocity and position update equations. In addition, particles

Algorithm 14.2: Self-organising Swarm Algorithm

Initialise particles in mapping layer randomly;
repeat
  for *each training vector* $x_0, x_1, ..., x_{n-1}$ *in turn* **do**
    Set g^{best} to be the input vector;
    Set p^{best} of each particle at its current position;
    Find particle with closest match to g^{best};
    Denote this particle as the firing particle;
    Update firing particle and its neighbour's velocity and position vectors;
  end
until *terminating condition*;
Assign class to each particle using known labels of training data;
Calculate classification accuracy using test data;

lying within a fixed neighbourhood of the firing particle also adjust their position vectors using the same equations, implicitly embedding a form of social communication between neighbouring particles.

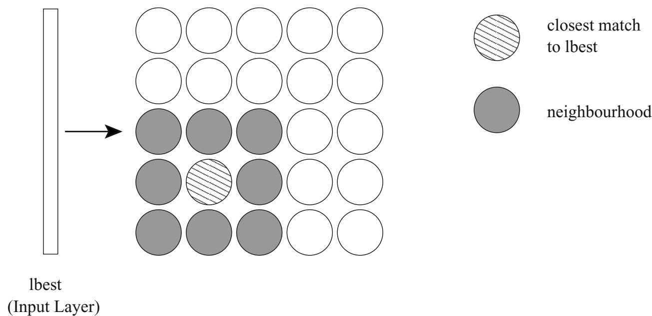


Fig. 14.6. A self-organising swarm (SOSwarm) with a two-dimensional mapping layer.

Once the map has been trained, each node in the mapping layer is assigned a class label using, for example, a simple majority voting scheme. In calculating the in-sample and out-of-sample classification accuracy, the distance between each input data vector and each mapping node is calculated, with the input data vector being assigned the class label of the closest mapping layer node.

14.6 SOSwarm and SOM

A close examination of the formulation of the PSA and the SOM suggests that there are links between the two paradigms. In the canonical SOM, the update of a firing node's weight vector is governed by:

$$x_i(t+1) = \eta(t)h(t)(\beta_i - x_i(t)) \tag{14.10}$$

where x_i is the firing node's weight vector, η is the time-varying learning rate and β_i is the input vector. Hence, after firing, the weight vector of the relevant node in the mapping layer and those of its neighbours which are defined by the neighbourhood function $h(t)$ are adjusted in order to more closely resemble the input vector. Ignoring the update of neighbouring nodes, and thinking of (14.10) in particle swarm terms, it is apparent that it can be written as a velocity update:

$$v_i(t+1) = \eta(t)(\beta - x_i(t)) \tag{14.11}$$

The component $\eta(t)$ in (14.11), in effect a weight parameter, is similar in concept to the weight parameter c_1 in (14.7). Hence, the canonical SOM update equation can be closely approximated by a reduced (nonmomentum) version of the canonical PSA update equation.

SOM-PSA Hybrids

The parallels between a SOM and a reduced form PSA suggest multiple possibilities for the creation of new hybrid algorithms for self-organisation. For example, the PSA embeds momentum, a form of personal particle history which is not included in the canonical version of SOMs. Like the learning rate in SOMs, the momentum term in the PSA velocity update is time-varying and it reduces over time in order to encourage particle convergence. The SOSwarm algorithm described above includes momentum.

Another interesting possibility, drawing on the use of peer learning in the PSA, is the incorporation of an additional 'peer-learning' term into the SOSwarm velocity update. For example, a topology consisting of a series of small overlapping neighbourhoods could be defined between the particles before the algorithm started, with (14.7) being extended by the addition of the term $c_3 r_3(y_{\text{local}} - x_i(t))$, where y_{local} is the location of a randomly selected neighbouring particle of $x_i(t)$. This social learning would tend to lessen the disruptive impact of an anomalous input vector during the learning process. This could prove especially useful in environments where training data is noisy or errorful.

Another approach is to recognise that SOSwarm extends SOM by including not only position but velocity, the time derivative of position. On the principle that including extra terms in a Taylor series enhances accuracy, it is reasonable to ask whether including acceleration (time derivative of velocity) terms as well as velocity and position terms could give a more effective or self-adapting algorithm. However, initial studies show mixed results [402].

14.7 Adaptive Resonance Theory

Adaptive Resonance Theory (ART), developed by Stephen Grossberg, Gail Carpenter and others [94, 95, 227, 228], describes a series of self-organising neural network (NN) architectures for problems such as pattern recognition, clustering, classification and prediction. Its original focus was the study of biological systems. It is based on physiological theories of cognitive abilities of humans and animals — in particular, how the brain learns over time — which may incorporate such things as the effects of changing ion concentrations on neurotransmitter operation. Typically, ARTs use unsupervised learning methods, though supervised methods are sometimes used, e.g., ARTMAP [95].

The main intuition behind ART models is that when an animal encounters an object, its brain tries to process it as follows. First, it compares the object to each of its stock of known categories. If the encountered object is 'reasonably' similar to one of its known categories, the object is considered to be known and is assigned to that category; otherwise, the object is considered to be *novel*, and a new category is created for it.

ARTs self-organise with the purpose of addressing the 'stability/plasticity' dilemma. This dilemma arises because any learning system must be able to adapt to important new information (exhibit plasticity) while not forgetting already learnt relationships (exhibit stability). In particular, the stability property means that previous learning should not be affected by irrelevant new information. The ART adaptively switches between plasticity and stability according to whether it detects *novelty* in the input. Of course, similar issues arise in the SOM, with the plasticity-stability dilemma being typically handled through a gradual reduction in the learning rate in the SOM, which in essence limits the plastic period of the map.

The various versions of ART are *online* learning approaches; that is, data items are presented to the system one at a time. Over time, the ART develops a memory of these inputs and which inputs should be assigned to the same category/cluster. When a new item is presented, its effect depends not only on its own characteristics (the 'bottom-up' current sensory experience of the learner) but also on the accumulation of previous learning (the 'top-down' expectation derived from previous learning and categorisation). The learning algorithms follow the 'leader follower clustering' approach. A cluster is represented by its centre (mean); only the cluster centre most similar to the new data item is adjusted; the adjustment is to make the 'resonating' cluster centre more like the new data item.

There are similarities to SOMs, but also significant differences. The main purpose of ARTs is to model cognition in humans and animals. Thus, the terminology used is very much inspired by biology, and less so by standard machine learning or mathematical terminology.

14.7.1 Unsupervised Learning for ART

Structure of ART Neural Network

In the basic unsupervised learning approach, an ART is a neural network containing two connected *layers* of neurons (nodes), called the *comparison layer* and the *recognition layer*, along with a vigilance parameter ρ and a reset function. The layers are also called the *fields* of the ART. Together, they make up the *attentional subsystem.* The reset function makes up the *orienting subsystem* of the ARTs.

The comparison layer corresponds to the SOM input layer while the recognition layer corresponds roughly to the SOM mapping layer, except that it is variable in size and behaves differently. Each recognition layer neuron i has a vector of weights w_i and has a connection to each of the other recognition layer neurons $1, 2, \ldots, i-1, i+1, \ldots, m$, allowing it to inhibit the other recognition neurons' outputs. This is called lateral inhibition and distinguishes ART approaches from SOM approaches, which actually reinforce learning of neighbouring neurons. The weight vector of a neuron is the centre of the cluster represented by that neuron. Initially, no recognition layer neurons are trained or 'committed'. As time goes on, and a new cluster is recognised, a new recognition layer neuron is committed to that cluster. In a sense, the ART NN starts with no active recognition layer neurons and builds them up one at a time.

The vigilance parameter $\rho \in [0, 1]$ is a threshold below which training is prevented. Its purpose is to address the 'stability/plasticity' dilemma. It controls whether already learnt relationships should be modified — and so partially forgotten — by the new item of information, or whether a new cluster should be created to represent that new item. It also controls the cluster granularity, as explained below.

The reset function (orienting subsystem) detects novelty and controls training, by comparing the recognition quality to the chosen value of the vigilance parameter ρ; based on this, it switches the ART between its stable and plastic modes. Thus, the orienting subsystem controls the overall dynamics of the attentional subsystem when a mismatch occurs between the comparison and recognition layers.

Working of ART Neural Network

The ART NN works as follows. The comparison layer is presented with an input vector x (as with the MLP input layer), for a period of time dependent on the type of ART and the training method used. Each recognition layer neuron i outputs a negative signal, proportional to the quality of match between w_i and x, to each of the other recognition neurons $1, 2, \ldots, i-1, i+1, \ldots, m$ and so laterally inhibits the outputs $w_1, w_2, \ldots, w_{i-1}, w_{i+1}, \ldots, w_m$. Then x is mapped to its 'best match' in the recognition layer, that is, the recognition layer neuron j whose weight vector w_j matches x most closely. The best

matching unit (BMU) is typically taken as the neuron with smallest Euclidean distance $\|x - w_j\|$ from x, though other measures of goodness of match may be used. This BMU j is thought of as the currently 'firing' recognition layer neuron. Thus, each recognition neuron can represent a category/cluster, and the input vectors are clustered into these categories.

Once x has been classified, the reset function compares the recognition match quality (the distance $\|x - w_{\mathrm{BMU}}\|$ between the BMU's cluster centre and the input datum) to the vigilance parameter ρ. If $\|x - w_{\mathrm{BMU}}\| < \rho$, x is considered to match the BMU; the BMU is updated or 'committed' to that input x, and training begins; that is, the BMU's weights are adjusted to be closer to x.

If $\|x - w_{\mathrm{BMU}}\| \geq \rho$, the BMU is inhibited until a new input vector is presented; a search procedure is started. During this search, the reset function disables recognition layer neurons one by one, until $\|x - w_i\| < \rho$ for some neuron i (the vigilance threshold is met by a recognition match). Training begins only when the search completes.

Otherwise (that is, $\|x - w_i\| \geq \rho$ for all neurons i, or no neuron is committed, as would happen on the first iteration) a new neuron is committed, with its cluster centred on the datum: $w_{\mathrm{new}} := x$. Thus, the number of clusters in the data is not predetermined but is itself discovered from the data.

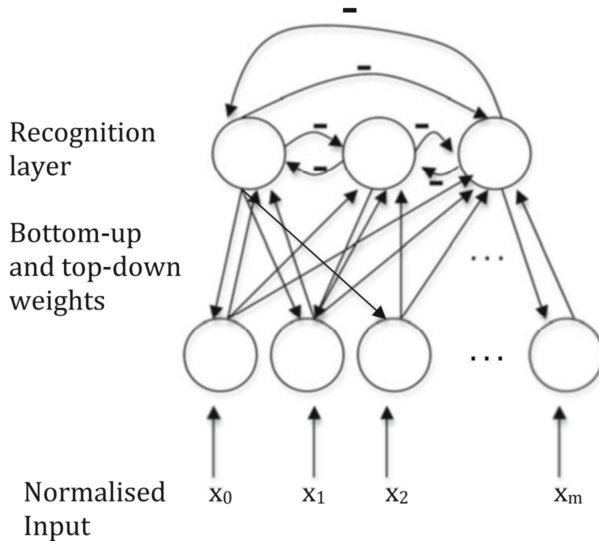


Fig. 14.7. Generic adaptive resonance network. The first (comparison) layer receives inputs from the environment and from the second (recognition) layer (mediated by a set of top-down weights). The second layer receives inputs from the first layer, mediated by a set of bottom-up weights. Note: not all bottom-up/top-down connections are shown here.

Choice of Vigilance Parameter

The choice of the vigilance parameter ρ crucially influences the behaviour of the ART. Its effect is to control the level of detail of the ART's memories. A large ρ gives fewer, more general categories, which can be thought of as 'broader brush stroke' memories. Conversely, a smaller ρ leads to more and higher precision categories, which can be thought of as memories with a higher level of detail. A very small value of ρ may give a cluster for each datum, an outcome of no benefit. Thus, a good value of ρ needs to be determined by the user, by trial and error, on an application-specific basis.

ART Training

ART training consists of the adjustment of the 'firing' recognition neuron's weights towards the input data vector x. It may be either *slow*, using differential equations, or *fast*, using algebraic equations. In slow training, the degree of training depends on the length of time for which the input vector is applied and is computed as a real number vector. In fast training, binary values are used. Fast learning is fast, efficient and works for many applications. However, slow learning is considered closer to what happens in brain processes (a major motivation for the developers) and has the advantage that it can be used with a continuously varying input vector.

14.7.2 Supervised Learning for ARTs

A supervised ART-based learning system may be obtained by combining two basic ARTs (each of the same type, either real number or binary), to give a structure known as ARTMAP or Predictive ART [95]. Here, the first ART takes the input data and the second ART takes the associated output data. The system then makes the smallest possible modification of the first ART's vigilance parameter that suffices to classify the data correctly.

14.7.3 Weaknesses of ART Approaches

ART NNs have been criticised for having several undesirable properties, including: being strongly dependent on the presentation order of the input data; not giving consistent estimates of the best cluster centres; and having high sensitivity to noise.

14.8 Summary

Unsupervised learning is a powerful approach for uncovering structures in populations of objects when little is known about the relationship between

these objects. In this chapter we have introduced a variety of NN models which adopt an unsupervised learning approach, including the well-known SOM algorithm. A short introduction to Adaptive Resonance Theory is also provided. Of course, other natural metaphors can be applied for the purposes of unsupervised learning, and a clustering methodology drawn from a particle swarm inspiration (self-organising swarm) was also introduced in this chapter.

One practical drawback of the neural net models discussed in the last two chapters is that the user must select both a model structure and the values of a number of parameters in order to apply the model. This presents a nontrivial problem given the huge number of combinatorial possibilities facing the user when making these choices. The results obtained from the networks are usually sensitive to these choices. Although rules of thumb exist to assist in this task, the construction of a quality neural network can be time-consuming, requiring substantial trial and error. This opens up the possibility of creating a hybrid model in order to automate these choices. The next chapter provides an introduction to neuroevolution, the use of evolutionary methodologies for this task.

15

Neuroevolution

In this chapter we discuss *neuroevolution* — the application of an evolutionary process to uncover quality neural network models. While the chapter will focus on the evolution of MLPs, the concepts can be carried over to the evolution of other NN structures as well.

The construction of an MLP entails the determination of the appropriate model inputs, model structure and connection weights. In all but toy problems this results in a huge search space of model possibilities which cannot be efficiently examined using a modeller-driven trial and error process. An evolutionary algorithm (EA) such as the GA provides scope to automate some or all of the MLP model generation process, by blending the model induction capabilities of an MLP with the optimising capabilities of an evolutionary algorithm (or more generally, any other optimisation algorithm).

There are several ways that a neuroevolutionary hybrid can be constructed. The first possibility is to use an EA to uncover a subset of good-quality model inputs from a possibly very large set of potential inputs. A second use of an EA is to evolve the node-arc structure of an MLP network. Weight optimisation in MLPs is also problematic, particularly for large networks with many weights. Even if attention is restricted to a specific learning algorithm (for example, backpropagation) a modeller is still required to make good choices for the parameters of the learning algorithm. An EA could be used to evolve the choice of learning algorithm and associated parameters. Of course, for a given (fixed) MLP structure, an EA could also be used to directly evolve good connection weights for each arc in the MLP. In summary, an EA can be used to select from choices for any or all of the following:

i. model inputs,
ii. number of hidden layers in the MLP,
iii. number of nodes in each hidden layer,
iv. nature of transfer functions at each node,
v. connection structure between each node, learning algorithm and associated parameters, or

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_15

vi. connection weights between each node.

The first step, as with all EC applications, is to develop a suitable representation of the problem. Two main approaches are used in neuroevolution, a direct encoding where the structure of the MLP can be read directly from the genotype, or an indirect encoding which specifies a set of construction rules that are iteratively applied to produce the MLP (Sect. 15.3). In this chapter we concentrate on the use of direct encodings.

15.1 Direct Encodings

A *direct encoding* means that the MLP structure and/or its weights can be directly read from the genotype. The following sections illustrate how various aspects of an MLP structure could be directly encoded on a genotype and then evolved using an EA.

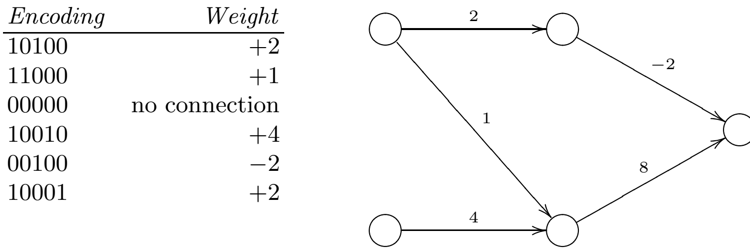
15.1.1 Evolving Weight Vectors

Suppose the objective is to uncover a good set of weights for a fixed MLP structure with two input, two hidden and one output node. This structure has six possible connections and therefore six weights. These weights can be directly encoded onto a chromosome. Figure 15.1 provides an example of how the weights for each of the six possible connections can be encoded as a binary chromosome. Each individual weight is encoded using a block of five bits, giving a total chromosome length of 30 bits. The first bit in each five-bit block indicates whether that weight is negative (a value of 0) or positive (a value of 1), with the remaining four bits (reading from left to right) encoding the weight value. A weight value of '00000' or '10000' indicates that there is no connection between two nodes. Of course, many other encodings of the weights could be used, including a higher-precision binary encoding or, preferably as it forms a more natural representation in this case, a real-valued encoding of the weights (producing a real-valued genotype).

Regardless of how the weights are encoded, the general evolutionary search process to find a high-quality set of weights for the MLP is the same. At the commencement of the training process each chromosome in the population of genotypes is decoded into a set of weights for the fixed MLP structure. An error measure for this network is obtained by passing the training data through the MLP and comparing the MLP's output with the known actual outputs for the training data. Lower errors represent higher fitness. An EA then acts on the population using fitness-based selection and the usual diversity-generation operators such as crossover and mutation in an effort to uncover better sets of weights over multiple generations.

Although the above method can be used to evolve weights for an MLP, it will not necessarily outperform specialist weight-adjustment techniques for

training MLPs such as quickprop [186]. Weight evolution does offer advantage in certain cases where backpropagation methods cannot be used, for example when a specialist error measure is required which is not a differentiable function, or in cases which backpropagation finds hard, for example highly multimodal error surfaces.



Encoding	*Weight*
10100	+2
11000	+1
00000	no connection
10010	+4
00100	−2
10001	+2

Fig. 15.1. Binary encoding of weights in an MLP. The first bit indicates whether the weight is negative (0) or positive (1) with the remaining four bits (reading from left to right) encoding the weight value. The weight 00000 indicates no connection between two nodes

15.1.2 Evolving the Selection of Inputs

Figure 15.2 and Algorithm 15.1 illustrate the general form of an EA-MLP hybrid when the MLP's structure and/or choice of inputs (as distinct from the MLP's weights) are being evolved. The elements of the MLP's structure are encoded onto a genotype. The weights for the connections in the MLP are obtained using a backprop (or other) learning algorithm and the resulting error measure is used to drive the evolutionary process.

If the intention is solely to evolve good sets of inputs for an MLP of otherwise fixed structure, a simple choice of representation is to use a binary string of length N where each element of the string is an indicator (0,1) as to whether each of the N inputs is excluded or included in the MLP. For each chromosome, corresponding to a specific selection of inputs, the MLP is trained using backprop and an error measure obtained which is then used to guide the evolutionary process in its search for the best set of inputs. While this process is intuitive, it does suffer from the *noisy fitness problem*. This issue is discussed in Sect. 15.1.5.

15.1.3 Evolving the Connection Structure

A slightly more complex task is to use an EA to evolve an MLP's connection structure. In this case the modeller must design a representation which is capable of encoding a broad range of possible network connection structures.

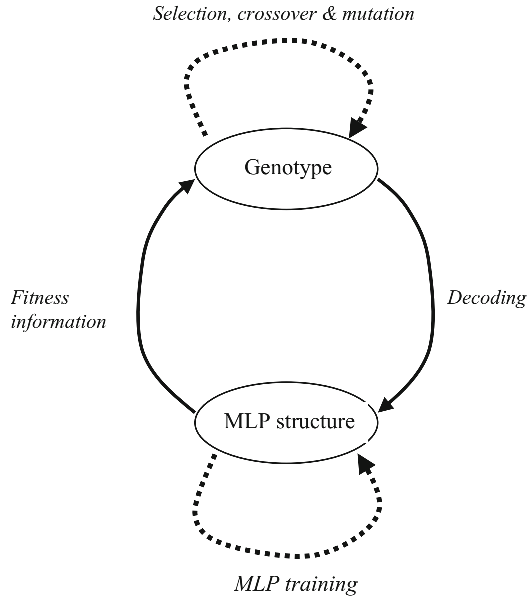


Fig. 15.2. Illustration of one form of a GA-MLP hybrid. The hybrid uses evolutionary learning to uncover a good structure but the weights are obtained using the backpropagation algorithm

Algorithm 15.1: Neuroevolutionary Hybrid Algorithm

Select genotypic representation for the MLP structure;
Generate an initial random population of n genotypes;

repeat
  for *each genotype in turn* **do**
    Decode the string into an MLP structure;
    Initialise the connection weights;
    Train the MLP using the backpropagation algorithm;
    Determine the fitness of the resulting MLP;
  end
  repeat
    Perform fitness-based selection of two parents;
    Apply crossover and mutation operators to generate new child solution;
  until *a new population of size n is created*;
  Replace the old population with the new one;
until *terminating condition*;

One way of directly encoding the connection topology is by means of a *connection matrix* or *adjacency matrix* of size $N \times N$, where N is the maximum number of possible nodes in the network (Figs. 15.3 and 15.4). The connection matrix consists of (0,1) values, each indicating whether that connection is used or *turned on* in the network. This representation implicitly allows the EA to select the number of hidden nodes, as a hidden node which is not connected to any input or output nodes is effectively redundant.

One drawback of this representation is that as N becomes large, the feasibility of employing a direct encoding declines, as the storage requirement of a connection matrix for N nodes scales at a rate of $O(N^2)$. Use of this representation also requires the modeller to define the maximum size of the MLP structure beforehand. A further problem is that if N is large, some of the initially generated networks could be complex, with many hidden layer nodes, even though the problem may not actually require this level of complexity. This could raise issues of poor performance generalisation out of sample.

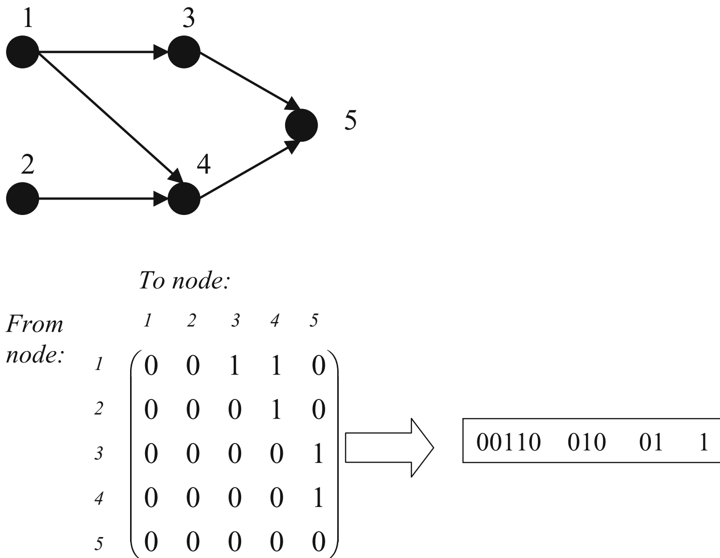


Fig. 15.3. A feedforward MLP connection matrix. The matrix can be concatenated into a binary string. Due to the feedforward architecture, the binary string only needs to contain the upper-right triangle of the matrix

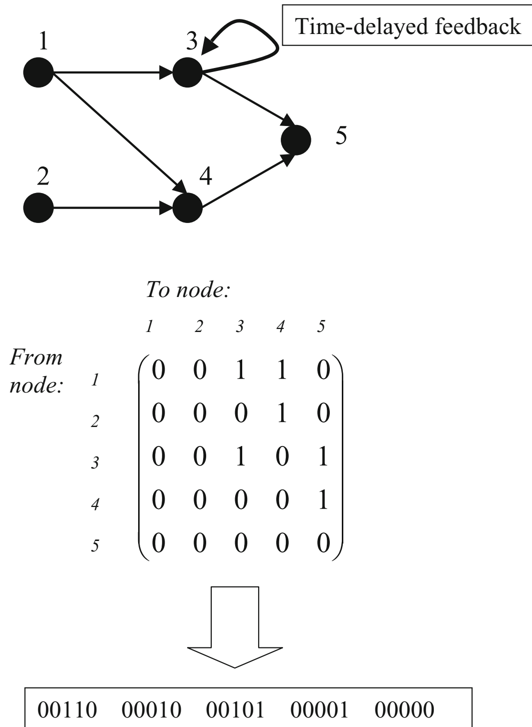


Fig. 15.4. A recurrent architecture encoded as a binary string. As feedback arcs are permitted, the binary string must encompass the entire connection matrix. The chromosome is formed by concatenating the matrix row by row

15.1.4 Other Hybrid MLP Algorithms

While the above discussion is framed in the context of using an EA to 'evolve' aspects of a neural network it is clear that many alternative optimising algorithms could be substituted for an EA in order to uncover good solution encodings. For example, if the object was to uncover a good connection structure (which can be represented as a binary-valued search problem) any optimisation algorithm capable of working with a binary representation such as BinPSO (Chap. 8) could be applied. Alternatively, if the objective was to uncover a good set of real-valued weights for a fixed MLP structure, any real-valued optimisation algorithm including, PSO, DE, ES, etc., could be used. For example, in using PSO to uncover good weight vectors for a fixed network structure, each particle could encode the weights for an entire MLP structure. Initially, the location of the particles in the swarm could be generated randomly and the fitness of each particle determined by passing the

training data through the MLP associated with that set of weights. The usual PSO velocity and location update equations could then be applied in order to undertake a search of the weight space [236]. Eberhart et al. [173] illustrate a comprehensive application of a PSO-MLP hybrid in which PSO is used to search for good inputs, for good network structures, and for good slope values for the transfer functions in an MLP structure.

15.1.5 Problems with Direct Encodings

Although the concept of evolving MLP structures using direct encoding schemes is quite intuitive, a number of problems arise in practice with the simple methods outlined in the above sections. These can reduce the efficiency and effectiveness of hybrid algorithms which use a direct encoding. Three issues are particularly troublesome:

i. noisy fitness,
ii. designing efficient crossover mechanisms, and
iii. generating over-elaborate MLP structures.

Noisy Fitness

When an evolutionary process is used to generate an MLP's structure and this is followed by the application of (for example) a backprop training process in order to determine the structure's fitness, the resulting fitness depends on both the MLP's structure and on the initialisation of the weights at the start of the (nonevolutionary) training process. Hence, a good network architecture could obtain a low fitness, and therefore be eliminated from the population, just because of an unlucky weight initialisation. One work-around for this problem is to train each MLP using several different weight initialisations and use the average or best of these results as the fitness of the genotype. A drawback of this approach is the extra run time required due to the multiple retrainings of each MLP.

Designing Efficient Crossover Mechanisms

While applications of directly encoded MLP-EA hybrids often produce good results, the efficiency of these hybrid algorithms can be lower than expected because of poorly designed crossover mechanisms. To illustrate this, assume that we are evolving the weights for a fixed MLP structure and that we have selected two parents for crossover (Fig. 15.5). Although the chromosomes for each parent will 'look different' in that not all the elements of each chromosome are identical in each locus (Fig. 15.6), in fact they represent the same MLP structure with the two hidden layer nodes permuted. This could produce an inefficient search process as the population of chromosomes starts to converge on a good region of the search space as crossover starts to produce children

which are less fit than their parents. This is known as the *permutation problem* or the *competing conventions problem* [242, 418].

In general, for a network with N hidden layer nodes, there are $N!$ functionally equivalent networks. To make matters worse, two MLPs with differing topologies can actually represent the same network (the *equivalent convention problem*).

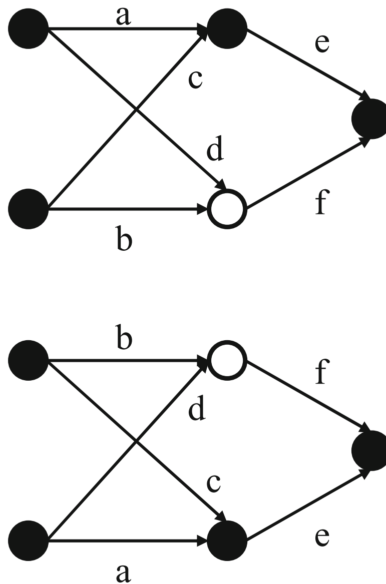


Fig. 15.5. Two identical MLPs with the same weights and structure, except their two hidden layer nodes are permuted

A number of different approaches to overcoming the permutation problem have been investigated with varying degrees of success (see Yao (1999) [677]).

More generally, this problem illustrates the issue of redundant representations which can arise in applications of natural computing methods. Representations are redundant if the number of possible genotypes exceeds the number of possible phenotypes (i.e. there is a many-to-one mapping between genotypes and phenotypes). Rothlauf and Goldberg [541] draw a distinction between *synonymously* and *nonsynonymously* redundant representations.

The former arise when genotypes that represent the same phenotype are similar to one another; the latter arise when the genotypes that represent a specific phenotype are different from one another. Nonsynonymously redundant representations make it difficult for variation operators such as crossover to work efficiently.

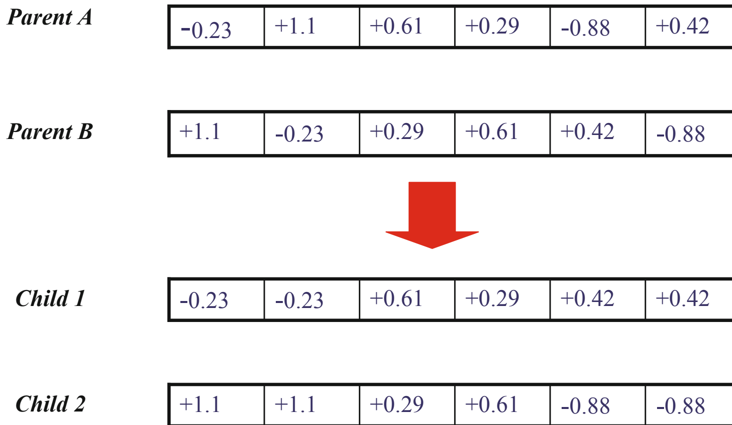


Fig. 15.6. Suppose the (identical) connection weights of parent A and parent B are in fact the optimal ones for the MLP (hidden layer nodes are permuted). Applying uniform crossover to the real-valued chromosomes produces children which are inferior to their parents

Over-elaborate MLP Structures

As mentioned in Sect. 15.1.3, over-elaborate MLPs can be generated when attempting to evolve good connection structures using representations such as an $N \times N$ matrix. This in turn can result in overfit of the training data and poor generalisation of the resulting MLP. One way of lessening this problem is to implement a penalty function in determining an MLP's fitness, whereby a network's fitness is reduced as its size increases.

15.2 NEAT

One approach, developed in 2002 by Stanley and Miikkulainen [591, 592, 593], which attempts to overcome the problems of evolving MLPs using a direct encoding, is NEAT (*neuroevolution of augmenting topologies*). NEAT simultaneously evolves both topology and weights. Proponents of NEAT claim that it:

i. applies a principled method of crossover (i.e. attempts to overcome the permutation problem),
ii. protects structural innovations (i.e. attempts to overcome the noisy fitness problem), and
iii. grows MLPs from a minimal structure (i.e. attempts to ensure that the final MLP is no more structurally complex than necessary).

In order to achieve this, NEAT uses three mechanisms, a novel MLP encoding which allows for 'sensible crossover', a speciation concept which offers some protection for structural innovations, and a seeding process whereby all initial MLP structures are of miminal size. Algorithm 15.2 provides an overview of NEAT. The algorithm is described in more detail in the following sections.

Algorithm 15.2: NEAT Algorithm

```
Generate an initial population of n (identical) genotypes of minimal
structure;
Decode one genotype into an MLP structure and assess fitness of all
members of the population;

repeat
    Select a random member of each species to represent that species;
    for each genotype in turn do
        Calculate shared fitness of genotype;
    end
    for each species in turn do
        Calculate number of members of that species in the next generation;
        Insert best current member of that species automatically into the
        next generation;
        Select best x% of each species for mating pool;
        repeat
            Randomly select parents from the mating pool and apply
            crossover to obtain a child;
            Apply mutation to newly-created child;
            Assign new child to an existing or a new species;
        until required number of children are generated;
    end
until terminating condition;
```

15.2.1 Representation in NEAT

The genetic encoding used in NEAT is illustrated in Fig. 15.7. Each gene encodes an individual connection between two nodes; therefore the genotype encodes an entire MLP structure and its associated weights. Each *connection gene* has four pieces of information, the index number of the input and output nodes for that connection, the connection's real-valued weight, an indicator as to whether that connection is 'enabled' (on) or 'disabled' (off), and the *innovation number* for the connection. Innovation numbers are assigned in sequence when a new connection is first created in the population of genotypes (when two nodes are linked for the first time), and *all* subsequent instances of that connection in the population have the same innovation number. Hence,

a connection gene between (say) nodes 1 and 4 will have the same innovation number for all members of the population of genotypes. The innovation number helps ensure that genotypes can be lined up sensibly during the crossover process.

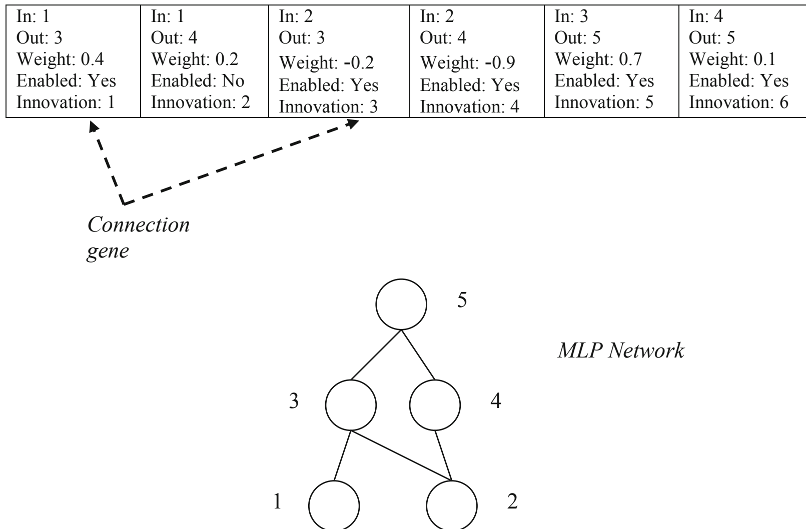


Fig. 15.7. A genotype in NEAT. This genotype has six connection genes (one of which is disabled) and encodes an MLP with two hidden layer nodes

15.2.2 Diversity Generation in NEAT

As in canonical ECs, new generations of genotypes are created using a select, reproduce and replace cycle. Each of these processes is described in the following sections.

Crossover in NEAT

The crossover operation in NEAT is based on the idea that nature only allows sensible, not random, crossover during reproduction. In contrast to the random crossover process in simple neuroevolutionary algorithms, NEAT encodes and uses information on the genotype's historical development (via its innovation numbers) in order to allow proper pairing of parent genotypes for crossover of their MLP structures.

Each connection gene representing a connection between two specified nodes will have the same innovation number for all genotypes in the population. Hence, genotypes can be lined up, and their common connection genes identified. The parents may also have disjoint and/or excess genes (genes which occur either inside or outside the overlapping range of the two parents' sets of innovation numbers), where one parent has a connection gene which the other does not. Figure 15.8 provides an illustration of the matching process highlighting matching and excess genes between two parent genotypes.

During the crossover process with two parents producing a single child, the child genotype inherits all connection genes which are common to both parents with the weight value being randomly selected from one of the parents. Disjoint and excess genes are inherited from the fitter parent. If a gene is enabled in one parent and disabled in the other, it is stochastically enabled/disabled in the child. A relatively strong selection pressure is typically applied in NEAT whereby the top 40% in each species (Sect. 15.2.3) are placed in a mating pool, with random selection from this pool in order to select parents. In generating children, a 25% cloning and a 75% crossover rate is suggested by Stanley and Miikkulainen (2002) [591].

One item of note in the operation of crossover in NEAT is that it does not usually play as significant a role as mutation in generating structural diversity in the population. As a child inherits all the disjoint and excess genes from its fitter parent, its basic structure will resemble that of the fitter parent. Hence, crossover primarily acts as a search of weight space around the fitter parent. In contrast, crossover in traditional neuroevolution, while running into the issue of competing conventions, does generate substantial structural diversity (akin to macromutation).

Mutation in NEAT

Three types of mutation are possible in NEAT, mutation of a real-valued connection weight, an 'add connection' mutation, and an 'add node' mutation. In an add connection mutation, a new connection gene is appended to a genotype which creates a connection between two existing nodes in an MLP. The weight for this new connection is generated randomly (Fig. 15.9). In the case of an add node mutation, a randomly selected existing connection is broken and a new node is placed at the break point. The connection into this new node is given a weight of 1 and the connection out of the node retains the old weight from the broken connection (Fig. 15.10). Typically, the weight mutation rate is higher than that for connections or node additions. Over time, the mutation process will tend to produce longer genotypes and hence more complex MLP structures.

15.2.3 Speciation

In evolving MLP structures, newly created topologies will often have limited fitness as their connection weights are not optimised when the new structure is

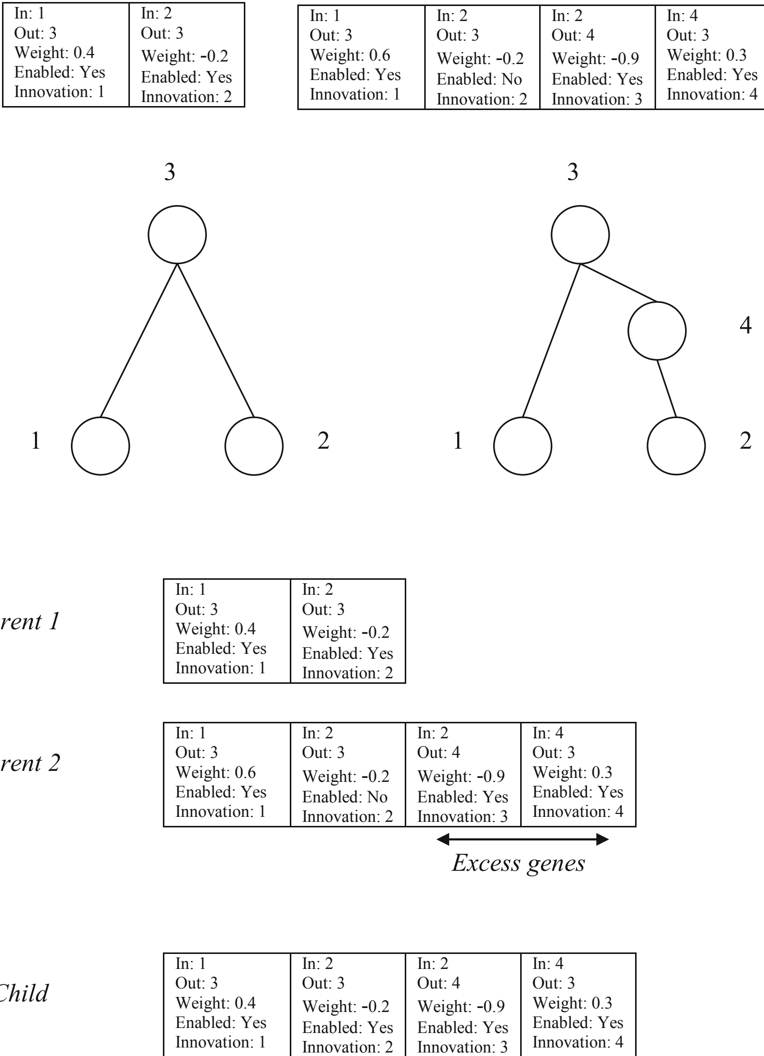


Fig. 15.8. Crossover in NEAT. Parent 2 is assumed to be the fitter; hence all its excess genes are inherited by the child. Stochastically, the connection between nodes 2 and 3 is turned on in the child, despite it being disabled in parent 2

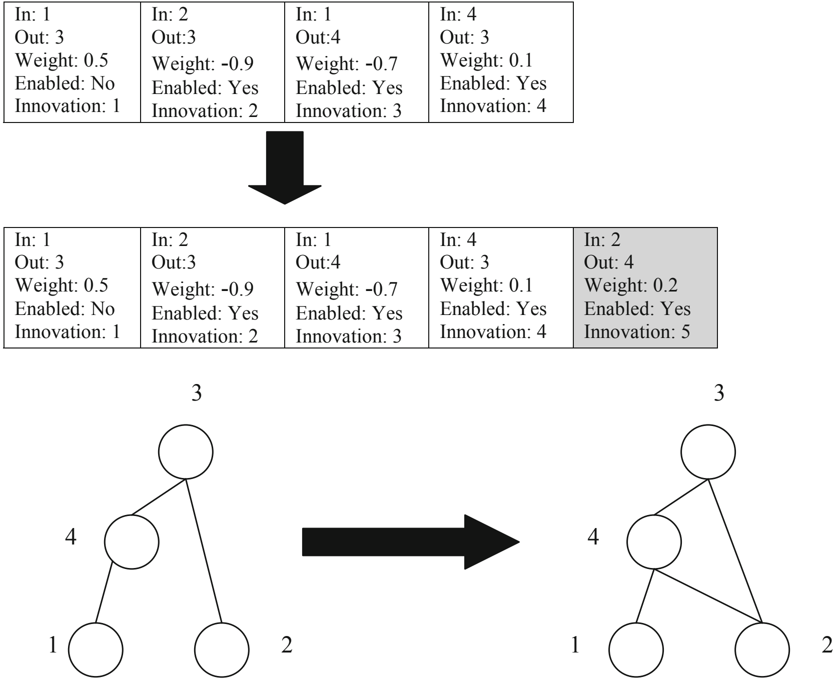


Fig. 15.9. An add connection mutation. Here a new connection is added between nodes 2 and 4 and a corresponding connection gene is inserted in the genotype

initially created. Hence the new structures may be promptly deselected before they have a chance to discover good weights.

To overcome this problem, NEAT uses speciation [385] in order to help protect new structures. The concept is similar to that of speciation in nature, where species compete in different ecological niches and the most direct competition for resources is between members of the same species. In NEAT, speciation is undertaken using the structural information contained in each genotype. The object is to determine which genotypes are most "similar" and therefore should be considered to belong to the same species. The degree of similarity between two genotypes is determined by looking at their connection genes: how many genes match, how many excess and disjoint genes are there, and what is the degree of similarity in the connection weights on the matching genes? A metric is calculated using:

$$\frac{c_1 E}{N} + \frac{c_2 D}{N} + c3\overline{W} \tag{15.1}$$

where N is the number of genes in the larger of the two chromosomes being compared, E and D are the number of excess and disjoint genes respectively,

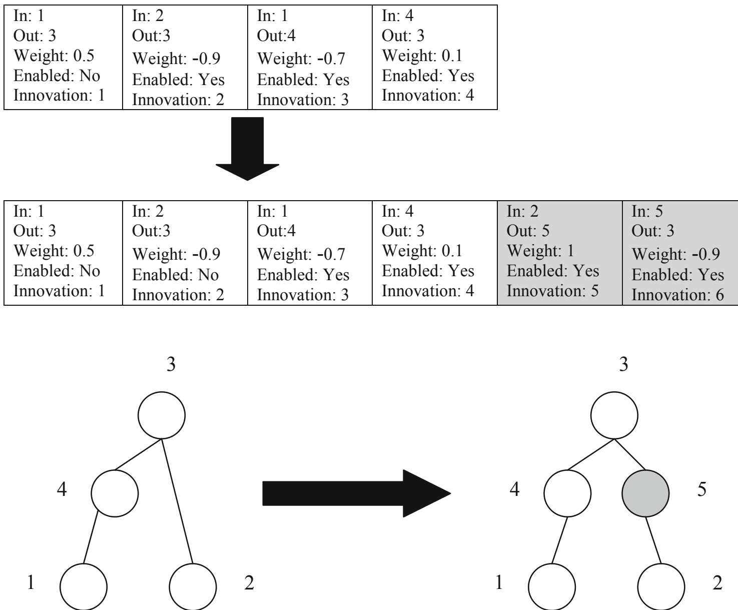


Fig. 15.10. An add node mutation. Here a new node is inserted between nodes 2 and 3 resulting in the addition of two new connection genes to the genotype

$\overline{W}$ is average weight difference of matching genes, and c_1, c_2, c_3 are the relative weights on each element of the metric.

When a new child genotype is created, the value of the above metric is calculated by comparing the similarity of the new genotype to a randomly chosen member of each species in the last generation of the NEAT algorithm. The new child is then assigned to the first species where the calculated distance is within a predetermined threshold value δ. If no species is found to be within this threshold distance, the genotype forms a new species.

Fitness-Sharing

In the selection step in NEAT, a fitness-sharing mechanism is used in order to encourage diversity in the population of genotypes. The use of fitness-sharing is driven by the observation that individuals within a species compete for the same resources and each species occupies a niche in the wider ecological environment. Shared fitness is calculated using:

$$f'(i) = \frac{f(i)}{\sum_{j=1}^{n} s(d(i,j))} \tag{15.2}$$

where $f(i)$ represents the original (unadjusted) fitness of genotype i and $f'(i)$ represents the shared (reduced) fitness of genotype i. The *sharing function s* provides a measure of the *density* of other species members within a given neighbourhood of a specific genotype i. In NEAT, a simple neighbourhood definition is used, namely the number of members of the species to which genotype i belongs. Hence, the shared fitness of a species member is its original fitness divided by the population size of that species. The use of fitness-sharing makes it difficult for any species to take over the population and helps protect and promote structural diversity.

The size of an individual species alters over time depending on whether the adjusted fitness of individuals in that species is higher or lower than the populational average fitness. The number of individuals in each species changes from one generation to the next according to:

$$N_j^{'} = \frac{\sum_{i=1}^{N_j} f_{ij}}{\overline{f}} \tag{15.3}$$

where $N_j^{'}$ and N_j are the new and old number of individuals in species j, f_{ij} is the adjusted fitness of individual i in species j and $\overline{f}$ is the average adjusted fitness of the entire population. In the reproduction process, each species is therefore assigned a different number of offspring in the next generation. Species reproduce by first eliminating the lowest performing members from the population. The best-performing $x\%$ of each species are then randomly mated to generate $N_j^{'}$ offspring, replacing the entire population of that species. An elitist selection process is also used whereby the best individual in each species automatically survives into the next generation, once the population has at least five members [591]. Species can become extinct, either when the number of new individuals in the species falls below 1, or when there is no change in the fitness of the best member of the species over multiple generations [591].

15.2.4 Incremental Evolution

NEAT begins with a uniform population of networks which have no hidden layer nodes and where all inputs are connected to the output node(s). Additional complexity in the form of hidden layer nodes or new connections is introduced as necessary via structural mutations. This approach implies that in earlier generations of the evolutionary process, search and optimisation is performed on small MLP structures, which makes weight optimisation within those structures easier.

Of course, a variety of other methods of initialisation could be used and the above approach will need to be altered for cases where the problem has a large number of input and/or output nodes. For example, using the above approach, a network with 40 inputs and 10 output nodes would require 400 connections, resulting in a high-dimensional search space. In contrast, initialisation of the population using, say, a hidden layer with three nodes would reduce the number of connections to 150.

15.3 Indirect Encodings

Apart from using direct encoding schemes it is possible to use an indirect encoding to generate and evolve MLPs. Examples of this approach include *grammatical* or *cellular encoding* [231, 331] (Chap. 18.2.3). In these methods the basic building blocks of the network are encoded in the form of a grammar (a set of rules which can be applied to produce a structure, in this case a complete MLP), and MLPs are developed by stringing together the building blocks defined in the grammar. These systems are generative in the sense that they do not assume the form of the MLP structure a priori.

A detailed discussion of grammars, and how they can be used to develop diverse structures, is provided in Chap. 17.

15.4 Other Hybrid Neural Algorithms

MLPs are only one family of neural networks, and concepts similar to those discussed in this chapter can be applied in order to automate the process of generating other forms of neural networks. Considering RBFNs for example, the modeller faces a number of design choices including:

- the number of model inputs,
- the locations of centres,
- the nature of RBF at each centre,
- the bandwidth of each RBF, and
- the weight on each centre's output.

While typically the final choices for the above parameters are made based on trial and error, there is no reason that the design process could not be automated. As for MLPs, the trick is to select an appropriate representation while minimising the effect of issues such as competing conventions and noisy fitness.

15.5 Summary

A practical difficulty in developing neural networks is that the creation of a quality network for a specific application can be a time-consuming process. Consequently, there has been significant interest in the possibility of automating some or all of this development process by creating hybrid algorithms for neural network construction. This chapter concentrates on the application of an evolutionary metaphor for the purposes of creating MLP networks, although the general principles can be applied to the creation of any form of neural network, and other optimisation algorithms could also be employed. A key issue arising is the need to carefully match the design of the diversity-generating operator to the choice of representation for the neural network.

In Chap. 16 we introduce another natural computing paradigm, *immuno-computing*, which draws inspiration from the workings of the immune system in order to create natural computing algorithms for optimisation and classification.

Part IV

Immunocomputing

16

Artificial Immune Systems

The immune system of vertebrates is comprised of an intricate network of specialised tissues, organs, cells and chemical molecules. The capabilities of the natural immune system include the ability to recognise, destroy and remember an almost unlimited numbers of *pathogens* (foreign or *nonself* objects that can enter the body, such as viruses, bacteria, multicellular parasites and fungi), and also to protect the organism from misbehaving self cells.

Two main strands of research comprise the field of *artificial immune systems* (AIS); namely, the modelling and simulation of the immune system in order to develop and test theories as to how it works, and the use of immune system metaphors in order to develop computational algorithms. This chapter concentrates on the latter.

The most common AIS algorithms can be broadly grouped into four categories (Fig. 16.1). This chapter concentrates on three of the most widely applied of these, the negative selection algorithm, which can be used for anomaly detection/classification, algorithms inspired by the clonal expansion process, which can be used for optimisation and classification, and danger theory, which gives rise to the dendritic cell algorithm which can be used for anomaly detection. A short introduction to natural immune systems is provided in the next section before these families of algorithms are examined.

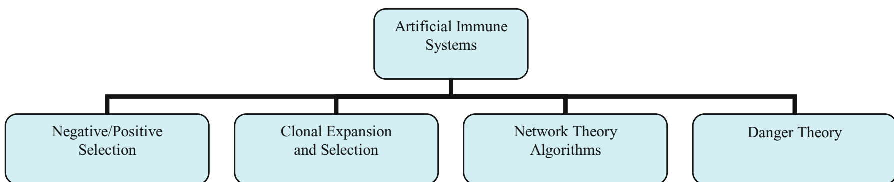


Fig. 16.1. Taxonomy of main AIS algorithms

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_16

16.1 The Natural Immune System

The human natural immune system has multiple-level defenses. The first lines of defence are barriers, such as skin and nasal hair, which physically block ingestion of pathogens. These barriers are supported by physiological defenses, including fluids secreted by the body (saliva, sweat and tears) which move pathogens out of the body and/or contain disruptive enzymes. In addition, humans have both an *innate* (or nonspecific) and an *adaptive* (or specific) immune system [48]. The innate immune system is present at birth and it does not adapt over a person's lifetime. In contrast, adaptive immunity is directed against specific pathogens and is modified by exposure to them, resulting in lifetime modification of the immune system. Thus a *memory* of previous invaders, and how to deal with them, is created and maintained by the adaptive immune system.

16.1.1 Components of the Natural Immune System

The immune system is comprised of a variety of molecules, cells and tissues. The most important cells are leukocytes (white blood cells) which can be divided into two major categories: phagocytes and lymphocytes. The first group belongs to the innate immune system while the latter group mediates adaptive immunity.

16.1.2 Innate Immune System

For a long time immunological research was dominated by the study of the adaptive immune system and it is only in recent years that we have begun to unravel the activities of the innate immune system, and its critical links with the adaptive immune system [405, 459]. The innate immune system uses a number of reliable signatures of foreign or nonself, such as *pathogen-associated molecular patterns* (PAMPs), to identify pathogens. PAMPs are molecular patterns that are broadly shared by pathogens but not by self molecules. If a nonself signature is detected the innate system triggers an inflammatory response in which components of the immune system attempt to wall off the invader and halt its spread. The inflammatory response results in symptoms such as redness and swelling that can occur at points of injury, as well as the fever and aches that can accompany infections. The inflammatory response is initiated by *Toll-like receptors* (TLRs) [458].

A total of ten TLRs are currently thought to exist in humans [458]. These recognise PAMPs, which are vitally important to the survival of pathogens such as bacteria, viruses, fungi and parasites. For example, TLR5 recognises a protein in the flagella used by bacteria to swim, TLR2 recognises a molecule which exists in the cell wall of bacteria and TLR3 binds to double-stranded viral DNA. It appears that TLRs have evolved over time to recognise molecules

that are fundamental components of a wide range of pathogens, and the ability to recognise these molecules has become encoded in our genome.

Therefore, TLRs play a key role in identifying the type of invading pathogen and mobilise the appropriate part of the immune system when required. The alarm sounded by the innate immune system when foreign molecules are detected is signalled by proteins known as cytokines that not only induce the inflammation response of the innate immune system but also activate the B and T cells that are needed for the adaptive response (see Sect. 16.1.3).

Cytokines are produced by various innate system cells including macrophages and dendritic cells, both of which are studded with TLRs. Macrophages circulate in the body looking for signs of infection. If they detect this they set off the inflammatory response, engulf the invader, and secrete cytokines which raise a general immune system alarm to recruit other cells to the site of infection. Dendritic cells (Sect. 16.1.4) ingest and subsequently present fragments of a pathogen's antigen to T cells in the lymph nodes and release signalling cytokines which indicate the level of danger associated with that antigen.

Innate immune systems (and TLRs) are not unique to humans. They also exist in many other species including insects and fish. Even plants are rife with TLRs [458].

Plant Immune Systems

Just like humans and other animals, plants have a wide range of potential pathogens to deal with, including viruses, bacteria, fungi, and nematodes (worms). In the first instance plants rely on physiological barriers (for example, the bark on a tree, thorns, or chemical secretions which are toxic to other organisms). These defenses can be overcome with plant pathogens such as bacteria gaining entry via wounds, or via water/gas pores, fungi gaining entry via plant epidermal cells, and insects invading via puncturing of the plant. The second layer of the plant's immune system is an innate immune system and this operates in a similar fashion to the innate immune system found in animals.

Unlike higher animals, plants do not have an adaptive immune system. They depend on the innate immunity of each cell and on systemic signals emanating from infection sites [307]. Pathogens are typically recognised either at the cell surface or inside the cell. In similar fashion to aspects of the innate immune systems in animals, individual plant cells can express receptors which recognise molecular structures (patterns) which are conserved over a broad range of pathogens. On the detection of pathogen molecules, a number of responses can be triggered in cells including cell wall thickening, production of anti-microbial compounds and host cell death (in an effort to stop the infection spreading to other cells) [307]. Defensive metabolites can also be transported within the plant in an effort to enhance resistance in tissue not yet affected by attack. Long-distance signalling in a plant may activate de

novo expression of resistance mechanisms in response to an attack elsewhere in a plant [260].

While the immune system in plants is clearly functional, the 'hard-wiring' of the immune system into plant genes causes problems when new pathogens emerge to which a plant species has no resistance. The only way for a plant species to obtain resistance to a new disease which the plant's innate immune system cannot deal with is via an evolutionary process (mutation) which can be slow. In contrast, vertebrates possess another line of immune defence, the adaptive immune system.

16.1.3 Adaptive Immune System

If a vertebrate's innate immune system cannot not swiftly remove a pathogen, the adaptive immune system swings into action. A key role is played in this process by lymphocytes which circulate constantly through the blood, lymph and tissue spaces. A major component of the population of lymphocytes is made up of B and T cells which are produced in the bone marrow. After their production in bone marrow, B cells remain there and mature whereas T cells migrate to the thymus for maturation.

B and T cells are capable of recognising and responding to certain nonself antigen (foreign molecules) patterns presented on the surface of pathogens (in the case of T cells) or antigen which has been expressed by an invading pathogen (in the case of B cells). A major role in the T cell recognition process is played by molecules of the *major histocompatibility complex* (MHC) [280]. These molecules act to transport peptides (fragments of protein chains) from the interior regions of a cell and present these peptides on the cell's surface. This mechanism enables components of the immune system to detect infections inside cells without having to penetrate the cell's membrane.

The control of adaptive immunity can be divided into two branches: *humoral immunity* which is controlled by B cells, and *cellular immunity* which is controlled by T cells. Humoral immunity is mediated by specially designed immunoglobulin proteins or *antibodies* contained in bodily fluids (or 'humors'), and it involves the interaction of B cells with antigens. Humoral immunity is aimed against extracellular foreign substances and micro-organisms [291]. Cellular immunity is aimed at intracellular micro-oganisms and plays an important role in the killing of virus-infected cells and tumours.

B Cells and T Cells

B cells and T cells have receptors on their surfaces which are capable of recognising antigens via a binding mechanism. The surface of a B cell contains

Table 16.1. Key immune system terms

Immune System Component	**Definition**
Pathogens	Foreign bodies including viruses, bacteria, multicellular parasites, and fungi.
Antigens	Any molecule that the immune system can recognise. Antigens may be nonself, for example, foreign molecules expressed by a pathogen. Antigens may also correspond to self molecules.
Leukocytes	White blood cells, including phagocytes and lymphocytes (B and T cells), for identifying and killing pathogens.
Antibodies	Glycoproteins (protein+carbohydrate) secreted into the blood in response to an antigenic stimulus that neutralise the antigen by binding specifically to it.

Y-shaped receptors (or *antibodies*). Antibody molecules possess two *paratopes* (each arm of the Y-shaped receptor, see Fig. 16.2). Paratopes consist of two regions, namely a constant region and a variable region. The constant region performs a number of functions, including the attachment of the antibody to the surface of the B cell and the variable region of the paratope is the part which can recognise and attach to specific antigens, or more precisely, to molecules such as proteins or fragments of proteins making up an antigen on the surface of a pathogen.

The regions on an antigen that a paratope can attach to are called *epitopes*. Identification of the antigen is achieved by a complementary matching between the paratope of the antibody and the epitope of the antigen. The match between the paratope and epitope need not be perfect. To increase the number of pathogens that the immune system can detect, individual lymphocytes can bind to a variety of antigens. This enhances the power of the immune system, as multiple lymphocytes will bind to an invading pathogen; therefore there will be multiple signals created in the immune system that an invader has been detected. The closer the match between paratope and epitope, the stronger the molecular binding between the antibody and the antigen, and the greater the degree of *stimulation* of the B cell.

A complex antigen can possess multiple epitopes and the T cell response against the antigen may focus on a subset of the antigen's epitopes (this phenomenon is known as *immunodominance*). The epitope on a molecule that provokes the most intense immune system response is considered to be *immunodominant*. Immunodominance can occur not only during infection but also following vaccination [536]. Ideally, a vaccine should circumvent immun-

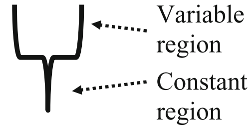


Fig. 16.2. Illustration of an antibody, showing the variable (paratope) and constant regions

odominance, thereby inducing responses to all epitopes, dominant and subdominant.

T Cell-Dependent Humoral Immune Response

When an antibody of a B cell binds to an antigen, the B cell becomes stimulated. The level of stimulation depends on the closeness or *affinity* of the match between the antibody and the antigen; high levels of affinity and stimulation corresponding to a close match. Once a threshold level of stimulation is reached, the B cell is *activated*. Before activation takes place, the B cell must be *costimulated* by a variant of the T cell population called *helper T cells*. When helper T cells recognise and bind to an antigen, they secrete cytokines which act as a signalling mechanism between the cells of the immune system. In addition to the cell-cell interaction, where the T cell can bind to a B cell, the secreted cytokines act on B cells to costimulate them.

Once the stimulation level of a B cell has reached a threshold level, the B cell is transformed into a *blast cell* and completes its maturation in the lymph nodes where a *clonal expansion* and *affinity maturation* process occurs. The object of the clonal expansion process is to generate a large population of antibody-secreting cells and memory B cells which are specific to the antigen. In the lymph nodes, activated B cells begin to clone at a rate proportional to their affinity to the antigen that stimulated them. These clones undergo a process of affinity maturation in order to better tune the cloned B cells to the antigen which initiated the immune system response. When new B cells are generated, the DNA strings that encode their antibody receptors (Fig. 16.2) are selected from gene libraries (Fig. 16.3) and are then subject to rearrangement via mutation and insertion processes. These processes mean that new forms of receptors are constantly created.

When a tailor-made detector is required for a specific novel antigen, the ability of the immune system to generate diversity is enhanced by means of a high mutation rate in the cloning process for the genes which encode the B cell's Y-shaped receptors (this process is known as *somatic hypermutation*). It is further enhanced by the preferential selection of the clones which best match the antigen. The evolutionary process of creating diversity and the subsequent selection of the variants of lymphocyte that become increasingly specific to an antigen is referred to as *clonal selection*.

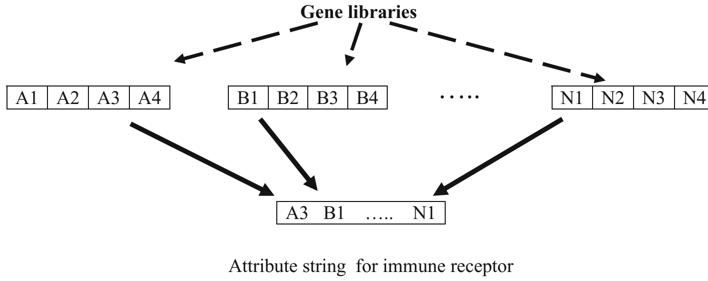


Fig. 16.3. Creation of an antibody molecule from gene libraries. The encoding of receptor molecules occurs through the concatenation of gene fragments from each gene library. Gene libraries act as a reservoir of gene fragments that have been found useful in the past for generating antibodies. Even with a relatively small number of genes in each library, a large number of distinct receptor molecules can be created. Somatic hypermutation increases the range of potential receptor molecules further.

In summary, the T cell-dependent humoral immune response is a series of complex immunological events. It commences with the interaction of B cells with antigens. The B cells which bind to the antigen are costimulated by helper T cells leading to proliferation and differentiation of the B cells to create B plasma and memory cells (Fig. 16.4). The new plasma B cells secrete antibodies (immunoglobulins) which circulate in the organism and mark the antigens by binding to them. These antigens and the associated pathogen are then targeted by the immune system for destruction. The steps in the process are:

i. Antigen-secreting pathogen enters the body.
ii. B cells are activated by the foreign antigen.
iii. B cells undergo cloning and mutation.
iv. Plasma B cells secrete immunoglobulins which attach to the antigen.
v. Marked antigens are attacked by the immune system.
vi. Memory of the antigen is maintained by B memory cells.

T Cell Tolerogenesis

A major challenge for the immune system is to ensure that only foreign or misbehaving self cells are targeted for destruction. In the normal creation of T cells their receptors are randomly generated and these cells could potentially bind to either self or nonself. To avoid auto-immune reactions where the immune system attacks its host organism, it is theorised that the cells are self-tolerised. In the case of T cells, this process of tolerogenesis takes place in the thymus. One mechanism for conferring self-tolerance to the lymphocytes

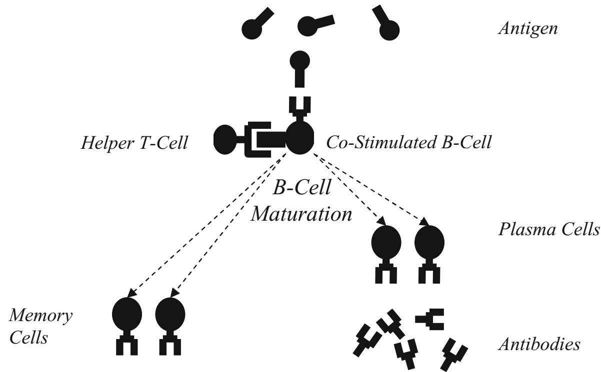


Fig. 16.4. B cell stimulation and differentiation

as they are maturing is exposure to a series of self-proteins. Any lymphocyte that binds to self-proteins is killed and only self-tolerised cells are allowed into the circulation system for lymphocytes. This represents a *negative selection process* as only nonself reactive T cells are permitted to survive.

Immune System Memory

If the immune system encounters an antigen for the first time, a *primary response* is provoked in the adaptive immune system and the organism will experience an infection while the immune system learns to recognise the antigen. In response to the invasion, a large number of antibodies will eventually be produced by the immune system which will help eliminate the associated pathogen from the body. After the infection is cleared, a memory of the successful receptors is maintained to allow much quicker *secondary response* when the same or similar pathogens invade thereafter. The secondary response is characterised by a much more rapid and more abundant production of the relevant antibody than the primary response. If a similar, but not identical, variant of the pathogen is later encountered a secondary response can be provoked by an antibody to the original antigen, which is a sufficiently close match for the antigens on the new pathogen. Therefore if a mutated version of the original pathogen is encountered, the immune system is already partly adapted to deal with it based on its previous learning. This is the concept underlying the process of immunisation against a disease using a nonharmful variant of that disease. Although there is debate as to the precise nature of how immune system memory is maintained, in broad terms the immune system maintains a population of long-lived lymphocytes or *memory cells*. Both T and B cells have memory variants. The creation of memory cells en-

sures that the results of past learning are physically encoded into the current population of lymphocytes.

16.1.4 Danger Theory

Although the concept of self and nonself provides a fertile ground for the development of algorithms for anomaly detection, a variety of alternative views of the working of the immune system exist. The *Danger Theory* proposed by Matzinger [393, 394] challenges the traditional self vs. nonself view of the immune system. A problem with the self vs. nonself perspective is that there are many instances of nonself that do not automatically trigger an immune reaction such as breathing air or ingesting food. The danger theory proposes that the immune system is activated by chemical signals of 'danger'.

As discussed in Sect. 16.1.3, T helper cells play a critical role in stimulating B cells when faced with an invading pathogen. In turn, it is known that T helper cells themselves require a costimulation signal from non-antigen-specific APCs (an antigen-presenting cell (APC) is a cell that displays foreign antigens complexed with major histocompatibility complexes (MHCs) on their surfaces). Under the danger theory, it is thought that APCs are activated by danger or alarm signals, such as molecular signals emitted by an injured cell (known as 'DAMPs', damage-associated molecular patterns), or the detection of the release of intracellular contents due to uncontrolled (necrotic) cell death. While there are several types of APCs, one of the most important are dendritic cells. These cells are able to collect an antigen along with any danger signals from their local environment and then integrate the signals to determine whether the environment is dangerous. If it is safe, on presenting the antigen to T cells, the T cells are tolerised to the antigen. On the other hand, if the environment is dangerous, the dendritic cell causes the T cell to become reactive when presented with the antigen.

While there is ongoing debate as to whether danger theory is a comprehensive framework of what drives immune system responses, there is no doubt that APCs play an important role in T cell stimulation. Many of the concepts of danger theory have become widely discussed in immunology [7]. The concept of danger theory has been applied to inspire computational algorithms, particularly the dendritic cell algorithm (DCA), which can be used for classification [223, 362]. The DCA is described in Sect. 16.4.

16.1.5 Immune Network Theory

Jerne [297] initiated the theoretical development of immune network theory in 1974. This views the immune system as a network of molecules and cells that can recognise each other and act in a self-organising manner to produce memory. In this theory, B cells interact to stimulate and suppress each other. Accordingly, the identification of antigens is not done by a single recognising set, but rather by a system-level network reaction. A variety of algorithms,

inspired by immune network theory have been developed, including aiNet [141], a modified version of CLONALG (Sect. 16.5.1) which incorporates a mechanism of suppression amongst the B cells.

16.1.6 Optimal Immune Defence

It is clear from even a brief description of the immune system that it is multilayered and complex. Shudo and Iwasa [577] point out that immune system defence activities have a cost to the organism, as does damage resulting from invading pathogens and harmful self cells. One way of thinking about the multiple mechanisms of the immune system is that the system seeks to minimise the sum of the damage caused by pathogens and the cost of defence activities. As the threat posed to the organism by a pathogen increases, more complex and costly modes of defence are called into action. The cost of maintaining a complex immune system may also help explain the variation in design of immune systems in different organisms. For example, only vertebrates possess adaptive immunity which relies on lymphocytes, and somatic hypermutation is found only in warm-blooded vertebrates [291].

16.2 Artificial Immune Algorithms

Even from the brief description of the natural immune system it is apparent that it is intricate, complex, and as yet imperfectly understood by immunologists. From a high-level perspective, it can be considered as a sophisticated information-processing system which possesses powerful pattern recognition and classification abilities. It also has the capability to adapt to new circumstances (problems) and can remember solutions to problems it has previously encountered. Hence, the immune system can serve as a metaphorical inspiration for the design of algorithms for optimisation and for pattern recognition.

Although a multitude of metaphors could be drawn from natural immune systems for the purposes of designing AIA, we will focus on three familes of these algorithms: negative selection algorithms, clonal expansion and selection inspired algorithms, and algorithms arising from a danger theory metaphor.

16.3 Negative Selection Algorithm

The negative selection algorithm is usually applied for classification purposes or to detect anomalies in the behaviour of a system. The basis of the negative selection algorithm is the ability of the immune system to discriminate between self and nonself, or more broadly to distinguish between two system states, normal or abnormal. Forrest et al. [199] developed a binary-valued negative selection algorithm analogous to the negative selection or self-tolerogenesis process during T cell maturation in the thymus. Later this was

extended to a real-valued representation. We concentrate on the real-valued algorithm in this chapter.

Canonical Real-Valued Algorithm

In implementing the algorithm, training data is usually normalised into the range [0,1] and a predetermined number of detectors are created at random positions in the input data space. During the training process (akin to tolerogenesis) any detector that falls within a threshold distance r_s of any member of the set of self-samples is discarded and replaced with another randomly generated detector. The replacement detector is also checked against the set of self-samples. The process of detector generation iterates until the required number N of valid detectors is generated. The 'rejection sampling' method attempts to ensure that all of the resulting detectors are potentially useful detectors of nonself. A variant on this training process occurs when a newly generated detector is discarded where the median distance between it and its k nearest self vectors is less than the threshold distance. The use of k nearest neighbours rather than just the nearest self vector makes the algorithm less susceptible to noise in the training data [214].

The pseudocode for the algorithm is described in Algorithm 16.1, where S is the set of self-samples, r_s is a predefined self-radius (a threshold distance) and it is assumed that the search space is bounded by an n-dimensional hypercube $[0,1]^n$ (Fig. 16.5).

Algorithm 16.1: Negative Selection Algorithm

```
Initialise the detector set D to be the empty set;
repeat
    Create a random vector x, drawn from [0,1]^n;
    for every s_i in S, i = 1,2,...,m do
        Calculate the Euclidean distance d_i between s_i and x;
    end
    if d_i > r_s for all i then
        Add x (a valid nonself detector) to set D;
    end
until D contains the required number N of valid detectors;
```

Once a population of detectors has been created they can be used to classify new data observations. To do this the new data vector is presented to the population of detectors and if it does not fall within a hypersphere of radius r_s of any detector, the data vector is deemed to be nonself. Otherwise, the new data vector is deemed to be self. A crucial point in the negative selection process is that the immune system does not require specific examples

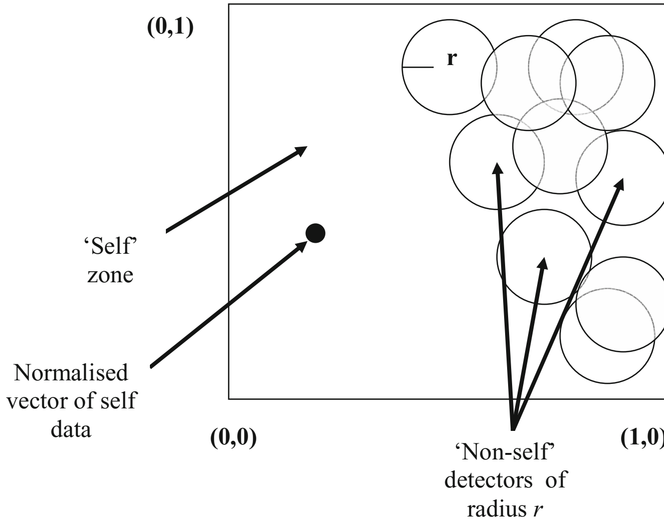


Fig. 16.5. Example of location of self/nonself zones

of nonself in creating its detectors. Potentially, the detectors can uncover any instance of nonself, even those never before encountered.

In using the negative selection algorithm, a choice must be made about the value of self-radius r_s and the number of detectors employed. The choice of value for r_s seeks to balance the detection rate and the false alarm rate of the system. If a small value of r_s is used the detection rate for nonself will be low and if a high value of r_s is set the false alarm rate will be high. Recent work by Gonzalez and Cannady [215] illustrates a hybrid AIS system which embeds an evolutionary algorithm for the purposes of automating parameter selection.

The selection of an appropriate distance or affinity measure is an important decision when designing an AIA. In real-valued applications of the negative selection algorithm, Euclidean distance is commonly used but there are many variants, including normalised Euclidean distance and Manhattan distance. Hamming distance can be used for applications requiring a binary representation. Another possibility when working with binary representations is to use *r-contiguous bits*, where two strings of length l are said to match if they have at least r contiguous bits in common. The value of r is set in order to balance the generalisability and the specificity of the detector. As $r \to l$, the detector becomes more specific, in that a smaller number of binary strings (and in the limit, only one type) will 'match' with it. A good discussion of different forms of representation and affinity measures is provided in [202, 300].

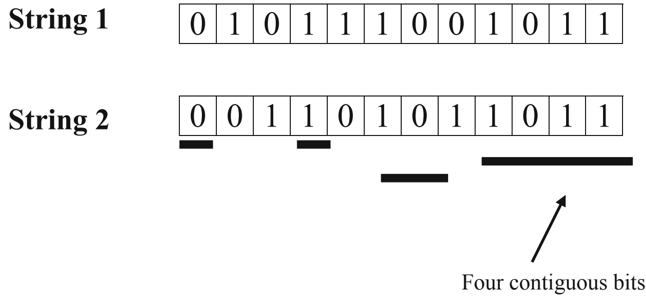


Fig. 16.6. The two binary strings have a maximum contiguous length of four. Hence, if r is set at 3, the strings are considered as matching. The strings have a Hamming distance of 4 as four bit-flips are required to convert one string into the other

Efficiency of the Algorithm

While the negative selection algorithm can produce an immune inspired classification system, it is known that the canonical algorithm scales poorly to high-dimensional problems [256, 596, 597, 620]. Intuitive reasons explaining the low efficiency of the algorithm include [102]:

i. It ignores the fact that examples of nonself may exist in some applications.
ii. The task of generating a population of valid detectors will grow rapidly as the size of self increases.
iii. In high-dimensional problems, a large number of detectors may be required in order to get adequate coverage of nonself space.
iv. Generated valid detectors may overlap, reducing the effective coverage of nonself space.

While not overcoming the above issues entirely, a number of attempts have been made to improve the canonical negative selection algorithm. For example, in real-world applications of AIS there may be historical examples of nonself available and these can be used to seed the detector creation process, thereby speeding it up.

Another approach is to use variable-size detectors [299]. In the canonical real-valued negative selection algorithm described above, the detectors have a fixed radius of detection whereas in the variable detector algorithm, the radius of each detector is permitted to vary. This allows areas of nonself which are far removed from any self vectors to be covered with a relatively small number of large radii detectors, and also allows for the insertion of smaller detectors to cover any gaps or holes in the nonself space between the large detectors (Fig. 16.7). Allowing the generation of irregularly-shaped detectors can also be useful in generating efficient coverage of nonself space.

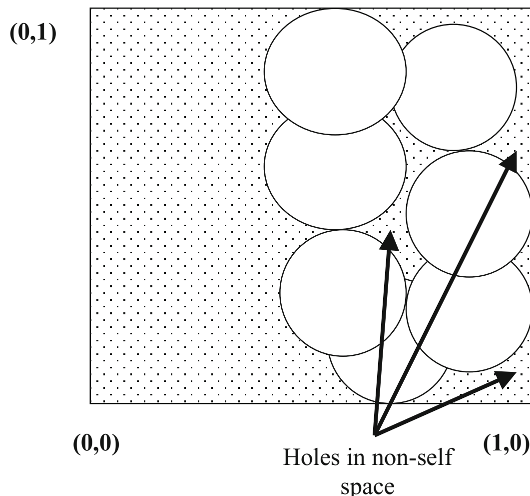


Fig. 16.7. The detectors do not cover the entire nonself region (here assumed to be the right-hand side of the square), leaving holes where nonself items could be incorrectly classed as self, in error

General Issues in Negative Selection

Negative selection algorithms are an example of 'one class' learning algorithms, where the classifier/anomaly detection system is constructed using only examples of one class. The concept of single class learning is not unique to negative selection and can be used with other classification algorithms, for example, one-class NNs [387], SVMs [558] and one-class GP [123]. One-class learning approaches have obvious application in cases where there there are few (or no) examples of one class available but there is a plentiful supply of examples of the other class (for example, detection of financial fraud [665]).

An obvious practical drawback (in addition to the efficiency concerns mentioned above) of the negative selection algorithm is that it is limited to a single class of nonself while many real-world classification problems are multiclass. Another potential drawback in applications of negative selection is that it is difficult to extract meaningful domain knowledge as to how classifications are being made or what the 'meaning' of specific detectors are.

Natural immune systems are capable of much more powerful classification behaviour than self/nonself. As already described, TLRs in the innate immune system are capable of distinguishing between multiple pathogenic signatures (Sect. 16.1.2) and triggering an appropriate immune defense. This provides ideas for the design of more powerful, multilayered, immune inspired classification algorithms. It should also be noted that natural immune systems typically look for a confirming (or second) signal before taking action against a

possible pathogen. A more comprehensive model of the immune system which explicitly considers this confirmation process has given rise to the dendritic cell algorithm, which is discussed next.

16.4 Dendritic Cell Algorithm

The potential to develop AIS algorithms using a danger theory metaphor was first identified by Aickelin et al. [7, 6]. The essence of danger theory is that the immune system does not respond naively to all nonself, but rather is triggered by danger, and in particular by chemical signals which occur when a cell dies an unnatural death (necrosis) due to cell stress or attack by a pathogen. A key role in the triggering of an immune system response is played by antigen-presenting cells (APCs) which can collect antigens, including potentially antigens from a harmful pathogen, in the vicinity of a damaged cell. These APCs can stimulate T cells, which in turn can stimulate the B cell clonal expansion process.

One of the key types of APC are dendritic cells (DCs), and the activities of these cells has inspired the *dendritic cell algorithm* (DCA) which can be applied for anomaly detection [223, 222]. Dendritic cells are able to collect antigen along with any danger signals arising in their local environment, and then integrate these signals to determine whether or not the environment is in fact dangerous.

DCs have three states, *immature*, *semimature* and *mature*. In their immature state, DCs collect antigen and monitor signals from local cells, danger signals from necrotic cells and safe signals from cells dying in a controlled fashion (apoptosis). Depending on the concentration of these signals, the DCs become either semimature (safe signals dominate) or mature (danger signals dominate), and migrate to the lymph nodes. At the lymph nodes they interact with helper T cells presenting antigen to them, and either tolerise the T cells to the antigen (if semimature) or sensitise the T cells to the antigen (if mature), thereby producing an immune system response. The antigen is presented in a *context*, being associated with either a safe or a dangerous environment. Hence, the DCs police tissue for signs of damage in the form of signals and also for potential culprits (antigens), which are found in the vicinity of the damage [221]. In considering the activities of DCs, it is more appropriate to consider them as a population rather than individually, as a significant immune system response only results when a multitude of DCs trigger T cells.

In the algorithm, four classes of signals are typically considered, and each is broadly inspired by mechanisms in the immune system.

i. *PAMPS*: in the natural immune system these are molecular signatures of bacteria and they can be recognised by TLRs (Sect. 16.1.2) on the surface of DCs. In the DCA, they are predefined patterns which indicate

the likely presence of an anomaly. The pattern is either present or not. In the DCA, the detection of PAMPS, in addition to generating a warning signal itself of the presence of an anomaly, also amplifies the expression of danger signals (see below).

ii. *Danger signals*: these indicate the likely presence of danger, drawing parallel with the presence of chemical signals of unplanned cell death in the natural immune system. In the DCA the signals are usually real-valued, and therefore can vary in strength depending on the local environment of the dendritic cell. The stronger the signal, the greater the likelihood of an anomaly being present.
iii. *Safe signals*: as for danger signals, these are usually real-valued, and therefore can vary in strength depending on the local environment of the dendritic cell. Safe signals result in a suppression of the immune system, and high levels of safe signals indicate the absence of an anomaly.
iv. *Inflammation*: In natural immune systems, inflammation is a response of the innate immune system to injury. In the DCA, the presence of an inflammatory signal (typically real-valued) amplifies the other categories of signal but is not itself an indicator of the presence or absence of an anomaly.

The DCs act as a population of multisensors [220] which can fuse the above signals in their local environment and produce differing output signals depending on the input signals they receive. Antigens in the DCA may be a string of numbers, bits or letters. As the algorithm runs, it learns to associate differing groups of antigens with specific signals. This allows it to label an antigen as normal or anomalous.

Operationalising the Algorithm

To operationalise the algorithm, problem-specific attributes must be mapped to the concepts of PAMPs, danger signals, safe signals, inflammatory signals, and antigens. There is no restriction that only one PAMP or danger/safe signal can be used and [221] provides an example which uses two PAMPS, two danger signals, two safe signals and one inflammatory signal. In order to ensure that all input signals have an equal chance to influence the output signals from the DC, they are preprocessed via normalisation into the range $[0, 1]$.

The output signals from a DC are of three types, *costimulatory* (CSM), *danger concentration*, and *safe concentration*. Each cell is assigned a random migration threshold on creation and the CSM values are incremented each time the cell receives an input signal. Once the cumulative CSM value of a cell exceeds its migration threshold the DC ceases to be immature and becomes either semimature or mature. As the migration threshold varies by cell, this implies that each cell samples inputs for differing periods of time; in other words, the lengths of their lifetimes vary.

A signal processing function C (16.1) must be defined for any specific application. A simple approach is to use a weighted average of PAMP (P), safe signal (S) and danger signal (D) concentration values. This average is multiplied by a factor depending on an inflammation value $I \in [0, 2]$ [222]. Weights are applied to each signal, denoted by W_P, W_S, W_D respectively.

$$C_{[\text{CSM, danger, safe}]} = \frac{W_P C_P + W_S C_S + W_D C_D}{W_P + W_S + W_D} \cdot \frac{1 + I}{2} \tag{16.1}$$

In the original implementations of the DCA for intrusion detection on computers, the signals were determined from a continuous time series of computer activity, with features such as number of data packets per second being used as a danger signal. Antigens in the algorithm are a stream of symbolic items (for example, process ID numbers in computer security cases) occurring alongside the time series of signals. In this representation, each data item (analogue to a tissue in the natural immune system) used for the learning process contains signal information from a time window, or at an instant, and an associated antigen type.

When a tissue 'sample' is presented to an individual DC, the relevant inputs are supplied, and each of the three outputs of the DC is calculated, and these outputs are cumulated in each signal category over the lifetime of the cell in its immature state. The associated antigen is then added to the antigen store of the DC. Over multiple time slots (multiple sample tissue presentations) the DC will collect multiple examples of antigens.

Input signals can interact with, for example, safe signals in a tissue acting to dampen the effect of danger signals in the same tissue. A user-defined weight matrix is required for any specific implementation, and as an exemplar, the weights in Table 16.2 were used for the signal processing function in [222]. We can see that, in this example, a safe signal suppresses the impact of a danger signal as this interaction has a negative coefficient.

W	**CSM**	**Safe**	**Danger**
PAMPs (P)	2	0	2
Danger signals (D)	1	0	1
Safe signals (S)	2	3	-3

Table 16.2. Sample weights for signal processing function

If the CSM threshold limit is reached for an individual DC, it is removed from the process of tissue sampling and becomes (semi)mature. Each DC therefore, only samples a subset of the total set of tissues and associated antigen available in the environment over time. When the DC transitions from the immature state, its cumulative output signals are assessed to form the context of all the antigen that it has collected. A higher total 'danger' output signal (cumulative danger signal > cumulative safe signal) results in

the assignment of context '1' to the DC, whereas a higher 'safe' output signal results in the assignment of '0' to the cell's context. All of the antigen sampled by the DC during its immature phase are assigned the same context.

A critical issue here is that the structure of the antigen itself does not determine the context; rather this is determined based on signal information from the tissue in which the antigen is found. As identical antigens may be found in different tissue samples, it is possible that different DCs will encounter varying local signals in these tissue samples and therefore assign a different context to the same antigen. Hence, the information in DCs is assessed at a populational level and when the tissue presentation stage of the algorithm terminates, a mean context value, in other words an *anomaly coefficient value*, for each antigen type is calculated using

$$\mathrm{MCAV}_x = \frac{Z_x}{Y_x} \tag{16.2}$$

where MCAV_x is the *mature context antigen value* (MCAV) coefficient for antigen type x, Z_x is the number of mature context antigen presentations for antigen type x and Y_x is the total number of antigen presented for antigen type x. The closer this value is to 1, the more likely it is that the antigen is associated with the danger signals as values of close to 1 indicate that the antigen is typically found in tissue which is exhibiting danger signals.

The resultant algorithm is population-based, with each cell in the population assigned a remaining life span value, which reduces as it is presented with more tissue samples. Different cells process signals acquired over different time periods, generating individual 'snapshots' of input information. When aggregated across the population, antigens are classified on the basis of the consensus opinion of whether a particular type of antigen is normal or anomalous. Pseudocode for the DCA is provided in Algorithm 16.2.

Summary

The DCA utilises a more nuanced view of immune system response than the traditional self/nonself framework, and is inspired by the process of T cell stimulation by antigen-presenting cells. The DCA has been applied in a number of areas, including computer intrusion detection [220, 221], and has produced good results. Although the DCA is a simplification of the underlying natural immune process, the algorithm is quite complex and requires definition of the relevant input signals, the signal processing function and the weight matrix in each problem instance. More recently, work by [595] indicates that the classification by a single agent in the DCA is equivalent to a statically weighted linear classifier, and this has cast doubt on the ability of the canonical DCA to fully capture classification problems which have complex, nonlinear decision boundaries in the signal space. Research on the theoretical analysis of the DCA, and on designing more powerful variants of the DCA, is ongoing [235, 234].

Algorithm 16.2: Dendritic Cell Algorithm

```
Initialise population of DCs and parameters for algorithm;
Set time step to zero;
repeat
    Sample data from time series (generate tissue sample);
    for each dendritic cell DC_i in turn, i = 1, ..., n do
        repeat
            Get antigen;
            Store antigen;
            Get signals;
            Calculate interim output signals;
            Update cumulative output signals;
        until CSM output signal from DC_i > migration threshold for DC_i;
        Migrate DC_i to lymph node;
        if safe output signal > danger output signal then
            Cell context is set to 0;
        else
            Cell context is set to 1;
        end
        Store cell information (list of its antigens and associated context) in
        S;
        Kill cell;
        Reset and replace DC_i in the population of DCs;
    end
    Increment time step to sample additional time series of signals and
    antigen;
until until time series of training data is exhausted;
for all antigens in store S do
    Increment antigen count for this antigen type;
    if antigen context = 1 then
        Increment antigen type mature count;
    end
end
for all antigen types do
    Set MCAV of antigen type := mature count/antigen count;
end
Output list of antigens and their MCAVs;
```

16.5 Clonal Expansion and Selection Inspired Algorithms

The primary idea underlying clonal selection theory is that when an antigen is detected, the antibodies that best recognise (match) it will proliferate via a cloning process, thereby greatly increasing the quantity of antibodies which can recognise the relevant antigen. In the cloning process, the newly generated antibodies are mutated in an attempt to better tune them to detect the antigen. The processes of clonal expansion and affinity maturation of B cells can be used to inspire the design of a family of algorithms which can be applied for optimisation and classification. Initially we outline three of the better-known algorithms from this family, which are primarily used for optimisation purposes, namely

i. the CLONALG algorithm,
ii. the B cell algorithm, and
iii. the real-valued clonal selection algorithm.

Following this we describe the artificial immune recognition system (AIRS) which is primarily used for classification purposes.

16.5.1 CLONALG Algorithm

In CLONALG there are two populations, a population of antigens (Ag) which corresponds to the environment, and a population P of antibodies (Ab) which corresponds to the population of current solutions. Each solution p_i is represented as a vector of attributes $p_i = (p_{i1}, p_{i2}, \ldots, p_{im})$ which correspond to a 'point' in the m-dimensional attribute space. After a random initialisation of the starting population of antibodies, an affinity value (a measure of how well an individual antibody 'fits' the antigen population) is calculated for each member of the antibody population.

In some cases, the creation of a function to measure 'fit' will be straightforward, in others it may require a good deal of thought. Taking a simple example of model calibration, the antibody could correspond to a vector of real numbers which parameterise or 'calibrate' a prespecified mathematical model, and the antigen environment could correspond to a set of test data, with a single antigen being a data vector, $x_j = (x_{j1}, x_{j2}, \ldots, x_{jn})$. In this application, the quality of an antibody, which can be 'decoded' to produce a specific instance of a mathematical model, can be assessed using a metric such as the mean squared error of the model fit to the entire test dataset (note: a low MSE value indicates a good fit and therefore a high level of affinity).

Next, n members from the population of antibodies are selected, preferentially members which have higher affinity (fitness), and clones are generated from these n items. The number of clones generated for each selected antibody is proportional to its fitness; thereby the fitter items generate a greater number of clones of themselves. This results in a new population P^{clone}. Then a

hypermutation operator is applied to each clone, in order to simulate an affinity maturation process, resulting in an altered population of clones P^{hyper}. The degree of mutation applied to each clone is inversely proportional to the clone's fitness; thereby the poorer quality clones are subject to higher levels of mutation and better clones are mutated less. The members of P^{hyper} then compete with P to determine the composition of the new population of antibodies at time step $t+1$. In order to maintain diversity in the population of antibodies, a small number d of new antibodies are randomly generated, replacing the poorest d items in the new population P. The algorithm then iterates until a terminating condition is triggered. Algorithm 16.3 presents an outline of the CLONALG algorithm [140, 142].

Algorithm 16.3: CLONALG Algorithm

```
Create an initial random population P of solution vectors (antibodies);
repeat
    Select a set F of parents, F ⊆ P, biasing the selection process towards
    better solutions;
    for each member of F do
        Create a population P^clone of clones from F, with better members of
        F producing more clones (clonal expansion step);
        Mutate each of these clones, in inverse proportion to their parent's
        fitness (the hypermutation step), giving population P^hyper;
        Select S, a subset of the better newly generated solutions P^hyper;
        Create R, a set of new random solutions;
        Replace poorer members of P with better solutions from S and R;
    end
until terminating condition;
```

Although CLONALG derives from an immune system metaphor, it embeds population-based search guided by selection and diversity generation. Hence, it bears some similarity to evolutionary algorithms. The most obvious difference between the algorithmic families is the method they use for generating variety when seeking to iteratively improve solutions. In addition to affinity (fitness) proportionate selection, CLONALG also uses (inverse) affinity proportionate mutation, affinity proportionate cloning, and does not use crossover.

Just as is the case for most heuristics described in this book, the general CLONALG framework can be implemented using different representations of antibody/antigen format, and therefore can be applied for a wide variety of real-world problems. Examples of the use of binary and integer-valued antibody representations in CLONALG are provided in [142].

16.5.2 B Cell Algorithm

Although CLONALG can be applied for function optimisation, a variant called the *B Cell algorithm* (BCA) was designed by Kelsey and Timmis [325] which is claimed to be particularly computationally efficient for function optimisation. Like CLONALG, the BCA is inspired by the clonal selection process but implements the mutation step in a notably different way, using a mutation operator called *contiguous somatic hypermutation*. This operator is inspired by a claim that mutation events tend to be concentrated in clusters or regions of genetic material rather than occurring at completely random locations [325].

In function optimisation there is no explicit antigen population which requires recognition; rather the aim is to uncover a vector of real numbers which optimises (maximises or minimises) the value of a function of interest. The BCA uses a binary (bit string) representation, with a double-precision (real) number being encoded using a 64-bit string. The total length of the binary string (antibody) will depend on the dimensionality of the search space. The affinity of a binary string is found by decoding it into a real-valued vector (a 'point' in the input space) and then determining the value of the objective function at this point in the input space.

In the algorithm, an initial population of P binary strings is randomly generated and their affinity is then determined using the objective function. Each B cell is then cloned to produce its own *clonal pool* C, with the size of this clonal pool being typically similar to P (i.e. each B cell gives rise to about P clones of itself). One clone is then selected randomly from this clonal pool and each element of its binary vector is subject to mutation (bit flipped) with a modeller-determined probability. The mutated clone then replaces its original version in the clonal pool.

Next, all other B cells in the clonal pool are subject to the contiguous somatic hypermutation operator. Under this operator, a random position (or *hotspot*) on the bitstring corresponding to that B cell clone is selected, along with a random length. The elements of the bitstring from the hotspot to the end of the selected length of contiguous region are then mutated (bit flipped) with a modeller-determined probability ($0 \leq r \leq 1$). In the case where the random length selected is longer than the number of elements remaining from the hotspot to the end of the bitstring, the mutation operation stops at the end of the string and does not wrap around to the beginning. Having completed the hypermutation step for all clones in C, if the best clone in the clonal pool has higher affinity than its parent B cell, it replaces the parent. The process iterates until a predetermined stopping condition is reached. Algorithm 16.4 outlines the pseudocode for the B cell algorithm.

In comparison with CLONALG, the processes of selection for clonal expansion, mutation, and selection for replacement, are quite different. In CLONALG, only the better antibodies in P are selected for cloning, whereas in the BCA, all members of P are selected. The mutation process in the two algorithms is distinct, with mutation being applied stochastically along the

Algorithm 16.4: B Cell Algorithm

```
Create an initial random population P (the parent set, typically three to five
members) of solution vectors (antibodies or B cells);

repeat
  for each member of P do
    Calculate its affinity (fitness);
    Create a clonal pool C (clonal expansion step);
    Select a clone c ∈ C randomly;
    Randomise the vector of c;
    Replace c in C by the newly randomised clone;
    for all members of C other than c do
      Apply the contiguous somatic hypermutation operator;
    end
    Calculate the affinity of each clone in C;
    if the best clone in C has higher affinity than the parent B cell then
      Replace the parent B cell with the best clone in C;
    end
  end
until terminating condition;
```

antibody bitstring in CLONALG, in contrast to the contiguous mutation operation in the BCA. Finally, a variant of elitism applies in the replacement process in BCA (as a parent is only replaced by a clone if the clone is better), whereas in CLONALG, the selection of survivors into the next interation of the algorithm is from the current population, a set of newly generated clones, and a set of randomly generated new antibodies.

In essence, the hypermutation operator in BCA applies a search process of varying locality around a 'parent' B cell depending on the selected length of the contiguous region. This capability to take steps of varying size on the search landscape enables the BCA to obtain good results with relatively small population sizes, with a typical size in the range $[3, 5]$ being suggested by [325]. However, in dynamic, or multimodal problems, a study by [634] shows that larger population sizes, and a relatively smaller number of clones in each clonal pool, are required.

16.5.3 Real-Valued Clonal Selection Algorithm

As seen above, the antibodies in the BCA use a binary string representation. An alternative approach for real-valued optimisation would be to use antibodies with a real-valued representation, and a paper by [93] introduces a variant of CLONALG, the *real-valued clonal selection algorithm* (RCSA), which does this. As for the general CLONALG algorithm, the best n antibodies are selected, and ranked in order of affinity (highest to lowest), for clonal expansion.

Each of these antibodies is allocated a number of clones in proportion to its ranking using [93]

$$N_{\text{clone}}^{i} = \text{round}\left(\frac{\beta \cdot n_{\text{pop}}}{i}\right) \tag{16.3}$$

where N_{clone}^{i} is the number of clones allocated to the antibody ranked in position i (where $1 \leq i \leq n$), n_{pop} is the size of the total population of antibodies, β is the multiplication factor for the cloning process, and the function round() ensures that the output number is an integer. The generated clones then undergo a hypermutation process and this is implemented in [93] by applying a Gaussian mutation to at least one element of the clone's vector (element(s) being selected stochastically), using

$$x_j^* = x_j(1 + p \cdot S(x_j) \cdot r) \tag{16.4}$$

where p is a small number, r is a Gaussian random variable drawn from $N(0,1)$ with mean of 0 and standard deviation of 1, and the parameter $S(x_j)$ is a scaling parameter, depending on the range of the input space on dimension j.

In the real-valued clonal selection algorithm, the population replacement step is modified from that of CLONALG, in that each selected antibody and its clones form a subpopulation, and only the best member of this subpopulation passes into the next generation, implementing an elitist selection process. Hence, n individuals survive from the clonal expansion and hypermutation steps. The remaining individuals required to fill out the population in the next generation, a total of $n_{\text{pop}} - n$ individuals, are generated randomly, thereby injecting diversity into the search process. The process iterates until a termination condition is met.

Parallels with Evolutionary Algorithms

There is an ongoing debate concerning the degree of similarity between the families of evolutionary and clonal selection algorithms. The distinctions typically drawn between clonal selection algorithms and evolutionary algorithms are summarised as follows in [79]:

i. each draws on a different source of inspiration and they have differing abstractions and terminology, and
ii. clonal selection algorithms and evolutionary algorithms employ differing diversity-generating and selection mechanisms.

It is apparent that clonal selection inspired algorithms also have similarities with the general evolutionary computation framework. All employ a population, and operate via a diversity-generation and fitness-preferential selection process. However, they do not have a crossover operator and the solutions therefore do not explicitly share information with each other.

16.5.4 Artificial Immune Recognition System

The clonal selection process has also inspired the design of classification algorithms. One of the best known of these is the *artificial immune recognition system* (AIRS) algorithm [651, 652] which uses a supervised learning paradigm. This approach takes inspiration from the adaptive immune system wherein after the immune system has been exposed to a pathogen, some B cells differentiate into memory cells, allowing swift recognition of that pathogen if it is subsequently encountered again. In essence, each of the memory cells is a 'detector' for an antigen which is associated with the relevant pathogen. At any point in time, a wide array of different B memory cells will be in circulation in the body, and as a population they are capable of multitarget detection/classification.

In AIRS the object is to create a population of memory cells using supervised learning, which can then be used to assign the correct class labels when presented with new data vectors. This is inspired by the capability of B memory cells to react to previously seen pathogens.

Typically, for a real-valued feature space, each memory cell will be a real-valued vector corresponding to a point in the feature space and will have an associated class label. The training process in the AIRS algorithm determines the location and associated class labels of the memory cells. Unlike the one-class learning of the T cell tolerogenesis inspired negative selection algorithm (Sect. 16.3), AIRS can be applied to multiclass classification problems.

AIRS Algorithm

The operationalisation of the AIRS algorithm is complex, but a high-level overview of the process is as follows [651]:

i. present the training data (antigens) one at a time to the system,
ii. generate a candidate memory cell, and implement a cloning, mutation and affinity maturation process to refine memory cell candidates,
iii. determine whether the candidate memory cell is added into the final memory cell pool,
iv. repeat above steps until all training instances are presented,
v. output is a population of memory cells, which can then be used, via a k nearest neighbour approach, to produce out-of-sample classifications.

A number of versions of the AIRS algorithm have been developed. One of the more common variants is AIRS2 [653, 650], which slightly simplified the original AIRS algorithm. Below we outline the relevant pseudocode for AIRS2 (Algorithm 16.5) and then describe this version of the algorithm, as outlined by [80], in some more detail.

In the pseudocode, each training data vector (antigen) is denoted as a_i, C is the memory cell pool, and ARB is an *Artificial Recognition Ball*. In AIRS, an ARB has exactly the same representation as a B cell, a real-valued feature

vector, along with an associated class label. The term is used to denote a representative of a set of clones or a group of similar B cells.

After training is complete, the created memory cell pool can be used for classification. The data vector to be classified is presented to each memory cell in the pool in turn, and the class label assigned to the data vector is that of the majority label of the k most simulated memory cells in the pool.

Algorithm 16.5: AIRS2 Algorithm

```
Normalise the training dataset;
Seed the memory cell pool C;
Set algorithm parameters;

for each training instance a_i do
    Select the memory cell c_match in C which has the highest stimulation
    when exposed to a_i;
    Clone c_match in proportion to its stimulation value for a_i;
    Mutate each clone and add all mutated clones to newly created ARB
    pool along with c_match;
    repeat
        Calculate stimulation value for each member of ARB pool when
        exposed to a_i;
        Allocate limited resources to members of ARB pool;
        Rank ARBs by resource level and remove ARBs with zero assigned
        resources;
        Clone and mutate a random selection of ARBs;
    until mean normalised stimulation values of ARBs >= affinity_threshold;
    Select the ARB with the highest stimulation and call it c_cand;
    if c_cand has higher stimulation than c_match then
        Add c_cand to C;
        if c_match and c_cand are sufficiently similar then
            Remove c_match from C;
        end
    end
end
Output final memory cell pool C which can be used for out-of-sample
classification;
```

Normalisation and Initialisation

To begin, all feature vectors are normalised so that all distances between antigens (training data vectors) and/or any memory cells are in the range $[0, 1]$. This can be achieved by normalising each vector component into the range $[0, 1]$. A Euclidean distance measure is usually adopted (assuming that

the feature space is real-valued). In order to ensure that the maximum distance between any two antigens or memory cells is in the range $[0, 1]$, the following step is added to the data normalisation process:

$$\text{Value}_{\text{normalised}} := \text{Value}_{\text{normalised}} \cdot \sqrt{1/n}. \tag{16.5}$$

The affinity measure can then be calculated as the normalised Euclidean distance between any two antigens or ARBs; so the smaller the distance between two items, the greater their affinity. Note, this means that in the AIRS algorithm, small affinity values indicate strong affinity, and values for affinity are also bounded in the range $[0, 1]$. The choice of normalisation process is not restricted to the above and any procedure which ensures that all distances are bounded within $[0, 1]$ can be used.

The next step is to seed the memory cell pool which will eventually contain the collection of recognition detectors that make up the final classifier. Typically, this is done by randomly selecting a subset of the antigens and adding them to the initial memory cell pool.

An affinity threshold $\text{affinity}_{\text{threshold}}$ is then calculated which is used later in the algorithm in order to determine whether a new memory cell is sufficiently close to an existing cell to replace it. The affinity threshold is calculated as the average affinity, calculated pairwise, over all training instances,

$$\text{affinity}_{\text{threshold}} = \frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{affinity}(a_i, a_j)}{n(n-1)/2} \tag{16.6}$$

where a_i and a_j are a pair of antigens, and affinity is measured as Euclidean distance in the normalised feature space.

Antigen Training

The training process involves a single pass through the training data. Each antigen in turn is individually exposed to the memory cell pool and a stimulation value is calculated between the antigen and each cell in the memory pool, with low affinity values (corresponding to high affinity) indicating high stimulation,

$$\text{stim} = 1 - \text{affinity}. \tag{16.7}$$

The memory cell of the same class as a_i with the highest stimulation is identified as the *best match* (c_{match}) for use in the affinity maturation process. If the set of memory cells with the same class as a_i is empty, then a_i is added to the set of memory cells, and is itself the c_{match}.

A number of mutated clones are created from c_{match} and, along with c_{match}, are added to a newly created ARB pool, with the number of mutated clones being calculated as

$$N_{\text{clone}} = \text{stim} \cdot \text{rate}_{\text{clonal}} \cdot \text{rate}_{\text{hypermutation}}. \tag{16.8}$$

The ARB pool therefore is a grouping of cells which are derived from the original (c_{match}) and is created anew as each antigen is presented during the training process. The terms $\text{rate}_{\text{clonal}}$ and $\text{rate}_{\text{hypermutation}}$ are user-defined parameters. The $\text{rate}_{\text{clonal}}$ is the number of clones that are created from a B cell (here, the best matching memory cell) and $\text{rate}_{\text{hypermutation}}$ determines the number of mutated clones that are derived from these (a scaling factor). These variant clones seed the ARB pool and are then refined in the next step of the algorithm. The object of this process is to develop a candidate memory cell which is most effective in correctly classifying the antigen.

Competition for Limited Resources

After the mutated clones of the best matching memory cells are added to the ARB pool a competition for resources, akin to survival of the fittest, is implemented in order to prune the size of the pool while maintaining the ARBs with greater stimulation to the training antigen. After the competition step, the surviving ARBs in the pool generate additional mutated clones using a similar process to that outlined for cloning c_{match}. The number of clones generated for each ARB in the pool is calculated using

$$N_{\text{clone}} = \text{stim} \cdot \text{rate}_{\text{clonal}}. \tag{16.9}$$

When these have been generated, the competition for *resources* begins and the aim of this is to reduce the number of ARBs that coexist in the ARB pool. First, the level of resource allocation to each ARB is calculated as

$$l_{\text{resource}} = \text{stim}_{\text{norm}} \cdot \text{rate}_{\text{clonal}} \tag{16.10}$$

where $\text{stim}_{\text{norm}}$ is the normalised stimulation value for that ARB and $\text{rate}_{\text{clonal}}$ is as already defined above. The normalised stimulation value for an ARB can be calculated using

$$\text{stim}_{\text{norm}}(\text{ARB}_i) = \frac{\text{stim}_{\text{ARB}_i} - \text{stim}_{\text{min}}}{\text{stim}_{\text{max}} - \text{stim}_{\text{min}}} \tag{16.11}$$

where $\text{stim}_{\text{ARB}_i}$ is the stimulation level when exposed to a_i for the ARB of interest, and stim_{max} and stim_{min} are the maximum and minimum stimulation levels across all ARBs in the pool.

Low levels of resources indicate that the ARB does not have a high stimulation when presented with the antigen. The members of the ARB pool are sorted by allocated resources in descending order. A maximum $\text{resource}_{\text{total}}$, a user-defined parameter, is available to the population of the pool, and resources are removed from ARBs at the end of the list until the total number of allocated resources across the pool is below the maximum limit. All ARBs with zero allocated resources after this step are removed from the pool. The surviving ARBs are then subject to a cloning and mutation process in order

to generate populational diversity and to refine their affinity to the antigen. The mutation process is designed so that more stimulated ARBs are mutated less.

The ARB refinement process continues until the mean normalised stimulation value across all ARBs is greater than a user-defined parameter ($\text{stim}_{\text{threshold}}$). This parameter is in the range $[0, 1]$, and as higher values are chosen, the ARBs are further refined, and become closer to the training antigen. Hence, the competition and ARB refinement process is as in Algorithm 16.6.

Algorithm 16.6: Competition and ARB Refinement

```
Starting from initial ARB pool;
repeat
    Stimulate each member of the ARB pool with the training antigen;
    Normalise ARB stimulation values;
    Allocate limited resources based on stimulation level of each ARB;
    Rank ARBs in descending order based on resource allocations;
    Prune ARBs with zero resources;
    Generate mutated clones of surviving ARBs
until mean normalised stimulation values of ARBs >= stim_threshold;
```

Memory Cell Selection

Once the ARB refinement process is completed, the ARB with the greatest normalised stimulation score is selected to become a memory cell candidate (c_{cand}). If the new candidate for the memory cell pool is a better fit (has higher stimulation) for the presenting antigen than the best existing memory cell (c_{match}) it is added to the pool. If the affinity between c_{cand} and c_{match}, is less than

$$\text{affinity}_{\text{threshold}} \cdot \text{param}_{\text{scale}} \tag{16.12}$$

where $\text{affinity}_{\text{threshold}}$ is calculated as above and $\text{param}_{\text{scale}}$ is a user-defined scale parameter, then c_{cand} replaces c_{match} in the memory pool and c_{match} is discarded. Otherwise c_{match} remains in the memory pool. This avoids the generation of a new memory cell which is very close to an existing memory cell. At this point, the training for this single antigen is complete, and system training recommences with the next antigen.

After all antigens have been presented and training is complete, the final pool of memory cells is the AIRS classifier and can be applied to determine the class of previously unseen data.

Summary

AIRS is an interesting algorithm in that it is one of the few examples of the use of an AIS for classification which uses a supervised learning approach. The algorithm has produced results which are competitive relative to other established classifiers [652, 650]. AIRS also has the feature that it does not require the user to preselect a model architecture as the architecture is evolved via mechanisms of diversity-generation and resource-based selection during the training process [80]. A drawback of the AIRS algorithm is that it has quite a complex implementation and also has multiple user-defined parameters.

16.6 Immune Programming

Niels Jerne [298] noted that there were similarities between the generative capabilities of the immune system and the generative capacity of linguistic grammars (an interesting aside on this point is provided by the title of his lecture on winning the 1984 Nobel Prize in Physiology or Medicine, namely 'The Generative Grammar of the Immune System'). In language, an infinite number of syntactically correct sentences can be generated and understood even where a person has never heard that precise sentence before. In the case of an immune system, the 'rules' governing the physical components and their workings can be considered as a grammar. This grammar, like the structures of language, is not fixed but can adapt over time.

Some work has been undertaken to develop hybrid AIS which combine concepts from immune systems, genetic programming and related grammatical computing methodologies (Chap. 17), such as grammatical evolution, giving rise to *immune programming* algorithms. Starting with the immune-GP (iGP) paper of Nikolaev, Iba and Slavov [450] there have been several explorative studies investigating these hybrids.

For example, Johnson [303] illustrated how a clonal-based optimisation algorithm could be combined with GP in order to generate programs to solve symbolic regression problems. In this study, a population of programs were generated from a function and terminal set in the usual manner using a ramped half-and-half procedure, and the quality of the initial population was assessed using a set of test data. Metaphorically, this step corresponds to a measurement of the affinity between the generated programs (the antibodies) and the test data (the antigens). The best programs were then selected for hypermutation, and multiple mutated versions of the programs were generated. The mutation step was implemented by selecting a random node in the GP program tree, and replacing the subtree below that node with a new randomly generated subtree. The selection and mutation steps could be performed in a large variety of ways. In Johnson's implementation, the top 20% of the population was selected for hypermutation and four mutated children were created

for each individual selected. In the next iteration of the algorithm, the population consisted of all the selected programs plus their mutated children. Unlike the case of traditional GP, a crossover step was not used.

While the study did produce interesting results, it did not attempt to explicitly embed immune concepts such as gene libraries or investigate alternative ways of implementing the selection and mutation process. The potential for further research was noted by the author. More recent, related work in [430] applies the clonal selection and expansion metaphor to generate stack-based assembly instruction computer programs.

16.7 Summary

The mechanisms of natural immune systems provide a rich metaphorical inspiration for the design of pattern-recognition and optimisation algorithms. In this chapter we have discussed how three of these metaphors, the negative selection process for T cells, the clonal selection and expansion of B cells, and danger theory, can be applied. While a large body of literature has already developed on these families of algorithms there is scope to extend the boundaries of AIS further.

Despite the powerful, multilevel learning capacity in natural immune systems, most AIS algorithms have drawn quite simplistic metaphorical inspiration from individual elements of the adaptive immune system. Natural immune systems display learning at all three levels of the POE (phylogeny, ontogeny, epigenesis) framework (Sect. 1.1.1). For example, the basic framework of the immune system (its major structures, etc.) is genetically determined and hence is subject to an evolutionary process (phylogenetic learning). Components of the system display the ability to differentiate during their maturation, or development, a form of ontogenetic learning. Finally, the adaptive immune system displays a capacity to respond in real time to changes in its environment (epigenetic learning), for example an invading pathogen. A key differentiating factor between the immune system metaphor and other biological metaphors such as evolution is that the immune system naturally operates over multiple timescales.

Future extensions of AIS will likely focus increased attention on the multilayered nature of the immune system and on the synergistic links between the innate immune system and the adaptive immune system [256].

Another challenging area for future development is the design of AIS specifically for application in dynamic environments. Given the highly dynamic environment faced by natural immune systems, a premium is placed on their being able to adapt rapidly. This suggests that well-designed AIS may be particularly relevant in solving dynamic rather than static problems.

However, it must also be noted most existing AIS algorithms are quite complex in structure, and as a result are not easy to analyse theoretically.

Constructing algorithms which draw on a more complete picture of the natural immune system will compound this issue further.

As we have seen in this chapter, immune systems starting from the innate immune system of a new-born infant are capable of considerable adaptation and development over time, with this development process being governed by a 'set of rules' (or a grammar) as to how parts of the immune system interact. In Chap. 17, we introduce general concepts of development processes and grammars, leading to a discussion of developmental and grammatical computing.

Part V

Developmental and Grammatical Computing

17

An Introduction to Developmental and Grammatical Computing

To say that the knowledge uncovered by developmental biologists has been under-exploited in natural computing is perhaps an understatement. Curiously, despite the relative lack of research attention that has been paid to these important biological processes, one of the fathers of Computer Science, Alan Turing, recognised the power of developmental systems and developed reaction-diffusion models to understand the mechanisms behind morphogenesis (the development of biological form) [638]. In recent years it is heartening to see researchers beginning to close this gap and start to explore the power of developmental processes such as genetic regulatory networks for problem solving, and the use of approaches such as self-modification of phenotypes and developmental evaluation. The surge in interest in developmental computing is illustrated by the creation of a new track dedicated to Generative and Developmental Systems which began in 2007 at the ACM Genetic and Evolutionary Computation Conference [72, 321, 347, 526, 586, 623] and which has run every year since. A special issue of the journal IEEE Transactions on Evolutionary Computation was also dedicated to this topic in 2011 [639]. In this chapter the concept of developmental computing is introduced with particular emphasis on grammatical computing, which uses grammars as a generative process in order to create structures of interest.

17.1 Developmental Computing

Developmental Computing refers to computational methods that have been inspired by developmental processes which occur in biology, that is the development of an embryo to a multicellular adult organism.

The simplest, and most common manner in which developmental systems have been adopted in natural computing is through the use of genotype–phenotype maps. Many of these approaches are developmentally so simple that perhaps it is an abuse of the term to call them developmental systems. However, the genotype–phenotype mapping is at the heart of developmental

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_17

biology. In one sense, we could then consider the simple genetic algorithm as being a form of developmental computing, as the genotype encodes the parameters of the problem under investigation and these are decoded into a 'solution' via the phenotype. Of course, this simple mapping omits virtually all of the critical aspects of real-world development processes. It has been observed that much of the diversity observed in the natural world can be traced to three features of developmental biology, namely, interactions between gene products, the temporal nature of gene expression, and shifts in the location of gene expression [32]. The first item highlights the significance of feedback loops to developmental processes. Related to this, a key feature of developmental systems in nature, and of all organisms in general, is that they are embedded in an environment. Interactions occur between a developing organism and its environment. Environment here can mean the state of the cell in which the genetic material is located, or the neighbouring cells and intracellular signaling events, and also the outward environment or habitat in which the organism is physically located. Feedback loops exist at all these levels of interaction with each playing a significant role in the outcome of the developmental process [210].

17.2 Grammatical Computing

Developmental and Grammatical Computing tend to go hand-in-hand, that is, a significant number of algorithms inspired by developmental systems have tended to adopt a grammatical representation. This arises because grammars are a natural representation for a generative model. They represent a set of production rules and the contexts in which those production rules (or transformations) are allowed to occur. *Grammatical Computing* refers to algorithms which have been inspired by the underlying formal representation of **a grammar**. As well as the obvious languages of communication as studied in linguistics, humans have also designed programming languages as problem-solving tools. Consequently, the formal concept of a grammar is widely adopted in the fields of linguistics and computer science [107].

In contrast to optimising algorithms such as the GA (Chap. 3) or PSO (Chap. 8) which generally attempt to optimise parameters within a specific predefined model structure, the generative approach of Grammatical Computing allows the automatic generation of both the model structure in addition to the appropriate parameters for that model. In other words, these algorithms are generally capable of model/structure induction.

Grammatical computing covers a spectrum of 'generative methods' including Lindenmayer Systems and the grammar-based approaches to genetic programming (GP) (Chap. 7) such as grammar-guided GP (G3P), grammatical evolution (GE), and tree-adjoining G3P (Chaps. 19 and 20).

Grammars have also been studied from a linguistic standpoint to understand the nature of spoken languages, how these languages and underlying

grammars originated, and how they subsequently evolved over time and use. In Evolutionary Computation grammars have also been used in the reverse sense, that is to evolve or find a grammar to fit a particular set of sentences or structures. This evolutionary approach to grammar learning will not be examined here. Instead we will focus on algorithms which exploit and incorporate grammars to solve problems, and in particular on grammar-based approaches to GP. This chapter will provide a short general introduction to the underlying concepts of Grammatical Computing largely focusing on the notion of a formal grammar.

17.3 What Is a Grammar?

A linguistic grammar can be loosely considered as a set of words and symbols of a language, together with a set of rules that specify the legal arrangements of these words and symbols in sentences of the language. In contrast to the standard or traditional use of grammars within Computer Science during compilation, as tools to parse sentences in programming languages to determine if they are syntactically correct, Grammatical Computing (GC) generally adopts a generative approach to their use. That is, a grammar provides a set of rules that is used to govern the generation of some structure. The use of the term **structure** is deliberately vague here, as grammars can represent all manner of different things, and it is this feature of grammars that makes them a particularly powerful formalism as a problem-solving device.

More generally then, a grammar can be considered as a set of components and the permissible ways that those components can be assembled in order to create a structure. These structures can range in complexity from a simple sequence of letters, to such things as a linear equation which fits a set of input–output data, a graph rewriting system, or an entire computer program.

Formally, a grammar can be defined as $G = (\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$ where $\mathcal{T}$ is an alphabet (or a set of *terminal* symbols), $\mathcal{N}$ is a set of intermediate or *nonterminal* symbols, $\mathcal{S}$ is a *root* or *start* symbol, and $\mathcal{P}$ is a set of production rules, which govern how elements of the alphabet and the set of nonterminals can be converted into structures which are comprised solely of elements from the alphabet. The set of all structures (the *language)* which the grammar can give rise to is denoted by $\mathcal{L}(G)$.

Hence, a formal grammar defines a language, which is a (possibly infinite) set of sequences of symbols that may be constructed by applying production rules to a sequence of symbols which initially contains just the start symbol. In generating a structure in the language, production rules are applied iteratively from left to right by replacing an occurrence of any nonterminal symbols with either other nonterminal or terminal symbols. A sequence of rule applications is called a derivation, and the resulting structure is referred to as a sentence in the language.

Taking a linguistic example to illustrate these ideas, Fig. 17.1 shows how a subset of production rules drawn from an English grammar can produce a syntactically correct sentence. The sentence commences from a root symbol (`<sentence>`) which is expanded into a `<noun-phrase>` and a `<verb-phrase>` (nonterminals), each of which in turn can be expanded further to produce lower levels in the derivation tree. Eventually, terminals (or words) are generated giving rise to a valid sentence. The process of expanding nonterminals into other nonterminals or into terminals is governed by a set of production rules which determine which expansions are syntactically valid or permissible. The infinitely large set of all possible sentences in the English language can therefore be specified by a relatively compact grammar.

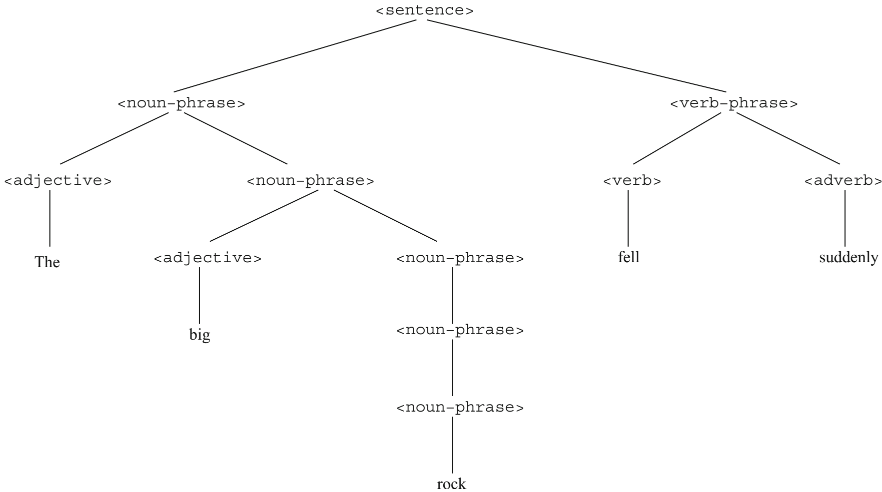


Fig. 17.1. A sample derivation tree illustrating how a subset of English grammar can produce a string of English words starting from a root symbol (*sentence*)

17.3.1 Types of Grammar

The foundations of the formal study of grammars were laid down in 1956 and 1957 by Noam Chomsky [107, 108]. Four different types of grammar were defined in his taxonomy, based on differing assumptions on the nature of the production rules, namely:

i. free grammars (also known as a type 0 grammar),
ii. context-sensitive grammars (also known as a type 1 grammar),
iii. context-free grammars (also known as a type 2 grammar), and
iv. regular grammars (also known as a type 3 grammar).

Each of these grammars is described below.

Regular Grammars

A regular grammar is one where every production rule is of the form:

$$\kappa \rightarrow \chi\gamma \qquad \text{or} \qquad \kappa \rightarrow \gamma$$

where κ and γ are nonterminals and χ is a terminal.

Context-Free Grammars

A grammar is context-free if every production rule is of the form:

$$\mathcal{I} \rightarrow \psi$$

where $\mathcal{I}$ is a nonterminal symbol and ψ is a string which is made up of nonterminal and/or terminal symbols.

Context-Sensitive Grammars

A context-sensitive grammar is one where the effect of a given production rule depends on the context in which it is applied. For example, a production rule in a context-sensitive grammar may have the form:

$$\alpha\mathcal{I}\beta \rightarrow \alpha\psi\beta$$

where α and β are strings made up of terminal and/or nonterminal symbols, $\mathcal{I}$ is a nonterminal, and ψ is a string made up of terminals and/or nonterminals. Hence, $\mathcal{I}$ is transformed into ψ in the context of β on its right and α on its left. Thus, in a different *context* the nonterminal $\mathcal{I}$ could transform into a different string.

Free Grammars

Free grammars impose no restrictions on the nature of production rules and hence they provide no restrictions on the sentences they produce. A production rule will therefore have the form:

$$\alpha \rightarrow \psi$$

where α and ψ are strings made up of terminal and/or nonterminal symbols, and there must, of course, be at least one nonterminal in α.

Any language generated by a grammar of type x is called a type x language. A hierarchy of grammars exists, as the set of all grammars of type x also includes all grammars of type $x+1$. For example, the set of all type 0 grammars includes grammars of all the other types.

Within GC, and in particular grammar-based Genetic Programming, a number of different types of grammars have been adopted, ranging from the common context-free to the less common but more powerful context-sensitive Logic and Attribute grammars. In more recent years Tree-adjoining and Tree-adjunct grammars have also been exploited to directly manipulate tree-based GP structures. Graph grammars [231] and Shape grammars [481] are examples of other grammars which have been employed in areas such as evolutionary design. Examples of some of these grammars, and the corresponding algorithms that adopt them, follow in the remainder of Part V of this book.

17.3.2 Formal Grammar Notation

When tackling a problem with GC, a suitable grammar definition must initially be defined. The grammar can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand.

The standard notation adopted is known as BNF (Backus–Naur form). In GC, a BNF definition is used to describe the output language to be produced by the system. BNF is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of *terminals*, which are items that can appear in the language, e.g., binary Boolean operators `and`, `or`, `xor` and `nand`, unary Boolean operators `not`, constants, `true` and `false`, etc., and *nonterminals*, which can be expanded into one or more terminals and nonterminals.

For example the grammar below can be used to generate Boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. As stated earlier, a grammar (G) can be represented by $G = (\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$, where $\mathcal{T}$ is the set of terminals, $\mathcal{N}$ is the set of nonterminals, $\mathcal{S}$ is a start symbol which is a member of $\mathcal{N}$, and $\mathcal{P}$ is a set of production rules that map the elements of $\mathcal{N}$ to $\mathcal{T}$. When there are a number of productions that can be applied to one element of N the choice is delimited with the '`|`' symbol. For example,

- $\mathcal{N} =$ `{<expr>,<biop>,<uop>,<bool>}`,
- $\mathcal{T} =$ `{and,or,xor,nand,not,true,false,(,)}`,
- $\mathcal{S} =$ `{<expr>}`,

And $\mathcal{P}$ can be represented as:

```
<expr> ::= (<expr> <biop> <expr>)       (0)
         | <uop> <expr>                 (1)
         | <bool>                       (2)

<biop> ::= and      (0)
         | or       (1)
         | xor      (2)
         | nand     (3)

<uop> ::= not

<bool> ::= true      (0)
         | false     (1)
```

The code produced will consist of elements of the terminal set $\mathcal{T}$. The grammar is used in a developmental approach whereby the search process generates the choice of production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely of elements from $\mathcal{T}$.

17.4 Grammatical Inference

As noted in the introduction to this chapter, the concept of a grammar can also be applied in reverse, whereby we can attempt to uncover a grammar from looking at a series of sample sentences. An example of this occurs when we attempt to create general rules for some phenomenon given our observations of examples of that phenomenon. This is a tricky task as multiple grammars can produce the same sentence and if we only have a limited number of samples to examine, it may not be possible to recover the unique grammar which underlies our observations. Examples of these approaches in the literature include Koza [340, p. 442–445] and Araujo [20].

17.5 Lindenmayer Systems

One of the first biologically inspired uses of grammars was that of the biologist Lindenmayer in 1968, to model the development of multicellular systems [370]. This work was popularised in Prusinkiewicz and Lindenmayer's book *The Algorithmic Beauty of Plants* [523]. Lindenmayer developed his formalism,

L-systems, which is effectively a grammatical representation encoding a set of rules governing the development of a plant from an embryo. The primary difference from a grammar as defined by Chomsky and an L-system is that the rewrite rules of an L-system are executed in parallel.

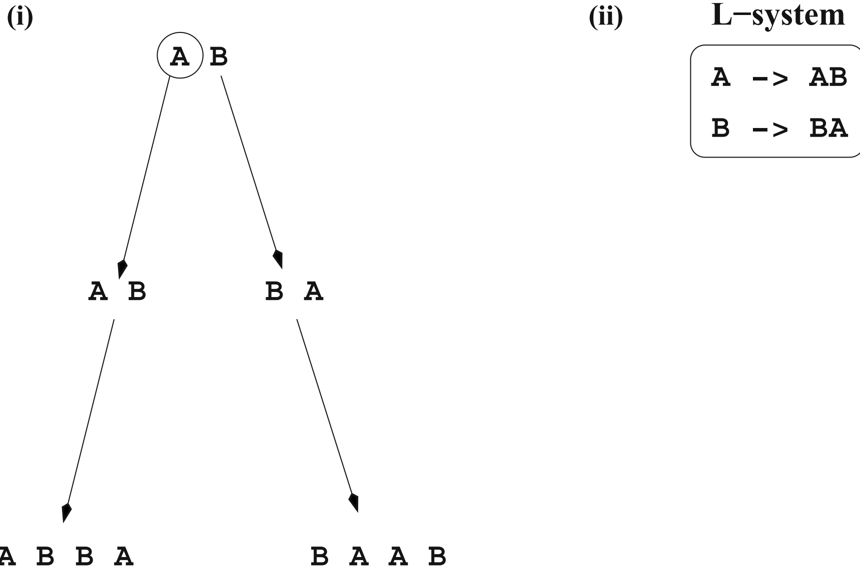


Fig. 17.2. An illustration of the string rewriting process as implemented in L-systems. The L-system (ii) is applied to the embryo string `AB` in (i). Both symbols in the embryo are simultaneously rewritten according to the rules of the L-system. The rules of the L-system can be applied iteratively to the current state of the developing string, so that `ABBA` is later rewritten as `ABBABAAB`

.

For example, given the seed string `AB` and the rules

```
A -> AB
B -> BA
```

`AB` becomes `ABBA` in one derivation step, that is the rule to replace `A` and the rule to replace `B` are executed at the same time. Fig. 17.2 further illustrates this rewrite or developmental process. A graphical interpretation of the L-system symbols is then required to visualise the simulated plant development. Typically this is achieved using turtle graphics as adopted in the programming language LOGO.

Later L-systems were extended to include cycles to model cell layers using map L-systems [371]. A number of applications of L-systems exist in the literature in addition to generating plant-like structures (e.g., [292, 523]), from

protein folding prediction [179], designing virtual creatures [284], pylons [543], and logos [467], to modelling of blood vessels [335].

As L-systems are primarily a formalism to study or model multicellular biological systems we will turn our attention to those forms of grammatical computing that have adopted grammars as algorithmic problem-solving devices. To this end, L-systems and map L-systems will be revisited later in this book from the perspective of genetic programming and developmental computing in Chaps. 18, 19 and 20. We will also later see an application of concepts from L-systems to design an optimisation algorithm based on the growth processes of plants (Sect. 25.6).

17.6 Summary

This chapter presented an introduction to developmental and grammatical computing, which are inspired by developmental biology, and the language communication devices of biological organisms as studied in the disciplines of linguistics and computer science.

The following chapters in this part of the book introduce the developmental and grammatical approaches to genetic programming, such as developmental GP inspired by Gruau's cellular encoding, grammar-guided genetic programming, grammatical evolution, and developmental TAG, in addition to algorithms inspired by genetic regulatory networks.

18

Grammar-Based and Developmental Genetic Programming

The use of grammars in genetic programming (GP) has a long tradition, and there are many examples of different approaches in the literature representing linear, tree-based and more generally graph-based forms. McKay et al. [403] presented a survey of grammar-based GP in the 10th Anniversary issue of the journal Genetic Programming and Evolvable Machines. In this and subsequent chapters, we highlight some of the more influential forms of grammar-based and developmental GP.

In grammar-based GP, the sets of GP terminals and GP functions are replaced by a grammar. This provides a significant advantage over standard GP as the grammar can be used to ensure the property of closure. That is, the GP practitioner is no longer restricted to the use of a single data type, or forced to use encodings such as strongly-typed GP. The grammar allows multitype GP, where the evolving structures can, for example, be comprised of both Booleans and reals sitting comfortably side by side.

Grammars in GP are traditionally adopted in a generative sense. That is, they provide the rules by which a sentence in the language defined by the grammar is expanded. As is characteristic of the natural world by which these algorithms are inspired, there are of course always exceptions. Banzhaf and colleagues have used grammars in the more traditional parsing approach common in computer science and compilers. In their developmental GP algorithm [30, 322, 323], the genome of each individual is used to specify a sequence of programming language primitives. During initialisation of the individual no attention is paid to the syntax of the represented primitives. As such, it is highly probable that each individual will represent a syntactically invalid program (i.e., sentence). The grammar is then used to parse the invalid sentence identifying where and how repairs should be applied to create a syntactically correct sentence.

The first popular grammar-based approach to GP was called *grammar-guided GP* (often referred to as G3P). A number of variations exist in the literature, with perhaps the most influential body of work being by Whigham [656].

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_18

In the following section we describe this tree-based approach to the use of grammars in GP.

Algorithm 18.1: Grammar-Guided Genetic Programming Algorithm

Define terminal set, function set and fitness function;
Define a grammar specifying the problem-specific language defined on the elements of the function and terminal sets;
Set parameters for GP run (population size, probabilities for mutation, crossover, etc., selection/replacement strategy, etc.);
Initialise population of solutions;
Calculate fitness of each solution;

repeat
  Select parents;
  Create offspring;
  Calculate fitness of each solution;
  Update population;
until *terminating condition*;

18.1 Grammar-Guided Genetic Programming

Grammar-guided genetic programming (G3P) is a tree-based form of GP, where each individual in the population is a *derivation tree*. A modified GP algorithm (see Chap. 7 Algorithm 7.1) incorporating a grammar definition is provided in Algorithm 18.1. In terms of implementation, the impact of adopting a grammar is that we need to have a special instance of the *initialisation* and *create offspring* steps, which take the rules of the grammar into consideration. To create each individual the user-defined grammar is consulted, and a path is selected through the grammar commencing from the start symbol. As per standard GP, it is possible to impose structural diversity onto the population during initialisation by adopting a ramped-half-and-half strategy where we impose limits on the depths of the derivation tree structures.

Let us walk through an example to visualise the process of initialisation, and to introduce the concept of a derivation tree. Taking the grammar in Fig. 18.1, encoded in BNF (see Chap. 17), which might be used to generate a simple PacMan controller, the start symbol is `<pacman>`. The start symbol `<pacman>` can be replaced with either a single `<line>` of code, or multiple lines of code `<line><pacman>`. During a random tree initialisation process there is a 50% chance either option is selected. For our purposes let us say `<pacman>` is replaced with `<line><pacman>`. The derivation tree of our expanding individual is presented in Fig. 18.2 (i).

```
<pacman> ::= <line>
           | <line><pacman>

<line> ::= <ifstmt>
         | <operator>

<ifstmt> ::= if (<condition>, <operator>, <operator>)

<condition> ::= isGhostAhead()
              | isGhostBehind()
              | isGhostRight()
              | isGhostLeft()
              | isWallAhead()
              | isWallBehind()
              | isWallRight()
              | isWallLeft()

<operator> ::= turnLeft()
             | turnRight()
             | moveForward()
             | moveBackward()
```

Fig. 18.1. An illustrative grammar which might be used to create a simple PacMan controller

The process of replacing a nonterminal symbol with the symbols on the right-hand side of a production rule is called a derivation step, and hence we use the term derivation tree for the tree which represents the state of the derivation at any point in time. Both `<line>` and `<pacman>` must now be expanded again randomly, and both of these nonterminals have two possible replacements. In the case of `<line>` there is a 50% chance that it will become either an *if statement* (`<ifstmt>`) or an `<operator>`. Flipping a coin in each case results in `<line>` replaced with `<operator>` and `<pacman>` being replaced with `<line>`. Fig. 18.2 (ii) and (iii) illustrates these two derivation steps. `<operator>` is replaced with `turnLeft()` (see step (iv)) and `<line>` is replaced with `<operator>` and finally `moveForward()` in steps (v) and (vi). The resulting program is trivial with PacMan trying to turn left and then move forward. Typically in a game like PacMan the program will be contained in a wrapping function which in this case is executed in a loop until a game termination state is reached.

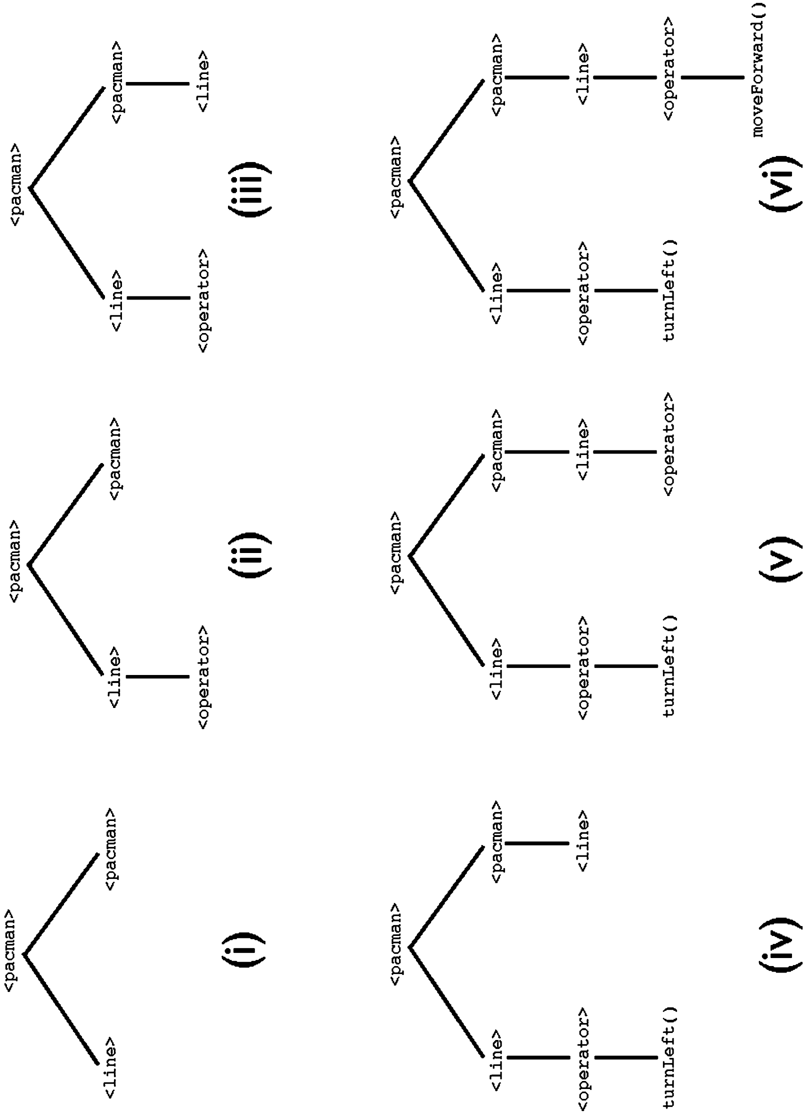


Fig. 18.2. An outline of an example derivation sequence in the creation of a grammar-guided GP individual

A more interesting PacMan controller would contain conditional statements. An example individual comprised of an <ifstmt> is illustrated in Fig. 18.3. In this case the individual is comprised of a single conditional statement which employs the isGhostAhead() function as the <condition>. If a ghost is detected directly in front of PacMan then this function returns a Boolean value of true and PacMan will attempt to moveBackward(), otherwise PacMan will take a step forward invoking the moveForward() function.

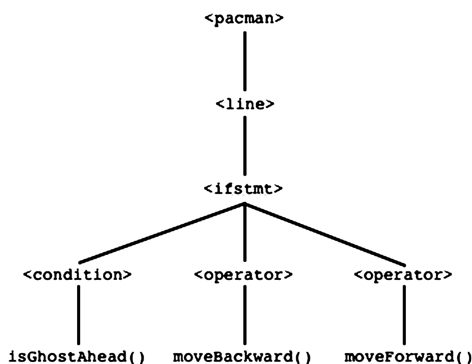


Fig. 18.3. A example PacMan controller which is comprised of a conditional IF statement

Once a population of derivation trees is created the fitness of each individual can be calculated in a normal GP-like manner. Fitness values can then be used to inform the selection stage of the evolutionary engine with genetic operators such as mutation and crossover being applied to generate new individuals. These genetic search operators are applied directly to the derivation trees.

In terms of *subtree* crossover this means that a crossover site is selected from the set of nonterminal symbols contained in the derivation tree of the first parent. The crossover site in the second parent is then selected from the subset of nonterminal symbols in the derivation tree of the second parent, which have the same value as the nonterminal symbol at the crossover site in the first parent. This process guarantees that when a derivation subtree is swapped from the second parent into the first parent a syntactically valid individual is created. Fig. 18.4 illustrates an example derivation subtree crossover in operation. In a similar manner, a mutation site is selected from the set of nonterminal symbols in the derivation tree. Legal mutants/replacements can then be selected from the grammar as per the derivation tree initialisation process.

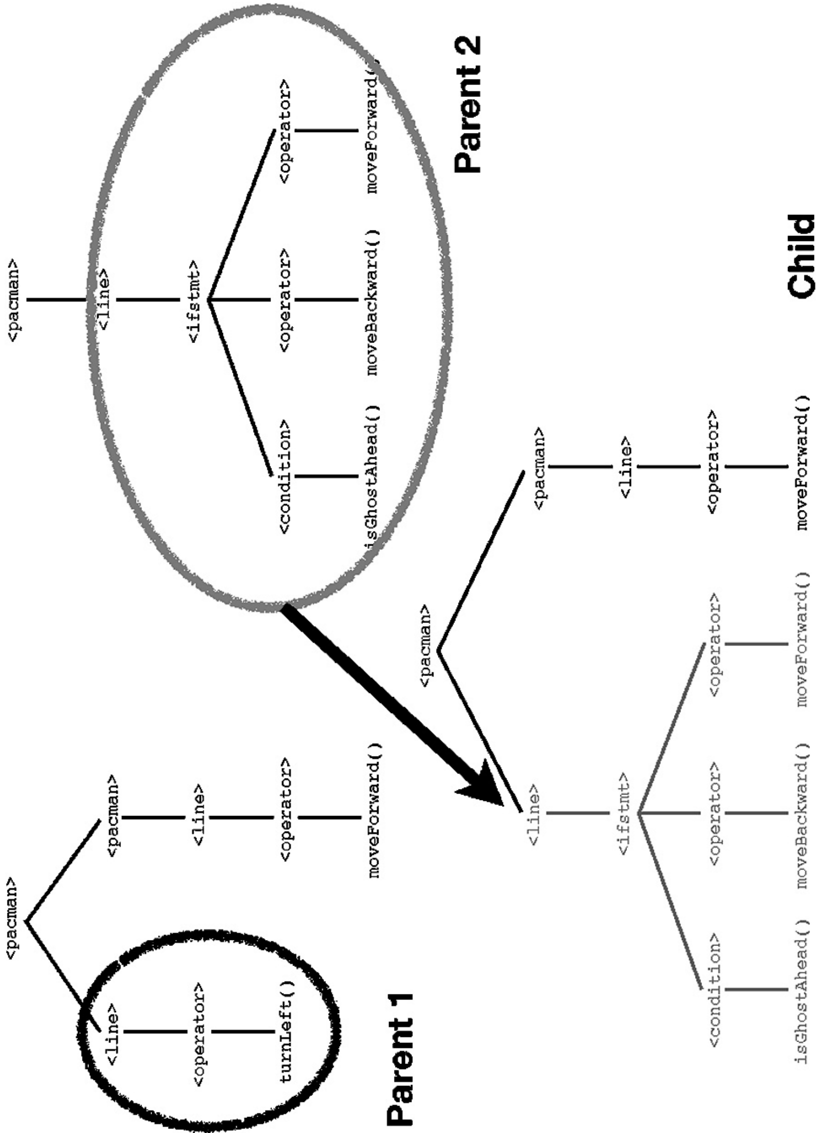


Fig. 18.4. An illustration of derivation subtree crossover

This representation has been extended to assign probabilities to each production rule in the grammar, allowing a bias to be explicitly introduced during the creation of the initial population, and during mutation events. In a further extension of this approach the probabilities can be modified/evolved during a run, allowing the population to develop a global model of a biased language [572].

18.1.1 Other Grammar-Based Approaches to GP

There are a large number of grammar-based approaches to GP [403]. In addition to those outlined earlier in this chapter some notable examples illustrating the diversity of grammars adopted include Logenpro [662], which utilises logic grammars. In later chapters we will revisit some of the most popular forms of grammar-based GP and their variants, including grammatical evolution (Chap. 19) with context-free, attribute, logic, meta and shape grammar variants. Also in Chap. 20 we will investigate tree-adjoining grammar (TAG) forms of GP, including TAG3P and an alternative form of grammatical evolution (TAGE).

18.2 Developmental GP

In this section we provide an overview of a sample of the primary GP methods which have been inspired by developmental systems. In the first example we provide an introduction to genetic L-system programming [292]. The second approach (binary GP) represents an algorithm which coevolves the underlying genetic code. The third, cellular encoding, is a neuroevolutionary algorithm which inspired the fourth highly successful algorithm outlined here, Koza's approach to the evolution of analog circuits. It is worth highlighting again that many of these approaches explicitly adopt grammars in their underlying representation. Specifically, genetic L-system programming, binary GP and cellular encoding adopt L-system grammars, context-free grammars, and graph grammars respectively.

18.2.1 Genetic L-System Programming

Jacob [292, 293] described an approach to GP based on L-systems, where he considers L-systems as 'rule-based, developmental programs', and uses a turtle graphics interpretation of the L-system to evolve 3-D images of plants. Fig. 18.5 outlines an illustration of how an L-system might be interpreted as a 2-D turtle graphic. A grammar is used to specify the legal construction of L-systems by the GP search engine. Typed mutation and subtree crossover operators are designed to manipulate the evolving L-systems.

A number of applications of L-systems and GP have appeared in the literature since Jacob's work. For example, an approach that used L-systems,

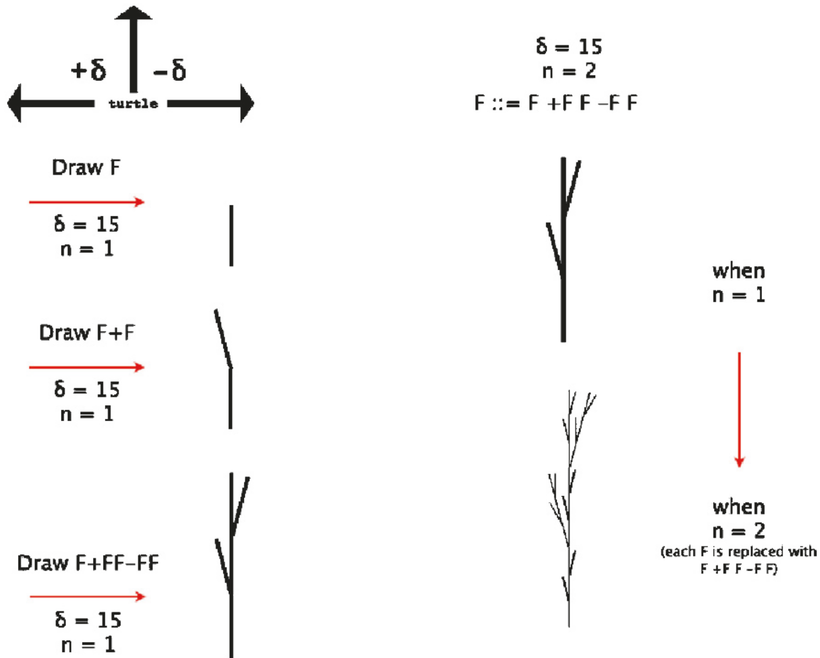


Fig. 18.5. An illustration of a turtle graphics interpretation of L-systems. F is the rule which results in a line being drawn in the direction specified by δ. The length of the line and δ are parameters of the system. On the left side of the image we see three examples of L-systems (i.e., F, F+F and F+FF−FF) drawn where $\delta = 15$. Another parameter is the number of iterations (n) or the number of developmental steps for which the rules of the L-system are executed. On the right we see the rule which states how each F in the developing image is replaced with F+FF−FF

similar to grammatical evolution (GE), tackled the problem of inferring protein secondary structures [179]. In another application of GE, L-systems were used to design logos [467]. Later in Chap. 20, we will see how L-systems have been used in an alternative, more flexible, developmental approach to GP.

18.2.2 Binary GP

Banzhaf introduced an approach to GP named *Binary GP* [30], and later studies provided a more in-depth analysis of its behaviour [322, 323]. As we will see in the following description the algorithm adopts a grammar in a repair process.

In Binary GP each primitive symbol of the programming language is assigned a binary code (its genetic code). A member of the population is a binary string with the individual being parsed into groups of bits referred to

as *codons*. Each codon is a predetermined fixed number of bits. Table 18.1 outlines an example Binary GP genetic code.

Symbol	*Binary Code*
$+$	000
$-$	001
$*$	010
$/$	011
w	100
x	101
y	110
z	111

Table 18.1. An example Binary GP genetic code comprising four problem-specific variables (w, x, y and z) and a set of arithmetic operators

A grammar for infix expressions utilising the primitive symbols presented in Table 18.1 might look as follows:

```
<expression> ::= <expression> <binaryoperator> <expression>
               | <variable>
<binaryoperator> ::= + | - | * | /
<variable> ::= w | x | y | z
```

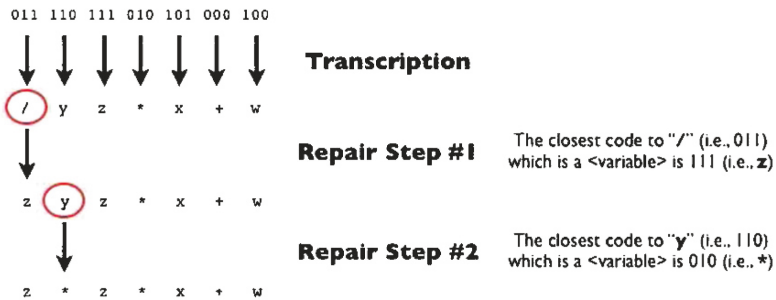


Fig. 18.6. A binary GP mapping from a randomly generated binary string divided into codons of three bits

For any randomly generated binary string individual we can then parse groups of three-bit codons at a time, in each case decoding the binary codon into its equivalent language primitive according to the lookup table. Fig. 18.6 outlines a sample mapping from a binary string to primitive symbols and its subsequent repair into a legal expression according to the infix grammar.

The repair process operates by detecting an *illegal* symbol, and subsequently replacing it with a legal primitive symbol chosen by calculating the Hamming distance of the genetic codes for all the available legal symbols which can be utilised legally in this context. The repaired Binary GP individual can then be evaluated in the usual GP manner and a fitness assigned to it.

Keller and Banzhaf extended the Binary GP system to evolve the genetic code itself [322, 323]. In a related study, Grammatical Evolution was adapted to evolve its genetic code using a metagrammar [474]. Both sets of studies found that it was possible to coevolve the code with the evolving solutions.

18.2.3 Cellular Encoding

A large number of papers which appear in the *Generative and Developmental Systems* track at recent GECCO conferences have focused on developmental approaches to neuroevolution (see Chap. 15 for an exposition of these methods such as NEAT). In earlier work in this area, Gruau developed a novel and influential representation which allowed GP to evolve the topology and weights of the edges of artificial neural networks.

Gruau's approach adopted graph grammars and was dubbed cellular encoding [231]. Effectively the graph grammar represented rules by which an embryonic graph (e.g., a single node) could be transformed into a mature neural network. The topology, edge weights and threshold functions could all be evolved using this representation. The GP individual (a tree) encodes the transformations which are to be applied to the embryonic *ancestor cell*. Fig. 18.7 outlines an example of how a simple graph grammar can be used to generate a multilayer perceptron.

Cellular encoding served as inspiration for a developmental approach to GP which is used to evolve analog circuits.

18.2.4 Analog Circuits

The application of GP to the evolution of analog circuits is one of its most noteworthy success stories, resulting in solutions which have been patented in their own right. As noted above, the approach is heavily inspired by Cellular Encoding, and applies a set of transformations encoded in a GP tree to a developing embryonic circuit initially comprised of a single wire. A large proportion of Koza's third [342] and fourth [343] books on GP are dedicated to analog circuit evolution, and a recent article in the 10th anniversary issue of the Genetic Programming and Evolvable Machines journal outlines the broad spectrum of human-competitive results which had been achieved by GP up to 2010 [344].

18.2.5 Other Developmental Approaches to GP

Other examples of developmental approaches to GP include those which adopt self-modification, such as ontogenetic programming [588] and self-modifying

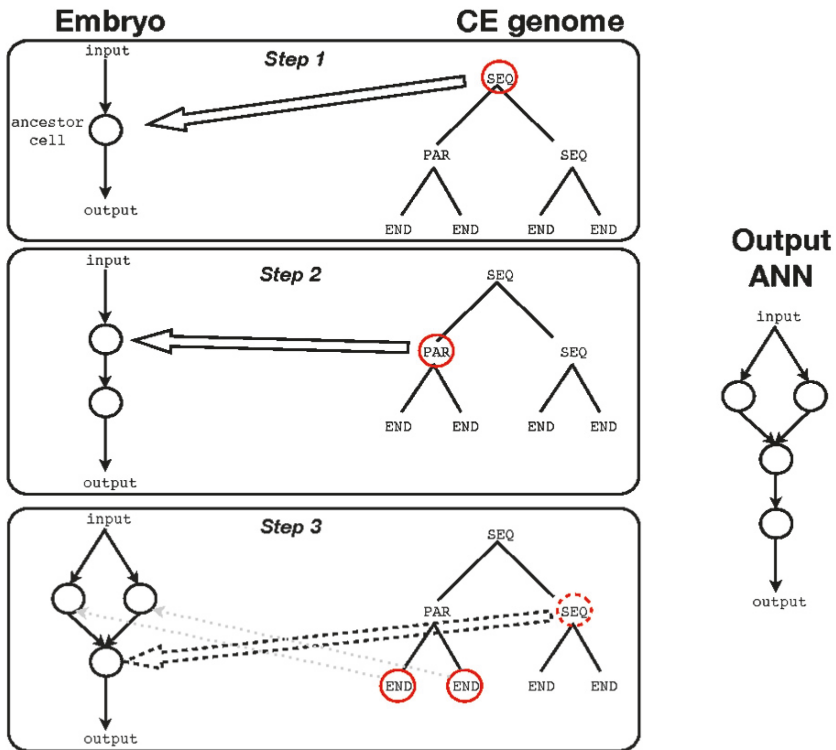


Fig. 18.7. An example Cellular Encoding mapping from a GP tree and embryonic ancestor cell to an ANN. Note a simplified GP tree is presented, additional functions available as part of the GP function set can, for example, modify the threshold of each node and the number of edges from each node

Cartesian GP [250, 251]. The aim here is to allow an executing program to modify its behaviour over the lifetime of its execution in order to enable the program to react to a changing problem environment.

Spector and Stoffel describe ontogenetic programming which uses a linear stack-based language, including a set of operators which can 'self-modify' the resulting program phenotypes [588]. For example, the operator `segment-copy` copies a part of the linear program over another part of the program, the operators `shift-left` and `shift-right` rotate the program to the left or the right. Spector and Stoffel also describe how a similar approach can be applied to tree-based GP using a `subtree-copy` function.

Self-modifying Cartesian GP (SMCGP) [250] is a similar approach to Spector and Stoffel, except the self-modification operators are applied to a different underlying representation, namely that of Cartesian GP [412], a graph-based encoding. SMCGP differs from CGP in that the nodes within the graph adopt

relative addressing. This allows subgraphs to effectively become modules by maintaining their semantics. At runtime, self-modification occurs due to the fact that nodes within the graph can be replaced with subgraphs. SMCGP is shown to have performance gains on a number of problems over the standard CGP.

The use of tree-adjoining grammars has been a significant step forward in grammatical and developmental approaches to GP, and in Chap. 20 we describe the TAG3P, DTAG3P and TAGE approaches. Tree-adjoining grammars are effectively grammars which describe tree-transformation rules and as such represent a self-modifying encoding for GP where each step of development is a valid, executable sentence/program.

18.3 Summary

This chapter presented an overview of both grammatical and developmental approaches to genetic programming. However, this is not the complete picture. In Chaps. 19 and 20 we introduce one of the most popular approaches to GP, namely grammatical evolution, and another influential grammar-based approach, tree-adjoining grammar-based genetic programming (TAG3P) which was recently expanded to include an advanced developmental process (developmental TAG3P).

19

Grammatical Evolution

Grammatical Evolution (GE), a form of grammar-based genetic programming (Chap. 18), is an algorithm that can evolve *computer programs*, *rulesets* or, more generally, *sentences* in any language [150, 460, 470, 472, 547]. Rulesets could be as diverse as a regression model, a set of design instructions, or a trading system for a financial market. Rather than representing the programs as syntax trees, as in GP (Chap. 7) [340, 514], a *linear genome* representation is used in conjunction with a grammar.

In addition to the standard set of evolutionary principles adopted in evolutionary computation, as described in Chap. 2, GE further extends inspiration taken from the biological analogy by employing neo-Darwinian principles of genetics that have been uncovered by molecular biologists. The most significant of these is the adoption of a distinction between the genotype and phenotype, similar to that which exists in nature. That is, through a mapping process, the genetic material (the genotype) contains the instructions that are used to control the development and day-to-day operation of a living organism (the phenotype). The molecules making up the genetic material (DNA) are distinct from the molecules responsible (proteins) for the phenotype. Each individual in GE, a variable-length linear structure, contains in its genome the information to select production rules from a grammar. Typically, each codon is either a group of bits (e.g., a byte, that is, eight bits) or an integer [288]. The primitive components of a GE genome are referred to as codons to reflect more closely their biological counterparts. A gene then is a collection of codons (Sect. 19.1).

It is in the notion of a genotype–phenotype mapping that the use of a grammar is exploited. The grammar contains the rules governing how the development of the phenotype is conducted, and as such can contain domain knowledge biasing the form a phenotypic solution can take. It is this use of grammars that provides GE with enormous flexibility in what it is capable of representing in an evolving population. Reflecting this is the wide range of application domains to which GE has been applied: highlights include architecture and engineering design [87, 399, 481, 485], fi-

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_19

nance [75, 122, 447], music and art [396, 397, 444, 467, 573], computer games [506, 507, 570, 571], animation [428, 429], ecosystem modelling [448], and autonomous networks [264, 265, 266].

Another particular benefit of GE is that the separation of the search and solution spaces allows the implementation of generic search algorithms without a requirement to tailor the diversity-generating operators to the nature of the phenotype. This removes one of the problems which can arise in genetic programming, that of *closure* (Chap. 7.1.2). It also separates GE from the evolutionary search engine, which can be replaced with any search algorithm which is capable of manipulating an encoding of rule choices (e.g., integers, reals, and bits). Examples of alternative search engines include PSO [461, 463] and DE [465].

In the remainder of this chapter we will introduce the GE methodology and illustrate how it might be applied to different problem domains. Later we will discuss some of the recent developments in GE. But first we will set the stage and provide background on the original inspiration for GE, with a short primer on the process of gene expression in eukaryotes (organisms each of whose cells contains a nucleus).

19.1 A Primer on Gene Expression

The GE system is inspired by the biological process of generating a protein from the genetic material of an organism. Proteins are fundamental in the proper development and operation of living organisms and are responsible for traits such as eye colour and height [366].

The genetic material (usually DNA, deoxyribonucleic acid) contains the information required to produce specific proteins. DNA encodes information using a four-letter alphabet (adenine, guanine, cytosine and thymine — A, G, C, T). In human chromosomes, roughly 3 billion nucleotides (letters) are strung together on two complementary strands which form a double helix.

When a gene's instructions are to be expressed, the double helix structure of DNA is opened to allow a single-strand copy of the gene's sequence to be *transcribed* onto RNA (ribonucleic acid). During this transcription process, thymine (T) is replaced by uracil (U). Not all portions of DNA encode proteins. In fact, only about 1-2% of human DNA does so, with the remaining portions of DNA being historically referred to as 'junk' DNA. Initially it was thought that junk DNA did not serve a useful function; however, it is now known that it encodes important cell-regulatory information and current estimates are that over 80% of DNA in the human genome "serves some purpose, biochemically speaking" [505].

Sequences of DNA and their corresponding RNA are comprised of *exons* or coding sequences, and *introns* or noncoding sequences. The RNA strand is spliced, or edited, into mRNA (messenger RNA) in a cell structure known as the spliceosome. In this process the intronic RNA is snipped out, and

the remaining exonic RNA is combined. Next, the mRNA carries the genetic instructions out of the cell nucleus into a structure called the ribosome in the cell's cytoplasm, which *translates* the instructions into amino acids. The mRNA is read one codon (which is made up of a group of three letters) at a time in the ribosome, and the amino acids specified by these letter sequences are fetched by tRNA (transfer RNA). The amino acids are strung together to produce a protein. The sequence of amino acids is very important as it plays a large part in determining the final three-dimensional structure of the protein, which in turn has a role to play in determining its functional properties.

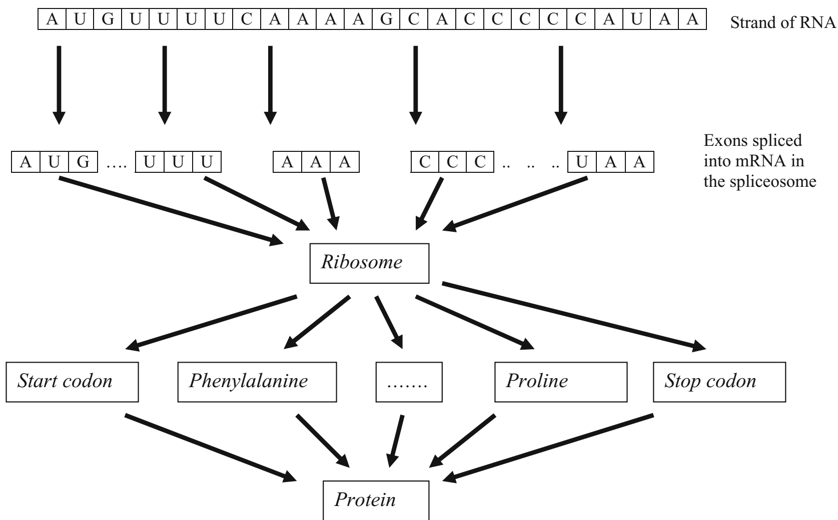


Fig. 19.1. Illustration of RNA to protein mapping. Initially, the strand of RNA is spliced and edited in the spliceosome. During this process, introns are edited out. Next, mRNA carries the genetic instructions to the ribosome where tRNA is generated in order to assemble the required amino acids (molecules). These molecules are then combined to form the protein

As the protein forms, it folds into a 3-D shape. Folding is at least partly determined by the affinity of the amino acids for water; in particular, hydrophobic amino acids fold towards the inside of the protein, away from the cell's cytoplasm. Thus, in summary, the genotype to phenotype mapping process is as follows:

$$\text{DNA} \rightarrow \text{RNA} \rightarrow \text{Protein}$$

The above description is a much-abbreviated version of the process of gene expression. A significant amount of research is currently taking place in an

effort to better understand the processes of gene expression, and how the constituent amino acid sequence of a protein folds into its native conformation. It has to be stressed that the process illustrated above of DNA transcribing to RNA and translated to protein was known as the Central Dogma of Molecular Biology, with a unidirectional flow of information being a defining characteristic. It has since been established that the information flow is not unidirectional: information can flow back through this chain of molecules.

The Translation Grammar

An important step in the genotype–phenotype map is the decoding of segments of mRNA into amino acids. Each codon of three letters specifies one of 20 standard amino acids, or one of three 'stop translating' signs — UGA, UAG or UAA — which in turn form the building blocks of proteins. If we consider that there are $4^3 = 64$ possible codons (since we are dealing with a base-4 alphabet where the primitives are A, U, G and C, and there are three available positions in a codon), it is clear that codons could encode more than 20 amino acids. There is not a one-to-one correspondence between codons and amino acids, and multiple codons can actually specify the same amino acid. Generally, the different codons which specify the same amino acid only differ in their last letter. For example, UCU, UCC, UCA and UCG all encode the amino acid, serine.

A key point to note is that the rules governing the decoding of codons into specific amino acids, and subsequently into proteins, can be considered as a grammar. Interestingly, although many organisms have the same basic genetic alphabet and generate the same amino acids, not all have the same mapping from specific codons to specific amino acids. For example, although most organisms read the RNA codon CTG as the amino acid leucine, many species of the fungus *Candida* translate this codon as serine [201].

There have been some studies in genetic programming where researchers have attempted to (co)evolve the underlying genetic code of their representation with some success [323, 324], including a study using metagrammars with GE [474].

19.2 Extending the Biological Analogy to GE

Through the adoption of a genotype–phenotype mapping process coupled to the use of a grammatical representation, GE can take advantage of its modular framework in a number of ways. Potential benefits of adopting genotype–phenotype maps are highlighted by O'Neill [460], and include:

i. A separation of the search (binary strings) and solution spaces (sentences) which removes the necessity to exclusively adopt a variable-length genetic algorithm (or even any evolutionary algorithm!) as is standard in GE

as the search engine. The search operators of the evolutionary algorithm themselves (e.g., the genetic operators of crossover and mutation) operate on an abstraction of the phenotype and as such do not have to take into consideration issues such as syntactic correctness of the phenotype, as the mapping process can be used to ensure this occurs automatically.

ii. An abstraction of a program's representation (a grammar) which can be used in a pluggable manner to generate sentences in arbitrary languages.

iii. Efficiency gains for an evolutionary search are possible through the adoption of a many-to-one mapping and a degenerate genetic code. This can be achieved by allowing neutral evolution to occur. Neutral evolution occurs when there are changes at the genotype level which are neutral or nearly neutral with respect to the fitness of the phenotype. For example, by allowing the functionality of the phenotype to be preserved while changes to the genotype occur, the population can potentially traverse otherwise infeasible regions of the search space. A further consequence of the many-to-one mapping is the maintenance of genetic diversity within a population by allowing many different genotypes to represent the same phenotype, thus helping to prevent loss of genetic material and premature convergence.

We will see later, in Chap. 21, how another desirable feature [460], alternative gene expression control — where feedback loops to the environment influence the development of the output phenotype — is implemented through Genetic Regulatory Networks when coupled to a tree-adjoining grammar in the TAGE algorithm (Chap. 21.2). We will now present an overview of how the genotype–phenotype mapping process occurs in GE through the mapping of a sample individual.

19.3 Example GE Mapping

When applying a grammar-based form of GP to any problem, first we must specify the grammar. The grammar dictates the language (or the structures) which the evolving population is searching. Each individual in the population typically represents a single sentence in that language. With GE, a BNF grammar definition is commonly employed to specify the language. As outlined in Sect. 17.3.2, BNF is a notation which specifies the rules by which a sentence may be constructed.

Adapting the example of evolving behaviours for the computer game Super Mario [506], we illustrate the mapping process of GE. The two inputs to the mapping are:

i. the genome (Fig. 19.3); and
ii. the grammar (Fig. 19.4).

Fig. 19.2. A screen capture from the implementation of Mario as used for the Mario AI competition.

The grammar specifies a language which defines a controller called a *behaviour tree*. The output sentences of this language are therefore instances of behaviour trees which determine the behaviour of the Mario character as he navigates the platform world which he inhabits. Figure 19.2 provides a screen capture from the Mario game as implemented for the Mario AI competition [624]. The goal of Mario is to reach the end of the level by successfully navigating the platforms and enemies as fast as possible, and collecting as many additional points along the way as possible. Points are collected for killing enemies and collecting bonuses such as gold coins. Behaviour trees (BTs) are a convenient way to organise behaviours in a hierarchical manner such that higher-level behaviours appear towards the roots of the trees with more primitive behaviours towards the leaves. As specified in Fig. 19.4, a BT in this case is comprised of one or more nodes (<Node>), where each node can be one of two types, a condition (<Condition>) or action (<Action>). Actions include the primitive behaviours moveLeft, moveRight, jump and shoot. Mario in this case can 'sense' two environment states, namely whether there is an obstacleAhead or an enemyAhead, and so we specify two kinds of <Condition> statements which handle detection of these states and then we must evolve an appropriate response by expanding the corresponding conditions <Action> component.

```
34 42 74 15 12 76 62 2 27 6 111 57 2
```

Fig. 19.3. The genome used to illustrate GE's mapping process. Note for simplicity we specify integer codon values directly. Some implementations of GE use binary codons which are first transcribed into their corresponding integer values

```
<StartSymbol> ::= sequence <BT> | selector <BT>
<BT>          ::= <BT> <Node> | <Node>
<Node>        ::= <Condition> | <Action>
<Condition>   ::= if(obstacleAhead) then <Action>;
                | if(enemyAhead) then <Action>;
<Action>      ::= moveLeft; | moveRight; | jump; | shoot;
```

Fig. 19.4. A grammar which could be used to generate behaviours for the computer game Mario. The grammar specifies a language which defines a controller called a *behaviour tree*. The output sentences of this language are therefore instances of behaviour trees which determine the behaviour of the Mario character

The start symbol (`<StartSymbol>`) for the grammar from which the mapping process commences is either a `sequence` or `selector` behaviour tree, where the content of the behaviour tree is defined through the expansion of the nonterminal `<BT>`, which has two possible rules which can be applied to it. A `sequence` behaviour tree executes all the child subtrees in sequence from left to right until one returns a fail (similar to logic AND of the subtrees), whereas a `selector` behaviour tree executes subtrees until one succeeds (similar to logic OR). To decide what production rule we use to replace the initial `<StartSymbol>` symbol with, we read the next available codon from the genome (i.e., the first integer value in Fig. 19.3), and plug the number of rule choices and the codon integer value into GE's mapping function below, which applies the mod operation (%), i.e., calculates the remainder of the division of the codon value (`c`) by the number of rule choices (`r`).

$$\text{Rule\#} \;=\; c \,\%\, r$$

This results in $34 \,\%\, 2 = 0$, so `<StartSymbol>` is rewritten as `sequence<BT>`. The mapping process continues by expanding the nonterminal `<BT>`.

Reading the second codon on the genome, this results in $42 \,\%\, 2 = 0$, so `<BT>` is now replaced with `<BT> <Node>`. The mapping process continues by reading the leftmost nonterminal in the developing phenotype and consulting the next available codon on the genome when a choice of production rules is available. The developing sentence (or phenotype) can be represented using a *derivation tree*. The remainder of the mapping process which results in a behaviour tree for Mario is outlined in Figs. 19.5 and 19.6. The corresponding derivation tree for each developmental step is also provided.

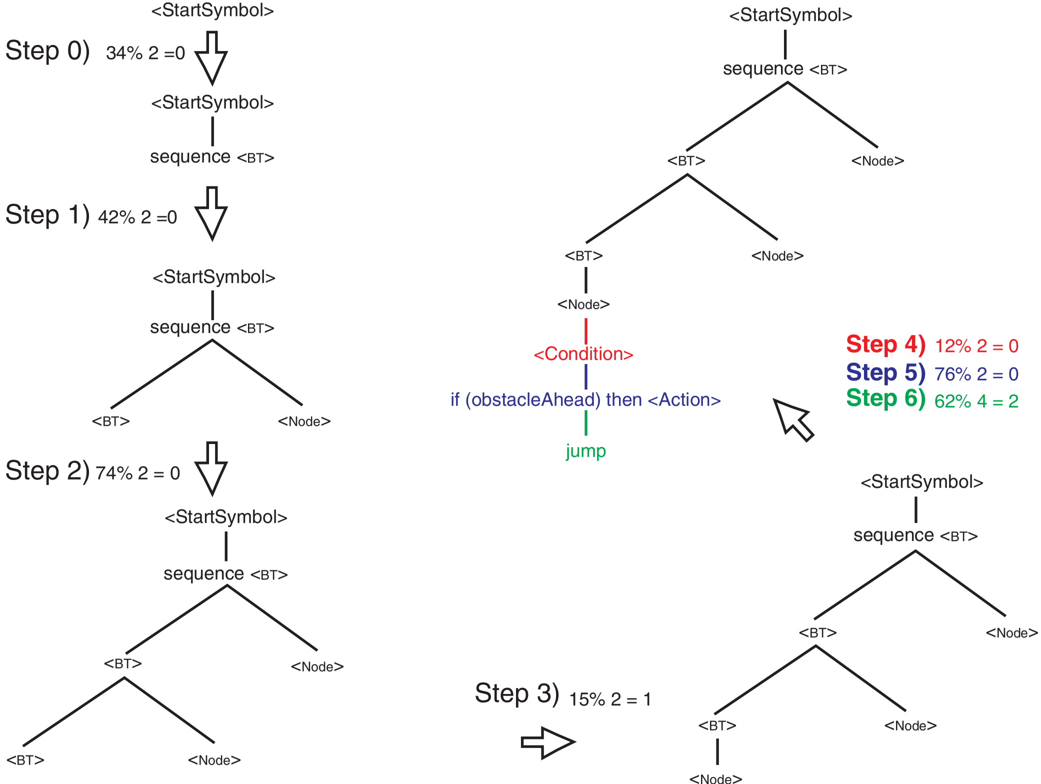


Fig. 19.5. An outline of the start of the GE mapping process which generates a behaviour tree controller for Mario from the grammar in Fig. 19.4 and using the genome from Fig. 19.3. The mapping is continued in Fig. 19.6

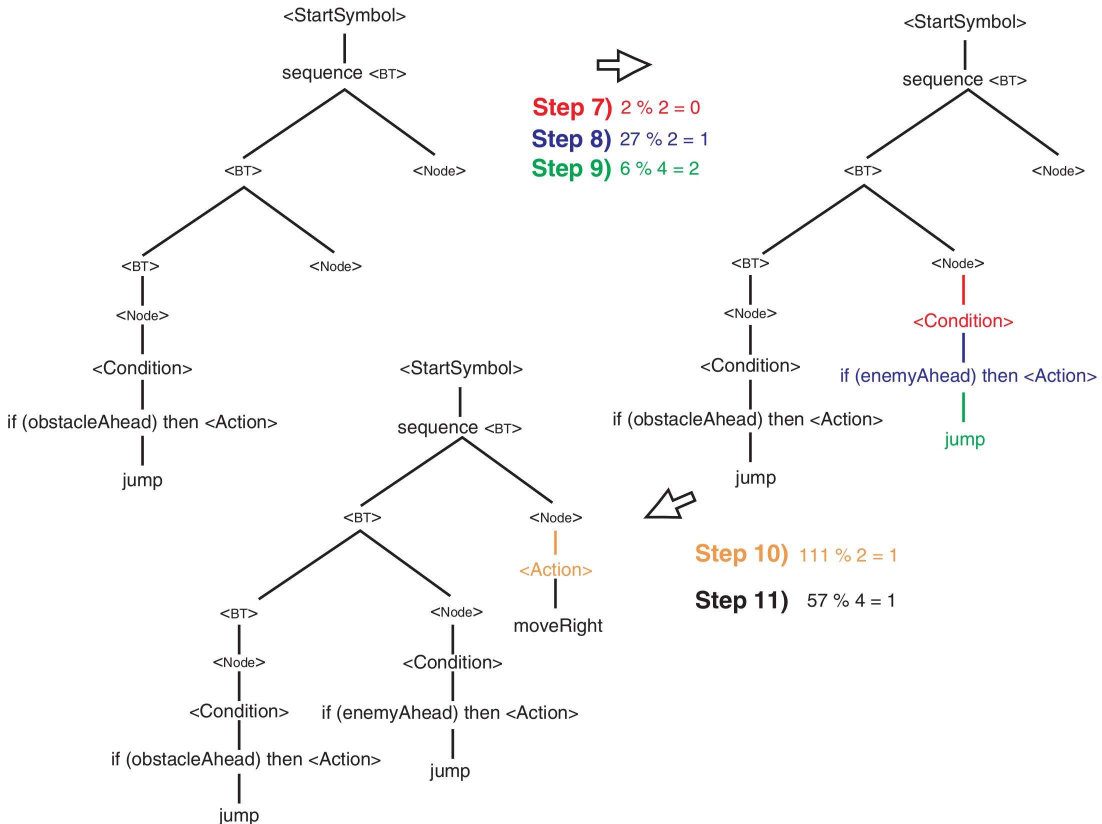


Fig. 19.6. A continuation of the mapping process started in Fig. 19.5. The resulting behaviour tree would cause Mario to jump if it detects an obstacle ahead, else jump if it detects an enemy ahead, else it moves to the right

During the genotype to phenotype mapping process, it is possible for individuals to run out of codons and still have outstanding nonterminal symbols which have not been completely mapped to terminals (language primitives). In order to resolve this issue the early versions of GE proposed the *wrap* operator. When applied, the codon *reading head* is returned to the first codon in the genome. As such, codons are reused when wrapping occurs. This is quite an unusual approach in evolutionary algorithms as it is entirely possible for certain codons to be used two or more times.

The technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [366]. It is possible that an incomplete mapping could occur, even after several wrapping events, and typically in this case the mapping process is aborted and the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

One potential complication is that any one codon can be used in different contexts (i.e., for different nonterminal symbols) and so create complex functional dependencies. A number of alternative strategies have been proposed, from a mapping function which removes the functional dependencies [320], to repair strategies which guarantee completely mapped individuals (e.g., [485]). For example, recursive rules which have the effect of expanding the number of nonterminals in a developing phenotype are temporarily removed from the grammar, thus restricting the selection of production rules which map nonterminals to terminals when reusing the genome with a wrap operation.

Algorithm 19.1: Grammatical Evolution Algorithm

```
Define terminal set, function set and fitness function;
Define a grammar specifying the problem-specific language defined on the
elements of the function and terminal sets;
Define the fitness function;
Set parameters for GE run (population size, probabilities for mutation,
crossover, wrapping, etc., selection/replacement strategy, etc.);
Initialise population of solutions;
Apply genotype–phenotype map;
Calculate fitness of each solution;

repeat
    Select parents;
    Create offspring;
    Apply genotype–phenotype map;
    Calculate fitness of each solution;
    Update population;
until terminating condition;
```

An outline of grammatical evolution is provided in Algorithm 19.1, which is a variant of the GP (Algorithm 7.1) and grammar-guided GP (Algorithm 18.1) algorithms presented earlier, in that it incorporates a genotype–mapping step. The primary difference to the grammar-guided GP algorithm is the application of the *genotype–phenotype map* to construct the program tree from the user-defined grammar and genome. In the canonical form of GE the *initialisation* and *create offspring* steps adopt (variable length) genetic algorithm style strategies on linear genome structures, which are typically comprised of either binary or integer values.

Grammatical evolution has received considerable attention since its introduction and there is a wide literature on the topic. Some of the more recent developments are focused on the various components (Fig. 19.7) of the GE approach, including the search engine, the grammar, and the mapping process itself. Rather than provide a comprehensive review of all research in this area, we will signpost some of the significant advances in the remainder of this chapter.

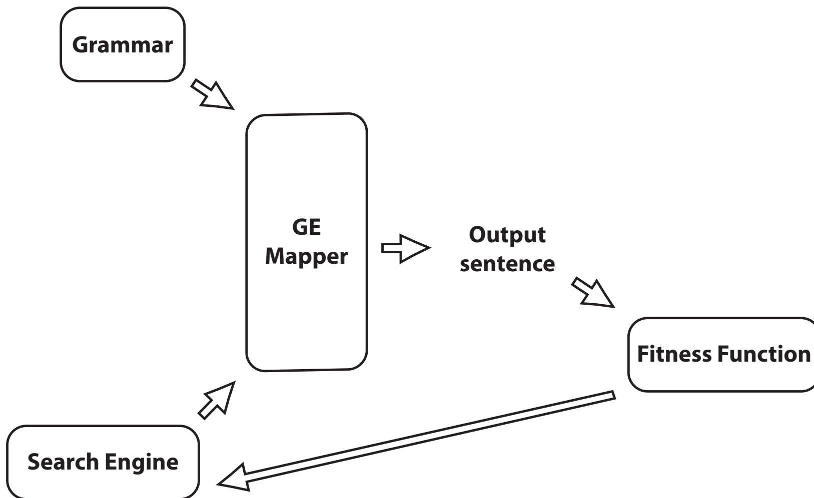


Fig. 19.7. Modular structure of grammatical evolution

19.4 Search Engine

The search engine in GE is responsible for generating the order of production rule choices by manipulating the genetic encoding of each member of the population. This is achieved using the diversity-generation operators of the search algorithm which need to be capable of exploring a variable dimension space whilst maintaining a balance between exploitation and exploration, in order to ensure effective search towards a target (e.g., the global optimum) without suffering from premature convergence towards local optima. Research directed towards the search engine component of GE has included replacing the standard variable-length genetic algorithm-like algorithm with alternatives such as differential evolution and particle swarm optimisation. Another vein of research has examined the behaviour (e.g., locality) of the evolutionary algorithm's search operators, in particular mutation when applied to both binary and integer genome encodings. Research has also focused on the impact of crossover on search performance when applied to both the genome and derivation trees. In this section we highlight some of this research.

19.4.1 Genome Encoding

The earlier studies on GE typically adopted 8-bit binary codons as the underlying genetic encoding of a rule choice (e.g., [470]), which must then be transcribed to their corresponding integer value. Alternative implementations of GE quickly arose which encoded codons directly using integer values (e.g., [319]). A study comparing binary and integer encodings and their corresponding bit-flip and integer mutations found that on the problems examined the integer encoding had a statistically superior performance [288]. Consequently implementations of GE such as GEVA [480] now adopt integer encoding as standard.

19.4.2 Mutation and Crossover Search Operators

A detailed analysis of GE's one-point crossover on variable-length genomes has found that it is superior to random search in the form of headless chicken crossover operators [482]. However, it has the undesirable consequence of producing so-called *ripple events*, as codons when moved during crossover from one parent to the other can be placed into a different nonterminal context, with the result that their meanings change. Because of this, GE's one-point crossover is referred to as *ripple crossover*. To avoid this issue, Harper and Blair investigated the application of more traditional grammar-based GP crossover operators which are applied to the derivation trees produced during GE's genotype–phenotype mapping. A number of alternatives were examined with great success [253, 254].

A number of studies have focused on GE's mutation operator. Oetzel and Rothlauf [542] examined the locality property of mutation events, finding that

```
<e> ::= <o><e><e> (0)
      | <v>       (1)
<o> ::= +         (0)
      | *         (1)
<v> ::= x         (0)
      | y         (1)
```

Fig. 19.8. A simple binary choice grammar to illustrate the impact of different types of mutation events in GE in Figs. 19.9 and 19.10

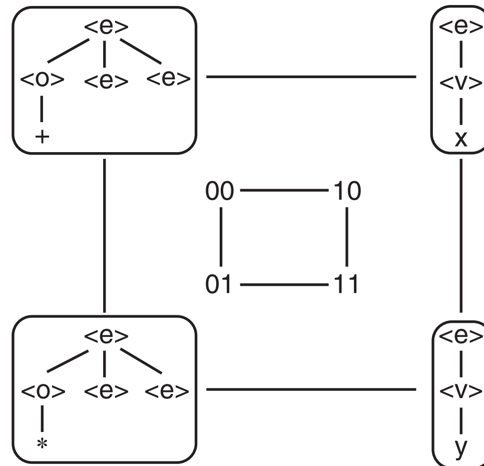


Fig. 19.9. An illustration of nodal and structural mutation events on the derivation tree for the first two codon choices. When mutating from 00 to 10, or from 01 to 11, we see structural changes, whereas mutating between 00 and 01, or between 10 and 11, results in only a single node value being modified

about 10% of events have poor locality. They also confirmed earlier observations [460] that a large number of mutations were neutral. Studies later analysed the different types of changes which can occur to the derivation trees as a result of integer mutation [84, 86]. Integer mutation events were decomposed into two different types, nodal and structural. Nodal events have the effect of changing terminal node values (i.e., nodes on the derivation tree which contain primitive symbols of the language). Structural events can change a derivation tree's size and structure due to their occurrence at nodes containing nonterminal symbols and the subsequent replacement of the underlying subtree of a different size. The impact of structural and nodal components of mutation when employing a simple binary choice grammar (Fig. 19.8) is illustrated in Figs. 19.9 and 19.10 for all possible genome instances of 2 and 3 codon lengths respectively. Here codons are simplified to be a single binary number (either

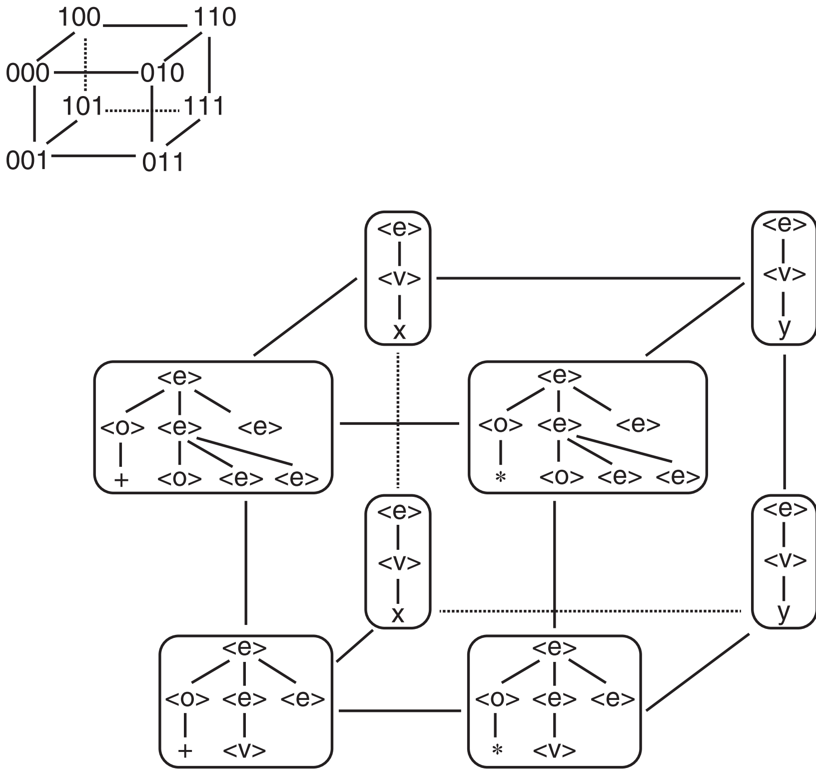


Fig. 19.10. An expansion of the illustration of nodal and structural mutation events on the derivation tree when considering three codon choices

0 or 1). These different types of mutation events which can arise were later exploited by designing separate mutation operators, a nodal mutation and a structural mutation, where it is was possible to more directly control the step-size of change which would occur. As a result, more predictable behaviours arose, which proved useful in improving the performance of an interactive architectural design environment [85, 88]. Adaptive forms of mutation operators have also been proposed [185].

19.4.3 Modularity

Mechanisms which allow modularity to develop in the evolving population have been shown to have performance gains when problems are nontrivial [341]. Researchers have explored different approaches to modularity in GE, ranging from static grammar-defined functions (a variation on ADFs) [469] to

dynamically defined variants using dynamic grammars (i.e., grammars which automatically update to incorporate new ADFs) [255] and metagrammar-defined ADFs [263]. More recently, Swafford et al. have examined a number of different approaches to identify and then incorporate subderivation trees as modules [608, 609, 610, 611, 612, 607]. In all of the above approaches, modularity is found to provide performance gains on problems which have sufficient 'difficulty' to warrant the overhead of increasing the search space by including mechanisms for modularity. McDermott has also investigated with success the use of higher-order functions when applied to evolutionary design [397].

19.4.4 Search Algorithm

The utility of alternative search engines, apart from variable length GA, has been examined in a number of studies. For example, particle swarm and differential evolution algorithms have been used to create grammatical swarm [463] and grammatical differential evolution algorithms [465], respectively. The grammatical swarm (GS) algorithm has shown particular promise and demonstrates a social learning approach to program generation referred to as *social programming.*

Grammatical Swarm

The grammatical swarm (GS) algorithm replaces the standard GA search engine in GE with a particle swarm algorithm. In GS each particle or real-valued vector represents choices of program construction rules specified as production rules of a Backus–Naur form grammar. The particle update equations are as described earlier for the continuous particle swarm algorithm (Sect. 8.2) with additional constraints placed on the velocity and dimension values. Velocities are bound to the range $[-V_{\max}, V_{\max}]$ with $V_{\max} = 255$, and each dimension is bound to the range $[0, 255]$. The real-valued dimension values are rounded up or down to the nearest integer, and the standard GE mapping function ($R = c \ \% \ r$, where R is the selected production rule, c is the codon integer value, and r is the number of production rules to choose from) is used.

Unlike its GE or GP counterparts, which predominantly use crossover-driven search coupled with selection, GS does not use explicit crossover or selection to generate programs. Instead the search process is driven by the movement of particles which are influenced by personal and social knowledge in the form of the positions of the g^{best} (or l^{best}) particle and the particle's own p^{best} position. The performance of GS has been compared to GE across a number of benchmark problems with encouraging results, suggesting that program generation using a social programming approach such as GS is a viable alternative to more traditional (GA-driven) genetic programming algorithms.

Grammatical Differential Evolution

Grammatical differential evolution (GDE) [465] adopts a differential evolution (see Chap. 6) learning algorithm coupled to a grammatical evolution (GE) genotype–phenotype mapping to generate programs in an arbitrary language.

The standard GE mapping function is adopted for the mapping process, with the real-valued vector components being rounded up or down to the nearest integer value. In the current implementation of GDE, fixed-length vectors are adopted within which it is possible for a variable number of elements to be required during the program construction genotype–phenotype mapping process. A vector's components may be used more than once if the wrapping operator is used, and in the opposite case it is possible that not all elements will be used during the mapping process if a complete program comprised only of terminal symbols is generated before reaching the end of the vector. In this latter case, the extra element values are simply ignored and considered introns that may be switched on in subsequent iterations. A diverse selection of benchmark programs from the literature on genetic programming are tackled using GDE to demonstrate proof of concept for the method.

19.5 Genotype–Phenotype Map

The genotype–phenotype map of GE has itself been the focus of research. One of the first variations studied an alternative mapping function to the straight modulo function when selecting production rules from the grammar. Dubbed the 'bucket rule' this results in every codon encoding a unique set of production rules, one for each nonterminal in the grammar [320]. On the problems examined, it was found that the bucket rule facilitated the exploitation of GE's property of intrinsic polymorphism (the same codon can have different meanings when applied to different nonterminal contexts). However, there was no clear performance gain on the problems examined when compared to the standard mod rule.

Other research has examined the importance of the mapping order, that is, the order in which nonterminals are selected for expansion in the growing derivation tree. The standard GE mapper adopts a depth-first expansion of the available nonterminals. A number of variations including breadth-first, random and evolved orders have been examined [182, 183, 184, 206]. An evolved order approach, named πGE, was particularly effective, resulting in performance gains over the standard depth-first mapping order [183, 477, 181, 180].

19.6 Grammars

Grammars play a central role in GE and it is not surprising that there has been considerable research in this area as a consequence. Given the flexibility of what can be represented with a grammar, GE has been applied to

a diverse array of problem environments and using many different grammars (e.g., context-free [468], attribute [479, 489], logic [319], shape [481], L-system [467], map L-system [485], graph [396], meta [462, 474], and tree-adjoining [426, 427, 424]).

The role of the grammar on the performance of GE has also been examined [261, 262, 442, 446], and a number of extensions and 'tricks' have been proposed including subtree-deactivation [478], grammar-defined functions (see Sect. 19.4.3), constants [147, 148, 149, 445], and introns [483]. Also, two special grammar symbols were introduced by Nicolau and Dempsey [445]. The first symbol `<GECodonValue>` is used to specify constant values within a predefined range. For example, to allow integer values between zero and five one would specify `<GECodonValue-0+5>`. The second symbol allows the explicit control of crossover sites. The `<GEXOMarker>` symbol is placed in the grammar by the user where it marks the positions where a crossover operation can take place at the genotypic level. When two genomes are selected for crossover, a crossover site corresponding to a `<GEXOMarker>` is chosen from each parent. The use of this approach allows the convenient implementation of two-point or even multipoint crossover operators, and even the evolution of crossover point locations.

19.7 Summary

This chapter presented an introduction to grammatical evolution, an influential form of grammar-based genetic programming, and one of the methodologies inspired by developmental systems and grammars. In Chap. 20 we outline a number of grammar-based GP algorithms which adopt the powerful tree-adjoining grammar representation, including a variant on GE called TAGE. Later, in Chap. 21, we will see how the TAGE approach has been adapted to a genetic regulatory network approach to GP. Tree-adjoining grammars are a particularly attractive proposition from a developmental GP perspective as they guarantee that at each stage of development, from the embryo to the full adult phenotype, the structure is complete and therefore executable.

20

Tree-Adjoining Grammars and Genetic Programming

A significant recent addition to the repertoire of grammar-based approaches to genetic programming (GP) is the use of tree-adjunct and tree-adjoining grammars (TAG). A TAG is a tree-generating grammar, and as such is a natural representation for GP and for computer programs. One of the primary benefits of a TAG is that the application of any rule within a TAG to an existing tree results in an executable tree, which overcomes the possibility of creating invalid programs (incomplete programs which cannot be executed). The creation of invalid phenotypes can occur in methods such as grammatical evolution without enforcing some sort of repair-like strategy during the genotype–phenotype mapping process.

It is this tree-generating (i.e., generative) property coupled with each transformation resulting in a valid phenotype that makes a TAG an especially interesting representation for developmental approaches to GP, as at each stage of development the emerging phenotype can be executed and therefore evaluated. In this chapter we illustrate how TAGs have been combined with G3P (Chap. 18) and GE (Chap. 19) to produce new approaches to GP, and also present a developmental approach to G3P based on TAG (DTAG3P).

Developmental (or at least evo–devo) approaches to GP are attractive at least partly because they potentially allow adaptation to occur faster than phylogenetic timescales typical of their purely evolutionary counterparts. In the ideal case, these algorithms work with genomes that encode a spectrum of phenotypic potential, the expressed phenotype being a function of the environmental state in which the genome finds itself at the outset of (and during) the development process. Another potential benefit from adopting developmental approaches to GP is due to the structural search space in which GP operates, in conjunction with the optimisation of the content (or parameters) of the structures. Navigating such a complex structural search space is a nontrivial problem, and the developmental processes which exist in nature are capable of producing impressive examples of complex, yet functional form. Developmental approaches to GP seek to tap into some of that potential to allow it to scale up to ever more challenging real-world problems.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_20

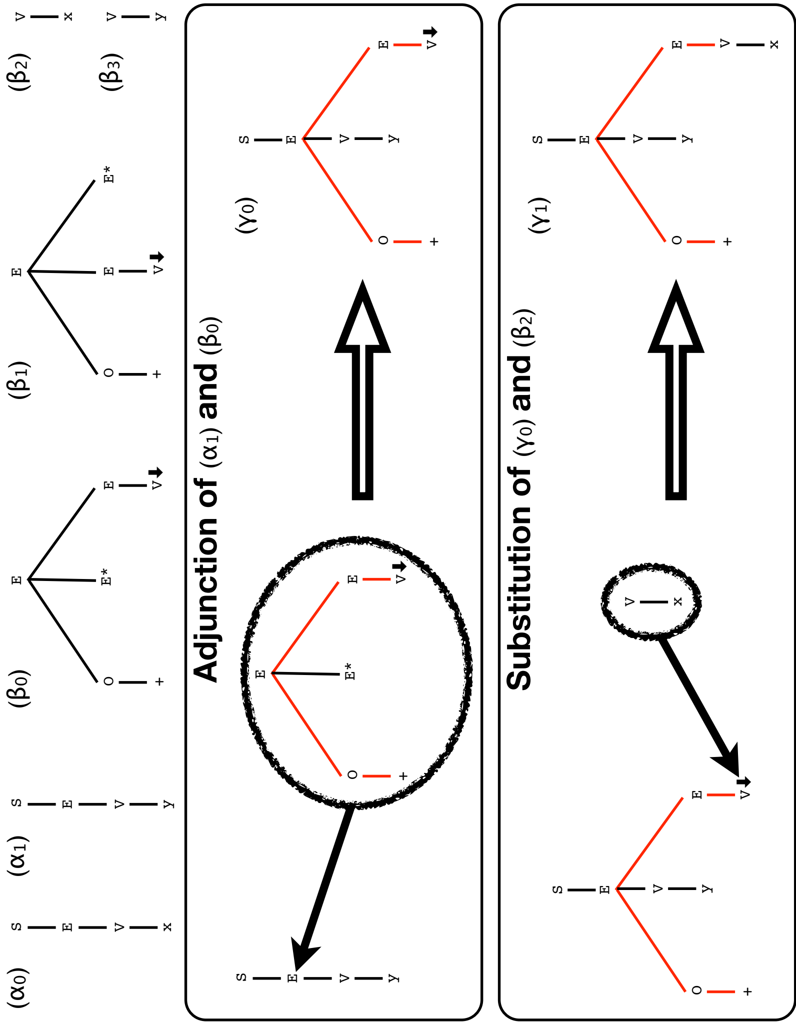


Fig. 20.1. The elementary trees of a simple TAG which describes a language which permits the addition of two variables, and examples of the composition operations of adjunction and substitution

20.1 Tree-Adjoining Grammars

Joshi et al. [309, 310, 311] introduced tree-adjunct and tree-adjoining grammars (TAG) which describe tree-generating systems. The set of languages produced by TAGs are a superset of those produced by context-free grammars (CFGs); in addition, TAGs are capable of representing some context-sensitive languages [311]. Context-free grammars (Sect. 17.3.2) are represented by a four-tuple $G = (\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$, comprised of a set $\mathcal{N}$ of nonterminal symbols of which one, $\mathcal{S}$, is the start symbol, a set $\mathcal{T}$ of terminal (or alphabet) symbols, and a set $\mathcal{P}$ of production rules that map the elements of $\mathcal{N}$ to $\mathcal{T}$. TAGs are similar in that they are also comprised of nonterminal, start and terminal symbols; however, as they are tree-rewriting systems, the equivalent of their production rules ($\mathcal{P}$) are called *elementary trees* which are comprised of two types, *initial* ($\mathcal{I}$) and *auxiliary* ($\mathcal{A}$, also referred to as β). TAGs are therefore described as a five-tuple $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{I}, \mathcal{A})$.

The defining behaviour of TAG centres around the two types of elementary trees, and the tree composition operators, *adjunction* and *substitution*, which manipulate them. All internal nodes in elementary trees are comprised of nonterminals, with leaf nodes either being a terminal or nonterminal. Auxiliary trees must contain a leaf node which has the same nonterminal symbol as the root of the auxiliary tree; this special leaf node is called the foot node and is critical to the operation of the TAG tree-rewriting system. All other leaf nodes in an auxiliary tree are labeled as substitution nodes.

The substitution operator of TAG replaces a nonterminal node on the frontier of an elementary tree with an initial tree rooted in the same nonterminal type. Adjunction works by generating a new derived tree γ through the composition of a β tree and an α tree (an α tree can be either an initial, auxiliary β or derived γ tree). Examples of adjunction and substitution are illustrated in Fig. 20.1 for a very simple grammar which permits the addition of the variables `x` and `y`.

20.2 TAG3P

A form of grammar-guided genetic programming (G3P: see Chap. 18) which exploits Tree-adjunct and Tree-adjoining grammars was proposed by Hoai et al. [274, 275, 276, 277]. TAG3P referred to the original tree-adjunct form [275], which was later improved upon by TAG3P+ [274] which adopts tree-adjoining grammars, and consequently includes the use of the substitution operator. The '+' label has since been dropped from the name and the tree-adjoining form is now also referred to as TAG3P. Conveniently, the CFG for a problem can be automatically converted to an equivalent TAG. Joshi et al. [311] have shown that for certain CFGs (those which are finitely ambiguous and which do not generate the empty string) there is an equivalent lexicalised TAG (LTAG) which describes the same language, and they also provide an algorithm to

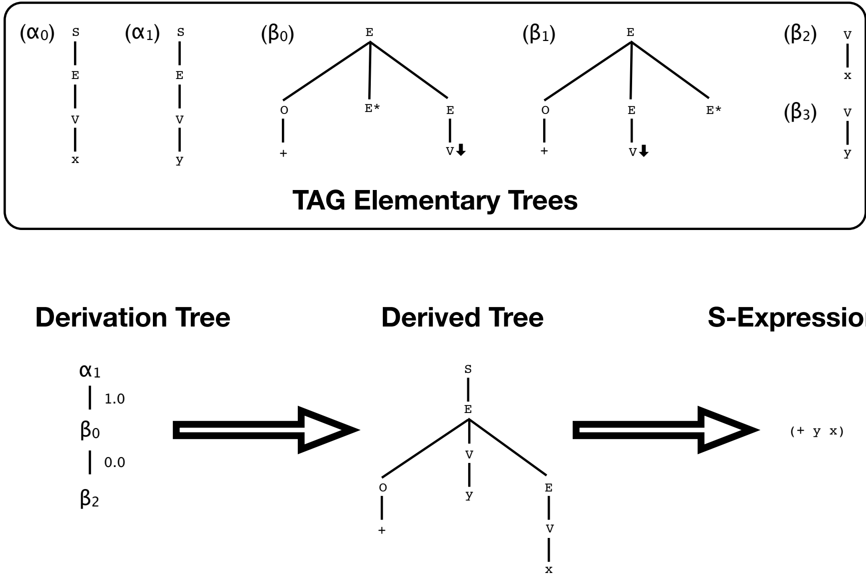


Fig. 20.2. Adopting the same TAG and the same derivation sequence as in Fig. 20.1 we can see the TAG3P derivation tree which results in a derived tree (labeled γ_1 in Fig. 20.1) and the corresponding S-expression which can be executed

convert from CFG to LTAG. An LTAG is a special type of TAG where each elementary tree must have at least one terminal/alphabet symbol.

When initialising the population, the depth of each derivation tree is determined randomly from some range and then, starting from the start symbol, elementary trees are randomly selected along with the index of adjunction sites. Figure 20.2 illustrates the derivation and derived trees which result from the application of composition operators to some of the elementary trees of an illustrative sample TAG. The standard derivation tree-based search operators of crossover and mutation as per G3P are employed. These genetic operators are designed to ensure that the syntactic integrity of the derivation trees are preserved after their application. As such, the standard grammar-guided GP algorithm outlined earlier in Algorithm 18.1 is employed by TAG3P. Later, duplication and truncation operators were added [277], which improved the performance of TAG3P.

20.3 Developmental TAG3P

TAG3P has been combined with L-systems to produce a developmental form DTAG3P [278]. A grammar is defined which specifies the construction of a D0TL-system. The L-system is then executed one iteration at a time. Each iteration of the L-system results in a TAG derivation tree which can subsequently be sent for evaluation (see Sect. 20.2 and Fig. 20.2 on how a TAG3P derivation tree is mapped for execution). This is one of the key properties of a developmental system which has been captured by DTAG3P, that is, each stage of development of the phenotype is open to evaluation. In DTAG3P, successive iterations of the L-system result in successively more complex versions of the phenotype, each iteration being evaluated on increasingly complex instances of the problem environment in a form of layered learning.

D0TL-systems are an extension of D0L-systems, which are a special instance of a deterministic L-system (DL-system) where only one production is permitted for each symbol from the L-system alphabet. The 'T' in D0TL-systems refers to the specialisation of the L-system to manipulate trees as the language primitives as opposed to the standard strings. In other words, the alphabet (primitive symbols) of a standard L-system are characters which result in sentences comprised of characters, whereas in a TL-system elementary trees of a TAG make up the alphabet. The execution of a D0TL-system results, therefore, in the development of tree structures.

TAG3P starts by prespecifying a finite number of production rules which will make up the D0TL-system. Each rule of the L-system is then evolved according to a D0TL-system grammar, and it must always contain at least one initial (I) elementary tree as the axiom from which development can commence. As the right-hand side of the production rules of the L-system may not contain the left-hand-side symbol of some of the productions, it is possible that some production rules of the L-system are never used during development.

20.4 TAGE

The G3P algorithms which adopt TAGs generate an initial population of programs in the form of TAG derivation trees (see Fig. 20.2). The derivation tree is the genotype of TAG3P. Recall that, in TAG3P, we first select the size of an individual's derivation tree from some range and then either randomly grow the tree or use a strategy like ramped-half-and-half.

An alternative grammar-based Genetic Programming approach to adopting Tree-adjoining Grammars is proposed with Grammatical Evolution (TAGE) [427, 425]. In TAGE there is a linear genome (as per GE) which encodes the construction of the TAG derivation tree. It is the genome (rules for constructing the TAG derivation tree) which is exposed to the genetic search operators of, for example, crossover and mutation.

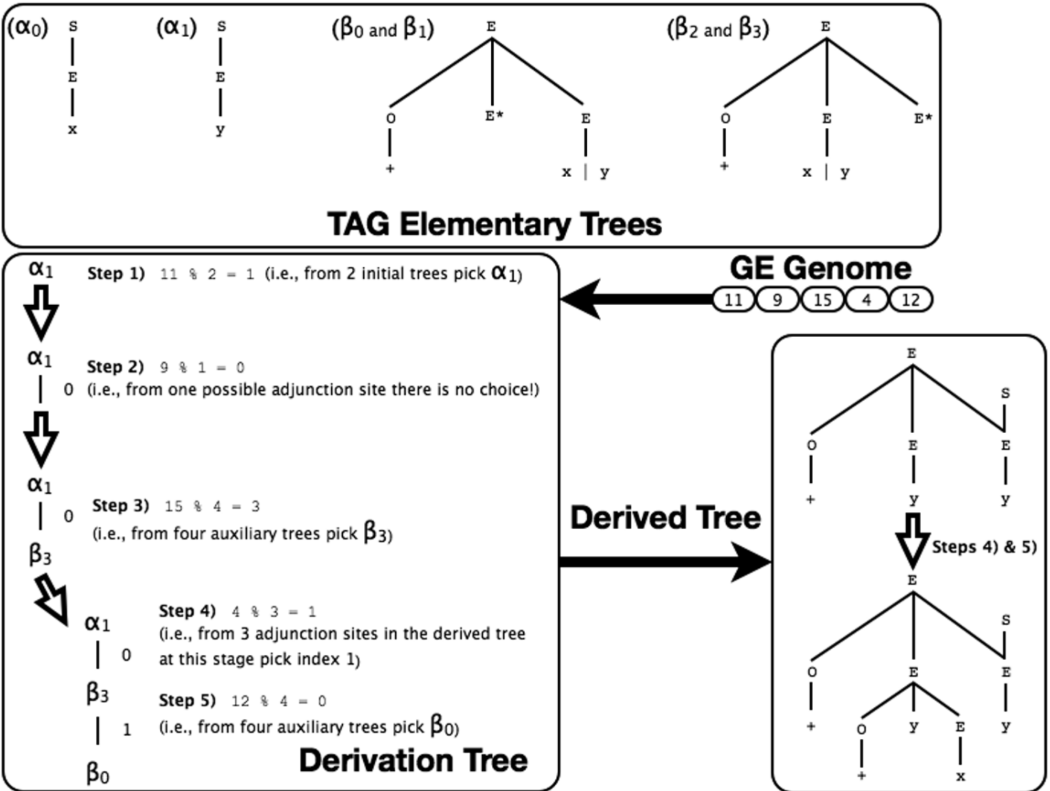


Fig. 20.3. An outline of the TAGE algorithm mapping process from linear genome to a derivation and derived tree. The derived tree is equivalent to the S-expression `(+ (+ y x) y)`. Note the shorthand used, for example, in the TAG elementary trees β_0 and β_1, where they are represented as a single tree with two choices for the right-most `E` (either `x` or `y`)

TAGE is similar in operation to the πGE algorithm (a variant of GE, Sect. 19.5) in that during the mapping process which creates the derivation tree, first a site of application of the next grammar rule is selected and then an appropriate rule (tree adjunction in TAGE, string replacement in πGE) is selected. In each case the site of application and the rule which is applied is determined by the genome. A list of all the possible adjunction sites in the growing phenotype is maintained, and when selecting a site from this list we read the next gene value and 'mod' it by the number of adjunction sites available. Figure 20.3 illustrates the TAGE mapping process. TAGE has been shown to significantly outperform standard grammatical evolution on a number of benchmark problems [427].

In Chap. 21 we will outline a developmental approach to GP (differential gene expression) which combines TAGE with genetic regulatory networks.

20.5 Summary

This chapter presented three forms of grammar-based Genetic Programming which employ tree-adjoining grammars (TAGs), namely TAG3P, DTAG3P and TAGE. The TAG formalism is particularly attractive for two main reasons. Firstly, it is a more powerful representation than context-free grammars as it is possible to represent some context-sensitive languages with TAGs. Secondly, as a TAG describes a tree-generating system at each stage of the development of the phenotype (solution), the intermediary phenotypes are executable entities in their own right. This allows TAG forms of grammar-based GP to adopt a developmental approach more closely related to their biological counterpart. In turn this potentially allows faster rates of adaptation to occur over a developmental (rather than solely phylogenetic) timescale, and also more effective structural search algorithms allowing these GP methods to be scaled up to ever more challenging real-world problems.

21

Genetic Regulatory Networks

It has been observed that much of the diversity in the natural world can be traced to three features of developmental biology, namely, interactions between gene products, the temporal nature of gene expression, and shifts in the location of gene expression [32]. The first item highlights the significance of feedback loops in developmental processes. In terms of developmental approaches adopted in Natural Computing, it is those algorithms which implement genetic regulatory networks (GRN) that capture the first two of these features (i.e., feedback loops regulating gene expression, and the temporal nature of gene expression). Much of the literature to date in the area of GRN algorithms represents attempts to examine GRN variants and tries to understand how they might operate [33, 249, 294, 364]. In this chapter we describe GRNs and outline examples of some of their practical uses including their potential as an approach for genetic programming [426, 449] and for image compression [629].

21.1 Artificial Gene Regulatory Model for Genetic Programming

Adapting an earlier genetic regulatory network (GRN) model [32, 33], Nicolau et al. [449] enable the model to include explicit input and output signals in the form of proteins. The inputs correspond to the problem variables and the output genes correspond to the model response to the input. The input proteins are allowed to interact with the GRN model as they are incorporated into the calculation of the regulation of each gene's expression. Specific genes are then identified as potentially producing the output signal/protein. The GRN model is allowed a period of time to 'react' to each set of input values before measuring the output signal which is then fed into the problem simulator/fitness calculation. Specifically, the algorithm is studied in the context of the pole balancing problem [35, 657]. The output signal is used to determine whether or not a force is applied to move the pole to the left-hand or to

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_21

the right-hand side. The inputs to the model are the four problem variables, namely, the position of the cart, the angle of the pole, the velocity of the cart, and the angular velocity of the pole.

Let us examine the model in more detail to understand the encoding and its operation. Each individual is comprised of a binary genome comprised of multiple genes. Each gene is made up of four regions: an enhancer site (32 bits), an inhibitor site (32 bits), a promoter site (32 bits), and finally the exon or expressed region (160 bits, i.e., five sets of 32 bits). A protein is produced from a gene by calculating each of the 32 bits which make up a protein using a majority vote for each bit position from the five sets of 32 bits in the exon.

The expression of a gene is regulated using the bit signatures of the enhancer and inhibitor sites and the concentration of each protein which currently exists in the model. The enhance (e_i) and inhibit (h_i) signals of gene i are calculated using (21.1) and (21.2):

$$e_i = \frac{1}{N}\sum_{j=1}^{N} c_j \exp(\beta(u_j - u_{\max})) \tag{21.1}$$

$$h_i = \frac{1}{N}\sum_{j=1}^{N} c_j \exp(\beta(u_j - u_{\max})) \tag{21.2}$$

where c_j is the concentration of protein j, u_j is the number of complementary bits between the protein j and either the enhancer or inhibitor site of gene i, and $u_{\max}$ is the maximum observed number of complementary bits between proteins and regulatory sites. N is the total number of proteins in the model, and β is a scaling factor which controls the significance of the protein/regulatory site match difference.

The expression of a protein at any point in time (p_i) is then calculated according to (21.3):

$$\frac{\mathrm{d}c_i}{\mathrm{d}t} = \delta(e_i - h_i)c_i - \phi(1.0) \tag{21.3}$$

where $\phi(1.0)$ is a term that scales protein production such that the sum of all protein concentrations is equal to 1.0, andδ is a scaling factor.

In addition to the proteins which are encoded on the genome, additional input proteins are added to the model. These extra proteins represent the problem variables, and they can also impact the expression of genes by extending (21.3) to become:

$$\frac{\mathrm{d}c_i}{\mathrm{d}t} = \delta(e_i - h_i)c_i - \phi\left(1.0 - \sum_{j=N+1}^{N_{ep}} c_j\right) \tag{21.4}$$

where c_j is the concentration of the extra input protein j, and $N+1,\ldots,N_{ep}$ are the indices of these input proteins.

Algorithm 21.1: Artificial Genetic Regulatory Model for GP

```
Initialise population;
for each generation do
    for each individual do
        Execute GRN model until it reaches a steady state;
        for each input do
            Set values of input gene(s);
            Execute GRN model for time period t;
            Read values of output gene(s);
            Update fitness;
        end
    end
    Select parents;
    Apply genetic operators;
    Carry out replacement;
end
```

21.1.1 Model Output

In this model (see overview in Fig. 21.1), each gene encodes one of two types of protein, either (i) a transcription factor or (ii) a product. Transcription factor proteins are used to regulate the expression of other proteins. The output signal(s) of the model are determined by the product proteins. The state of the product proteins, which determines the output signal, is itself a function of the expressed transcription factors and the problem-specific input proteins. In the Nicolau approach the GRN model is allowed to *stabilise to a steady state* before any attempt is made to read an output signal. How these product proteins are interpreted to produce the model output is an open problem. Nicolau et al. examined two approaches where either the relative concentration of a particular product protein was used or the *tendency* of a protein concentration (amounting to the derivative of protein concentration). In terms of the pole-balancing problem, the first approach interpreted the relative concentration of a product protein which was above or below the value of 0.5 to be a push right or push left signal. In the tendency approach, if the derivative is positive the cart is pushed right, otherwise it is pushed left.

Another issue faced when using an artificial gene regulatory model is the synchronisation of the regulatory model to the problem domain. Signals (input proteins) from the problem environment need to be absorbed quickly enough by the GRN model to produce an output signal in time, so the speed of response of the GRN model must be evolved to match the environment.

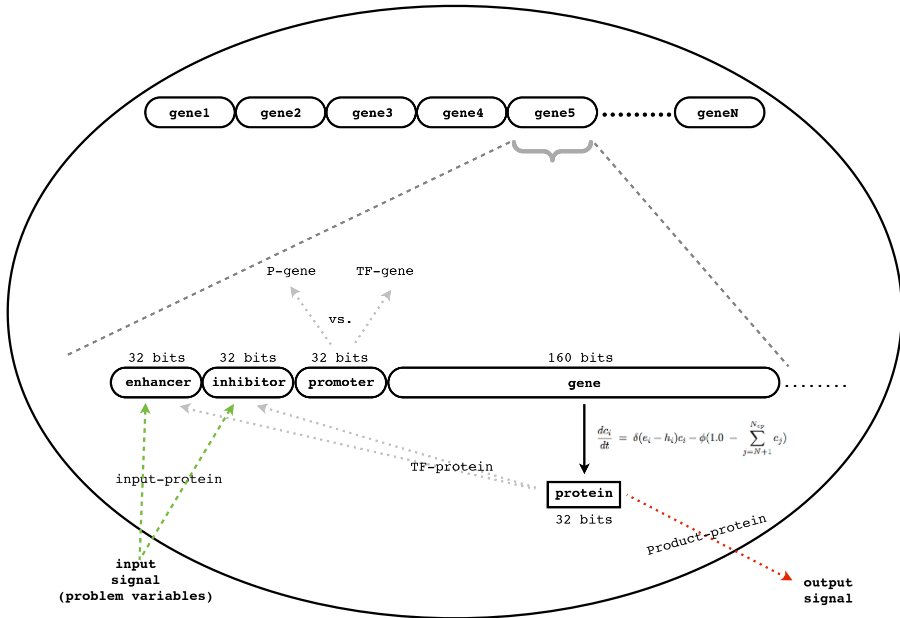


Fig. 21.1. An overview of the genetic regulatory network model as presented by Nicolau et al. [449]. In the bottom part of the figure we zoom in to reveal the detailed structure of each gene. The bit signature of the `promoter` region determines the gene type (i.e., product or transcription factor (TF) in this case). At any point in time the degree of enhancement/inhibition of any gene is a function of the number and strength of matches of `input-proteins` and `TF-proteins` to the `enhancer` and `inhibitor` bit sequences. A gene which is 'switched on' produces a protein, and the type of the protein is determined by the signature in the promoter region. `Product-proteins` can be used to provide output signals for the model, and `TF-proteins` regulate the expression of P-genes and TF-genes. `Input-proteins` correspond to variables of the problem domain

21.2 Differential Gene Expression

Building upon the earlier work which developed a computational model of genetic regulatory networks and their subsequent application to genetic programming, Murphy et al. [426] adapted the tree-adjunct grammar approach to grammatical evolution (TAGE) and coupled it to a form of genetic regulatory network which adopts *differential gene expression*. Differential gene expression is an important feature of biological development because from the same genotypic repertoire it is possible to express many different phenotypic states. When applied to genetic programming this allows the same genome

to encode different phenotypic programs, allowing temporal adaptation of the expressed phenotype to a changing environment.

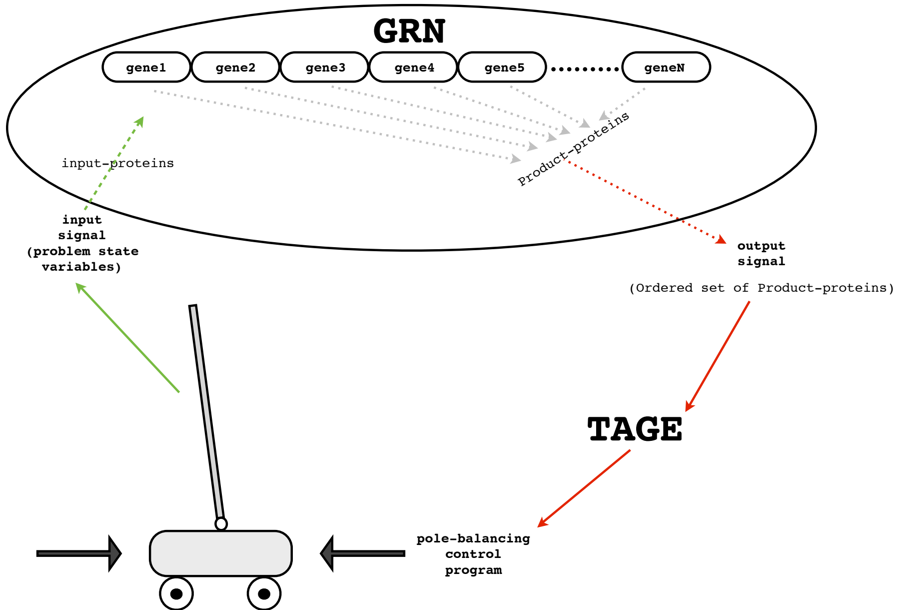


Fig. 21.2. An illustration of the differential gene expression approach combining GRNs and grammars (TAGE) [426]. The GRN produces `product-proteins` which at each problem time-step are sampled, and a relative ordering of their states determined. The ordered product-proteins states are then interpreted by the TAGE mapping process to select rules from a tree-adjoining grammar resulting in the construction of a pole-balancing control program. At each time step of the problem environment, the state variables are passed back to the GRN model which is allowed to adjust to the new environmental state before the product-protein states are reinterpreted

We outline this approach to developmental and grammatical genetic programming in Fig. 21.2, focusing in particular on the mechanism for differential gene expression. The 'trick' to effectively combine TAGE with GRNs is the use of the product-proteins expressed by the GRN as mapping inputs to TAGE. As in the earlier study by Nicolau et al. [449] it is not immediately obvious how best to interpret the product-proteins as inputs to the TAGE mapping process, and a number of alternative strategies have been examined. Product-proteins are sorted based on either their concentration or concentration tendency, and

the protein ordering determines the order in which the state of the protein is used to select rules from the TAGE grammar to generate a phenotypic GP expression tree.

When applied to the pole-balancing problem, the environment variables are passed to the GRN at each time step and the GRN is allowed to stabilise for a finite number of steps, at which point a product-protein ordering is determined and a new GP expression tree is generated to determine the force applied to the pole.

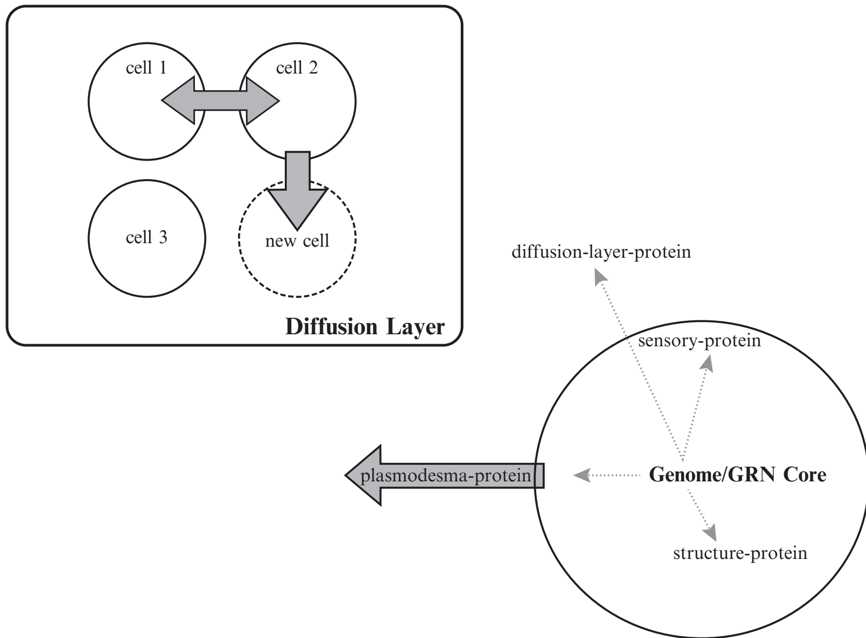


Fig. 21.3. An overview of the multicellular genetic regulatory network model as presented by Trefzer et al. [449]. In the bottom right-hand corner we zoom in to a cell and see the various proteins which can be expressed (structure, sensory, plasmodesma and diffusion). Expression of an (East-facing) plasmodesma protein in Cell 1 and the corresponding expression of a (West-facing) plasmodesma protein in Cell 2 results in a protein tunnel allowing transfer of proteins between cells. When a (South-facing) plasmodesma protein is expressed in Cell 2 the previously empty space will be used to grow a new cell (a copy of Cell 2). Proteins emitted into the diffusion layer can be ‘sensed’ by any cell which has expressed the correct sensory protein

21.3 Artificial GRN for Image Compression

It is noteworthy that all of the GRN models presented in this chapter to this point are unicellular in their instantiation, and it will be interesting to see how these methods develop as they expand to multicellular models. Multicellularity allows parallel computing and the specialisation of functionality. This specialisation would correspond to a GRN model's ability to decompose a problem, and will be a critical step in the scalability of these algorithms to harder real-world problems [155, 187, 216].

Trefzer et al. [629] have proposed a multicellular form of artificial gene regulatory networks and applied it to the image compression problem (Fig. 21.3). A simplified model of regulation is adopted where the binding sites which activate the expression of a gene match only a single unique protein. Cell communication can occur via diffusion of protein signals emitted from a cell into a diffusion layer between the cells, and through a direct tunneling mechanism between cells, which is inspired by plasmodesmata protein tunnels in plants. If a plasmodesma tunnel is opened to an 'empty/dead' cell, cell growth occurs by replicating the contents of the plasmodesma's origin cell. Each cell contains a GRN, and each cell represents a single pixel of the output image, where the pixel is a coefficient for a frequency component of the image. The resulting solutions can provide better compression rates than JPEG for the images examined.

21.4 Summary

We have presented computational tools inspired by biological genetic regulatory networks and illustrated how they have been applied to problems such as image compression and genetic programming. As our knowledge of developmental biology deepens we would expect that there will be many extensions on the approaches presented here.

Part VI

Physical Computing

22

An Introduction to Physically Inspired Computing

Physical systems and processes, just like biological ones, can inspire the design of computer algorithms. In this chapter we provide a short introduction to a range of physical phenomena, and in Chaps. 23 and 24 we outline a range of algorithms which draw inspiration from aspects of these phenomena. The nature of this material is necessarily somewhat technical, so readers who are looking for a high-level overview of physically inspired algorithms may wish to initially skip to Chap. 23, and then return to this chapter for a more detailed discussion of the physical principles which underlie the algorithms.

22.1 A Brief Physics Primer

This section gives a quick introduction to some major areas of physics and outlines how they fit together. Subsequent sections in the chapter look in a little more detail at classical mechanics, thermodynamics, quantum mechanics, and models of annealing. Of course, it is not possible to provide a highly detailed discussion of all of these topics in a single chapter, so we emphasise the aspects of each area which are most important in helping to understand the range of physically inspired algorithms which have been developed thus far.

22.1.1 A Rough Taxonomy of Modern Physics

Figure 22.1 gives a rough taxonomy of modern physics, with continuous arrows showing which areas arise from others by relaxation of assumptions, or introduction of new principles to existing theories. The diagram also shows the sources of inspiration for the best-known physically inspired algorithms.

Classical Mechanics: As developed by Galileo and Newton, particles/bodies are acted on by (vector) *forces* which are thought of as instantaneously acting at a distance. Equivalent formulations in terms of the scalar *energy* were developed by Euler, Laplace, Lagrange, Hamilton and others.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_22

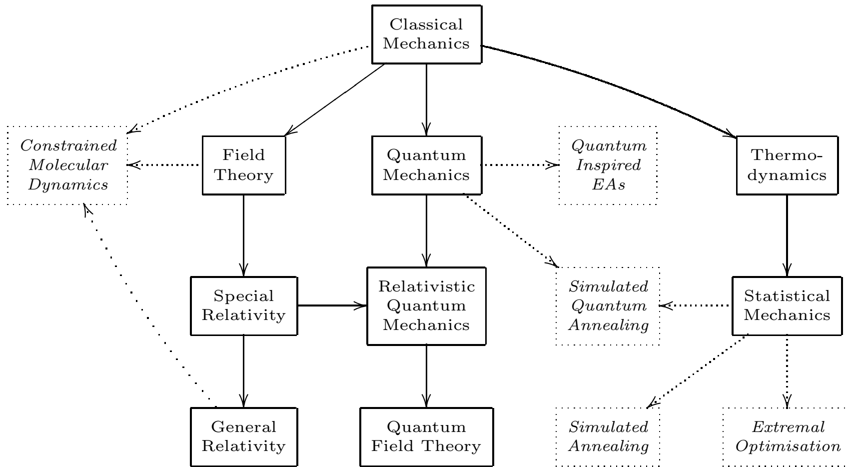


Fig. 22.1. Connections among areas of physics and algorithms

Field Theory: Due to Maxwell and others. An object may generate a (scalar) potential field V which gives rise to a vector field ∇V, e.g., gravitation and electromagnetism (the 'classical' fields). Other bodies are then influenced by this field. Maxwell showed that a field can carry energy through space in the form of waves: this leads to a finite speed of signal propagation, removing instantaneous 'action at a distance' of forces. In particular, Maxwell's electromagnetism equations show that light has a constant finite speed in vacuo.
Special Theory of Relativity: Developed by Einstein (and, independently, by Poincaré and Lorentz), derived by adding the axiom of equivalence of inertial reference frames to the constant speed of light derived from Maxwell's equations of electromagnetism.
General Theory of Relativity: Also due to Einstein, derived by adding to the Special Theory the *principle of equivalence* between a uniform gravitational field and an acceleration (i.e., the equivalence of inertial and gravitational mass).
Quantum Mechanics: Due to Planck, Bohr, Schrödinger, Heisenberg, Pauli, Einstein, and others. *Quantisation* means the replacement of physical quantities (or *observables*) such as position, momentum, or energy by operators on a Hilbert (vector) space of *wavefunctions.*
Relativistic Quantum Mechanics: Dirac reformulated Schrödinger's equation for the electron in a relativistically invariant form such that the wavevectors have extra degrees of freedom, which turn out to exactly model the spin of the electron: hence the wavevectors are called spinors in this setting. It was from this equation that Dirac predicted the existence of antimatter (specifically, the positron).

Quantum Field Theory (QFT): Due to Feynman and others, inspired by Dirac's work. Here, the fields themselves, including the wavefunction, are quantised (called second quantisation): thus the force carriers are particles. The first example was Feynman's Quantum Electrodynamics (QED), derived using a form of least action principle for path integrals.

Thermodynamics: Due to Carnot, Boltzmann and others. Deals with macroscopic measurable properties, e.g., pressure and temperature, of a system composed of microscopic components (e.g., a gas of molecules), and the ideas of thermodynamic equilibrium, thermodynamic processes and entropy. It explains why physical systems seek the lowest state of free energy, for example, in annealing of metals.

Statistical Mechanics: A modern approach to thermodynamics, explaining macroscopic measurable quantities in terms of statistics of distinguishable particles. The statistical physics Ising model of spin lattices is used to describe frustrated spin glass states of magnetic materials and inspires the Simulated Annealing and Simulated Quantum Annealing algorithms.

In the following sections, we explain some terms from these areas of physics that arise in the physically inspired algorithms we consider.

22.2 Classical Mechanics

In Chap. 23 we will describe the simulated annealing, simulated quantum annealing and constrained molecular dynamics algorithms. For all of these algorithms, 'energy' is a core concept. Hence, we now introduce the ideas of energy and the Hamiltonian. The construction of an artificial Hamiltonian to suit a particular problem is often at the heart of the design of a physically inspired algorithm. The Hamiltonian will also prove useful in Chap. 24 on quantum inspired evolutionary algorithms, since quantum mechanics is derived from classical mechanics by constructing a quantum Hamiltonian in terms of operators, and the Schrödinger Equation describes the evolution of a quantum wavefunction in terms of this quantum Hamiltonian.

22.2.1 Energy and Momentum

Energy is the ability to perform work, e.g., move an object.[1] The energy of a system may be regarded as being made up of two parts: the *kinetic* energy K, which is due to motion; and the *potential* energy, or *energy of interaction*, V, which is stored in the forces between objects. Strictly speaking, potential energy is the energy of position, i.e., the energy an object has because of its position $x = (x_1, x_2, x_3)$ relative to other objects in space. Energy is a scalar nonnegative quantity. The total of kinetic and potential energy is a *conserved*

[1] An object of negligible size is called a *particle*; otherwise, it is called a *(rigid) body.*

or *invariant* quantity: it remains the same even if the internal configuration of the system changes (assuming no outside influences).

The *velocity* (*vector*) of an object is $v := \dot{x} := \frac{\mathrm{d}}{\mathrm{d}t}x$, the time derivative[2] of the position vector x.

A particle of mass m and velocity v has *momentum* $p := mv$. Momentum is a vector quantity, since velocity is a vector. The *total momentum* of a system is the vector sum of the individual momenta in the system: it is also a conserved quantity in that it cannot change in the absence of influences from outside the system (Newton's First Law). There are other conserved quantities in classical mechanics, such as angular momentum of rigid bodies.

In Newtonian terms, a *force* is any influence which can accelerate (change the velocity of) an object. Newton's second law, $F = ma$, relates the force vector F applied to an object, the object's mass m and the resulting acceleration $a := \dot{v}$. Since a force changes the velocity of a particle or body, it changes its kinetic energy, and so does work. A *conservative* force is one for which the amount of work (e.g., in moving a particle) done is independent of the path taken. Such processes are *reversible*; the same cannot be said of nonconservative forces. Examples of conservative forces are gravity and electromagnetism.

Given an arrangement of particles in space, a conservative force can be represented by a (scalar) potential field $V = V(x_1, x_2, x_3)$ (i.e., a number defined at each point in space) whose gradient $\nabla V := \left(\frac{\partial}{\partial x_1}V, \frac{\partial}{\partial x_2}V, \frac{\partial}{\partial x_3}V\right)$ is a vector field giving the force between particles; if there is no other force opposing this, the particles will move in response to this force, reducing the potential energy and converting the excess potential energy to kinetic energy, but keeping the sum $K + V$ constant. For example, if an apple is held at a height above the surface of the earth and then released, gravitational forces will pull the earth and apple together, converting the potential energy to kinetic energy (manifested as heat energy of the constituent atoms of the earth and apple).

22.2.2 The Hamiltonian

We now introduce the *Hamiltonian* or total energy of a system. It is important in optimisation algorithms such as Simulated Annealing and Simulated Quantum Annealing, where it (or, rather, an analogue of it) is constructed from a potential energy term, namely the objective function, and a kinetic energy term, representing some modification to the objective function. In both Simulated and Simulated Quantum Annealing, the kinetic term is gradually reduced to zero as the algorithm progresses (respectively by reducing temperature or the strength of a quantum field), so that by the end of the algorithm's run, the total energy which the algorithm has minimised is in fact equal to the objective function.

[2]Note that differentiation with respect to time t is commonly denoted by a dot over the variable, a notation introduced by Newton.

As noted above, Newton's dynamical laws are vector laws of the form $F = ma$, relating the acceleration vector a of an object of mass m to the force vector F acting on it. These laws were reformulated by Euler, Lagrange, Hamilton and others in terms of the scalar, energy, leading to unifying views of mechanics. The most important of these views for us is the *Hamiltonian* approach, closely related to the *Lagrangian* approach.

In both of these views, given a system of particles and bodies, we consider the *configuration space* $\mathcal{S}$ of the system: each point of $\mathcal{S}$ represents an arrangement in space of all the particles and bodies, that is, all the locations of all parts of the system. $\mathcal{S}$ will have some high dimension N. For example, for n particles, $\mathcal{S}$ will have $N = 3n$ dimensions since it captures the three position coordinates of each of the n particles. As time goes on, the single point q of $\mathcal{S}$ which represents the system will move around in $\mathcal{S}$ according to some law which completely describes the classical behaviour of the system.

This law can be obtained from a single scalar function. In the Lagrangian view, the function is also called the *Lagrangian* $\mathcal{L}$ and is defined on the $2N$-dimensional tangent bundle $T(\mathcal{S})$ of $\mathcal{S}$. The Lagrangian is the kinetic energy minus the potential energy. Though less relevant to us, the Lagrangian is important for the following reason. As our system evolves, the point q representing it in $\mathcal{S}$ will move along some curve C; the (*Hamilton*) *principle of least action* says that the motion of q through $\mathcal{S}$ will be such that the *action* (the integral of $\mathcal{L}$ along C) will be a minimum. Thus, if we regard the action as a measure of curve length, C is a geodesic.

In the Hamiltonian view, the function is called the *Hamiltonian* $\mathcal{H}$ and is defined on the *phase space* $\mathcal{P}$, which is another name for the $2N$-dimensional cotangent bundle[3] $T^*(\mathcal{S})$ of $\mathcal{S}$. The Hamiltonian is the sum of kinetic and potential energy, that is, the total energy of the system, expressed not in terms of position and velocity but rather in terms of position and momentum. Let the coordinates of a point x in $\mathcal{S}$ be $x_1, \ldots, x_N$, sometimes called *generalised position coordinates*. Let $p_1, \ldots, p_N$ be the corresponding *generalised momentum coordinates*. Then the point P in the phase space $\mathcal{P}$ corresponding to $x \in \mathcal{S}$ has coordinates $(x, p) = (x_1, \ldots, x_N, p_1, \ldots, p_N)$ and the value of the Hamiltonian at this point is $\mathcal{H}(x, p) = \mathcal{H}(x_1, \ldots, x_N, p_1, \ldots, p_N)$.

The Hamiltonian $\mathcal{H}$ gives rise to *Hamilton's dynamical equations*,

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial \mathcal{H}}{\partial x_i}, \qquad \frac{\mathrm{d}x_i}{\mathrm{d}t} = \frac{\partial \mathcal{H}}{\partial p_i}, \qquad i = 1, \ldots, N \tag{22.1}$$

which describe the time evolution of our original system as the trajectory in the phase space of the point P.

The phase space $\mathcal{P}$ has the important property that it is a symplectic manifold, equipped with a *Poisson bracket*, $\{\ ,\ \}$, which takes two scalar fields

[3] The cotangent bundle (and tangent bundle) are in general *manifolds*. They are not linear spaces, nor do they even possess a single coordinate system for the whole space.

A and B defined on $\mathcal{P}$ and returns a third such scalar field, $\{A, B\}$. The time evolution of any scalar physical quantity $A = A(x, p)$ is given by its Poisson bracket with the Hamiltonian: $\dot{A} = \{A, \mathcal{H}\}$. It can be shown that $\{\mathcal{H}, \mathcal{H}\} = 0$, the principle of conservation of energy in another guise. The Poisson bracket has very useful properties, and it is this symplectic or *canonical* structure arising from the Poisson bracket which is generalised in the construction of Quantum Mechanics.

To give intuition on this point, consider the case of a single particle of mass m, moving in a field given by a potential $V = V(x_1, x_2, x_3, t)$ depending on position and time. V defines the potential energy of the particle due to the field. The kinetic energy of a particle of mass m and velocity v is $\frac{1}{2}m\|v\|^2 = \frac{1}{2m}\|p\|^2$ which in this case gives $\frac{1}{2}m(\dot{x_1}^2 + \dot{x_2}^2 + \dot{x_3}^2) = \frac{1}{2m}(p_1^2 + p_2^2 + p_3^2)$. Thus

$$\mathcal{H} = \frac{p_1^2 + p_2^2 + p_3^2}{2m} + V. \tag{22.2}$$

For example, if the particle is in the earth's gravitational field, with position coordinates x_1, x_2, x_3 where x_3 is its distance from the earth, we can take $V = mgx_3$ where $g \approx 9.8$ ms^{-2} is the acceleration due to gravity. (22.2) gives

$$\mathcal{H} = \frac{p_1^2 + p_2^2 + p_3^2}{2m} + mgx_3. \tag{22.3}$$

22.3 Thermodynamics

(Simulated) Annealing (introduced in Sect. 23.1) may be thought of as minimising, through a slow cooling, the (*Helmholtz*) *free energy* of a system, in the sense of the thermodynamic principle that *every system seeks to achieve a minimum of free energy.* It may also be thought of as entropy maximisation, in that the system is relaxed to thermal equilibrium. We now explain these basic terms.

A thermodynamic system is taken to consist of a large number of distinguishable particles (typically of the order of Avogadro's number, 6.022×10^{23}). In a gas or liquid, the particles are free to move; in a solid they are more or less fixed but can vibrate about their mean positions. Thermal energy generally has two components, the kinetic energy of random motion of particles and the potential energy of their mutual positions. Each particle has several degrees of freedom in which energy can reside, translational degrees (which generally dominate — these are the velocity components in each of the three coordinate directions), along with any rotation or vibration the particle may have.

As before, we consider the system at a given time as being in a state (often called a *microstate*) in phase space $\mathcal{P}$. We 'roughly' divide $\mathcal{P}$ into subsets called *macrostates.* The meaning of two states x and y lying in the same macrostate is that macroscopically we cannot distinguish between them: the temperature, pressure, etc., of x and y may be identical to the best precision

of our measuring instruments. This division into macrostates is somewhat arbitrary but is adequate for our purposes, including the definition of entropy.

The *entropy* S of a system microstate $x \in \mathcal{P}$ is $k_B \ln \Omega$, where k_B is Boltzmann's constant and Ω is the volume in $\mathcal{P}$ of the macrostate containing x. The definition of entropy is relatively robust because of the log function, and the fact that the macrostates generally have very different volumes.

Thermal equilibrium (or *thermodynamic equilibrium*) is the overwhelmingly most likely macrostate (of volume almost equal to that of $\mathcal{P}$ itself) and thus the state of maximum entropy. At thermal equilibrium, the principle of *equipartition of energy*[4] holds, i.e., the energy of the system is (statistically) dispersed equally among all the degrees of freedom of the system.

The *temperature* T of the system is a measure of the (average) energy per degree of freedom. If the system is not in thermal equilibrium, then the temperature is different at different locations in the system (more energy at some locations), giving a temperature potential and resulting temperature gradient vector field. This causes a flow of heat energy in the system and the ability to perform useful work. Since only temperature *differences* (gradients) can provide useful work, a system in thermal equilibrium cannot provide any useful work, regardless of the actual quantity of (heat) energy in the system. A disturbance to a system not in thermal equilibrium is likely to disperse energy more uniformly among degrees of freedom and so increase entropy, bringing the system closer to thermal equilibrium.

Because random allocation of units of energy among degrees of freedom is combinatorially much more likely to result in an *even* distribution of energy among degrees of freedom, we see that large-volume macrostates correspond to more evenly spread energy, while smaller-volume macrostates correspond to unevenly-distributed (very peaked) distributions of energy among degrees of freedom. Thus, low entropy means high energy (temperature) differences in the system, high temperature gradient, and high useful work; while high entropy means low energy differences in the system and low useful work. Hence, entropy may be viewed as a measure of the *unavailability* of a system's energy to do work. This leads to the interpretation of low entropy as availability of 'high-quality' energy and vice versa.

The *Helmholtz free energy*[5] A is a thermodynamic potential which measures the 'useful' work obtainable from a closed system at a constant temperature. $A = U - TS$ where U is the *internal* or total energy of the system (i.e., the Hamiltonian). It attains a minimum at thermal equilibrium, that is, when

[4]The Statistical Mechanics *Theorem of Equipartition of Energy*, which can be derived from Newton's Laws, states that *the total energy of a system of a large number of particles, exchanging energy among themselves through collisions, is, on average, shared equally among all the particles' degrees of freedom.*

[5]The *Gibbs Free Energy* $G = U - TS + PV$ is the same concept, with the added requirement that the system's volume remain unchanged (here, P is pressure, while V is the system's volume). As we rarely have an equivalent of volume in our annealing or indeed chemical reaction algorithms, it is less relevant to us.

entropy S is maximised. Hence, at nonzero temperatures, free energy is the quantity minimised in natural processes such as annealing: this is discussed in more detail below.

The *First Law of Thermodynamics* is just the Principle of Conservation of Energy, applied to heat energy.

The *Second Law of Thermodynamics* states that, statistically, entropy increases over time; i.e., the system's energy, which may originally have been concentrated in relatively few degrees of freedom, tends to spread itself among all degrees of freedom.[6] Thus the system tends towards thermal equilibrium: differences in temperature, pressure, and density all tend to even out; and entropy measures how far this evening out process has progressed.[7] This law explains the irreversible nature of physical processes involving large numbers of components. From a Statistical Mechanics viewpoint, this law arises because, combinatorially, even distributions of energy across degrees of freedom are much more likely (i.e., bigger phase space volumes) than those having energy concentrated in relatively few degrees of freedom.

22.3.1 Statistical Mechanics

Given the large numbers of particles and possible degrees of freedom in any physical system, we describe macroscopic measurable quantities in terms of statistics of distinguishable particles.

For each microstate x of a classical thermodynamical system, denote its energy (Hamiltonian) by $\mathcal{H}_x$. Let $\beta = \frac{1}{k_B T}$ where $k_B \approx 1.38 \times 10^{-23}\mathrm{JK}^{-1}$ is Boltzmann's constant.[8] The probability $P(x)$ that the system occupies microstate x is proportional to the *Boltzmann factor* $\mathrm{e}^{-\beta \mathcal{H}_x}$. The constant of proportionality is $Z := \sum_x \mathrm{e}^{-\beta \mathcal{H}_x}$ (the sum being over all microstates), known as the *partition function* of the system. That is, the probability $P(x) = \frac{1}{Z}\mathrm{e}^{-\beta \mathcal{H}_x}$: the energy of state x is proportional to the natural log of its probability. The partition function is central in statistical mechanics because all the other thermodynamic properties of the system can be expressed in terms of Z or its derivatives. For example, the free energy is $A = -\frac{1}{\beta} \ln Z$ and the entropy is $S = -\frac{\partial A}{\partial T} = \frac{\partial}{\partial T}(\frac{1}{\beta} \ln Z) = \frac{\partial}{\partial T}(k_B T \ln Z)$.

Simulated Annealing and Quantum Annealing (see Sects. 23.1 and 23.2) both use at their core a Monte Carlo sampling approach due to Metropolis et al. [408] which incorporates controlled uphill steps, as well as downhill

[6] Because of this, it is sometimes phrased 'Heat flows from a hotter to a colder place'.

[7] There is an information-theoretic interpretation of entropy as a measure of our uncertainty about a system. Thermal equilibrium maximises entropy because we lose all information about the initial conditions except for the summary statistics: temperature, etc. Maximising entropy maximises our ignorance about the details of the system: it is *only* in this sense that entropy measures 'disorder'.

[8] β is often called the *inverse temperature*, though its dimension is inverse energy.

steps, in the search for a minimum. This Metropolis(–Hastings) algorithm (Algorithm 22.1) can draw samples from any probability distribution $P(x)$, and generates a Markov chain in which each state x_{t+1} depends only on the previous state x_t. It uses a *proposal density* $Q(x'|x_t)$, which depends on the current state x_t, to generate a new proposed sample x'.[9] This proposal x' is 'accepted' as the next value x_{t+1} if a number a drawn[10] from the uniform distribution on $[0, 1]$ satisfies (see [258]):

$$a < \frac{P(x')Q(x_t|x')}{P(x_t)Q(x'|x_t)}. \tag{22.4}$$

Otherwise (the proposal is not accepted), the current value is retained: $x_{t+1} := x_t$.

Algorithm 22.1: Metropolis Algorithm

```
Choose starting x_0;
repeat
    Generate proposed x' based on x_t;
    Let α := P(x')Q(x_t|x') / (P(x_t)Q(x'|x_t));
    if α > 1 then
        x_{t+1} := x';
    else
        x_{t+1} := x_t with probability 1 − α;
    end
    Increment t;
until terminating condition;
```

Because P only occurs in the ratio $\frac{P(x')}{P(x_t)}$, we require only that a function proportional to the density can be calculated. In our case, the function is usually (though not necessarily) taken as the Boltzmann factor $\mathrm{e}^{-\beta\mathcal{H}_x}$.[11] If Q is symmetric, that is, $Q(x_t|x') = Q(x'|x_t)$, as in Metropolis's original algorithm [408], then it suffices to compute $\frac{P(x')}{P(x_t)}$. If Q is not symmetric then the more complicated formula $\frac{P(x')Q(x_t|x')}{P(x_t)Q(x'|x_t)}$ (as in [258]) must be used.

[9] For example, the proposal density could be a Gaussian function centred on the current state x_t.

[10] This is the 'Monte Carlo' aspect of the algorithm.

[11] The proportionality constant is then, of course, the partition function Z, though the Metropolis algorithm does not need to compute it.

22.3.2 Ergodicity

A system's evolution or process is called *ergodic* if, over a long period, the time spent by the system in some region of the phase space is proportional to the volume of the region.

More precisely, a *weakly ergodic* evolution (*weak ergodicity*) means that the system forgets its initial state during the process of evolution, while a *strongly ergodic* evolution (*strong ergodicity*) means that the system converges to a stationary distribution regardless of the initial state. From the point of view of algorithm design, we prefer a search algorithm to be as ergodic as possible, and (for a strongly ergodic evolution) the state to which the system converges to be the state we seek (e.g., an optimum).

It will be seen later that certain physical systems such as magnets or spin glasses are not very ergodic but that the type of fluctuations introduced by annealing or quantum annealing can make them more so, so that they more easily reach the ground (lowest) energy state. The Simulated Annealing and Simulated Quantum Annealing algorithms based on these processes try to mimic this enhancement of ergodicity for objective functions.

22.4 Quantum Mechanics

Quantum mechanics is an extension of classical mechanics that models behaviours of natural systems that are observed at very short time or distance scales. An example of such a system is a subatomic particle, such as a free electron or photon (a particle of light); but a quantum system may consist of many particles.

Quantum mechanics arises from classical (nonrelativistic) mechanics by applying a formal *quantisation* procedure:[12] variables representing observable physical quantities such as position, momentum, energy, spin, etc., are replaced by linear operators on a suitable vector space. Such operators are called *observables*. The vector space is actually a *Hilbert space*, a complete inner product space, over the complex numbers $\mathbb{C}$; its elements are complex-valued (deterministic) functions of time and space coordinates.

This Hilbert 'phase' space plays a role analogous to that of classical phase space.[13] An element ψ of the Hilbert space is associated with the system. It describes the *quantum state* the system is in; ψ is variously called the system's *state vector* or *wavefunction* or *wavevector*. The only values that may be observed for a physical quantity are the eigenvalues of the corresponding linear

[12] In an analogous way, quantum field theories arise from classical field theories.

[13] In a classical phase space for n particles, we deal with real-valued functions of $6n$ dimensions (each particle giving three spatial and three momentum coordinates). In quantum Hilbert 'phase' space (really a configuration space) we deal with complex-valued functions on a $3n$-dimensional space. However, much of the mathematical structure carries over.

operator on the Hilbert space. When an eigenvalue is observed, the system's state vector ψ must be a corresponding eigenvector[14] of the operator. The operator's eigenstates form a basis for the Hilbert Space: thus, any wavefunction may be written as a linear combination of these eigenstates.[15]

22.4.1 Observation in Quantum Mechanics

The standard 'Copenhagen' interpretation of Quantum Mechanics is that this abstract wavefunction allows us to calculate probabilities of outcomes of concrete experiments. The squared modulus $|\psi|^2$ of the wavefunction ψ is a probability density function (PDF). It describes the probability that an observation of, for example, a particle will find the particle at a given time in a given region of space. The wavefunction ψ satisfies the linear *Schrödinger equation*:[16]

$$\mathrm{i}\hbar \frac{\partial}{\partial t}\psi = \mathcal{H}\psi \tag{22.5}$$

where $\mathrm{i} = \sqrt{-1}$, $\hbar = \frac{h}{2\pi}$ and $h \approx 6.62 \times 10^{-34}$ Js is Planck's constant. This differential equation describes the time evolution (the dynamical behaviour[17]) of the wavefunction, and so the PDF, at each point in space: as time goes on, the PDF becomes more 'spread out' over space, and our knowledge of, say, the position of the particle becomes less precise. This spreading continues until an observation is carried out; then, the wavefunction *collapses* nonlinearly to a particular classical state (eigenstate), in this case a particular position, and the spreading out of the PDF starts all over again.[18]

Before the observation we regard the system as being in a linear combination of all possible classical states (called *superposition of states*); then the act of observation causes one such classical state to be chosen, with probability given by the PDF. This random element in any measurement means we cannot infer the initial state of the system from the measurement result.

The essentially complex nature of the wavefunction means that even if its amplitude is constant, it has a varying phase.[19] Thus, the wavefunction may interfere with itself, e.g., if a barrier with slits is placed in a particle's

[14] Also called an *eigenfunction* or *eigenstate*: we think of it as a 'classical state'.

[15] Converting from representing a wavefunction ψ in terms of position eigenstates to writing ψ in terms of momentum eigenstates (and vice versa) is done by Fourier transform.

[16] The basic (dynamical) equation of motion: the quantum Hamiltonian of the wavefunction is equal to an (imaginary) constant times the time derivative of the wavefunction. The Schrödinger equation is a diffusion equation.

[17] The Hamiltonian contains all dynamical information: we get the wavefunction at time $t + \delta t$ (for δt small) by operating on the wavefunction at time t with $\frac{1}{\mathrm{i}\hbar}\mathcal{H}$.

[18] This collapse is called *state vector reduction*.

[19] The *phase* of a wave is the argument or angle of a complex number: it is unrelated to phase space.

'path'. This interference may be constructive or destructive, the probability of detecting a particle in a given position may go up or go down.

The probability of finding a particle is greatest at locations where the modulus (or absolute value) $|\psi|$ of the complex wavefunction ψ is largest; the probability is 0 if $|\psi| = 0$. Since the probability of finding a particle somewhere in space should be 1, we commonly restrict attention to *normalised* wavefunctions ψ, namely, those whose integral over all of space, $\int_{\mathbb{R}^3} |\psi(x)|^2 = 1$. That is, we restrict attention to the unit ball (the set of all vectors of norm 1) in our Hilbert space. Then allowable operators on the Hilbert space are the *unitary* operators, that is, the norm-preserving ones.

Observables may be either continuous (e.g., position of a particle) or discrete (e.g., the energy of an electron in a bound state in an atom). Some discrete observables may take only finitely many values, e.g., there are only two possible values for a given particle's spin in a given direction ('up' or 'down').[20]

22.4.2 Entanglement and Decoherence

A crucial distinction between classical and quantum physics (perhaps *the* crucial distinction) is the phenomenon of quantum *entanglement*: the wave function of a system composed of many particles cannot be separated into independent wave functions, one for each particle. There is *one* wavefunction for the whole system, which evolves according to the Schrödinger equation until a measurement occurs. A measurement made on one particle produces, through the collapse of the total wavefunction, an instantaneous nonlocal effect on other particles with which it is entangled, no matter how far apart they are: all are forced to immediately adopt a particular eigenstate. Entanglement leads to an explosion of the dimension of the relevant Hilbert space. The correct Hilbert space for a quantum system of m particles, each of which can be in n distinct locations, is the m-fold tensor product, rather than the m-fold direct sum. Its dimension is m^n, rather than mn as would happen classically. This is a crucial difference between quantum and classical physics, and is one reason for the better performance of quantum computers on certain problems [576]. Since the 1980s, physicists have come to view the nonlocal correlations of entangled quantum states as a new kind of nonclassical resource to be exploited, e.g., in quantum computers.

The issue of what exactly constitutes a 'measurement' is a thorny one. A modern viewpoint is that the 'quantumness' of a system is in the nonlocal entanglement effects among particles. Then, when a quantum system interacts with a macroscopic system (e.g., its environment, or a measuring device) the quantum nature of the system 'leaks' into the environment in a

[20]Spin is incorporated by necessity in Dirac's relativistic wave equation, an extension of the Schrödinger equation which is consistent with Special Relativity. In fact, spin is one of the arguments of the wavefunction in Dirac's formalism.

thermodynamically irreversible way: the phase relations are spread so ‘thinly’ among the many possible degrees of freedom that different elements of the combined wavefunction of system and environment can no longer effectively interfere with each other; we say they have *decohered.* This *decoherence* shows up as an apparent collapse of the wavefunction: decoherence causes superpositions of states to decay; and the faster the decay, the larger the scale of the superposition. Thus, quantum collapse is not, in this view, some effect of a human observer but rather a consequence of the enormous number of degrees of freedom (that is, dimensions in the Hilbert space) in any macroscopic system.

22.4.3 Noncommuting Operators

The symplectic or *canonical* structure arising from the Poisson bracket in phase space is generalised in the construction of Quantum Mechanics: a *commutator* [,] with similar properties is defined by $[A, B] := AB - BA$ for any two operators A and B. The momentum operator associated with the kth coordinate direction x_k is defined as the differential operator:[21]

$$p_k := \mathrm{i}\hbar \frac{\partial}{\partial x_k}. \tag{22.6}$$

Then $[x, p]$ can be shown to be nonzero (in fact $[x, p] = \mathrm{i}\hbar\mathbf{1}$, where $\mathbf{1}$ is the identity operator), telling us that the momentum p and position x operators do not commute: they are *complementary* and there are eigenvalues of each which are not eigenvalues of the other. From this, via Fourier analysis, comes Heisenberg's *Uncertainty Principle*, $\mathrm{d}p\mathrm{d}x \geq \hbar/2$: noncommuting observables cannot simultaneously be measured to arbitrary precision.[22]

Identifying $p_k = \mathrm{i}\hbar\frac{\partial}{\partial x_k}$ thus, and recalling (22.2) for the energy of a single particle, we may rewrite the (quantum) Hamiltonian of a particle as

$$\begin{aligned}
\mathcal{H} &= \frac{p_1^2 + p_2^2 + p_3^2}{2m} + V \\
&= -\frac{\hbar^2}{2m}\Big[\Big(\frac{\partial}{\partial x_1}\Big)^2 + \Big(\frac{\partial}{\partial x_2}\Big)^2 + \Big(\frac{\partial}{\partial x_3}\Big)^2\Big] + V \\
&= -\frac{\hbar^2}{2m}\Big(\frac{\partial^2}{\partial {x_1}^2} + \frac{\partial^2}{\partial {x_2}^2} + \frac{\partial^2}{\partial {x_3}^2}\Big) + V \\
&= -\frac{\hbar^2}{2m}\nabla^2 + V.
\end{aligned} \tag{22.7}$$

Here, multiplication of operators is just composition, so $(\frac{\partial}{\partial x_k})^2 = \frac{\partial}{\partial x_k}(\frac{\partial}{\partial x_k}) = \frac{\partial^2}{\partial {x_k}^2}$; and we have used the standard notation ∇^2 for the div operator $\frac{\partial^2}{\partial {x_1}^2} +$

[21] This is the *correspondence principle*: the key point of (canonical) quantisation.

[22] Classical mechanics is the limiting case of quantum mechanics as $h \to 0$. In this case, all commutators are 0 so all operators commute, and there is no uncertainty. It is also the limiting case of special relativity as $c \to \infty$, where c is the speed of light.

$\frac{\partial^2}{\partial {x_2}^2} + \frac{\partial^2}{\partial {x_3}^2}$. Then the particle's Schrödinger equation $\mathrm{i}\hbar\frac{\partial}{\partial t}\psi = \mathcal{H}\psi$ becomes:

$$\mathrm{i}\hbar\frac{\partial}{\partial t}\psi = -\frac{\hbar^2}{2m}\nabla^2\psi + V\psi. \tag{22.8}$$

Apart from the imaginary factor i, this recalls other equations arising in physics: the time rate of change of a quantity (ψ) is described by a diffusion term (the second spatial derivative[23]) and a drift term (the potential V).

22.4.4 Tunnelling

In *quantum tunnelling* a particle can jump from one side of a physical or energy barrier to another, without having to occupy all intermediate positions. This can be explained by the wave nature of the particle.[24] Its wavefunction is spread through space and, in particular, is nonzero on the opposite side of the barrier, giving a finite probability of observing the particle there. The narrower the barrier the greater the probability of tunnelling. From this it follows that the expected time to tunnel through a barrier depends not only on the height of the barrier, but also on its width. For a fixed height, the mean tunnelling time is shorter for a narrower barrier.

Such situations arise when a particle is trapped in a potential well of finite height, for example, an electron in an atom. Another physical example of tunnelling as a mechanism is where a neutron may be spontaneously emitted from a ^{238}U Uranium nucleus in radioactive decay. In this example, the potential well is caused by the *strong nuclear* force among nucleons,[25] which is repulsive at very short distance scales but becomes strongly attractive as nucleon separation increases.

22.4.5 Quantum Statistical Mechanics

The formal description of partition function carries over to quantum physics, where $\mathcal{H}$ becomes an operator: then $Z = \sum_x \mathrm{e}^{-\beta\mathcal{H}_x}$ is often written as the basis-independent *trace*, $Z = \mathrm{Trace}(\mathrm{e}^{-\beta\mathcal{H}})$. The exponential of an operator is defined using the Taylor series $\mathrm{e}^t = 1 + t + \frac{1}{2}t^2 + \frac{1}{3!}t^3 + \cdots$ for the exponential function. Here, we regard the product of operators as meaning composition, so AB is defined by $(AB)(\psi) := A(B(\psi))$. Thus, in a quantum setting,

[23] If this were the only term we would get the classical Heat Equation $\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$, which also arises in finance as the Black–Scholes(–Merton) equation for option pricing.

[24] The tunnelling effect is a property of any wave and can be observed, for example, when light waves hit, at a glancing angle, a boundary where the refractive index changes.

[25] *Nucleon* is a generic term for nuclear particles, namely, protons and neutrons.

$$Z = \mathrm{Trace}(\mathrm{e}^{-\beta\mathcal{H}}) = \mathrm{Trace}\left(\mathbf{1} + \mathcal{H} + \frac{1}{2}\mathcal{H}^2 + \frac{1}{3!}\mathcal{H}^3 + \cdots\right). \tag{22.9}$$

In applications to algorithms, a cost function to be optimised is commonly mapped to a quantum Hamiltonian $\mathcal{H} = \mathcal{H}_{\text{pot}} + \mathcal{H}_{\text{kin}}$ consisting of two parts, the objective function $\mathcal{H}_{\text{pot}}$, and a 'kinetic' perturbation term $\mathcal{H}_{\text{kin}}$. Thus the *Suzuki-Trotter formula* for the exponential of the sum of two operators $A+B$ becomes important for estimating the partition function:

$$\mathrm{e}^{A+B} = \lim_{M\to\infty} \left(\mathrm{e}^{A/M}\mathrm{e}^{B/M}\right)^M \tag{22.10}$$

This formula is applicable even if the commutator $[A, B] \neq 0$.

22.5 Quantum Computing

The special properties of quantum systems have been used to construct computers based on principles beyond those of classical computing machinery; these computers are provably able to attack certain problems more efficiently than classical computers. *Digital* quantum computers use qubits, a quantum version of bits, while *Adiabatic* quantum computers slowly evolve a quantum system to optimise some quantity such as (free) energy.

We discuss quantum computers briefly here, because their workings are the inspiration for quantum inspired evolutionary algorithms (Chap. 24) and simulated quantum annealing (SQA: Sect. 23.2).

A good overview of quantum computing, including more detail on the underlying physics than is given here, can be found in [273].

22.5.1 Two-State Systems and Qubits

A quantum particle's spin has only two possible values relative to a direction of measurement: "up" or "down". This is an example of a *two-state system*.[26] In such a system, the quantum state ψ is a linear superposition of just two eigenstates, say $|0\rangle$ and $|1\rangle$, in the standard Dirac bra-ket notation; that is,

$$\psi = \alpha|0\rangle + \beta|1\rangle. \tag{22.11}$$

Here, $|0\rangle$ and $|1\rangle$ are orthogonal basis vectors for a two-dimensional complex Hilbert space, and $\alpha, \beta \in \mathbb{C}$, with $|\alpha|^2 + |\beta|^2 = 1$ for ψ normalised.

A two-state system where the states are normalised and orthogonal, as here, may be regarded as a *quantum bit* or *qubit*.[27] It is thought of as being in

[26] Realisations using multiple particles have also been used.

[27] Geometrically, a qubit is a projective two-dimensional complex Hilbert space $P\mathbb{C}^2$: it is a compact two-dimensional complex manifold, called the Riemann (or Bloch) sphere.

eigenstates $|0\rangle$ and $|1\rangle$ simultaneously, until a measurement projects ψ onto the basis $\{|0\rangle, |1\rangle\}$: the state vector collapses to $|0\rangle$ (with probability $|\alpha|^2$) or $|1\rangle$ (with probability $|\beta|^2$). The normalisation relation $|\alpha|^2 + |\beta|^2 = 1$ captures the fact that precisely one of $|0\rangle$, $|1\rangle$ must be observed, so their probabilities of observation must sum to 1.

22.5.2 Digital Quantum Computers

A *digital* (or *standard* or *circuit theory*) quantum computer is one which uses qubits instead of the (classical) bits used by usual computers. Paul Benioff [50] first considered a Turing machine which used a tape containing the equivalent of qubits. Richard Feynman [189], in the course of trying to simulate quantum systems on classical computers, developed examples of physical (quantum) computing systems that are not equivalent to the (deterministic) Turing machine. He showed that, because of entanglement, no classical computer (not even a probabilistic one) can efficiently simulate quantum systems, as it would need exponential time or resources to do so. For example, an n-bit classical system is characterised by n times the amount of information needed to characterise a one-bit system. However, the amount of information needed to characterise a general entangled state of n qubits is $O(2^n)$: the Hilbert space has 2^n dimensions, so an entangled state is a superposition of 2^n n-qubit basis (eigen)states, and a complex coefficient is needed for each basis vector. Benioff and Feynman were the first to point out that quantum systems can be used to compute, and Feynman noted that they could efficently do computations that a classical computer could not (e.g., simulate themselves!).

In a digital quantum computer, computations are performed by applying a sequence of unitary operators, called *gates*, to a system of qubits, without performing a measurement until the end of the computation.[28] In the 1990s, interest in these "standard" qubit-based computers was stimulated by the publication of algorithms which were provably faster than classical algorithms could be for the same problem: Shor's algorithm [574, 575] for factorising a large integer of n bits in $O(n^2)$ time — compared to $O(\exp(n^{1/3}))$ time classically — for which he received the Nevanlinna Prize in 1998; and Grover's unstructured database search algorithm [229, 230] which reduced the complexity quadratically from $O(n)$ classically to $O(\sqrt{n})$. The only other applications (so far) for which quantum computers have been proven to outperform classical computers are simulation of quantum systems, and computation of discrete logarithms. Note that quantum computers are not known to be able to solve NP problems in polynomial time (in fact, it is widely believed that they are not able to); in particular, factorisation is not an NP problem.

[28] These ideas are described further in Chapter 24, where they are used in new types of evolutionary algorithms.

22.5.3 Quantum Information

A crucial difference between a qubit and a (classical) bit is that multiple qubits can exhibit entanglement. This allows a set of qubits to be highly correlated. Entanglement also allows many states to be acted on simultaneously, unlike bits that can only have one value at a time, and so the quantum computer can extract global (nonlocal) information. This is sometimes called *quantum parallelism* [154], and gives a possible explanation for the power of quantum computing: because the state of the quantum computer (i.e., the state of the system considered as a whole) can be a quantum superposition of many different classical computational states, these classical computations can all be carried out at the same time. For more information on quantum algorithms and entanglement, including its possible use in cryptography, see [91].

Much of the art of designing quantum algorithms is in finding ways to make efficient use of the nonlocal correlations. There are quantum algorithms that achieve an exponential speedup over any possible classical algorithm. A quantum computer is usually taken to be *globally phase coherent*, that is, having every qubit entangled with every other qubit (global phase coherence is, for example, assumed in Grover's proof of unstructured database search speedup).

However, quantum information has properties that make it more difficult to deal with, in some ways, than classical information:

- The uncertainty principle says that if A and B are noncommuting (complementary) observables, then measuring A will necessarily affect the result of any subsequent measurement of B. The act of acquiring information about a quantum system unavoidably disturbs the system: the *disturbance principle.*
- Quantum information cannot be copied with perfect fidelity (this is the *no-cloning principle* [663, 158]); for, if it could be, then we could measure the copy without disturbing the original, thus defeating the disturbance principle.
- John Bell [47] showed that quantum information is, because of entanglement, typically encoded in nonlocal correlations among different parts of the system: something without any analogy in classical systems, and the main reason that we cannot efficiently simulate a quantum system with a classical computer.
- These nonlocal correlations are extremely unstable in practice, because of *decoherence*: the system interacts with its environment and its information becomes entangled with the information in the environment; since most of the information is in the nonlocal interactions, we lose the ability to recover our original system's information. Shor discovered quantum error correction techniques to counteract this leakage, essentially reducing the decoherence rate to acceptable levels.

Standard quantum computers have been built: one of seven qubits, implemented using nuclear magnetic resonance (NMR) applied to seven chosen

nuclei of 10^{18} identical molecules, has used Shor's algorithm to factorise 15 as 3×5 (see [641]). The record size so far of an NMR digital quantum computer is 12 qubits [434].

It is now an engineering problem to develop globally phase coherent quantum computers of 100 qubits or more (which could break current encryption standards in an acceptable time); however, decoherence makes this problem very difficult. Analogies of these, implemented on classical machines, form the basis of Quantum-Inspired Genetic Algorithms, described in Chap. 24.

22.5.4 Adiabatic Quantum Computation

More generally, we can build quantum computation systems which are not made up of qubits. Such quantum computers are not necessarily "digital": they evolve with time and are closer in spirit to the original analogue computers, in that a physical system modelling the computation to be performed is set up, and then allowed to evolve with time.

The *quantum adiabatic theorem* [70] describes the behaviour of a quantum system when its Hamiltonian (total energy) varies slowly in time:

Theorem 22.1 (Born, Fock). *A perturbed system remains in its instantaneous eigenstate if the perturbation is applied sufficiently slowly, and if there is a gap between that state's eigenvalue and the Hamiltonian's other eigenvalues.*

Intuitively, this says that a quantum system subjected to sufficiently slowly varying external conditions can modify its functional form so as to remain in the same relative (pure) eigenstate of the system Hamiltonian at all times; however, if the conditions change rapidly, the functional form of the state has insufficient time to adapt, so it stays unchanged, ending in a linear superposition of states, i.e., *not* an eigenstate, of the final Hamiltonian. A *quantum adiabatic evolution* is one where the system is varied slowly enough that Theorem 22.1 applies, and the system stays in the same eigenstate of $\mathcal{H}$; often this is the ground (lowest energy) state. These evolutions are the main idea of *Adiabatic Quantum Computing* (AQC):

- the Hamiltonian is regarded as a multivariate cost function to be optimised;
- if energy landscape barriers are high but very narrow, quantum mechanical fluctuations in an observable because of its noncommuting with the Hamiltonian can help in tunnelling through them: so a high strength "tunnelling" field is applied which adds a noncommuting quantum kinetic term;
- roughly speaking, this gives a very flat energy landscape for which the Hamiltonian easily reaches its ground state;
- then the field strength is slowly reduced to 0, allowing the Hamiltonian to find the ground state of the desired system (provided there is no crossing of energy levels with the ground state during the evolution).

Because of the similarity to heat annealing of metals (Sect. 22.6), ACQ is also known as Quantum Annealing (QA). The adiabatic theorem guarantees that the result of the computation will be correct, provided that the computation (perturbation) is carried out slowly enough.

It has been shown [4] that an Adiabatic Quantum Computer can perform any computation achievable by a "standard" qubit-based quantum computer: they are polynomially equivalent in power. As with classical analogue computers, devices based on AQC generally need to be tailored to the specific problem they are being applied to.

22.6 Annealing and Spin Glasses

In the process of *annealing*, as used by smiths for centuries, a material (for example, a metal alloy) is initially heated. By injecting energy into the material during the heating phase, its component atoms are able to assume random or *disordered* states. If the metal is cooled sufficiently slowly, it remains approximately in thermodynamic equilibrium.[29] It will settle or *freeze* into a minimum energy crystalline structure with the individual atoms arranging themselves into a regular array. This has desirable physical characteristics as it produces a strong and malleable or ductile structure. If the cooling is performed too quickly (*quenching*), the metal will freeze into a local, rather than global, minimum energy state and the atoms will not form a regular array. This can produce a metastable state with fault lines and defects in the metal, leading to fractures if the metal is stressed. This state or *phase*[30] of the metal is analogous to the disordered yet solid structure of glass. The annealed phase is analogous to a regular, ordered crystal.

A particular case of annealing, used as inspiration in Simulated Annealing (see Sect. 23.1), is where a ferromagnetic alloy is *slowly* cooled to give a magnet. Physically, magnetism is caused by rotating particle charges (spins). Each such spin gives rise to a microscopic dipole which may be aligned either up or down along a given axis. Above a critical *Curie* temperature, T_c, the minimum of free energy $A = \mathcal{H} - TS$ occurs at states with no magnetisation: the spins are completely disordered. As the temperature is reduced below T_c, a phase change occurs. Ordered clusters start to appear and the material begins to show magnetism; the minimum of A shifts towards states with higher order of magnetisation. Finally, at $T = 0$, the global minimum of free energy, and hence that of energy $\mathcal{H} = A+TS$, occurs at the highest order of magnetisation; the microscopic spins have all become aligned.

Spin glasses are a phase of magnetic alloys which bear the same relation to magnets as glasses do to crystals; hence the name. In rapidly cooled or *quenched* spin glasses, different spin-spin interactions are randomly ferromagnetic or antiferromagnetic and frozen in time, which is known as *frozen*

[29] When thermodynamic equilibrium is maintained, the process is called *adiabatic*.
[30] This use of 'phase' is not related to either phase space or the phase of a wave!

disorder. This disorder leads to competing interactions among the spins, called *frustration*; none of the spin states is able to satisfy all of the interactions. Thus the potential energy landscape becomes very rugged, with local and global minima separated by potential energy barriers. This disorder and frustration is reminiscent of the search landscapes occurring in applications, such as combinatorial optimisation problems; hence it seems worthwhile to adapt features of the physics of spin glasses to search algorithms for these applications. In particular, the frustration effect is analogous to the conflicting constraints often encountered in optimisation applications. There is a large body of work on the physics of spin glasses, which has enhanced our understanding of the structure of the spin glass energy landscape, and of the slow (glassy) dynamics of many-body systems in the presence of frustration and disorder: consequently, it can aid our understanding of computationally difficult problems. Indeed, research has found annealing approaches particularly useful for combinatorial optimisation problems and this is described in more detail in Sects. 23.1 and 23.2.

In [330], it is noted that slow annealing of a material may avoid the frustrated spin interactions of a quenched spin glass, by allowing the material to adopt a regular noncompeting pattern of spins, settling into a ferromagnet phase rather than a spin glass phase. However, Hamiltonians encountered in applications may have a spin glass phase, even at low temperatures, with many degenerate near-to-ground states of nearly equal energy. This phase is very stable at low temperatures, leading [330] to conclude:

- even with frustration present, one can make clear improvements over a random starting configuration;
- there will be many good near-optimal solutions; a good search algorithm should find some;
- no one of these near-equal low-energy states is significantly better than the others, so seeking the global optimum may not be productive.

22.6.1 Ising Spin Glasses

A common Statistical Mechanics model used for (among other things) spin glasses is the *Ising model*, named after the physicist Ernst Ising. It models a discrete collection of spins, each of which can take the value 1 or -1. Typically, the spins are the lattice points in a regular grid of dimension 1, 2 or 3. The spins S_i are considered to interact in pairs, with an interaction (i.e., potential) energy J_{ij} which has one value when the two spins are the same (*aligned*), and another value when the spins are different (anti-aligned). The product of two aligned spins is 1 and the product of two anti-aligned spins is -1. The energy of a configuration of N such spins is modelled as:

$$\mathcal{H} : \{-1, 1\}^N \longrightarrow \mathbb{R} \quad \text{with} \quad \mathcal{H} = -\sum_{i>j}^{N} J_{ij} S_i S_j. \tag{22.12}$$

Thus the configuration space (of order 2^N) is a Cartesian product of N copies of $\{-1, 1\}$ and is isomorphic as a set to the search space of an EA using a bit representation, with $\mathcal{H}$ playing the same rôle as a fitness function.

The spins can be modelled as vertices of a weighted graph, with an edge (of weight J_{ij}) connecting vertices i and j if and only if $J_{ij} \neq 0$. This weighted graph completely specifies the Ising model.

When the Ising model is applied to a spin glass, the J_{ij} interactions are quenched variables which vary randomly both in magnitude and sign, according to some distribution $\rho(J_{ij})$, which in the standard models is taken to be normal about 0:

$$\rho(J_{ij}) = A \exp\left(\frac{-J_{ij}^2}{2J^2}\right). \tag{22.13}$$

In two or more dimensions, the Ising model undergoes a phase transition between an ordered and a disordered phase. This transition occurs at a critical temperature, T_c, above which the spins are disordered and below which is the ordered ("spin glass") phase. Below T_c, *freezing* occurs.

There are two competing pictures of the physics of spin glasses. The *mean-field* picture applies to infinite-range models such as the SK spin glass, while the *droplet* picture applies to short-range models. The mean-field picture may not be valid for physical spin glasses, where the interactions are effectively short range, but may still be of use in applications of such models to optimisation problems, where interactions may be long range.

The Ising model is a statistical model and, as in Sect. 22.3.1, the probability of a given configuration x of spins is the Boltzmann factor with inverse temperature β:

$$P(x) = \tfrac{1}{Z}\mathrm{e}^{-\beta\mathcal{H}}\,. \tag{22.14}$$

To generate configurations for simulation using this probability distribution, a variant of the Metropolis Algorithm (Algorithm 22.2) may be used.

Algorithm 22.2: Metropolis Algorithm for Ising Spin Models

```
repeat
    Randomly pick a spin;
    Calculate the contribution to the energy due to this spin;
    Flip the value of the spin and calculate the new contribution;
    Let ΔH = new energy − previous energy;
    if ΔH < 0 then
        Keep the flipped value;
    else
        Keep the flipped value with a probability e^(−βΔH);
    end
until terminating condition;
```

The change in energy only depends on the value of the spin and its nearest neighbours in the Ising model graph. Thus, if the graph is sparse (that is, on average, a vertex is not connected to many other vertices), the algorithm is fast. This process will eventually produce a choice from the distribution.

22.6.2 Quantum Spin Glasses

A quantum spin glass is a classical spin glass with the addition of a kinetic (quantum tunnelling) term to the Hamiltonian. In quantum spin glasses, both thermal and quantum fluctuations can cause the order-disorder phase change. There are two types of quantum spin glass: vector spin glasses, where the fluctuations cannot be affected by adjusting a laboratory field; and classical spin glasses perturbed by a quantum tunnelling term (the useful type), where a transverse laboratory field Γ can affect the fluctuations. This transverse Ising spin glass (TISG) model is the most studied [99]. The term *quantum annealing* is used for this 'annealing' by reduction of the transverse field (by analogy with thermal annealing). As well as being able to penetrate very high but narrow barriers (which makes the energy landscape more accessible to local moves), quantum annealing has the ability to 'sense' the whole configuration space simultaneously via a delocalised wave function.

Experimental results [3, 77] on a TISG (a sample of $\mathrm{LiHo}_{0.44}Y_{0.56}\mathrm{F}_4$) have shown that the relaxation behaviour depends on the path taken. The sample was brought from a high temperature paramagnetic phase to a low temperature glassy phase along two paths from $(0.8, 0)$ to $(0.02, 7)$ in the T-Γ plane. Along the classical annealing path, the transverse field Γ was kept at 0 while T was slowly reduced; then Γ was switched on. Along the quantum annealing path, Γ was kept high, then T was reduced, with Γ being reduced (slowly, i.e., adiabatically) only on reaching the final temperature. It was found that the quantum approach gave states with relaxation up to thirty times faster than thermal annealing. This indicates that quantum annealing is much more effective at exploring the configuration space in the glassy 'disordered' phase than thermal annealing. Crucially, it has been found [557] that even in the presence of all three ingredients of a spin glass — frustration, randomness and long-range dipolar interactions — the spin glass phase of $\mathrm{LiHo}_x Y_{1-x}\mathrm{F}_4$ is destroyed by any nonzero transverse field, which shows the ability of quantum tunnelling to explore a rugged potential energy landscape with high barriers.

The Hamiltonian of this model (based on (22.12) with spins aligned in the z direction and a kinetic term in the x direction added) is:

$$\mathcal{H} = -\sum_{i>j}^{N} J_{ij} S_i^z S_j^z - \Gamma \sum_{i=1}^{N} S_i^x = \mathcal{H}_{\mathrm{pot}} + \mathcal{H}_{\mathrm{kin}} \tag{22.15}$$

where Γ is the tunnelling strength and the weights J_{ij} are randomly distributed as in (22.13). Γ is thought of as the 'strength' of the quantum kinetic term $\mathcal{H}_{\mathrm{kin}} = -\Gamma \sum_{i=1}^{N} S_i^x$.

It was first pointed out by Feynman (see [550]) that a quantum-mechanical system may, in many ways, be regarded as a classical system embedded in a world with one extra dimension, called imaginary time. The Suzuki-Trotter formalism maps a d-dimensional quantum Hamiltonian to an effective $(d+1)$-dimensional classical Hamiltonian, with the extra time-like dimension (called the *Trotter dimension*) discretised into M steps. At each index $k = 1, \ldots, M$ along this new dimension, there is a d-dimensional *Trotter slice*, and the Hamiltonian now represents a system of spins in a $(d+1)$-dimensional lattice. Now the Suzuki-Trotter formula, (22.10), can be used to compute the partition function

$$Z = \mathrm{Trace}(\mathrm{e}^{-\mathcal{H}/T}) = \mathrm{Trace}(\mathrm{e}^{-(\mathcal{H}_{\mathrm{pot}}+\mathcal{H}_{\mathrm{kin}})/T}) \tag{22.16}$$

and this eventually gives the effective classical Hamiltonian in the Mth Trotter approximation as

$$\mathcal{H} = \sum_{i>j}^{N}\sum_{k=1}^{M} K_{ij} S_{ik} S_{jk} - K \sum_{i=1}^{N}\sum_{k=1}^{M} S_{ik} S_{ik+1} \tag{22.17}$$

where $K_{ij} = J_{ik}/(MT)$, $K = \frac{1}{2}\ln\coth(\Gamma/(MT))$, k is the index in the extra Trotter dimension, i is the position in the original Ising model, and S_{ik} is the Ising spin on the lattice site (i,k). In principle, this equivalence of quantum Hamiltonian with a one-dimensional higher classical Hamiltonian only holds exactly as $M \to \infty$, but in practice there is an optimum M. In units where $\hbar = 1$, M should be of the order of $1/T$ (of course, $M \to \infty$ as $T \to 0$).

One then proceeds to study and optimise the equivalent classical Hamiltonian using Monte Carlo techniques, as in Algorithm 22.2, which leads to Simulated Quantum Annealing (SQA) as described in Sect. 23.2.

Recently [550, 584] it has been shown that it is possible to study the thermodynamics of a d-dimensional classical system by reducing it to the study of the ground state(s) of a d-dimensional quantum system, the reverse of the above use of equivalent classical Hamiltonian. In [585] it is shown that SA itself can be 'simulated' using a quantum algorithm, Quantum Simulated Annealing (QSA);[31] and that when SA requires $1/\delta$ steps to get within a certain error bound of the optimum, then QSA will only require $1/\sqrt{\delta}$ steps.

22.7 Summary

Physically inspired computing algorithms draw on our understanding of the properties of physical processes. In this chapter we provide a short primer on aspects of these processes as well as the basic physics needed to develop an appreciation of them. In the next two chapters we outline a range of algorithms whose design has been inspired by these processes.

[31] QSA is a *quantum* algorithm, very different from the *classical* algorithm, SQA.

23

Physically Inspired Computing Algorithms

Following the introduction to a range of physical phenomena in the last chapter, this chapter describes a number of algorithms which are metaphorically inspired by these. As was the case with biologically inspired algorithms, the degree of faithfulness of each of these metaphors to the original natural process varies, and multiple algorithms could be devised depending on which aspect of the underlying phenomenon is chosen.

Initially, we introduce the simulated annealing algorithm (Sect. 23.1) and the simulated quantum annealing algorithm (Sect. 23.2). This is followed by a discussion of the constrained molecular dynamics algorithm (Sect. 23.3), physical field-inspired algorithms (Sect. 23.4), and the extremal optimisation algorithm (Sect. 23.5).

23.1 Simulated Annealing

Simulated annealing (SA) [97, 98, 330, 510] is a global optimisation algorithm whose workings are loosely inspired by the physical process of the thermal annealing of metals or glass (Sect. 22.6). In the context of designing a search algorithm, the physical process of thermal annealing can be considered as being the search for the global minimum of an energy landscape. By analogy, the SA algorithm tries to find the global minimum of an objective function, which could be defined on $\mathbb{R}^n$ or, more generally, on a space of 'states' (or candidate solutions), e.g., valid tours of the network in a TSP (see Sect. 9.3.2 for an introduction to the travelling salesman problem).

A simulated annealing algorithm can be run using a single trial solution which is modified randomly during each iteration of the algorithm. The search process usually proceeds 'downhill', metaphorically from a *higher* to a *lower* energy state, assuming the goal is minimisation of the objective function. Modifications which improve the existing trial solution are always accepted. However, occasional 'wrong-way', or uphill, moves are accepted with a nonzero

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_23

probability. This can allow the search process to escape a local minimum, which would trap a standard hill-climbing or other greedy approach.

The SA algorithm is based on the Metropolis algorithm of statistical mechanics given as Algorithm 22.1 in the previous chapter, with the addition of a temperature schedule which controls the probability of uphill moves. This permits a gradual tuning of the exploration-exploitation balance away from exploration towards more intensive exploitation as the search progresses. A basic SA algorithm is provided in Algorithm 23.1.

Algorithm 23.1: Simulated Annealing Algorithm

```
Set initial temperature and determine the annealing schedule;
Generate an initial feasible solution and assess the cost of this solution;
repeat
    Randomly generate new solution within a neighbourhood of the current
    solution;
    Assess cost of new solution;
    if cost of new solution < cost of previous solution then
        Accept the new solution;
    else
        Accept the new solution with a probability p;
    end
    Adjust temperature;
until terminating condition;
```

This approach, with suitable parameter modification, allows for discrimination between the objective function's large-scale behaviour (at higher temperatures, near the start of the algorithm's run) and its finer-scale behaviour (at lower temperatures, later in the run). Early in the run, gross features of the system appear. The algorithm finds a broad area in the search space where a global optimum should exist, following the large-scale behaviour without regard to small local optima found on the way. At lower temperatures, the finer details of this area emerge and, with high probability, the algorithm will find a near-global optimum, or even the global optimum.

As noted earlier (Sect. 22.3), thermodynamic equilibrium corresponds to minimising (Helmholtz) Free Energy, $A = \mathcal{H} - ST$, with S being entropy and T being temperature. When $T = 0$, this reduces to the Hamiltonian. Since SA reduces a temperature parameter to 0, it may be thought of as minimising either (or both) the free energy or the Hamiltonian. Of course, in search algorithms such as SA or Simulated Quantum Annealing (SQA) (Sect. 23.2), we may modify the Hamiltonian, e.g., by adding extra terms.

23.1.1 Search and Neighbourhoods

As with any search algorithm, the way in which moves are made from one state (or 'point' in the search space) to another is crucial. In SA, as in Simulated Quantum Annealing (SQA) (Sect. 23.2) and Extremal Optimisation (EO) (Sect. 23.5), a *neighbourhood structure* is defined. Let X denote the set of all possible states (assumed finite). For each state $x \in X$, there is a set N_x of states to which transitions from x may happen. N_x is called the *set of neighbours* or *neighbourhood* of x. We require that the neighbour relation be symmetric, i.e., $x \in N_y$ if and only if $y \in N_x$. In general, the neighbours of x (the possible moves) are specified by the user of the algorithm, in a way that depends on the particular problem. This neighbourhood structure defines a graph on the search space: the vertex set is the set of states X; and there exists an edge xy if and only if $x \in N_y$ (or, equivalently, $y \in N_x$); thus, the graph is undirected. The performance of the SA algorithm depends crucially on the choice of neighbourhood structure. If the neighbourhoods are too small, the diameter of the corresponding graph will be too large and the simulated process will not be able to move around the set X sufficiently quickly to reach the optimum in reasonable time. Conversely, if the neighbourhoods are too large, with sizes of the same order as the size of X, then the process effectively carries out a random search through X, with a distribution close to uniform. Usually in SA (and in SQA and EO), the neighbour of x is generated by making a small change (perturbation) to x: thus these algorithms are generally *perturbative* in nature.

The `neighbour()` method that generates the next state should be biased towards states of roughly the same cost/energy as the current one, since after a few iterations the SA algorithm is probably at a state of much lower cost than the average state. For example, in the TSP, a state is just a particular tour: we may use Lin's k-optimality idea [369] to define the neighbours of a tour x as those tours that can be obtained from x by a 'k-opt' move, namely, removing k edges of x and reassembling the k resulting sections of the tour in a different order, using k new edges. For example, a special case of 2-opt is exchanging a pair of consecutive cities in the tour x (Sect. 9.6 illustrates an example of this).

23.1.2 Acceptance of 'Bad' Moves

A key component of the algorithm is the acceptance rule for 'wrong way' moves, those in which the solution disimproves during an iteration of the algorithm. Traditionally[1] in SA the acceptance probability for these cases is taken to be:

$$p_i = \mathrm{e}^{-\Delta f / T_i} \tag{23.1}$$

[1] By analogy with the Boltzmann factor described in Sect. 22.4.5, with Boltzmann's constant k_B scaled to 1.

where T_i is the *temperature* at iteration i and Δf is the difference in the objective function value between the current and the previous trial solutions (this is positive if the current solution is poorer than the previous one). In determining whether to accept a poorer solution, two factors are relevant: how much worse the current solution is than the previous one (much poorer solutions are less likely to be accepted); and the value of the temperature parameter. For given values of T_i and Δf, the value p_i is calculated and a random number is generated uniformly on the interval $(0, 1)$. If the generated random number is less than p_i, the move to a poorer solution is accepted. Thus SA (and, as will be seen later, SQA and EO) is a *stochastic* local search algorithm.

The higher the value of T_i, the higher the likelihood that a poorer solution will be accepted. Hence, T is a tuning parameter and its value is varied during the algorithm's run according to an annealing or *cooling schedule*. The cooling schedule determines the probability with which a poorer solution will be accepted at each stage of the algorithm. The slower the rate of cooling, the more likely the algorithm is to find a good quality solution. However, slower cooling schedules also increase the algorithm's run time. A common cooling schedule is exponential cooling, where the initial temperature value is decreased to a fixed lower value over n discrete steps according to $T_{i+1} = \alpha T_i$, where $0 < \alpha < 1$. That is, $T_{i+1} = \alpha^i T_1$.

23.1.3 Parameterisation of SA

The crucial choices in SA are the methods of neighbourhood generation, and temperature adjustment (the annealing schedule). There exist convergence theorems [209] that say that if the temperature parameter is reduced sufficiently slowly, namely, no faster than $N/\log i$, where N is the system size and i the time so far, then the system state will remain 'close' to the minimum energy possible for that temperature, and will asymptotically converge to the optimum. The analogy from thermodynamics is that if cooling is sufficiently slow, then the system will remain very close to thermodynamic equilibrium at all times. Thus, if T_{i+1} is of the logarithmic form $T_1/\log i$, for T_1 sufficiently large, then the system will converge to the global optimum and will not be trapped at a local optimum. This schedule can be very slow to converge and sometimes, for reasons of expediency, faster annealing schedules are used. In this case, convergence is not guaranteed and the algorithm might be more properly called *Simulated Quenching* (SQ) by analogy with the idea of quickly cooling (quenching) a magnetic alloy.

Usually, a fixed number of moves must be accepted at each temperature before proceeding to the next (lower) temperature. This is done to give reasonable sampling statistics for the current temperature.

Values for α, and the initial and final temperature values are problem-dependent and the efficiency of the algorithm is sensitive to the choice of values. As a rough rule of thumb, an α value of 0.9, and a choice of initial

temperature T_1 such that there is an initial acceptance probability of poorer solutions of around 0.8, are commonly used.

23.1.4 Extensions of SA

The choice of neighbourhood system may be regarded as another way in which SA may be parameterised, and much research has addressed the question of how to choose a neighbourhood system that will allow the SA algorithm to perform well. In combinatorial optimisation problems, the structure of the problem often leads to a 'natural' choice of neighbours, e.g., the 2-opt or 3-opt criterion for a 'close' tour in the TSP. In continuous optimisation problems, there may be less (obvious) structure to make use of, and so more freedom in neighbourhood generation. Even after having chosen a neighbourhood system, new trial solutions need not be generated purely randomly: the distribution may be weighted (possibly with weight depending on progress of the algorithm) to favour certain trial solutions, e.g., those 'close' to the current solution.

For example, in *Adaptive Simulated Annealing* (ASA), also called Very Fast (Simulated) Reannealing (VFR) [289, 290], the algorithm parameters such as temperature, neighbourhood radius, step size, etc., are adjusted by the algorithm during its run, with the intention that the user need not choose precise starting values of these (SA performance can be very sensitive to initial choice of the parameter values). At the beginning of the algorithm's run, the whole of the search space is explored to a coarse resolution. Later in the run, there is more exploitation and the state is directed towards favourable areas found so far. This changing of the resolution is analogous to multigrid or variable mesh approaches used in, for example, finite difference algorithms for solving partial differential equations.

Additionally, the algorithm need not be restricted to a single trial solution. For example, the SA paradigm could be used with a populational approach, whereby a population of trial solutions are maintained with diversity being generated using recombination and mutation. In the latter case, parents can be selected randomly from the current population, with the resulting child competing against its parents for survival in a Boltzmann trial. Other hybrid SA possibilities are outlined in [145, 386]. One interesting combination of SA and neural nets is described in [105], where transiently chaotic dynamics are introduced into neural networks, giving rise to an optimisation process similar to simulated annealing (not stochastic but rather deterministically chaotic). This new method is called chaotic simulated annealing and has been applied to the TSP and other problems.

23.1.5 Concluding Remarks

SA is a popular optimisation algorithm as it is relatively easy to code, makes few assumptions regarding the function to be optimised, is capable of handling

a variety of boundary conditions and constraints, gives a statistical 'ergodic' guarantee of finding an optimum (provided the cooling is done slowly enough) and is quite robust. However, the effectiveness of the algorithm does depend on several factors, including the methodology for selecting neighbours of the previous solution, the annealing schedule, the design of the probability transition function, and the choice of parameters.

Criticisms of SA and related algorithms centre on: its slowness (if the proven convergent logarithmic annealing schedule is used); loss of ergodicity if a faster annealing schedule is used; the difficulty of tuning the parameters α, T_1, etc., to suit particular problems; and other implementation difficulties, such as often there being no obvious free parameter in the application problem which is analogous to the temperature T.

23.2 Simulated Quantum Annealing

The *simulated quantum annealing* (SQA) optimisation algorithm[2] takes inspiration from the physical process of adiabatic quantum computing or quantum annealing (Sect. 22.6.2). It uses quantum fluctuations rather than the thermal fluctuations of SA to explore the landscape. Again, a candidate solution is regarded as a system 'state' and the objective function value is regarded as 'energy', i.e., the Hamiltonian. Simulated quantum annealing was proposed by Finnila et al. in 1994 [190].

SQA is loosely based on the phenomenon of quantum tunnelling (Sect. 22.4.4). In optimisation terms, this corresponds to making a jump on a landscape and this capability can allow a solution to 'tunnel' through a (narrow) barrier on the landscape in one jump, rather than having to climb over the barrier via a succession of wrong-way moves as in SA. In some landscapes (Fig. 23.1), such as those that commonly arise in combinatorial optimisation, thermal annealing inspired algorithms such as SA can find it difficult to escape from poor-quality local minima, as they may be surrounded by high, though possibly narrow, barriers. Thermal fluctuations only see the height of the barrier. There is also the problem of entropy: the number of configurations grows exponentially with the number of variables: 2^n configurations for n Ising spins. A classical algorithm can only examine one configuration at a time; if there is no 'guiding' gradient to roughly steer the algorithm from any start point towards the global optimum (as happens, e.g., in golf course or deceptive landscapes), the algorithm must visit a significant fraction of the 2^n possible configurations. In this case, SA is not much better than random search.

In the SQA algorithm, as in SA, the current state x is replaced by a randomly selected neighbour state x' if x' has lower energy. The process is

[2]We refer to this algorithm as the Simulated Quantum Annealing (SQA) algorithm, to avoid confusion with the physical quantum annealing process; but, particularly in the physics literature, the name 'Quantum Annealing' seems to be used for both meanings.

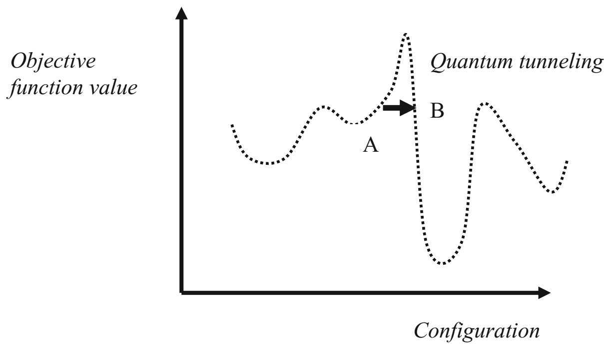


Fig. 23.1. Illustration of quantum tunnelling on a landscape. SA would require several 'wrong way' moves to jump over the barrier between the local (A) and the global optimum (B) basins. With a quantum tunnelling mechanism, this movement can potentially occur in one step

controlled by the tunnelling field strength, a parameter which controls the size of the neighbourhood explored by the algorithm. The field strength begins high, so the neighbourhood extends through the full search space. This permits long jumps on the landscape, facilitating explorative search. The tunnelling field strength is slowly reduced during the run, promoting local exploitation. Eventually the neighbourhood shrinks to just a few states close to the current state.

It is important to note that both SA and SQA are based on the physics of adiabatic processes, that is, on systems which remain at or near thermodynamic equilibrium throughout their evolution. Furthermore, SA and SQA are both based on the Metropolis algorithm, with the SA temperature parameter playing a role similar to that of the SQA tunnelling strength. Thus they are both stochastic local search algorithms and both generally use a perturbative approach to neighbour selection. However, in standard SA (as opposed to adaptive SA) the neighbourhood radius remains the same during the search, and the temperature determines the probability of moving to a higher energy state, whereas in SQA, the tunnelling field strength controls instead the neighbourhood radius, i.e., the average distance between the current state x and the next x'. The possibility of tunnelling through barriers can be shown to enhance the 'ergodicity' (Sect. 22.3.2) of SQA (compared to SA, which relies on thermal fluctuations) and so improve landscape exploration [130]. The appropriate definition of the tunnelling field depends on the nature of the problem being considered.

SQAs have been applied to a variety of combinatorial problems and [40, 392] illustrate their application to the TSP. A comprehensive discussion of

these algorithms, along with several applications, can be obtained in [129] and [130].

23.2.1 Implementation of SQA

Implementing SQA means using adjustable quantum fluctuations in the problem rather than thermal ones. For a given optimisation problem, the main tasks in developing an SQA approach are:

i. how to describe (and, usually, simplify) the Hilbert space for the problem. Real-world optimisation problems generally involve enormously large-dimension Hilbert spaces, so an alternative Quantum Monte Carlo approach, often using path integrals, is required; and
ii. how to formulate a quantum Hamiltonian $\mathcal{H} = \mathcal{H}_{\text{pot}} + \mathcal{H}_{\text{kin}}$, where the potential energy term $\mathcal{H}_{\text{pot}}$ is typically the cost or other objective function to be minimised, and the kinetic energy term is a perturbation term: a noncommuting operator analogous to the physical transverse field. It is useful to write $\mathcal{H} = \mathcal{H}_{\text{pot}} + \Gamma(t)\mathcal{H}_{\text{kin}}$, thus explicitly showing the time dependence of the tunnelling field strength: $\Gamma(t) \to 0$ as $t \to \infty$.

Thus SQA is more flexible than SA since not only can the ‘annealing schedule’ be chosen, but there may be a range of choices for the kinetic term which is then multiplied by $\Gamma(t)$.

23.2.2 SQA Application to TSP-Type Problems

An instance of the TSP is an (undirected) graph $G = (V, E)$ with $|V| = n$, together with a set of weights $\{d_{ij} : i, j = 1, \ldots, n\}$ some of which may be infinite in general. The problem is to find a tour of minimum length which starts and ends at the same vertex, and visits each vertex exactly once. This can be phrased as minimising an Ising Hamiltonian, as follows (see [392, 549] for further details).

A given tour can be represented by an $n \times n$ matrix $T = (t_{ij})$ with: $t_{ij} = 1$ if edge ij is on the tour (i.e., city j is visited immediately after city i); and $t_{ij} = 0$ otherwise. Thus a valid tour has a T matrix with exactly one 1 in each row and in each column, all other entries being 0. For an undirected graph, a tour and its reverse have the same length, and we consider the undirected tour matrix $U = (u_{ij}) = T + T'$ where T' (representing the reverse tour of T) is the transpose of T. U is a symmetric matrix with precisely two 1s in each row and column. The length of a tour is:

$$\mathcal{H}_{\text{pot}} = \frac{1}{2} \sum_{i,j=1}^{n} d_{ij} u_{ij} \tag{23.2}$$

Defining Ising spins by $S_{ij} = 2u_{ij} - 1$, we can rewrite this Hamiltonian in terms of S_{ij} as:

$$\mathcal{H}_{\text{pot}} = \frac{1}{2} \sum_{i,j=1}^{n} d_{ij} \frac{1 + S_{ij}}{2} \tag{23.3}$$

which is similar to the Hamiltonian of noninteracting Ising spins on an $n \times n$ lattice, with fields d_{ij} on the lattice points. The frustration is caused by the global constraints on the spin configurations that force them to conform with the structure of U.

$\mathcal{H}_{\text{kin}}$ is then chosen (with a high degree of arbitrariness) to induce fluctuations generating the important elementary moves of the problem, namely, changing a valid tour into another valid tour. Deciding which configurations are to be neighbours of which others is crucial, since it determines the problem's effective landscape. The 2-opt perturbation can be used as it is guaranteed to transform a valid tour into another valid tour. Here, if ab and cd are edges not incident with a common vertex, then they are deleted and replaced in the tour by ac and bd. The whole 2-opt move can be represented (in U-matrix terms) by four spin-flip operators:

$$S^{+}_{\langle c,a \rangle} S^{+}_{\langle d,b \rangle} S^{-}_{\langle b,a \rangle} S^{-}_{\langle d,c \rangle} \tag{23.4}$$

where each $S^{\pm}_{\langle i,j \rangle}$ is defined to flip an Ising spin variable (defined as $S_{\langle i,j \rangle} = 2u_{ij} - 1 = \pm 1$) at position (i,j) and at the symmetric position (j,i); that is, $S^{\pm}_{\langle i,j \rangle} = S^{\pm}_{i,j} S^{\pm}_{j,i}$. Then we get a (time-dependent) TSP quantum Hamiltonian implementing the 2-opt moves:

$$\begin{aligned} \mathcal{H}_{\text{TSP}} &= \mathcal{H}_{\text{pot}}(U) + \mathcal{H}_{\text{kin}} \\ &= \sum_{\langle ij \rangle}^{n} d_{ij} \frac{1 + S_{ij}}{2} - \frac{1}{2} \sum_{\langle ij \rangle} \sum_{\langle i',j' \rangle} \Gamma(i,j,i',j';t) S^{+}_{\langle i,i' \rangle} S^{+}_{\langle j,j' \rangle} S^{-}_{\langle j,i \rangle} S^{-}_{\langle j',i' \rangle} \end{aligned} \tag{23.5}$$

Γ is real and positive and depends not only on the edges but also on time.

An actual Schrödinger annealing evolution is computationally impractical, because of the large Hilbert space, so Metropolis-type Monte Carlo methods as mentioned earlier are used. The kinetic part of the quantum Hamiltonian as given requires calculation of very many matrix entries of exponential operators in the Trotter discretised system, which is computationally very expensive. For this reason, it is replaced by a standard transverse Ising form, which is trivially Trotter-discretised:

$$\tilde{\mathcal{H}}_{\text{TSP}} = \sum_{\langle ij \rangle}^{n} d_{ij} \frac{1 + S_{ij}}{2} - \Gamma(t) \sum_{\langle ij \rangle} [S^{+}_{\langle j,i \rangle} + \text{H.c.}] \tag{23.6}$$

As it stands, this simpler kinetic term is not guaranteed to move from a valid tour to another valid tour in the search space, but this can be ensured by exclusively using 2-opt moves to generate new states in the Monte Carlo algorithm. It can be shown that this simplification still gives a working SQA.

In testing on the `pr1002` instance of the standard benchmark TSPLIB [392, 549], SQA is found to anneal more efficiently than SA. However, SQA does not appear to be uniformly superior to SA. SA outperforms SQA on the Random Boolean Satisfiabilty (3-SAT) problem [549]; and recent research [516] investigating extensions of the approach to the multivisit TSP and bottleneck TSP appear to show sensitivity of SQA to the particular flavour of TSP addressed, and to the network structure.

The conclusion seems to be that SQA has the potential to outperform SA, but may require domain knowledge in order to build the kinetic energy term.

23.3 Constrained Molecular Dynamics Algorithm

The constrained molecular dynamics (CMD) algorithm was introduced by Poli and Stephens in 2004 [515]. It is related to the field-based algorithms discussed in Sect. 23.4 below in that it explicitly uses (a model of) a gravitational field as one of the influences in the algorithm. However, it models other effects as well.

This optimisation algorithm was inspired by the area of molecular dynamics which seeks to gain understanding of the properties of interacting physical bodies such as atoms and molecules.[3] A multitude of physical forces act on these bodies, which are generally treated as particles (i.e., of negligible size). The theoretical analysis of these systems is difficult, because of the number of particles involved: even in classical mechanics the three-body problem has no analytical solution, because solving the Lagrangian dynamical equations would involve 18 integrations.[4] Instead, in molecular dynamics, the properties of these systems are typically studied using computer simulations. In implementing these simulations, differing levels of fidelity to real-world interactions of particles (for example, classical forces of molecular mechanics, quantum effects, etc.) can be employed, depending on the objective and the desired accuracy of the simulation.

In CMD, the physics of masses and forces, that is, classical (Newtonian) mechanics, is used to guide a search process on a landscape. A population of particles is allowed to move across a landscape and their movement and interactions are governed by a small subset of possible real-world forces, namely gravity, friction, and coupling forces (springs). The movement of the particles is constrained so that they slide across the fitness landscape. Thus the CMD

[3] It uses the tensor formalism of Grassmann as applied by Einstein and Grossmann to gravitation, and also appears to derive inspiration from the 'rubber-sheet' geometry of spacetime deformed by the presence of massive bodies. Thus it seems to be one of the few algorithms in the literature to be influenced by (general) relativity, though there is no equivalent of the concept of bodies' speeds in a reference frame being bounded by the speed of light.

[4] There are five special cases, found by Lagrange, where an analytic solution (in elementary functions) is possible.

algorithm is one of the few optimisation algorithms that explicitly use the concept of a physical field.

The particles are acted upon by gravity, which tends to pull them to lower points on the landscape, assuming that the goal is to find the global minimum point on the landscape. However, if the only force acting on each particle were gravity, this would produce a local optimising algorithm where each particle could only walk downhill in its base of attraction. To avoid this, each particle i, of mass m_i and position vector x_i (where $i = 1, \ldots, n$), is given a momentum[5] $m_i v_i = m_i \dot{x}_i$, and thus a resulting kinetic energy $\frac{1}{2} m_i v_i^2$, which can help it to escape local optima. It is also linked to a set of other particles via a spring mechanism. Hence, the movement of one particle affects the movement of others, as linked particles are accelerated towards one another. This can help particles to escape from low fitness areas of the landscape. Each particle is subject to friction as it moves over the landscape and this acts to retard its movement, promoting focussed search. Hence, different friction values can be used to promote or inhibit a particle's exploration of the landscape.

As mentioned, an analytic solution is generally impossible, so numerical approaches are used. In such approaches, derivatives are approximated by finite differences. The CMD algorithm uses forward differences, for example, the time derivative $\dot{x}(t) = v(t)$ of position $x(t)$ at time t (that is, the velocity at time t) is approximated by the forward finite difference:

$$v(t) \approx \frac{x(t + \delta t) - x(t)}{\delta t} \tag{23.7}$$

where δt is the size of the timestep in the discretisation. Rearranging this gives $x(t + \delta t) = x(t) + v(t) \cdot \delta t$, which is used in an iteration of the algorithm. A similar approximation is used for acceleration[6] $a(t) = \dot{v}(t) = \ddot{x}(t)$, which gives the iteration step for $v(t + \delta t) = v(t) + a(t) \cdot \delta t$. Algorithm 23.2 provides a high-level outline of the CMD algorithm.

In each iteration of the algorithm, each particle i is moved from its current position x_i by applying to it a position change vector, namely $v_i \cdot \delta t$, the current velocity vector multiplied by the time step. In turn, this velocity vector is changed by an amount $a_i \cdot \delta t$. Here, a_i is the particle's acceleration; by Newton's third law, this is the ratio F_i/m_i (of the force F_i acting on the particle to the particle's mass m_i). F_i may be a vector sum of forces from several broad classes:

- no external forces (that is, the force on the particle arises purely from the constraints on its motion; in other words, from the geometry of the landscape);

[5] Recall that differentiation with respect to time t is commonly denoted by a dot over the variable, so that $\dot{x}_i$ denotes the velocity of particle i.

[6] Newton's dot notation is extended to denote the second derivative with respect to time t by a double dot over the variable, so that $\ddot{x}_i$ denotes the acceleration (that is, the time derivative of velocity) of particle i.

Algorithm 23.2: Constrained Molecular Dynamics Algorithm

```
Randomly generate an initial population of n solutions;
repeat
    for i = 1 to n do
        a_i = F_i(fitness surface, x_1, ..., x_n, v_i)/m_i;
        v_i = v_i + a_i · δt;
        x_i = x_i + v_i · δt;
    end
until terminating condition;
```

- forces due to particle-particle interactions (i.e., tensions in connecting 'springs', or fields generated by other particles, which depend on all the particles' positions);
- forces due to interactions with an external field, dependent on this particle's position; and
- friction/viscosity/damping type forces: generally a function of, and in the direction opposite to that of, the particle's velocity.

To indicate this, F_i is written above as $F_i(\text{fitness surface}, x_1, \ldots, x_n, v_i)$. The precise calculation of each particle's acceleration is described in [515] and interested readers are referred there for a full description (in tensor notation) of this step.

The fact that CMD searches the space using a population of particles makes it a global search algorithm. Other than the initial random choice of population, its action is deterministic (as are Newton's laws, on which it is based). There are some similarities between CMD and PSO in that both algorithms conduct a search of a space using concepts such as velocity, momentum and particle interaction. However, their source of inspiration differs (social interactions vs. molecular interactions) and particles in CMD have no memory and no explicit intelligence.

The CMD algorithm is an example of an optimisation algorithm whose design is inspired by properties of physical systems at a molecular level. Initial results from applying the algorithm to a series of test functions [515] indicate that it can solve optimisation problems and future work is awaited to further test the effectiveness and efficiency of the approach on other problems. So far, the only types of force considered in CMD appear to be 'square-law' fields (where the force decreases as the square of the distance; as do, for instance, the electromagnetic and gravitational forces), linear forces (the type of tension arising in a spring) and viscous/friction forces (proportional to the speed of movement). A future research direction would be to widen the scope to emulate other types of interaction arising in physics, e.g., the strong nuclear force (hadron colour force).

In addition to CMD, there have appeared in recent years a number of other approaches inspired by physical fields; some of these are discussed in the next section.

23.4 Physical Field Inspired Algorithms

A number of other approaches inspired by physical fields, some loosely, some more closely, have been introduced in the last decade or so; we look at some of these now. Their uses to date seem to be quite tightly coupled with the application domain; it is not clear whether it is possible to separate out the domain-specific processing into well-defined modules (as can be done, for example, in SA, where the neighbour generation routine is domain-specific), which would warrant describing these approaches as metaheuristics. Three examples are outlined in this section to illustrate this family of algorithms, namely central force optimisation, the gravitational search algorithm and the binary gravitational search algorithm.

23.4.1 Central Force Optimisation

Central Force Optimisation (CFO) [197, 198] is a nature inspired deterministic metaheuristic for multidimensional search and optimisation, based on the metaphor of gravitational kinematics. It seeks to locate the global optima of an objective or fitness function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ in a bounded search space $X \subseteq \mathbb{R}^n$. This f may be smooth, continuous or discontinuous, unimodal or highly multimodal, and may be subject to a set of constraints on the decision variables $x_1, \ldots, x_n$. CFO 'flies' a set of N_p 'probes' through the space during an interval of discrete time steps: the analogy is to particle masses moving in a gravitational field. Each probe's trajectory is governed by its initial position and the equations of motion arising from the total force on the probe. The probes are attracted to high mass regions of the space, which are analogous to regions of high fitness (high objective function value). CFO allows some decoupling and parameterisation of fitness and mass, since the mass is a user-defined function of the objective function value.

CFO is a populational algorithm, but is closer to particle swarm approaches than evolutionary algorithms, since CFO uses concepts of position, velocity and acceleration, and the information exchanged among probes is about the (phenotypical) fitness each sees, rather than any equivalent of genetic information. However, CFO's deterministic nature distinguishes it from PSO.

The location of probe i at time step t is given by its position vector $x^i(t) = (x^i_1(t), \ldots, x^i_n(t))$. Then its velocity $v^i(t) = \frac{1}{\Delta t}(x^i(t) - x^i(t-1))$ where Δt is the time interval from $t-1$ to t. The fitness ('mass') at a particular probe i's location at time step t is

$$M^i(t) = f(x^i(t)) = f(x_1^i(t), \ldots, x_n^i(t)).$$

By partial analogy with Newton's law of gravitational attraction, the 'gravitational' acceleration $a^{ij}(t)$ of probe i caused by another probe j at time t is given by a user-defined function of $M^i(t)$ and $M^j(t)$:

$$a^{ij}(t) = G \cdot \frac{U(M^j(t) - M^i(t)) \cdot (M^j(t) - M^i(t))^\alpha}{\|x^j(t) - x^i(t)\|^\beta}(x^j(t) - x^i(t)) \tag{23.8}$$

Here, G is a 'gravitational constant', α and β are user-chosen parameters, and U is the Heaviside step function $U : \mathbb{R} \longrightarrow \mathbb{R}$, where $U(x) = 1$ for $x \geq 0$ and $U(x) = 0$ otherwise (see Chap. 13, Fig. 13.5). Then the total acceleration of probe i at time t is the vector sum

$$a^i(t) = \sum_{j=1, j\neq i}^{N_p} a^{ij}(t) \tag{23.9}$$

of the accelerations due to each other probe in the space. This gives the new position at the next time step as $x^i(t+1) = x^i(t) + v^i(t)\Delta t + \frac{1}{2}a^i(t)(\Delta t)^2$.

The rationale given in [197] for using fitness differences $M^j(t) - M^i(t)$, rather than the actual fitness values, is "to avoid excessive gravitational 'pull' by other very close probes". However, this use of the difference $M^j(t) - M^i(t)$ is physically unrealistic, and can lead to negative masses and thus repulsive forces: to avoid this possibility, the Heaviside step function is used. In Newtonian gravity, the values of the exponents α and β are 1 and 3 respectively, but the user of CFO may choose to modify these and so depart from a physically realistic setup. CFO's convergence is found to be sensitive to the exponent values, and the values $\alpha = \beta = 2$ are used in [197], because they give reasonable performance.

The main steps of the CFO algorithm are given in Algorithm 23.3. In the algorithm, the time step Δt is taken to be 1 for simplicity, and all initial velocities are set to 0. Initially, the probes are uniformly distributed across the search space, though this may be adjusted based on domain knowledge. During the run, any errant probe whose velocity takes it outside the search space is returned to a location within the search space.

In [197], CFO is compared to other standard approaches on a number of test problems. It is found to give similar quality of solution, but requires fewer function evaluations. CFO may be susceptible to becoming trapped in a local optimum, but adaptive approaches may address this. In [527], evidence is presented that CFO does not perform well with random initialisation, and the population size and initial population have a significant effect on the results obtained. As the difficulty and dimension of the problem increases, CFO will be more sensitive to the population size and initialisation criteria: to obtain acceptable results, CFO must start with a large population size. The implication is that to set a suitable population size, we must either have some a priori information about the problem difficulty, or carry out a number of trial runs with different population sizes.

Algorithm 23.3: CFO Algorithm

$t := 0$;
Initialise the probes' position vectors at time 0, $x^1(0), \dots, x^{N_p}(0)$, to a uniform or similar distribution over the search space;
for *each probe* $i \in \{1, \dots, N_p\}$ **do**
  Set initial velocity $v^i(0) := 0$;
  Set initial acceleration $a^i(0) := 0$;
  Compute initial fitness $M^i(0) = f(x^i(0))$;
end
repeat
  for *each probe* $i \in \{1, \dots, N_p\}$ **do**
    Compute new probe position
    $x^i(t+1) = x^i(t) + v^i(t)\Delta t + \frac{1}{2}a^i(t)(\Delta t)^2$;
    if $x^i(t+1)$ *is outside the search space* X **then**
      Adjust $x^i(t+1)$ to relocate probe i within X;
    end
    Update fitness $M^i(t+1) = f(x^i(t+1))$;
    Set velocity $v^i(t+1) = x^i(t+1) - x^i(t)$;
    Compute acceleration $a^i(t+1)$ using Eqs. (23.8) and (23.9);
  end
until $t >$ *maximum time or other terminating condition is met*;

23.4.2 Gravitational Search Algorithm and Variants

The Gravitational Search Algorithm (GSA) was introduced in [527] and a binary-valued variant, Binary Gravitational Search Algorithm (BGSA), was presented in [528]. These are stochastic algorithms inspired by Newton's law of gravitational attraction.

GSA works on an isolated system of N masses or *agents*, which are particles in the search space $\mathbb{R}^n$, and attract each other by the gravitational force: thus, masses directly communicate. Each particle has four attributes: position, inertial mass, active gravitational mass, and passive gravitational mass (all masses are taken to be equal, by the Principles of Equivalence from General Relativity). The position $x^i = (x_1^i, \dots, x_n^i)$ corresponds to a solution of the problem, and a fitness function determines and adjusts the masses. Heavy particles correspond to good solutions and move more slowly than lighter ones. Over time, particles are attracted by the heaviest mass(es), which represents an optimum solution. Particles obey the following laws:

Law of gravity: Each particle i attracts each other particle j by a gravitational force between the two particles; this force is directly proportional to the product of their masses and inversely proportional to the Euclidean distance $R_{ij} = \|x^j - x^i\|_2$ between them. (R_{ij} is used instead of the more

physically realistic R_{ij}^2, because R_{ij} provides better results in all experiments.)

Law of motion: The acceleration, or (rate of) change in the velocity, of a particle is equal to the force acting on it divided by its inertial mass.

Suppose there are N particles (agents). The position of each particle i is written as a function of time, $x^i = x^i(t)$. At time t, the force acting on mass i from mass j is defined as follows:

$$F^{ij}(t) = G(t)\frac{M_i(t)M_j(t)}{R_{ij} + \varepsilon}(x^j(t) - x^i(t)) \tag{23.10}$$

where M_j is the (active) gravitational mass of agent j, M_i is the (passive) gravitational mass of agent i, $G(t)$ is the gravitational constant at time t, and ε is a small constant. Then the total force acting on i is:

$$F^i(t) = \sum_{j=1, j\neq i}^{N} F^{ij}(t) = \sum_{j=1, j\neq i}^{N} G(t)\frac{M_i(t)M_j(t)}{R_{ij} + \varepsilon}(x^j(t) - x^i(t)) \tag{23.11}$$

To give a stochastic characteristic to GSA, the authors scale each summand $F^{ij}(t)$ of the total force acting on agent i by a (different) random number r_j drawn from the interval [0, 1]:

$$F^i(t) = \sum_{j=1, j\neq i}^{N} r_j F^{ij}(t) = \sum_{j=1, j\neq i}^{N} r_j G(t)\frac{M_i(t)M_j(t)}{R_{ij} + \varepsilon}(x^j(t) - x^i(t)) \tag{23.12}$$

Hence, by Newton's laws, the acceleration of agent i at time t is $a^i(t) = F^i(t)/M_i(t)$. The new velocity of agent i is then defined as its current velocity plus its acceleration. The current velocity is multiplied by a random number r_i in the interval [0, 1] to give a randomised characteristic to the search. This gives the position and velocity of i as follows:

$$v^i(t+1) = r_i v^i(t) + a^i(t); \qquad x^i(t+1) = x^i(t) + v^i(t). \tag{23.13}$$

The gravitational constant, $G = G(G_0, t)$, is initialised to G_0 and is reduced over time to control the search accuracy. This is analogous to the temperature parameter in SA.

A heavier mass is interpreted as a fitter and so more efficient agent. Thus, better agents have higher gravitational attractions and move more slowly. The masses are calculated using the fitness, as follows:

$$m_i(t) = \frac{f_i(t) - \text{worst}(t)}{\text{best}(t) - \text{worst}(t)}; \qquad Mi(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)} \tag{23.14}$$

where $f_i(t)$ is the fitness value of agent i at time t, and worst(t) and best(t) are, respectively, the worst and best fitness of any agent at time t.

To enhance performance, GSA follows the approach that towards the beginning of the run, exploration should predominate, but as the run proceeds, the effort spent on exploration must decrease, and that on exploitation must increase. GSA controls the exploration-exploitation balance by allowing only the top $|K_{\text{best}}|$ agents to attract the others, where K_{best} is an ordered set containing agents ranked in decreasing order of fitness (mass). The set size $|K_{\text{best}}|$ is chosen to be a linearly decreasing function of time, with initial value K_0. To begin, all agents gravitationally attract all others, but as time passes, K_{best} decreases in size, and, at the end, only one agent gravitationally attracts the others. Thus, (23.12) can be written as:

$$F^i(t) = \sum_{j \in K_{\text{best}}, j \neq i} r_j F^{ij}(t) \tag{23.15}$$

The main steps of the Gravitational Search Algorithm are given in Algorithm 23.4.

Algorithm 23.4: Gravitational Search Algorithm

Identify search space;
Perform randomised initialisation;
repeat
    Perform fitness evaluation of agents;
    for $i = 1$ **to** N **do**
        Update $G(t)$, best(t), worst(t) and $M_i(t)$;
        Calculate the total force on agent i;
        Calculate acceleration and velocity;
        Update agent's position;
    end
until *terminating condition*;

GSA is noted to have the following characteristics [527]:

- higher performing agents have greater mass, so other agents tend to move towards the best agent(s);
- GSA is a memory-less algorithm;
- distinguishing between gravitational and inertial masses, though nonphysical, allows flexibility in GSA. A bigger inertial mass means less response to outside forces, a slower motion of the agent in the search space and so a more precise search. A bigger gravitational mass leads to greater forces on other agents, allowing faster convergence.

GSA contrasts with PSO [527]:

- In PSO, the direction of agent i is determined using only two best positions, p_i^{best} and g^{best}. However, in GSA, agent i's direction is determined using the overall force applied by all other agents;

- In PSO, updating is performed without considering solution quality or distance between agents, and fitness values are not important. However, in GSA the force is proportional to the fitness value, and inversely proportional to the distance between solutions, allowing the agents to perceive the search space around themselves through the force;
- PSO uses p_i^{best} and g^{best} as a memory for updating the velocity. However, GSA is memory-less.

GSA also contrasts with CFO [527]:

- In both algorithms, the positions and accelerations of the probes (agents) are inspired by particle motion in a gravitational field, but the precise formulations differ;
- CFO is deterministic while GSA is stochastic;
- GSA's and CFO's expressions for acceleration, motion and calculation of masses differ;
- The gravitational parameter G varies in GSA but is constant in CFO.

GSA is found to perform well compared to PSO and a real-valued GA (RGA), particularly on unimodal high-dimensional functions and multimodal functions [527].

The Binary Gravitational Search Algorithm (BGSA)

The Binary Gravitational Search Algorithm (BGSA) [528] was developed because it is natural to encode solutions as binary vectors in many problem types, such as feature selection, dimensionality reduction, data mining, unit commitment, and cell formation. The binary search space is considered as a hypercube $\subseteq \{0,1\}^n$: in this space, an agent may move to other corners of the hypercube by flipping various bits.

In the binary version of GSA, moving through a dimension means that the corresponding variable changes from 0 to 1 or vice versa. In BGSA, the updating procedures for force, acceleration and velocity are as for GSA, with the difference that in BGSA, position updating means switching between values 0 and 1. BGSA updates the velocity as in (23.13), and sets the current bit of the new position to be either 1 or 0 with a probability dependent on this velocity.

BGSA uses the principles that:

- a large absolute value of velocity means a nonoptimal position, so we must provide a high probability of changing the position of the agent with respect to its previous position (from 1 to 0 or vice versa);
- a small absolute value of the velocity indicates that the current position of the agent is close to optimal, so we must provide a small probability of changing the position. The agent's position is good and should not be changed.

Thus, we require a transfer function $S : \mathbb{R} \longrightarrow [0, 1]$ to map each velocity component (dimension) d to the probability of updating position, with the properties that for small $|v_d^i|$, the probability of changing x_d^i must be ≈ 0, and for a large $|v_d^i|$, the probability of changing x_d^i must be high. Thus, $S(v_d^i)$ must increase with increasing $|v_d^i|$. To achieve this, S is defined as the absolute value of tanh, the hyperbolic tangent:[7]

$$S(v_d^i) = |\tanh(v_d^i)| \tag{23.16}$$

Once $S(v_d^i)$ is computed, a random number $r \in [0, 1]$ is generated. Then the agent moves as follows:

$$x_d^i(t+1) = \begin{cases} x_d^i(t) & \text{if } r \geq S(v_d^i(t+1)) \\ 1 - x_d^i(t) & \text{otherwise.} \end{cases} \tag{23.17}$$

The second case is simply complementing the bit $x_d^i(t)$. To achieve a good rate of convergence, the velocity $|v_d^i|$ is limited to at most $v_{\max}$, with $v_{\max}$ chosen to be 6, based on experimental work.

On a suite of 25 test functions, BGSA obtains competitive results compared to a GA and Binary PSO [528].

23.4.3 Differences Among Physical Field-Inspired Algorithms

In [527] GSA explicitly avoids using a 'square-law' field, where the force decreases as the square of the distance, as occurs in nature in the electromagnetic and gravitational forces. Instead, it uses a field where the force decreases linearly with distance, because this is found to give better performance, experimentally. Furthermore, in [527], the difference of the position vectors of the two bodies is divided by its own norm: thus, it is normalised to a unit vector in that direction, so the magnitude of the force only depends on the masses of the two particles. In addition, only the k particles of greatest mass are considered, which is an arbitrary choice and departs from physical behaviour. Similarly, in sample applications, [197] chose to emphasise the role of mass (by raising the mass difference to the power $\alpha = 2$ rather than 1) and de-emphasise the role of separation (by raising the norm of the difference of position vectors to the power $\beta = 2$ rather than 3 as is physically more realistic).

Thus, the degree of agreement with nature of CFO, GSA and related algorithms is much looser than that of CMD (Sect. 23.3) or the approaches of [546], each of which uses a 'square-law' of the separation distance.

[7] Note that tanh is also used as a squashing function in multilayer perceptrons: see Chap. 13, Fig. 13.4.

23.5 Extremal Optimisation Algorithm

Extremal Optimisation (EO) is an optimisation (meta)heuristic which uses local search. It is based on the statistical physics idea of *self-organised criticality* (SOC), in particular the Bak-Sneppen model [23, 24]. This idea describes a class of dynamical systems that have a critical point as an attractor. This class consists of nonequilibrium systems which evolve by way of sudden large changes ('avalanches'). This use of ideas from nonequilibrium physics marks this approach as distinctly different from equilibrium physics approaches such as SA and SQA. The Bak-Sneppen model of SOC has been applied to such diverse phenomena as the dynamics of sand piles, natural evolution via punctuated equilibrium (mass extinction events) and earthquakes. In the application to evolution, a number of species in a given environment are thought of as coevolving, with the weakest (most poorly adapted) species becoming extinct. The species are thought of as vertices in a graph, with an edge indicating that the fitness values of the two vertices connected are dependent on each other (e.g., the two species may compete for resources in the environment).

Boettcher and Percus [63, 64, 65] initially developed EO to attack combinatorial optimisation problems, based on the observation that critical points exist in NP-complete problems, while there exist many widely dispersed near-optimum solutions which are separated by barriers, causing hill-climbing algorithms to become trapped. EO attempts (as do SA and SQA) to escape such local optima. EO outperforms SA on certain graph partitioning problems (for example, bipartitioning problems) [63]. It has since been applied to (among other things) general optimisation [66] and spin glasses [411].

EO is an iterative approach that works with a single candidate solution at a time, unlike population-based approaches. A solution in EO is a configuration of n components, analogous to the set of species coevolving (interacting) in a given environment; the components may be thought of as 'building blocks' of the solution. The initial candidate solution may be randomly generated, or may be based on domain knowledge or the result of a previous search process. EO's main idea is to make local modifications to the 'worst' components. Thus EO is a perturbative local search. The worst component is replaced by a randomly selected component, which is accepted *whether or not* this is a good move.

Algorithm 23.5 gives an overview of EO (assuming a minimisation objective function). Here, a configuration (solution) S is made up of n components (variables) $x_1, \ldots, x_n$, and $0 \leq f_i \leq 1$ stands for the fitness of variable x_i. $C(S)$ is the cost of S and is usually, though not necessarily, taken to be a linear function of $f_1, \ldots, f_n$. N_S is the set of neighbours of S (valid moves from S), as in SA.

Thus, EO uses a fitness measure, but the measure must be applicable to individual components of the configuration. This requires an appropriate encoding of the problem. This assignment of varying quality measures to particular components is a major difference between EO and algorithms such as

Algorithm 23.5: Extremal Optimisation Algorithm

Generate initial configuration S of n components;
Set $S_{\text{best}} := S$;
repeat
  for $i = 1$ **to** n **do**
    Evaluate f_i for each variable x_i;
    Find j satisfying $f_j \leq f_i$ for all i, i.e., x_j has the 'worst fitness';
    Choose $S' \in N_S$ such that x_j must change;
    Set $S := S'$;
    if $C(S) < C(S_{\text{best}})$ **then**
      Set $S_{\text{best}} := S$;
    end
  end
until *terminating condition*;
Return S_{best} and $C(S_{\text{best}})$;

EAs and ACO. In particular, EAs only ever judge components indirectly, by evaluating the individual consisting of those components; while EO measures components directly and is a fine-grained search.

This approach gives a robust hill-climbing-like exploitation behaviour, with an exploration behaviour similar to multiple-restart search. Overall, solution quality plotted with time shows punctuated equilibrium-like effects, giving periods of gradual improvement interspersed with large drops in quality (crashes). These crashes allow the algorithm to escape local optima, and are an emergent effect of EO's random modification of the worst component. In [67], EO is compared to other optimisation metaheuristics such as SA, GAs, and Tabu Search; and there it is noted that

> by persistent selection against the worst fitnesses, EO quickly approaches near-optimal solutions. Yet, large fluctuations remain at late runtimes (unlike in SA ...) to escape deep local minima and to access new regions in configuration space.

23.6 Summary

This chapter presented an introduction to a number of algorithms which are inspired by the properties of physical systems. These algorithms illustrate how physical, as distinct from biological, phenomena can inspire the design of optimisation algorithms. In addition to algorithms which are solely derived from a physical inspiration, research has also emerged on a series of hybrid algorithms which draw loose inspiration from both physical and evolutionary processes. In the next chapter (Chap. 24) we introduce the best-known example of this, the quantum inspired evolutionary algorithm.

24

Quantum Inspired Evolutionary Algorithms

In this chapter we introduce a family of algorithms whose workings draw inspiration from aspects of quantum mechanics in order to develop a series of hybrid quantum evolutionary algorithms. Initially, the chapter provides a short introduction to quantum systems and then describes the design of both hybrid binary-valued and hybrid real-valued quantum evolutionary algorithms.

It is appropriate to here reiterate that an algorithm designer may choose the degree of faithfulness to nature of an algorithm, so as to result in an algorithm which is efficient on the problems to which it is applied, while not necessarily being a perfect copy of nature. As described in Sect. 22.5.2, Feynman [189] showed that, because of entanglement, no classical computer can efficiently simulate quantum systems, as it would need exponential time or resources to do so. Thus, the degree of inspiration we can take from quantum mechanics is perforce limited; and all algorithms in this chapter are at best partial approximations of nature. However, as our criterion of usefulness is not fidelity to nature but rather effectiveness in problem solving, we discuss them here: despite their physical incorrectness (particularly in working with a space of dimension $2n$ rather than the 2^n dimensions of the Hilbert space), they have given rise to a rich thread in the literature.

24.1 Qubit Representation

In the language of evolutionary computation a system of m qubits (Sect. 22.5.1) may be referred to as a *quantum chromosome*, and written as a matrix with two rows,

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_m \\ \beta_1 & \beta_2 & \dots & \beta_m \end{bmatrix}, \tag{24.1}$$

where (for a normalised system) we require that for each i, $\alpha_i^2 + \beta_i^2 = 1$. A key point about such quantum systems is that they can compactly convey information on a large number of possible system states. In classical bit strings,

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_24

a string of length m can represent 2^m possible states; but a quantum space of m qubits has 2^m *dimensions* (as a complex manifold).

Thus, a single qubit register of length m can *simultaneously* represent *all* possible bit strings of length 2^m, e.g., an eight-qubit system can simultaneously encode 256 distinct strings.

This suggests that it may be possible to modify standard evolutionary algorithms to work with very few or even a single quantum individual, rather than having to use a large population of solution encodings. The qubit representation can also help to maintain diversity during the search process of an evolutionary algorithm, due to its capability to represent multiple system states simultaneously.

The use of a *qubit representation* also opens up the possibility of creating hybrid quantum evolutionary algorithms which scale well while also having good diversity maintenance characteristics.

However, it must be noted that because a classical computer cannot efficiently simulate entanglement (Sect. 22.5.3), the degree of fidelity of the algorithms of this chapter to quantum computing algorithms is partial at best.

24.2 Quantum Inspired Evolutionary Algorithms (QIEAs)

The application of quantum concepts to design optimisation algorithms is currently an area of active research interest. For example, quantum inspired concepts have been applied to the domains of evolutionary algorithms [240, 241, 433, 671, 670], grammatical evolution [401], social computing [669], neurocomputing [208, 363, 635], and immunocomputing [301, 368]. A claimed benefit of these algorithms is that because they use a quantum inspired representation, they can maintain a good balance between exploration and exploitation. It is also suggested that they offer computational efficiencies as use of a quantum representation can allow the use of smaller population sizes than typical evolutionary algorithms. However, as noted above, they are not true quantum algorithms, cannot simulate entanglement, and must not be regarded as carrying over all of the abilities of quantum computing, such as quantum parallelism (Sect. 22.5.3).

This chapter concentrates on two distinct forms of quantum inspired evolutionary algorithms, the first working with binary encodings and the second working with real-valued encodings. The binary encoding version bears more fidelity to the quantum metaphor, though neither is completely faithful.

24.3 Binary-Valued QIEA

There are many ways that a quantum system could be defined in order to encode a set of binary (solution) strings. For example, in the following three-

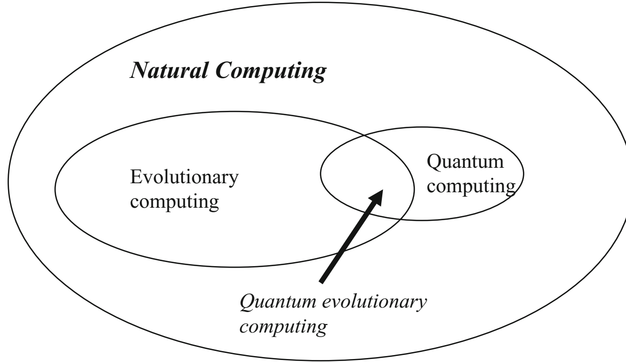


Fig. 24.1. Quantum inspired evolutionary computing

qubit quantum system, the quantum chromosome is defined using the three pairs of (probability) amplitudes below:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{\sqrt{3}}{2} & \frac{1}{2} \\ \frac{1}{\sqrt{2}} & \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix}. \tag{24.2}$$

The squared moduli of these numbers are the probabilities that a qubit will be observed in a particular eigenstate rather than another. Taking the first qubit, the occurrence of either state 0 or 1 is equally likely as both α_1 and β_1 have the same absolute value. From the definition of the three-qubit system, the (quantum) state of the system is given by:

$$\tfrac{\sqrt{3}}{4\sqrt{2}}|000\rangle+\tfrac{3}{4\sqrt{2}}|001\rangle+\tfrac{1}{4\sqrt{2}}|010\rangle+\tfrac{\sqrt{3}}{4\sqrt{2}}|011\rangle+\tfrac{\sqrt{3}}{4\sqrt{2}}|100\rangle+\tfrac{3}{4\sqrt{2}}|101\rangle+\tfrac{1}{4\sqrt{2}}|110\rangle+\tfrac{\sqrt{3}}{4\sqrt{2}}|111\rangle. \tag{24.3}$$

For intuition on this point, consider the system state $|000\rangle$. The associated probability amplitude for this state is derived from the probability amplitudes of the 0 state for each of the three individual qubits: $\frac{1}{\sqrt{2}} \cdot \frac{\sqrt{3}}{2} \cdot \frac{1}{2} = \frac{\sqrt{3}}{4\sqrt{2}}$. The associated probability of this state is then $\left(\frac{\sqrt{3}}{4\sqrt{2}}\right)^2 = \frac{3}{32}$. Working in this way, the associated probabilities of each of the individual states $|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle$ are $\frac{3}{32}, \frac{9}{32}, \frac{1}{32}, \frac{3}{32}, \frac{3}{32}, \frac{9}{32}, \frac{1}{32}, \frac{3}{32}$ respectively.

Algorithm 24.1 provides an example of a canonical binary QIGA [240]. Initially, the population of quantum chromosomes (which could be of size 1) is created as

$$Q(t) = \{q^1(t), q^2(t), \ldots, q^n(t)\},$$

where n is the population size, and each member of the population consists of m individual qubits (the superscripts here are just labels, not exponents). The α and β values for each qubit are set to $\frac{1}{\sqrt{2}}$ initially, in order to ensure

Algorithm 24.1: Binary Quantum Inspired Genetic Algorithm

Set $t = 0$;
Initalise $Q(t)$, the population of quantum chromosome(s);
for $j = 1$ *to* n **do**
  Create $p^j(t)$ by undertaking an observation of $Q(t)$;
end
Evaluate $P(t)$ and select the best solution from this population;
Store the best solution in $B(t)$;
while $t < t_{\max}$ **do**
  $t = t + 1$;
  Create $P^*(t)$ by undertaking observations of $Q(t-1)$;
  Evaluate the population of solutions $P^*(t)$;
  Compare the best solution in $P^*(t)$ with that of $B(t-1)$ and store the better of these in $B(t)$;
  Update $Q(t)$ using $B(t)$;
end

that the states 0 and 1 are equally likely for each qubit (i.e., the probability of either state 0 or 1 is $\left(\frac{1}{\sqrt{2}}\right)^2 = 0.50$). If there is domain knowledge that some states are likely to lead to better results, this can be used to seed the initial quantum chromosome(s). Once a population of quantum chromosomes is created, these can be used to create a population

$$P(t) = \{p^1(t), p^2(t), \ldots, p^n(t)\}$$

of binary strings (solution encodings) by performing 'observations' on the quantum chromosomes. One way of performing the observation step on qubit i of the jth individual $p^j(t)$ is to draw a random number $r \in [0, 1]$. If $r > |\alpha_i(t)|^2$, the corresponding bit $p_i^j(t)$ in the observed individual $p^j(t)$ is assigned state 1; otherwise, it is assigned state 0.

Because of the stochastic nature of the observation step, the QIGA could be implemented using a single quantum chromosome, where this chromosome is observed multiple times in order to generate the population $P(t) = \{p^1(t), p^2(t), \ldots, p^n(t)\}$. Alternatively, a small population of quantum chromosomes could be maintained, with each chromosome being observed a fixed number of times in order to generate $P(t)$.

In the **while** loop of Algorithm 24.1, an update step is performed on the quantum chromosome(s). This update step could be performed in a variety of ways, for example by using pseudogenetic operators, or by using a suitable quantum gate (see below). How ever the step is undertaken, its essence is that the quantum chromosome is adjusted so as to make it more likely that the best solution found so far will be generated in the next iteration of the algorithm. As the optimal solution is approached by the QIGA system, the values of each element of the quantum chromosome tend towards either 0 or 1, cor-

responding to a high probability that the quantum chromosome will generate a specific solution vector p^j when observed. For example, a quantum chromosome $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ will generate an observed solution chromosome of [0 0 0] with probability 1, regardless of the choice of the parameter r in the observation step.

24.3.1 Diversity Generation in Binary QIEA

Unlike the canonical GA, the binary QIGA does not typically employ distinct crossover and mutation operators. Instead a single diversity-generation step is used. There are two common approaches to this, the first drawing greater inspiration from quantum mechanics than the second. Each is described below.

Quantum Gates

The quantum equivalent of a classical operator on bits is a quantum *gate* or *evolution* (not to be confused with the concept of evolution in Evolutionary Algorithms). It transforms an input to an output, e.g., by a rotation, Hadamard or CNOT (controlled NOT) gate [576], and operates without measuring the value of the qubit(s). Thus, it effectively does a parallel computation on all the qubits at once and gives rise to a new superposition. Since the Hilbert space of a system of n qubits has finite dimension 2^n, an operator on the space (such as a gate) may be represented by a $2^n \times 2^n$ matrix. As before, we require that the matrix be unitary (in particular, self-adjoint), so that the updated coefficients α_i', β_i' for the ith qubit will still satisfy $|\alpha_i'|^2 + |\beta_i'|^2 = 1$.

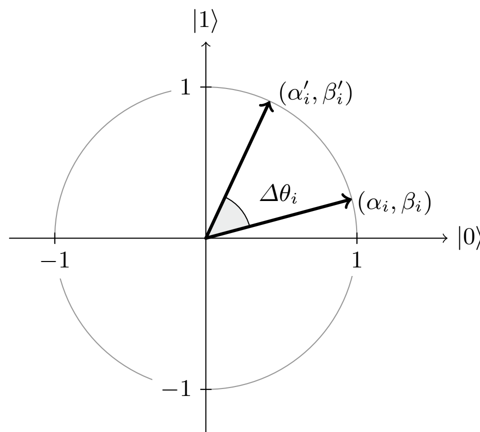


Fig. 24.2. Quantum rotation gate

This idea of quantum gate has been used successfully in QIEAs. However, to date, researchers appear to exclusively use real coefficients (as, for example, in Fig. 24.2), rather than the complex coefficients arising in Quantum Mechanics; therefore, a possible extension of these algorithms would be to use complex coefficients and take absolute values during an observation.

In the mutation step, the intent is to adjust the values on the quantum chromosome (the quantum probability amplitudes) so that fitter individuals are more likely to be observed from the quantum chromosome in the next iteration of the algorithm. This adjustment is undertaken using information from the best-performing individual in the (nonquantum) population and the probability amplitudes on the quantum chromosome are altered using a quantum rotation gate as per (24.4).

$$\begin{bmatrix} \alpha_i' \\ \beta_i' \end{bmatrix} = \begin{bmatrix} \cos(\Delta\theta_i) & -\sin(\Delta\theta_i) \\ \sin(\Delta\theta_i) & \cos(\Delta\theta_i) \end{bmatrix} \begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} \tag{24.4}$$

In order to determine the appropriate value for $\Delta\theta$ so that mutation is steered towards the best current member of the observed population, a look-up table is used. One simple way of implementing this is to create a binary chromosome x, using the current quantum chromosome q. Hence, $q = \begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_m \\ \beta_1 & \beta_2 & \dots & \beta_m \end{bmatrix}$ and for each element q_i in turn, if $\alpha_i^2 \leq 0.5$, then $x_i = 0$; otherwise $x_i = 1$. Table 24.1 provides the look-up table for $\Delta\theta$. The parameter a determines

Table 24.1. A look-up table for $\Delta\theta$

x_i	b_i	$\Delta\theta$
1	1	0
0	1	a
0	0	0
1	0	$-a$

the mutation step size. If a bit has the same value in both x_i and b_i, the best element, then the quantum chromosome is not altered. To illustrate this, consider the case where there is a single element in the quantum chromosome $q = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$. As $\alpha_1^2 > 0.5$, $x_1 = 1$ and assume $b_1 = 1$. Therefore, $\Delta\theta = 0$ according to the look-up table. Applying the quantum gate (24.4), the values for $\begin{bmatrix} \alpha_1 \\ \beta_1 \end{bmatrix}$ remain unchanged, since $\begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} = \begin{bmatrix} \cos(0) & -\sin(0) \\ \sin(0) & \cos(0) \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$.

Alternatively, if $x_i > b_i$, the corresponding value of the quantum chromosome is reduced by $-a$, making the observation of a '1' when looking at the quantum chromosome less likely in the future. While Table 24.1 illustrates a simple quantum gate, it does not explicitly consider the quadrant that $\begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix}$ is

in (α_i and/or β_i could be negative). Therefore, a variant on the above method for updating the ith qubit value $\begin{bmatrix}\alpha_i \\ \beta_i\end{bmatrix}$ is as follows:

$$\begin{bmatrix}\alpha'_i \\ \beta'_i\end{bmatrix} = \begin{bmatrix}\cos(\xi(\Delta\theta_i)) & -\sin(\xi(\Delta\theta_i)) \\ \sin(\xi(\Delta\theta_i)) & \cos(\xi(\Delta\theta_i))\end{bmatrix} \begin{bmatrix}\alpha_i \\ \beta_i\end{bmatrix}, \tag{24.5}$$

where $\xi(\Delta\theta_i) = s(\alpha_i, \beta_i) * \Delta\theta_i$. The terms $s(\alpha_i, \beta_i)$ and $\Delta\theta_i$ are the rotation direction and the angle respectively. A sample lookup table is illustrated in Table 24.2.

In order to provide some intuition on the table, consider the case where x_i and b_i are 1 and 0 respectively, and the condition $f(x) > f(b)$ is false. If the qubit is located in the second or the fourth quadrant, the adjustment $s(\alpha_i, \beta_i) * \Delta\theta_i$ should be positive in order to increase the probability of occurrence of the state '0'. Conversely, if the qubit is located in the first or the third quadrant, the adjustment $s(\alpha_i, \beta_i) * \Delta\theta_i$ should be negative in order to increase the probability of occurrence of the state '0'. In cases where $x_i = b_i$, the value of $s(\alpha_i, \beta_i) * \Delta\theta_i$ is set to 0. The step size for $\Delta\theta_i$ is problem-specific and it

Table 24.2. A rotation gate lookup table

x_i	b_i	$f(x) > f(b)$	$\Delta\theta_i$	$s(\alpha_i, \beta_i)$			
				$\alpha_i\beta_i > 0$	$\alpha_i\beta_i < 0$	$\alpha_i = 0$	$\beta_i = 0$
0	0	False	0	0	0	0	0
0	0	True	0	0	0	0	0
0	1	False	Delta	+1	−1	0	±1
0	1	True	Delta	−1	+1	±1	0
1	0	False	Delta	−1	+1	±1	0
1	0	True	Delta	+1	−1	0	±1
1	1	False	0	0	0	0	0
1	1	True	0	0	0	0	0

controls the speed of convergence of the algorithm. The step size is usually set to a small value, for example, [240, 648] suggest $\Delta = 0.01\pi$, although trial and error may be required to set a good value.

Quantum Mutation

A simpler form of mutation (24.6) and (24.7) which draws more inspiration from the standard GA mutation operator than from an 'evolution' in quantum systems is defined by [671, 670]:

$$Q_{\text{pointer}}(t) = a * B_{\text{best}}(t) + (1 - a) * (1 - B_{\text{best}}(t)) \tag{24.6}$$

$$Q(t+1) = Q_{\text{pointer}}(t) + b * r \tag{24.7}$$

where $B_{\text{best}}(t)$ is the best solution found by iteration t and $Q_{\text{pointer}}(t)$ is a temporary quantum chromosome which is used to guide the generation of $Q(t+1)$ towards the form of B_{best}. The term r is a random number drawn from a $N(0,1)$ normal distribution. The parameters a and b control the balance between exploration and exploitation, with a governing the importance attached to $B_{\text{best}}(t)$ and b governing the degree of variance generation, centred on $Q_{\text{pointer}}(t)$. Values of $a \in [0.1, 0., 5]$ and $b \in [0.05, 0.15]$ are suggested by [670, 671]. Another simple method for adapting in the quantum chromosome [670] is as follows:

$$\alpha(t+1) = [2 - B_{\text{best}}(t)] * 0.5 \tag{24.8}$$

Under this mechanism, the bits in the current best individual will be more likely to be generated during observation of the quantum chromosome in the next iteration of the algorithm.

24.4 Real-Valued QIEA

An alternative, real-valued, QIEA methodology was suggested by da Cruz et al. [126]. Like the binary-valued QIGA, real-valued QIGA maintains a distinction between a quantum population and an observed population of, in this case, real-valued solution vectors. However, in this approach, the quantum population $Q(t)$ is comprised of N quantum individuals q_i with $i = 1, 2, \ldots, N$, where each individual i is comprised of G genes q_{ij} with $j = 1, 2, \ldots, G$. Each of these genes is a pair $q_{ij} = (p_{ij}, \sigma_{ij})$ of real values where p_{ij} and σ_{ij} respectively represent the mean and the width of a square pulse. Representing a gene in this manner has a loose parallel with the quantum concept of superposition of states as a gene is specified by a range of possible values, rather than by a unique value, and the act of observing a particle (here, periodically sampling from an associated distribution) can be regarded as a wave function (quantum state) collapsing to a classical state upon observation. The real-valued QIEA, assuming that a single quantum chromosome is employed, is outlined in Algorithm 24.2. Note that the population $C(t)$ is maintained between iterations of the algorithm.

24.4.1 Initialising the Quantum Population

A *quantum chromosome*, which is observed to give a specific solution vector of real numbers, is made up of several quantum genes. The number of genes is determined by the required dimension of the solution vector. At the start of the algorithm, each quantum gene is initialised by randomly selecting a value from within the range of allowable values for that dimension. A gene's width value is set to the range of allowable values for the dimension. For example, if the known allowable values for dimension j on the solution vector are $[-75, 75]$ then q_{ij} (dimension j in quantum chromosome i) is initially determined by

Algorithm 24.2: Real-Valued Quantum Inspired Genetic Algorithm

Set $t = 0$;
Initialise the quantum chromosome $Q(t)$;
while $t < t_{\max}$ **do**
  Create the PDFs and corresponding CDFs for each gene locus using the quantum individual;
  Create a temporary population $E(t)$ of K real-valued solution vectors through a series of 'observations' via the CDFs;
  if $t = 0$ **then**
    $C(t) = E(t)$;
  else
    Let $E(t) =$ outcome of crossover between $E(t-1)$ and $C(t-1)$;
    Evaluate $E(t)$;
    Let $C(t) = K$ best individuals from $E(t) \cup C(t-1)$;
    Evaluate $C(t)$;
  end
  With the N best individuals from $C(t)$;
  $Q(t+1) =$ Output of translate operation on $Q(t)$;
  $Q(t+1) =$ Output of resize operation on $Q(t+1)$;
  $t = t + 1$;
end

randomly selecting a value from this range, (say) -50. The corresponding width value will be 150. Hence, $q_{ij} = (-50, 150)$. The square pulse need not be entirely within the allowable range for a dimension when it is initially created as the algorithm will automatically adjust for this as it executes. The height of the pulse arising from a gene j in chromosome i is calculated using

$$h_{ij} = \frac{1/\sigma_{ij}}{N} \tag{24.9}$$

where N is the number of individuals in the quantum population (of course, N could be set at 1). This equation ensures that the probability density functions (PDFs) (see next subsection) used to generate the observed individual solution vectors will have a total area equal to 1. Figure 24.3 provides an illustration of a quantum gene where $N = 4$.

24.4.2 Observing the Quantum Chromosomes

In order to generate a population of real-valued solution vectors, a series of observations must be undertaken using the population of quantum chromosomes (individuals). A pseudointerference process between the quantum individuals is simulated by summing up the square pulse for each individual gene across all members of the quantum population. This generates a separate probability density function (PDF) — just the sum of the square pulses — for each gene

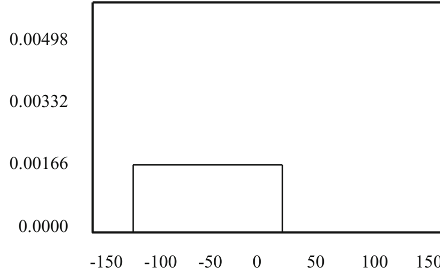


Fig. 24.3. A square pulse, representing a quantum gene, with a width of 150, centred on -50. The height of the pulse is $\frac{1}{150} = 0.00166\dot{6}$.

and (24.9) ensures that the area under this PDF is 1. Hence, the PDF for gene j on iteration t is:

$$\mathrm{PDF}_j(t) = \sum_i^j q_{ij} \tag{24.10}$$

where q_{ij} is the square pulse of the jth gene of the ith quantum individual (of N). To use this information to obtain an observation, the PDF is first converted into its corresponding Cumulative Distribution Function (CDF):

$$\mathrm{CDF}_j(x) = \int_{L_j}^{x} \mathrm{PDF}_j(t)dt \tag{24.11}$$

where U_j and L_j are the upper and lower limits of the probability distribution. By generating a random number r from the interval (0,1), the CDF can be used to obtain an observation of a real number x, where $x = \mathrm{CDF}^{-1}(r)$. If the generated value x is outside the allowable real-valued range for that dimension, the generated value is limited to its allowable boundary value. A separate PDF and CDF is calculated for each of the G gene positions. Once these have been calculated, the observation process is iterated to create a temporary population with K members, denoted $E(t)$.

24.4.3 Crossover Mechanism

The crossover operation takes place between $C(t)$ and the temporary population $E(t)$. This step could be operationalised in a variety of ways with [126] choosing to adopt a variant of uniform crossover. After the K crossover operations have been performed, with the resulting children being copied into $E(t)$, the best K individuals $\in C(t-1) \cup E(t)$ are copied into $C(t)$.

24.4.4 Updating the Quantum Chromosomes

The N quantum chromosomes are updated using the N best individuals from $C(t)$ after performing the crossover step. Each quantum gene's mean value is altered using:

$$p_{ij} = c_{ij} \tag{24.12}$$

so that the mean value of the jth gene of the ith quantum chromosome is given by the corresponding j^{th} value of the ith ranked individual in $C(t)$ (the 'translate' step in Algorithm 24.2).

The next step is to update the corresponding width value of the jth gene (the 'resize' step in Algorithm 24.2). The objective of this process is to vary the exploration/exploitation characteristics of the search algorithm, depending on the feedback from previous iterations. If the search process is continuing to uncover many new better solutions, then the exploration phase should be continued by keeping the widths relatively broad. However, if the search process is not uncovering many new better solutions, the widths are reduced in order to encourage finer-grained search around already discovered good regions of the solution space. There are multiple ways this general approach could be operationalised. For example, [126] suggests use of the $\frac{1}{5}$ mutation rule from Evolutionary Strategies [531] (Sect. 5.1.4), whereby we set:

$$\sigma_{ij} = \begin{cases} \sigma_{ij} g & \text{if } \phi < 1/5 \\ \sigma_{ij}/g & \text{if } \phi > 1/5 \\ \sigma_{ij} & \text{if } \phi = 1/5 \end{cases} \tag{24.13}$$

where σ_{ij} is the width of the ith quantum chromosome's j^{th} gene, g is a constant in the range $[0, 1]$ and ϕ is the proportion of individuals in the new population that have improved their fitness.

24.5 QIEAs and EDAs

Although the real-valued QIEA can claim a parallel with the quantum concepts of superposition of states and wave function collapse to a classical state upon observation (sampling from a gene's PDF), it draws less inspiration from quantum mechanics than its binary cousin as it does not use an analogue to a qubit representation. Also, the real-valued QIEA uses uniform distributions (which do not arise physically); and it only allows for constructive (not destructive) interference: furthermore, that interference is among wave functions of *different* individuals.

A key distinction between quantum inspired algorithms and other naturally inspired algorithms such as GA and PSO is that in quantum inspired algorithms, a probability vector (or a small population of these) rather than a population of individual solution vectors is manipulated by the algorithm and used to guide the exploration of the search space.

This approach has similarities to those in binary and continuous estimation of distribution algorithms (EDAs) (Sect. 4.7). Real-valued QIEAs in particular are strongly similar in design to univariate continuous EDAs, suggesting that the canonical real-valued QIEA would struggle in cases where there are substantial epistatic relationships between variables. There are also similarities between the binary-valued QIEA and binary EDAs (Sect. 4.7) such as population-based incremental learning (PBIL) [26] and the compact genetic algorithm (cGA) [252]. However, the mechanisms for generating diversity in the quantum chromosome, particularly the use of a pseudoquantum gate, are different from the typical diversity-generating mechanisms used in EDAs. Readers are referred to [509] for a good discussion of some of the linkages and differences between EDAs and quantum inspired algorithms.

24.6 Other Quantum Hybrid Algorithms

As already noted, quantum concepts have been applied to a number of different families of natural computing algorithms. An illustration of one of these hybrids, quantum binary PSO, is described now.

Quantum Binary PSO

Two versions of binary PSO, *BinPSO* and *Angle Modulated PSO*, were described in Sect. 8.6. An alternative version of binary PSO has been proposed by Yang et al. [669]. This approach draws inspiration from quantum mechanics in that the population of binary particles which encode the solutions is generated through observations of an underlying quantum particle.

The primary quantum concepts employed are the superposition of states and the collapse of the quantum state of a particle to a single classical state via the process of observation. Unlike many applications of quantum concepts to evolutionary algorithms, two populations of the same size are maintained: one population of binary encodings (the solutions); and a corresponding population of the same size of quantum particles. The population of quantum particles is updated from one iteration of the algorithm to the next by applying a variant of the PSO velocity update equation to the population of quantum particles. Each of the quantum particles is then observed in turn, in order to generate a corresponding binary particle whose fitness can be assessed.

The Algorithm

In the algorithm, a population of N quantum particle vectors, each of length m, is created, defined as follows:

$$Q(t) = \{q^1(t), q^2(t), \ldots, q^N(t)\}, \tag{24.14}$$

where

$$q^j(t) = (q_1^j(t), q_2^j(t), \dots, q_m^j(t)), \quad j = 1, 2, \dots, N \tag{24.15}$$

and $q_i^j(t)$ denotes the probability of the ith bit of the jth particle being 0 at time t, for each $i = 1, 2, \dots, m$, and each $j = 1, 2, \dots, N$. Therefore, the range of possible values for each $q_i^j(t)$ is the closed interval $[0, 1]$. The population of binary particles is of the same size and dimensionality as the population of quantum particles; therefore:

$$X(t) = \{x^1(t), x^2(t), \dots, x^N(t)\}, \tag{24.16}$$

where

$$x^j(t) = (x_1^j(t), x_2^j(t), \dots, x_m^j(t)) \tag{24.17}$$

and $x_i^j(t) \in \{0, 1\}$ represents the binary value of particle j in position i corresponding to an observation of the quantum particle $q_i^j(t)$, for each $i = 1, 2, \dots, m$ and each $j = 1, 2, \dots, N$.

After the quantum population has been created, or updated as the algorithm executes, a population of binary-encoded particles is created by observing the corresponding quantum particle. Hence, an observation of the first quantum particle is used to generate the first binary particle, etc. The observation mechanism is as follows. For each $q_i^j(t)$ where $i = 1, 2, \dots, m$, $j = 1, 2, \dots, N$, generate a random number r in the range $[0, 1]$. If the random number is greater than $q_i^j(t)$, then $x_i^j(t) = 1$; otherwise, $x_i^j(t) = 0$.

In order to drive the search process in the population of quantum particles from one iteration of the algorithm to the next, a variant on the PSO velocity update step is applied. Initially, an m-dimensional (quantum) particle denoted as $q_{g^{\text{best}}}(t)$ is calculated using the global best binary particle $x_{g^{\text{best}}}(t)$,

$$q_{g^{\text{best}}}(t) = \alpha x_{g^{\text{best}}}(t) + \beta(1 - x_{g^{\text{best}}}(t)) \tag{24.18}$$

and then $q_{p^{\text{best}}}$ is calculated for each individual member of the quantum population using the $x_{p^{\text{best}}}(t)$ of their associated binary vector:

$$q_{p^{\text{best}}}(t) = \alpha x_{p^{\text{best}}}(t) + \beta(1 - x_{p^{\text{best}}}(t)) \tag{24.19}$$

The numbers α, β, where $\alpha + \beta = 1$, $0 < \alpha, \beta < 1$, are control parameters which determine the speed of adaptation towards g^{best} and p^{best}. Finally, each member of the quantum population is updated using:

$$q(t + 1) = wq(t) + c_1 q_{p^{\text{best}}}(t) + c_2 q_{g^{\text{best}}}(t) \tag{24.20}$$

where $w + c_1 + c_2 = 1$ and $0 < w, c_1, c_2 < 1$. Algorithm 24.3 lists the steps in the algorithm.

The quantum binary PSO algorithm is a hybrid of quantum and PSO concepts. Unlike the canonical real-valued PSO algorithm, two populations are maintained, and the hybrid algorithm replaces the two-step process of calculating velocity updates and then moving the population of particle, with a single (quantum) particle update step.

Algorithm 24.3: Quantum Binary Particle Swarm

```
Set t = 0;
Initialise the populations Q(t) of quantum particles and X(t) of binary
particles;
Evaluate the fitness of each (binary) individual in X(t);
Determine the binary particle with highest fitness, and store as x_gbest;

repeat
    t = t + 1;
    Update all individuals in Q(t) using Eqs. (24.18)–(24.20);
    Observe each individual in Q(t) in turn to get each individual in X(t);
    for each i ∈ {1, ..., N}, each j ∈ {1, ..., m} do
        Generate a random number r in the range [0, 1];
        if r > q_i^j then
            x_i^j = 1;
        else
            x_i^j = 0;
        end
    end
    Evaluate fitness of each individual in X(t);
    Update x_gbest for X(t) and individual x_pbest s if necessary;
until terminating condition;
```

24.7 Summary

Quantum effects are a natural phenomenon and, just like evolution, or immune systems, can serve as an inspiration for the design of computing algorithms. Thus far, the primary quantum concept 'borrowed' in designing quantum evolutionary algorithms is that of superposition of states. Although this has produced some interesting results, there is particular opportunity to further extend the range of quantum effects into hybrid algorithms. For example, it would be interesting to investigate whether some analogy of local confinement could be applied to nearby candidate solutions in a space being searched by an EA. Preservation of locality of reference could possibly protect good building blocks from damage by inhibiting mutation. A local confinement field could also enhance local exploration once a rough location had been found for a global optimum by an algorithm such as SA or an EA.

Part VII

Other Paradigms

25

Plant-Inspired Algorithms

In previous chapters we have introduced a wide array of natural computing algorithms and have illustrated how these fit into a broad taxonomy of algorithmic families. A recent addition to this taxonomy is a series of algorithms which are derived from studies of plant behaviours. In this chapter we initially outline a variety of interesting plant behaviours, several of which offer potential to inspire the design of computational algorithms. We then describe an illustrative sample of plant-inspired algorithms.

25.1 Plant Behaviours

Plants represent some 99% of the eukaryotic biomass of the planet and have been highly successful in colonising many habitats with differing resource potential. The success of plants in 'earning a living' suggests that they have evolved robust architectures and problem-solving mechanisms. Typically plants have a modular architecture (roots, branches, shoots and buds are modular) which makes them robust to damage. Just like animals or simpler organisms such as bacteria (Chap. 11), plants have evolved multiple problem-solving mechanisms including complex food foraging mechanisms, environmental-sensing mechanisms, and reproductive strategies. Although plants do not have a brain or central nervous system, they are capable of sensing environmental conditions and taking actions which are 'adaptive' in the sense of allowing them to adjust to changing environmental conditions. Examples of plant adaption to the environment include:

- foraging for light, water and other nutrients,
- an ability to defend against herbivores and other attackers, and the
- ability to 'remember' past events.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_25

25.2 Foraging

Plants, unlike animals, can create their own food. Most plants and algae use photosynthesis to convert carbon dioxide and water into a variety of organic compounds (particularly carbohydrates) using energy from sunlight, releasing oxygen as a waste product. The basic equation for photosynthesis is

$$CO_2 + H_2O + \text{sunlight} \rightarrow (CH_2O)_n + O_2, \tag{25.1}$$

with nitrogen playing an important role in proteins, which help catalyse this process. Despite the plentiful supply of nitrogen in the atmosphere (N_2) plants cannot directly use nitrogen in this form, and rely on absorbtion of nitrogen compounds from the soil through their root network for their required supply.

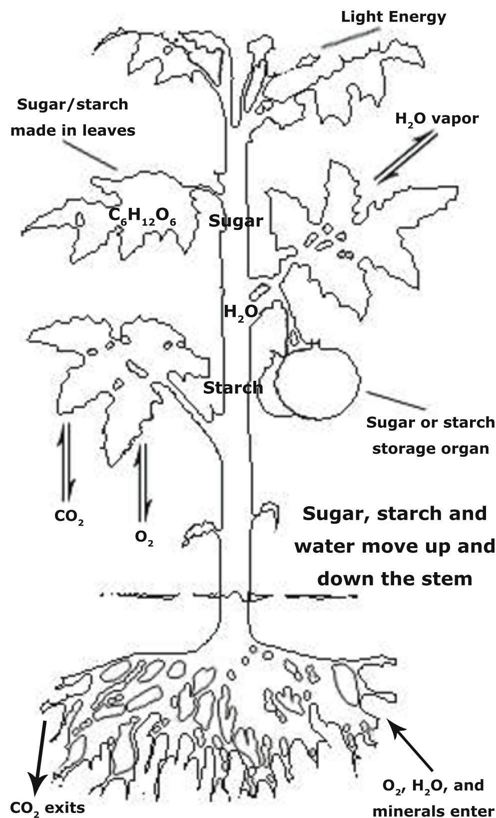


Fig. 25.1. Illustration of plant transpiration

Virtually all of a plant's water requirement is obtained through its root network and passes upward from the roots to the leaves, where it evaporates

through openings in the leaf surface called *stomata*. As water vapour diffuses out of the stomata, CO_2 diffuses in. The rate of exchange is some 400:1 (water to CO_2); hence, availability of water resources is a critical factor in determining plant growth. If a plant is enduring a water drought, its stomata will close in order to avoid water loss, at the expense of carbohydrate production. The movement of water and other nutrients from the soil upwards through the plant is known as *transpiration* (Fig. 25.1).

Plants have developed a wide variety of mechanisms to ensure that they can successfully capture the necessary raw materials for food generation, with their leaves (and stem) capturing sunlight, and their root network capturing water and other necessary nutrients. Hence we can loosely speak of plants as 'foraging' from their environment, seeking good access to sunlight and to nutrient-rich soil patches below ground.

25.2.1 Plant Movement and Foraging

One notable aspect of plants is that most are *sessile*, or fixed in place by their root network, as a result of an evolutionary decision several hundred million years ago to gather energy via photosynthesis rather than resource hunting via movement around their environment [630]. As resource location is unlikely to be static, even a sessile plant needs to be able to move to some degree in order to acquire necessary minerals, light and water resources. Plants tend to grow towards light and take actions such as accelerating the growth of stems or branches to reacquire light if they are shaded by other plants or objects. Leaves are placed and positioned by petioles to minimise self-shading, and leaves can move in response to changing light conditions to face the optimal direction of light [632]. Plants can change their morphology in response to changes in the availability of resources [630], with plant growth and development continuing throughout their lifecycle.

Although we do not usually think of plants as being able to move, this is largely because plants typically move at much slower rates than animals. Plants can move in response to a variety of stimuli including:

i. light (phototropism),
ii. gravity (geotropism),
iii. water (hydrotropism), and
iv. touch (thigmotropism).

The movement can be directionally towards the source of stimulation (positive tropism) or away from it (negative tropism). Plant shoots typically forage for (seek) light, whereas plant roots have a negative tropism for light. Other stimuli that can impact on plant root movements include neighbour competition for resources and the nature of soil composition in and around the plant's root network. One of the earliest formal studies of plant movement in response to these stimuli was undertaken by Charles Darwin and published in his famous work *The Power of Movement in Plants* (1880) [128].

Phototropism

Plants monitor their visible environment continuously. Although it has long been known that plants respond to light, the mechanism that was responsible for this was unknown before Darwin's experiments. His experiments found that the tips of shoots respond positively to light (specifically to short wavelength blue light) and the stem bends in order to orientate the plant towards the light source. Somewhat surprisingly perhaps, plants have a greater number of distinct photoreceptors than humans (11 in the case of plants and five in the case of humans) [101], perhaps because of the direct importance of light to plant survival. In addition to providing energy, plants use information gathered from light as cues for growth, to regulate the timing of their circadian clock and therefore their internal processes, and to regulate their flowering process.

Not all plants exhibit the same degree of phototropism, and light foraging mechanisms evolve in line with the challenges posed by the habitat faced by a particular plant species [226]. Morphological plasticity such as plant growth or plant bending is likely to be a high-cost foraging strategy and therefore may be less common in resource-poor environments. As one example of environmental influences, the majority of turf grasses exhibit low foraging precision, as the basal meristems of the grasses are too remote from the leaves they subtend to allow fine adjustments in leaf position. In other words, the plant has poor directional control in attempting to forage for light. However, this plant architecture confers a strong selective advantage in pasture environments as it allows for rapid recovery following grazing and trampling by vertebrates, producing a trade-off between precision in foraging for light and the capacity to recover from defoliation [226].

Although it is widely known that plants can bend towards light, the process by which plants move towards light is not immediately obvious. Typically the bending process involves *differential cellular elongation*. The elongation process is stimulated by the hormone auxin. Auxin triggers a chemical reaction which weakens cell walls, thereby allowing them to elongate. Elongation of cells on one side of the plant creates unequal growth leading to a bending of the plant. As auxin levels tend to be higher on the shaded side of the plant, the plant bends towards light. Plants can also 'move' via growth of their stem, branches and roots.

Geotropism

The root network of a plant also moves, the root tips responding to gravity. Cells in the root cap contain ball-like structures called *statoliths* which are heavier than other contents of cells in the root cap, thereby falling to the bottom of the cell under the influence of gravity. This allows the plant to know which way is 'down' and thereby guide root growth [101]. The statolith structures (called *statocytes*) found in some plant cells near the root tip bear

close parallels to similar functioning structures in the human inner ear which contribute to our balance. The inner ear contains semicircular, fluid-filled canals which lie at right angles to each other. As we move around sensory hairs are stimulated as the fluid moves. The fluid vestibules contain *otoliths*, small crystalline stones that sink in response to gravity, which also stimulate the sensory hairs, providing sensory inputs to let us know whether we are upright or horizontal.

Hydrotropism

Hydrotropism is the response of plant growth to water. Roots are generally positively hydrotropic, and as the root cap detects water, the root responds by bending towards the moist soil by means of differential cellular elongation.

Thigmotropism

Many plants display a response to touch. For example, tendrils in climbing plants such as vines or bindweed, will tend to coil around a supporting object (with differential cellular elongation taking place on the side opposite the supporting object leading to coiling behaviour). An interesting consequence of this mechanism is that the plant is left with more resources for growth as it does not need to invest as much energy in supportive tissue.

25.2.2 Root Foraging

While the most obvious plant growth occurs above ground, a significant component of plant activity takes place out of sight below ground in its root network. The root network of a plant serves a number of functions including the provision of structural anchorage for the shoot and branches, in addition to the capture of water and critical nutrients such as nitrogen and phosphorous from the soil. While a wide variety of root system architectures exist, a common architecture consists of a primary root (or 'radicle') with lateral roots branching off this (Fig. 25.2).

The structure and location of a plant's root network has direct implications for its ability to obtain necessary resources. Critical decisions are faced concerning the direction of root network growth and the level of resource investment in growth of the root network versus that of the stem and branch network. Species of plants display diverse root network architectures, some plants having narrow root networks with low branching angles (therefore focusing on exploitation of local resources) and other plants exhibiting widely dispersed root networks (thereby having greater explorative capabilities). In the same way that plant growth above ground is plastic and responsive to environmental conditions, root growth varies in response to several factors including [89]:

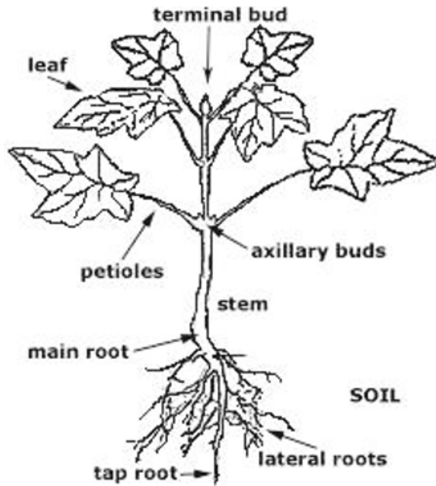


Fig. 25.2. Main and lateral roots

i. genetic factors (plants respond differently to environmental cues),
ii. soil conditions, including resource distribution,
iii. plant nutrient requirements, and
iv. the presence (or absence) of neighbouring plants.

We can consider that the root foraging process is a form of 'search' but the nature of this search process is highly complex. As resources are not equally distributed in the soil, plants will tend to place more root mass into areas with higher resource concentrations. However, no single foraging strategy is observed across all plant species. The degree of sensitivity of root placement in response to resource availability varies by species and it is known that root placement is also partially stochastic [89].

As plant root networks are simultaneously foraging for several resources which will likely not all be found in the same place, the root network architecture design problem faced by an individual plant is multifaceted. The current evidence suggests that the tips of plant roots (the root apical meristem) have the capability to 'sense' water and nutrients and use this information to direct root growth. However we do not yet have a complete understanding of the root foraging process [89].

Another aspect of root system development is that it takes place in a complex ecological environment. As already mentioned, nitrogen is important for plant growth as it plays a pivotal role in metabolic processes for energy release in plants. In order to access required nitrogen supplies, plants can form symbiotic relations with soil microorganisms such as bacteria and fungi wherein the plant obtains nitrogen in exchange for photosynthate.

Of course, soil contains many other organisms that can impact on an individual plant, including pathogens, herbivores and competitors. The latter case is particularly interesting, with a variety of strategies being adopted by plant root networks when a neighbouring root network is detected. In some cases the response is avoidance but in others the response is competitive, with a plant seeking to increase root growth in the vicinity of neighbouring roots akin to the territoriality seen in some animal species.

There is evidence that plant roots can distinguish between encounters with their own roots and alien roots of other plants, and in some cases between roots of own-species plants and those of alien-species plants [101]. The precise mechanisms of self-recognition and own-species recognition are poorly understood but it is thought that self-recognition is mediated internally, with foreign roots being recognised via messenger molecules in the soil [579]. Enhanced understanding of these mechanisms could lead to the development of plant-inspired algorithms for classification.

25.2.3 Predatory Plants

There are a small number of plants that exhibit atypical foraging behaviours. Two of the best-known examples are the ‘dodder plant’ (or *Cuscuta pentagona*) and the ‘Venus flytrap’ (or *Dionaea*).

Dodder Plant

The dodder is unusual as it has no leaves and lacks chlorophyll, which is necessary for photosynthesis. Therefore, the plant is incapable of generating its own food supply. Instead this vine-like plant finds and attacks a host plant by burrowing into its vascular system and draining off the nutrients it requires. The dodder plant identifies potential targets by detecting volatile chemicals released by its favoured host targets (particularly the tomato vine) akin to ‘smelling’ the host target [100]. When the dodder’s behaviour is examined using time-lapse photography, the dodder shoot tip is seen to move in a circular motion, initially randomly, and then growing and rotating in the direction of the nearest preferred host.

More generally, plants can release a variety of volatile organic compounds (VOCs) into the air in response to stimuli including insect attack and other stressors. Plants can also detect a variety of airborne volatile chemicals and produce a physiological response, for example, release chemicals to make their leaves less palatable to attacking herbivores. Whilst these signals can sometimes be detected, and acted upon, by close-by neighbouring plants, the plants are not communicating or ‘talking to each other’ per se, but rather neighbouring plants can be considered as eavesdropping on chemical messages released by the stressed plant which are intended for other leaves on the same plant [100].

Venus Flytrap

Another example of predatory foraging behaviour is provided by the Venus flytrap which typically lives in nitrogen-depleted areas [205] and which supplements nitrogen resources by capturing and devouring small insects. In the Venus flytrap, an action potential is generated (action potentials are momentary changes in electrical potential that travel along the surface of a cell with constant velocity and magnitude) whenever an upper trap hair is bent. For the trap to close a second action potential must be generated within approximately 40 seconds (corresponding to a second movement of the insect in the trap). The requirement for the second (or costimulation) signal corresponds to a form of memory. Other examples of processes akin to memory in plants have been noted by plant researchers [101].

25.3 Plant-Level Coordination

Like all complex multicellular organisms, plants require a means of coordinating the activities of cells in different locations. Examples which illustrate intraplant coordination include downstream signalling of the availability of photosynthetic products (carbohydrates) to support plant growth, or upstream signalling from the root network to the shoot. Studies looking at signaling have examined how leaf or root damage, or resource stress (insufficient access to water or light), can promote or retard growth and development elsewhere in the plant. Leaf shading can result in the development of new branches or leaves; a shortage of water can result in enhanced root growth in an attempt to locate new water resources. The coordination of these actions indicate that information is transmitted from the leaves and roots to other parts of the plant. A variety of communication processes have been uncovered in plants, including chemical messaging, electrical signaling, and communication via volatile organic compounds. Higher plants possess two key vascular transport systems, with water and soluble nutrients being transported from root hairs to all parts of the plant via a transport system called the xylem. A second transport system called the phloem carries sugars (photosynthate) from the leaves to other parts of the plant (including the roots) (Fig. 25.3).

Plant Neurobiology

The ability of plants to 'communicate internally' has led to a significant interest in plant signaling mechanisms and in trying to better understand the mechanisms by which plants convert environmental information into actions. One controversial area of this study is called *plant neurobiology*, which views plants as information-processing organisms with complex processes of communication [73]. Plant neurobiologists point to the electrical excitability of plant cells and also to the presence of several proteins (specifically, neurotransmitter-like

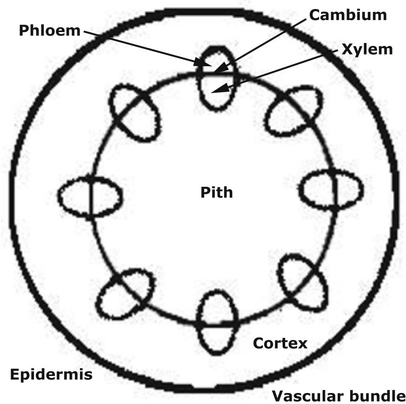


Fig. 25.3. Cross-section of plant stem illustrating the vascular cambium (the tissue that carries water and nutrients throughout the plant). Phloem layers form on the outer surface and xylem layers form on the inner surface

chemicals) which play a role in animal neuron systems, and suggest that there are some functional similarities between information processing activities in plants and animals.

Critics of the plant neurobiology perspective point out that there is no evidence for the existence of structures such as neurons, synapses or a 'brain' in plants [12, 191, 602] whereas other scientists, whilst agreeing with this perspective, note that metaphors can still have value in terms of assisting our understanding of phenomena [630, 631, 633]. Trewavas (2007) [633] comments that organisms do not necessarily need to have brains to exhibit problem-solving capabilities, citing the cases of bacterial chemotaxis and the growing literature on self-organising structures which examines the ability of locally interacting elements or agents to solve complex problems in an emergent fashion (swarm intelligence), or the ability of complex systems to 'self-organise'. Trewavas notes that [633]

> This bottom-up approach would seem tailor-made to fit higher plant organization with its multiple growing points and modular structure. A 'small world' structure is evident in higher plants, consisting of clumps of locally communicating cells within meristems, coupled via longer-range simpler communication to the activities of others.

The debate as to whether plants are 'intelligent' is a very old one [372] and it is interesting to note that Darwin himself used a 'root-brain' metaphor in some of his writings [128]

> ... the tip of the root acts like the brain of one of the lower animals, the brain being seated within the anterior end of the body receiving impressions from the sense organs and directing the several movements.

As commented by [191], "...complexity of signaling molecules [mechanism] does not necessarily imply complexity of message ..." so we need to caution against simplistic assumptions that the existence of complex processes in plants automatically implies a deep problem solving capability.

25.4 A Taxonomy of Plant-Inspired Algorithms

While there is an active and continuing debate as to the level of cognition in plants, it is evident that plants have evolved complex sensory and regulatory systems that allow them to modify their growth and other internal processes in response to changes in conditions [101]. Until recently, little attention was paid to the possible utility of plant metaphors for the design of computational algorithms but the last few years have seen the development of a range of plant algorithms. Broadly speaking, the majority of these fall into three categories, algorithms inspired by:

i. plant propagation behaviour,
ii. light foraging behaviour (branching algorithms), and
iii. purported swarm behaviour of plant root networks.

Examples of each of these are discussed in the following sections.

25.5 Plant Propagation Algorithms

Plants have a repertoire of processes by which they propagate themselves, including seed dispersal and root propagation. Three algorithms which have been inspired by these processes, the *invasive weed algorithm* [406], the *paddy field algorithm* [519], and the *strawberry plant algorithm* [548], are discussed below.

25.5.1 Invasive Weed Optimisation Algorithm

Effective seed dispersal plays an important role in ensuring the survival of plant species, and in turn this depends on the ability of the plant to propagate its seeds into resource-rich areas. Hence, this process can metaphorically provide inspiration for the design of optimisation algorithms.

The *invasive weed optimisation algorithm* (IWO) (pseudocode provided in Algorithm 25.1), based on the colonisation behaviour of weeds, was proposed by Mehrabian and Lucas (2006) [406]. The inspiration for the algorithm arose from the observation that weeds, or more generally, any plant, can effectively colonise a territory unless their growth is carefully controlled. Two aspects of this colonising behaviour are that weeds thrive in fertile soil and reproduce more effectively than their peers in less fertile soil, and the dispersal of seeds during plant reproduction is stochastic.

The three key components of the algorithm are:

Algorithm 25.1: Invasive Weed Algorithm [406]

Generate p_{initial} seeds and disperse them randomly in the search space;
Determine the best solution in the current colony and store this location;
repeat
    Each plant in the population produces a quantity of seeds depending on the quality of its location;
    Disperse these new seeds spatially in the search space, giving rise to new plants;
    If maximum number of plants ($p_{\max} > p_{\text{initial}}$) has been exceeded, reduce the population size to $p_{\max}$ by eliminating the weakest (least fit) plants, simulating competition for resources;
    Assess the fitness of new plant locations and, if necessary, update the best location found so far;
until *terminating condition*;

i. seeding (reproduction),
ii. seed dispersal, and
iii. competition between plants.

Mehrabian and Lucas operationalised these mechanisms in the following way in the IWO algorithm.

Seed Production

Each plant produces multiple seeds, based on its fitness relative to that of the other plants in the current colony of weeds. A linear scaling system is used whereby all plants are guaranteed to produce a minimum number $N_{\min}^{\text{seeds}}$ of seeds, and no plant can produce more than a maximum number $N_{\max}^{\text{seeds}}$ of seeds. The number of seeds produced by an individual plant x in the current generation is calculated as:

$$N_x^{\text{seeds}} = \frac{\text{fitness}_x - \text{colfitness}_{\min}}{\text{colfitness}_{\max} - \text{colfitness}_{\min}}(N_{\max}^{\text{seeds}} - N_{\min}^{\text{seeds}}) + N_{\min}^{\text{seeds}} \qquad (25.2)$$

where $\text{colfitness}_{\max}$ and $\text{colfitness}_{\min}$ are the maximum and minimum fitnesses in the current population, and fitness_x is the fitness of the individual plant x.

Seed Dispersal

While the IWO algorithm employs the notions of fitness and reproduction, unlike the GA, the IWO does not use genetic operators in the creation of populational diversity. Exploration of the search space is obtained via a simulated seed dispersal mechanism. The seeds associated with each plant are dispersed by generating a random displacement vector and applying this to

the location of their parent plant. The displacement vector has n elements corresponding to the n dimensions of the search space, and is obtained by generating n normally distributed random numbers, with a mean of 0 and a standard deviation calculated using (25.3):

$$\sigma_{\text{iter}} = \left(\frac{\text{iter}_{\text{max}} - \text{iter}_{\text{curr}}}{\text{iter}_{\text{max}}} \right)^n (\sigma_{\text{max}} - \sigma_{\text{min}}) + \sigma_{\text{min}} \tag{25.3}$$

where $\text{iter}_{\text{curr}}$ is the current algorithm iteration number, iter_{max} is the maximum number of iterations, σ_{max} and σ_{min} are maximum and minimum allowable values for the standard deviation, n is a nonlinear modulation index, and σ_{iter} is the standard deviation used in the current iteration in calculating the seed displacements.

The effect of this formulation is to encourage random seed dispersal around the location of the parent plant, with decreasing variance over time. This results in greater seed dispersal in earlier iterations of the algorithm, promoting exploration of the search space. Later, the balance is tilted towards exploitation as the value of σ_{iter} is reduced. The incorporation of the nonlinear modulation index in (25.3) also tilts the balance from exploration to exploitation as the algorithm runs.

Depending on the scaling of the search space, the same value of σ_{iter} could be applied when randomly drawing each element of the displacement vector. Alternatively, differing values of σ_{initial} and σ_{final} could be set for each dimension if required.

Competition for Resources

Competition between plants is simulated by placing a population size limit p_{max} on the colony. The plant colony starts with a population of size p_{initial}. The population increases as new plants grow in subsequent generations. Once the p_{max} population limit is reached, parent plants compete with their children for survival. The parent and child plants are ranked by fitness, with only p_{max} plants surviving into the next generation. This mechanism ensures that the best solution found to date cannot be lost between iterations (elitism).

Performance of the Algorithm

The IWO is a conceptually simple, numerical, non-gradient-based, optimisation algorithm. So far, due to its novelty, there has been limited investigation of its effectiveness, scalability and efficiency. Mehrabian and Lucas [406] report GA and PSO-competitive results from the IWO algorithm with settings of 10 to 20 weeds, maximum and minimum numbers of seeds per plant of 2 and 0 respectively, and a nonlinear modulation index value of 3. Competitive results for the IWO algorithm are also reported by [36, 407] and [568].

The algorithm requires that several problem-specific parameters be set by the modeller, including the maximum and minimum number of seeds that a

plant can produce, the values for $\sigma_{\max}$, $\sigma_{\min}$ and $\text{iter}_{\max}$, and the initial and the maximum population size. However, the determination of good values for these parameters is not necessarily a trivial task, particularly in poorly understood problem environments.

Recent work has extended the application of IWO into clustering where each individual seed consists of a string of up to n real vectors of dimension d, corresponding to n cluster centre coordinates (in d-dimensional space $\mathbb{R}^d$) [376]. Apart from the IWO algorithm, a number of other algorithms which draw inspiration from seed dispersal behaviour have been proposed, including the *paddy field algorithm*.

25.5.2 Paddy Field Algorithm

The *paddy field algorithm* was first proposed by Premaratne, Samarabandu and Sidhu (2009) [519]. This algorithm (pseudocode provided in Algorithm 25.2) draws inspiration from aspects of the plant reproduction cycle, concentrating on the processes of pollination and seed dispersal.

Given a vector $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, we view it as a location in an n-dimensional space, and let $y = f(x)$ be the 'fitness' or 'quality' of that location. Each seed i therefore has a corresponding location x_i, and a corresponding fitness $y_i = f(x_i)$. The paddy field algorithm manipulates a population of these 'seeds' in an attempt to find a good solution to the optimisation problem of interest. The algorithm consists of five stages, sowing, selection, seeding, pollination, and dispersion [519]. Each of these are described below.

Sowing

An initial population P of p seeds is distributed (sown) at random locations in the search space.

Selection

The seeds are assumed to grow into plants, and each plant i has an associated fitness value y_i determined by the output of the underlying objective function when evaluated at the plant's location. The plants are ranked by fitness, and the best n plants are then selected to produce seeds.

Seeding

Each plant produces a number of seeds in proportion to its fitness. The fittest plant produces $q_{\max}$ seeds and the other plants produce varying amounts of seeds, with the number of seeds produced by plant i calculated using:

$$s_i = q_{\max} \frac{y_i - y_t}{y_{\max} - y_t}. \tag{25.4}$$

Here, the term $y_{\max}$ is the fitness of the best plant in the current population, and y_t is the fitness of the lowest ranked plant selected in the previous step. Although the algorithm describes this step as 'seeding', it can more correctly be considered as the process of growth of flower structures in order to enable pollination.

Pollination

Only seeds which have been pollinated can become viable and, to determine this portion, a simulated pollination process is applied, whereby the probability that a seed is pollinated depends on the local density of plants around the seed's parent plant (as a greater density of plants locally increases chances of pollination). A radius a is defined, and two plants are considered to be neighbours if the distance between them is less than a (that is, each lies within a hypersphere of radius a centred on the other). The pollination factor U_i of plant i (where $0 \leq U_i \leq 1$) is then calculated using:

$$U_i = \mathrm{e}^{v_i/v_{\max}-1} \tag{25.5}$$

where v_i is the number of neighbours of the plant i and $v_{\max}$ is the number of neighbours of the plant with the largest number of neighbours in the population. The effective number of viable seeds produced by a plant i from the inital s_i seeds it produced is therefore:

$$s_i^{\text{viable}} = U_i s_i. \tag{25.6}$$

Dispersion

The s_i^{viable} pollinated seeds are then dispersed from the location of their parent plant i such that the location of the new plant (grown from the dispersed seed) is determined using $N(x_i, \sigma)$ where x_i is the location of the parent plant and σ is a user-selected parameter.

The above five steps are iterated until a termination condition is reached. In summary, the fittest plants give rise to the greatest number of seeds, and search is intensified around the better regions of the landscape uncovered thus far. Variants on the paddy field algorithm include [337].

25.5.3 Strawberry Plant Algorithm

Although many plants propagate using seeds, some employ a system of 'runners', or horizontal stems which grow outwards from the base of the plant (Fig. 25.4). At variable distances from the parent plant, if suitable soil conditions are found, new roots will grow from the runner and in turn produce an offspring clone of the parent plant. An example of this behaviour is provided by modern strawberry plants which can propagate via seeds and by runners. This has inspired the development of an optimisation algorithm based on this phenomenon [548]. The algorithm is based on the following ideas:

Algorithm 25.2: Paddy Field Algorithm [519]

Generate an initial population P of p plants, each located randomly in the search space;
Choose values for iter_{max} and n;
Set generation counter iter = 1;
repeat
  Calculate fitness y_i of each plant $i = 1, \ldots, p$ and add this fitness vector $y = (y_1, \ldots, y_p)$ to the matrix containing the location of all p plants;
  Using these fitnesses, sort the population in descending order of fitness (assuming the objective is to maximise fitness);
  for $i = 1$ **to** n *(the top n plants)* **do**
    Generate seeds for each selected plant;
    Implement pollination step;
    Disperse pollinated seeds;
  end
  Replace old population with new plants;
  Set iter := iter + 1;
until iter = iter_{max};
Output the best location found;

- healthy plants in good resource locations generate more runners,
- plants in good resource locations tend to send short runners in order to exploit local resources,
- plants in poorer resource locations tend to send longer runners to search for better conditions, and
- as the generation of longer runners requires more resource investment, plants generating these will create relatively few of them.

The algorithm therefore seeks to balance exploration with exploitation, with increasing local exploration over time as plants concentrate in the locations with best conditions for growth. Salhi and Fraga (2011) [548] report competitive results from this algorithm when it is applied to a number of real-valued benchmark optimisation problems. Algorithm 25.3 presents an adapted version of the algorithm based on [548].

25.6 Plant Growth Simulation Algorithm

Plants exhibit a considerable degree of phenotypic plasticity which can be generally described as the 'response of organisms to environmental conditions or stimuli' [455]. One aspect of this is 'developmental plasticity', which can be generally defined as 'the developmental changes that follow the perception and integration of environmental information'. The ability of plants to adapt to changing environmental conditions via direction of shoot, leaf and root growth

Algorithm 25.3: Strawberry Propagation Algorithm (adapted from [548])

Generate an initial population $\{p_1, \ldots, p_m\}$ of m plants, each located randomly in the search space;
Choose values for maxgen and y (see below);
Set generation counter gen = 1;
repeat
  Calculate fitness of each plant and store in vector N where the ith component of N is $N_i = \text{fitness}(p_i)$, $i = 1, \ldots, m$;
  Sort the components $N_1, \ldots, N_m$ of N into descending order (assuming the objective is to maximise fitness);
  for $i = 1$ **to** $m/10$: *top 10% of plants* **do**
    Generate y/i short runners for each plant (y is a user-defined parameter which defines the intensity of local search around each of the fitter plants);
    if *any of the new locations has higher fitness than that of the parent plant* **then**
      Move the parent plant to the new location with the highest fitness ($r_i \rightarrow p_i$);
    else
      Discard the new locations; the parent plant stays at its current location;
    end
  end
  for $i = m/10 + 1$ **to** m: *indices for remaining plants* **do**
    Generate one long runner for each plant not in the top 10% and select the location of the end-point r_i for that runner randomly in the search space;
    if *the new location has higher fitness than that of the parent plant* **then**
      Move the parent plant to the new location ($r_i \rightarrow p_i$);
    else
      Discard the new location; the parent plant stays at its current location;
    end
  end
  Set gen := gen + 1;
until gen = maxgen;
Output the best location found;

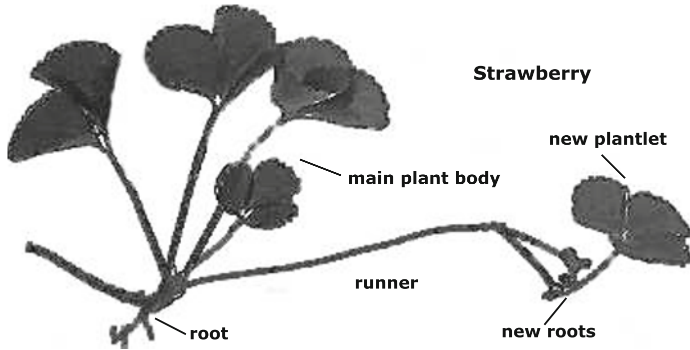


Fig. 25.4. Stawberry plant with runner stems

provides a rich vein of metaphorical inspiration for the design of optimisation algorithms based on plant resource foraging behaviours. In this subsection we describe an algorithm which is inspired by the light foraging process, the *plant growth simulation algorithm* (PSGA).

A key aspect of plant growth is that the initial stem of the plant gives rise over time to branches and leaves as it grows. The location and number of branches and leaves is (at least in part) a function of the resources in the plant's environment. Plants can display different growth patterns above ground. These can be broadly classified as being either *monopodial* or *sympodial*. Monopodial (from the Greek words for 'one' and 'foot') growth occurs when the plant's growth is led by the development of its stem. While lateral branches and leaves can be expressed off the stem, these are subordinate to the stem, which continues to grow upwards over time. In contrast, sympodial growth occurs when plant growth is led by the new leader shoots which branch off the original stem.

The actual process of growth is driven by the production of new cells in meristem tissue (embryonic undifferentiated cells), which can be found just below the shoot tip (shoot apical meristem) or at the tips of branches (lateral meristems) or just inside the root cap in root tips (apical meristems). As the new cells are produced, the stem, branch or root grows longer.

The Hungarian biologist Aristid Lindenmayer, in studying plant growth, developed L-systems (Sect. 17.5) to model recursive growth patterns, and a key item of L-systems is the notion of rewriting, whereby a complex structure can be generated from a starting 'seed' by successively replacing parts of the growing structure using a set of production or rewriting rules.

Metaphorically, the plant growth process can be considered as being the exploration of an environment in a search for a good architecture which allows the plant to capture resources effectively. Tong et al. (2005) [625] drew inspiration from this idea to develop the PGSA. This initial work inspired a number of follow-on studies and applications of the algorithm, including

[237, 616, 626, 646]. The PGSA constructs a virtual plant growth simulation in which a simulated plant grows in the search space and attempts to find the optimal solution (the light source) [626]. The growth of the plant is driven by a pseudophototropic mechanism whereby the degree of growth of the main shoot and branches depends on the quality of the solution at various points on each (a proxy for the level of light). In essence, each branch metaphorically undertakes a search in a local area of the search space, with the plant's growth pattern being biased towards regions which display higher fitness.

25.6.1 The Algorithm

The PGSA takes inspiration from L-systems in terms of growth and branching system design. New branches are assumed to grow from nodes on the main stem, or from nodes on previously generated branches. Each new branch is assumed to have a turning angle of 90° (i.e. it grows perpendicularly to its parent branch) and the definition of the length of the new branch (branching length) is determined by the nature of the optimisation problem at hand. For example, in the initial application of the algorithm to an integer programming problem [625] the branch length was set as 1. The number of new branches grown from a selected node (corresponding to a tip with meristem cells) in each iteration of the algorithm is $2n$, where n is the dimensionality of the search space. Therefore, starting from a seed point (the initial solution), the 'plant' grows and branches at nodes on the growing plant.

The next issue is how to select a node for branching in each iteration of the algorithm. This process is loosely based on elements of a morphogenetic model of plant development in which the concentration of growth hormone (morphactin) at a node determines whether the cells at that node will start to grow and produce branches. In the design of the optimisation algorithm, the concentration of the growth hormone at a node corresponds to the relative fitness of the coordinates of the location of that node for the optimisation problem of interest. The selection process for choosing the next branching node, and therefore the next region of the search space to be explored, is biased towards the location of the nodes of higher fitness. Hence, each node is a possible solution to the optimisation problem and the best one found during the simulated growth process is output by the algorithm.

$$\begin{cases} C_{Mi} = \frac{g(B_0)-g(B_{Mi})}{\Delta_1}, & (i = 1, 2, \ldots, k); \\ \Delta_1 = \sum_{i=1}^{k} g(B_0) - g(B_{Mi}). \end{cases} \tag{25.7}$$

Initially, the plant grows a stem from its root node (denoted B_0) and the stem is assumed to have k nodes which have a better environment (or fitness) than the root node. The quality of the environment at each node is calculated using the fitness function g and the morphactin concentration at each node B (denoted C_B) is calculated as the difference between the fitness of the

root of the plant (denoted by $g(B_0)$) and the fitness of that node i ($g(B_{Mi})$) (assuming a minimisation problem) divided by the sum of these differences for all nodes $i = 1, 2, \ldots, k$. Only the k nodes with fitness greater than the root are considered in the calculation (25.7). The fitness of each node is calculated as being relative to the root and relative to that of all other nodes. By inspection of (25.7) the sum of the scaled fitnesses must be 1 ($\sum_{i=1}^{k} C_{Mi} = 1$).

To select the node from which the next branch will be grown, the fitnesses for all nodes are laid out in the range $0 \rightarrow 1$ and a random draw is made from this interval, with the node corresponding to that interval being selected as the preferential node from which a branch will be grown in the next iteration of the algorithm. Assume that there are q nodes on the new branch that have a better environment than the root node (B_0). The 'originating node' from which the branch grew (node 2 in (25.8)) has its morphactin concentration set to 0 and is ignored for the rest of the algorithm, and the morphactin concentrations for all nodes on the plant are recalculated using (25.8). The term Δ_1 in (25.8) calculates the sum of the fitnesses of all nodes on the plant's stem (omitting node 2, from which the new branch grew) relative to that of the root node, and Δ_2 in (25.8) calculates the sum of the fitnesses of all nodes on the new branch relative to that of the root node.

$$\begin{cases} C_{Mi} = \frac{g(B_0) - g(B_{Mi})}{\Delta_1 + \Delta_2}, & (i = 1, 3, \ldots, k) \\ C_{mj} = \frac{g(B_0) - g(B_{mj})}{\Delta_1 + \Delta_2}, & (j = 1, 2, \ldots, q) \\ \Delta_1 = \sum_{i=1, i \neq 2}^{k} (g(B_0) - g(B_{Mi})) \\ \Delta_2 = \sum_{j=1}^{q} (g(B_0) - g(B_{mj})) \end{cases} \quad (25.8)$$

As before, the sum of the scaled fitnesses must be 1 ($\sum_{i=1, i \neq 2}^{k} C_{Mi} + \sum_{j=1}^{q} C_{mj} = 1$). The branching process iterates until a preset terminating number of branching iterations is reached.

In summary, the algorithm traverses a search space by implementing a search process which is loosely modelled on plant growth mechanisms. The plant covers a region of the search space, with the nodes on the plant representing possible solutions to the problem, and the value $g(B_i)$ being the objective function value at a node i. The algorithm biases its search process towards the regions of higher fitness (exploitation) whilst maintaining some explorative capability.

While competitive results have been reported by a number of studies using the algorithm, one aspect of the algorithm that has received less attention thus far is its computational efficiency. As each new branch is added, the morphactin concentrations for all nodes must be recalculated as they depend on relative measures (the fitness values need not be recalculated). Implementation of the algorithm also requires the definition of problem-specific branch

lengths and the implementation of a mechanism to determine the number and placement of nodes on the main stem and subsequent branches.

Algorithm 25.4: Plant Growth Simulation Algorithm (adapted from [625])

Choose values for maxgen and branch length;
Define a mechanism for node placement on the stem and subsequent branches;
Set generation counter gen = 1;
Generate an initial root at location B_0;
Define node locations on stem;
Calculate fitness of each node on stem using $g(B_i)$ and store location of highest-fitness node;
Calculate morphactin concentration at each node using (25.7);
repeat
    Select branching node (stochastically but biased by concentration);
    Add $2n$ lateral branches at this node;
    Calculate fitness of new nodes on added branches;
    If any of these locations have higher fitness than best found to date, replace best location;
    Update morphactin concentrations at each node on the plant using (25.8);
    Set gen := gen + 1;
until gen = maxgen;
Output the best location found;

25.6.2 Variants on the Plant Growth Simulation Algorithm

There are many ways to operationalise the growth and phototropic mechanisms in the PGSA and each choice gives rise to an optimisation algorithm with different characteristics. A variable branch growth mechanism is implemented by [680] with the rate of growth of an individual branch in each iteration being determined by its level of photosynthesis activity, which in turn is determined by the light intensity on the branch (proxied by its fitness) and its rate of respiration. The higher the rate of simulated photosynthesis in the branch, the faster its rate of growth. In this version of the algorithm, branches also move in the search space towards increasing light intensity, simulating a 'bending to light'. The effect of the two mechanisms is to promote greater exploitation of higher-fitness regions of the search space.

A more complex set of growth and branching behaviours are adopted by [90]. In this variant of the PGSA, the growth and branching behaviour at nodes depends on their relative fitness, with sympodial branching occurring if a node is of high fitness relative to other nodes on the plant (i.e., the stem or

branch elongates from the node in the current direction of the stem/branch, and child branches are grown from the node in a lateral direction), monopodial branching occurring if the node is of moderate fitness (i.e., the current stem or branch is extended in its current direction). If a node is of low relative fitness, no growth or branching occurs at the node. This approach allows for more intensive search via branching around the higher-fitness points, growth out of moderate fitness regions for branches of moderate fitness and no growth or branching at low fitness nodes. The differential branching mechanism is supplemented by a simulated leaf growth process, whereby the region around the end points of new branches is searched using a local search algorithm (simulating leaf growth around the branch end point, with the location of the highest fitness point in this region (denoted a 'leaf point') being recorded.

25.7 Root-Swarm Behaviour

As described in Sect. 25.2.2, plant roots forage for nutrients in a complex and sometimes hostile underground environment. While each root acquires environmental information locally to itself, an open question is whether (and to what extent) the information gathered by each root is processed in any collective sense. When we ponder the richness of root network structures, for example Dittmer (1937) estimated that a winter rye plant possessed some 13.8 million roots / root hairs [161], this raises the question as to whether root networks bear functional similarity to an integrated sensor array.

Baluska, Lev-Yadun and Mancuso (2010) [28] describe three possible communication channels between root tips, namely:

i. secreted chemicals and released volatiles,
ii. electrical fields, and
iii. vertical signaling within the root (upwards).

The first two of these channels embed stigmergic communication (Chap. 9). While each of these mechanisms are plausible, the degree to which each is used and the degree to which signals from each mechanism can spread across a root network and interact with each other is not known. It is known that the root apex (tip) plays a critical role in directional growth decisions of a root and that most of the environmental sensing of the root also takes place here [579]. However, the relatively low computational capacity of a plant cell makes it difficult to envisage that any complex information-processing is taking place in isolation in individual roots.

A number of authors, including [28], have drawn a parallel between root systems and models of swarm intelligence, noting that each typically consist of relatively simple agents, with local sensing capabilities, which solve complex problems as a result of information dissemination between the agents. These complex problems include [28]

where to grow, whether to grow at all, to fight or retreat in face of competitive root systems, whether to enter into symbiotic relationships with mycorrhiza fungi and Rhizobium bacteria

25.7.1 Modelling Root Growth in Real Plants

A number of studies have attempted to apply a root-swarm metaphor in differing ways. Ciszack et al. (2012) [111] applied a swarm model in order to simulate the real-world growth behaviour of a root network. The developed model simulates the growth of a population of plants growing from initial 'seeds' (each seed growing at slightly different rates) in order to determine the behaviour of the root network of the entire population.

A simplifying assumption was made in the model that each plant (seed) produces a single main root, and each root tip is conceptualised as a 'particle' moving in an environment where the associated root length is the temporal history of that particle's movement or 'growth'. Each particle has a position and a velocity, and a series of simple rules are defined to govern the interactions of roots with each other. These interactions can occur anywhere along the length of the root and they include dynamics of spatial repulsion, attraction and heading alignment between roots from individual plants.

The key findings from the simulation were that the developed model could produce root growth patterns which were qualitatively similar to those observed in the real-world growth of root networks for maize seeds, producing realistic-looking root networks which exhibited trajectory alignment and root clustering behaviours as seen in nature. In contrast, control results from a random growth control model did not produce realistic-looking root networks.

25.7.2 Applying the Root-Swarm Metaphor for Optimisation

The results of [111] do not necessarily imply that plant root networks actually engage in swarm-like behaviour. However, the work does indicate that it would be interesting to investigate the potential for the development of robust swarm optimisation algorithms, whose workings are loosely inspired by plant root mechanisms, even if the mechanisms are as yet imperfectly understood. Simoes et al. (2011) [579] take up this idea and observe that a root system can be considered as attempting to explore an unknown environment or, in other words, as searching for a solution to a complex multiobjective optimisation problem. The study develops a swarm system based on simple root agents and applies this as a decentralised control system for a swarm of robots performing a collective exploration task.

Unlike ant pheromone pathways, it is 'expensive' and difficult for a plant to change its root network architecture and this raises the question as to what types of problems might be best suited to root inspired swarm algorithms. An interesting conjecture in [579] is that ant systems may be better suited

for decision-making in dynamically changing environments whereas root inspired algorithms may be better suited for collective decision-making under conditions of uncertainty.

25.8 Summary

Despite the vast array of plants and associated plant behaviours which are exhibited in the natural world, little inspiration has been taken from these mechanisms, as yet, for the design of computational algorithms. Most of the algorithms developed thus far are relatively recent in design and further work is required in order to assess their utility and to assess more fully whether they represent truly novel problem-solving mechanisms or whether they are qualitatively similar to existing natural computing algorithms. Given the lack of research in this area to date, there is rich potential for future work.

26

Chemically Inspired Algorithms

An as yet relatively under-explored area of natural computing is the use of chemical phenomena as a source of inspiration for the design of computational algorithms. Of course, chemical processes play a significant role in many of the phenomena already described in this book, including (for example) evolutionary processes and the workings of the natural immune system. However, so far, chemical aspects of these processes have been largely ignored in the design of computational algorithms. In this chapter, we initially provide a short primer on some concepts from chemistry and then — using these — describe a number of optimisation algorithms loosely inspired by the processes of chemical reactions.

26.1 A Brief Chemistry Primer

Chemistry studies the way in which atoms of elements combine to produce molecules, and the properties of such molecules. We now explain these terms.

An *element* is a pure chemically distinct substance (species of matter); examples include hydrogen (chemical symbol H), carbon (C), oxygen (O), gold (Au) and plutonium (Pu). An *atom* is the smallest chemically indivisible unit of an element, and has a particular number of protons in its nucleus (its *atomic number*) and when isolated and nonionised has the same number of electrons in shells about the nucleus. Thus, an element is characterised by its atomic number. For example, hydrogen has atomic number 1, while carbon has atomic number 6.

Chemistry is interested in how the electron shells of atoms interact, but not the nuclei; thus, there may exist different isotopes of an element having different numbers of neutrons in the nucleus, such as Carbon-12, ^{12}C, and Carbon-14, ^{14}C, but chemically they are the same. Thus, an atom has a type (the element it is part of, characterised by its atomic number), mass, effective radius, charge (normally uncharged), position, and momentum.

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_26

A *molecule* is a chemically stable connected group of atoms. Some elements occur naturally as individual atoms, e.g., helium; but most occur as molecules of two or more atoms, e.g., hydrogen, which occurs as a molecule of two atoms, H_2, as does oxygen, O_2. A *compound* is a chemical substance made up of two or more elements; examples include water, H_2O, methane, CH_4, etc.

Chemically, a molecule is made up of a set of atoms connected by bonds: a *bond* is created when one or more electrons are shared among several atoms, allowing the electrons to fill lower energy states and so produce a more stable structure — the molecule — than isolated atoms. The *valency* of an element is the number of electrons an atom of it can provide to form bonds with other atoms. There is often a distinction made between covalent and ionic bonds, but this is somewhat arbitrary and is loosely based on the distribution of the time the electrons spend about the different atoms. In a covalent bond it is roughly even, while in an ionic bond it is more uneven, leading to one atom or group of atoms having an effective negative charge (negative ion) and another group having an effective positive charge (positive ion). A bond does not have to be between two atoms, though the standard network diagrams of molecules may make this appear to be so. For example, in benzene (C_6H_6), each C atom has valency 4 and each H atom has valency 1; the six C atoms are arranged in a ring, each C having an associated H atom; of the four electrons each C atom can provide for bonding, one is shared with its associated H atom, one with each of its two C neighbours, and the remaining fourth electron is shared among all six C atoms (the six shared electrons are delocalised, forming two π rings). Another example of electron delocalisation is any metal: the outermost electrons may be viewed as being shared by all atoms in the piece of metal.

A molecule is characterised by the number and types of atom (elements) together with the interatomic bond lengths, angles, and torsions. Two molecules are considered to be distinct when they contain different elements, numbers of atoms, bond length, angle, or torsion; even if all of these are the same, mirror images of molecules are considered to be distinct.[1] A molecule viewed as a particle also has physical attributes such as charge, position, momentum and energy.

In a chemical reaction, reactants (input species to the reaction) chemically react to form products (output species of a reaction) by the creation of bonds. The rate of the reaction depends on the ambient temperature, typically doubling for every 10 degree increase in temperature. Generally, reactants are not completely converted to products, but an equilibrium is reached where the relative concentrations of chemical species depend on the initial concentrations and ambient temperature. The kinds of reaction possible and the species of products formed depend not only on the reactants but on what combinations

[1] In organic chemistry, the basis for life, molecules of the same type generally have the same chirality or 'handedness'; in the case of biologically active molecules, for example, amino acids are almost always D (dexterous or right-handed) while sugars are L (left-handed).

are energetically feasible: molecules must obey valency and other combination rules.

Reactions are traditionally denoted by $A + B \rightarrow C$ or even $A + B \leftrightarrow C$ or $A + B \Leftrightarrow C$ to indicate that reactions rarely use up 100% of the reactants, but that to a small extent the products may react in reverse to give the reactants.

As described earlier, by equipartition of energy, physical systems such as chemical reactions seek to minimise their free energy: the particular mechanism here is the settling of electrons into lower energy states than they could achieve in isolated atoms, the throwing off of excess energy, and the resulting increased stability of the molecule. Thus, there is potential in principle to use minimisation of energy during chemical reactions as a proxy for optimisation.

26.2 Chemically Inspired Algorithms

This idea of energy minimisation during chemical reactions is implemented in chemical reaction optimisation (CRO), developed in a series of papers by Lam, Li and collaborators, beginning with [356]. It is the main topic of this chapter.

More recently, another chemically inspired algorithm appeared [9, 10], which uses chemical inspiration in a somewhat different way. We treat this briefly in Sect. 26.2.2.

26.2.1 Chemical Reaction Optimisation (CRO)

Chemical reaction optimisation (CRO) is an optimisation algorithm deriving inspiration from some principles of chemical reactions. The development herein largely follows that of [356], the originators of CRO. Many of the concepts from the previous chapters on physically inspired computing (especially Chap. 22), such as potential energy and kinetic energy, are used here also.

In CRO, a molecule ω represents a candidate solution through its internal 'molecular structure': this incorporates both the molecule's state of motion and its chemical structure. The particular molecular structure is problem-specific, in that the structure must be able to express a feasible solution to the problem. For example, if the feasible solution set is the set of n-dimensional vectors comprising positive real numbers $\mathbb{R}^n_+$, then any vector with n elements whose values are positive real numbers is a valid molecular structure, and no valid molecular structure can contain numbers with nonpositive values. A change in molecular structure is equivalent to moving to another feasible solution [356].

In the kind of chemical reaction process modelled by CRO, the molecules are viewed as being in a gaseous state, enclosed in a walled container. Each molecule possesses both potential energy (PE) — associated with its molecular structure — and kinetic energy (KE) — associated with its motion. The algorithm manipulates molecules by a series of 'collisions' which may give

rise to 'reactions', modifying both the molecular structure and potential and kinetic energy values. It improves the best-so-far solution, with the aim of optimisation. Specifically, CRO loosely mimics the way molecules interact on a microscopic level:

- physically: by collisions with a container wall or each other, only affecting physical properties such as energy; the originators of CRO call these collisions 'ineffective'; and
- chemically: where new products (output species of a reaction) may be chemically formed from the reactants (input species to the reaction); we call these collisions '(chemically) effective'.

Although in a real chemical reaction, molecules cannot collide or react unless they are in close physical proximity, CRO ignores physical locations: they have no meaning in the algorithm. Any members of the population of molecules may react together.

26.2.2 Artificial Chemical Reaction Optimisation Algorithm (ACROA)

The artificial chemical reaction optimisation algorithm (ACROA) was introduced in [9] and applications developed in [10]. In ACROA, atoms and molecules may move and collide continuously in a viscous medium within a two-dimensional bounded space. Time passes in discrete steps. The algorithm can be thought of as a simulation of reactants in a vessel of fixed volume. The vessel is thought of as containing a spatially uniform mixture of N species $R_1, \ldots, R_N$ of chemical reactants interacting by way of M specified chemical reactions. The way the reactants for ACROA are encoded depends on the particular problem of interest: binary, real, string encoding, etc. The encoding scheme influences the formation of *reaction rules*, which define the interaction among reactants which may lead to production of a new product.

Having defined the objective function and encoding, ACROA begins by initialisation of the algorithm parameter N_{reac}, denoting the number of species of reactant. The initial state is a set of N_{reac} initial reactants in solution. Over time, reactants are consumed and produced by chemical reactions; at each time step, reactants are updated and the termination criteria checked. The ACROA algorithm terminates when it reaches a state when no more reactions can occur: this is termed an 'inert solution'.

Reactants are selected probabilistically depending on their potentials and concentrations in the solution. The product of one reaction may be a reactant of other reactions. There are two main types of reaction:

Consecutive: consecutive reactions occur in serial (one after the other, for example, $A + B + C \leftrightarrow AB + C \leftrightarrow ABC \leftrightarrow A + BC$)

Competing: competing reactions occur in parallel, and which occurs and to what degree depends on a number of conditions.

For simplicity, there is an equal probability for monomolecular or bimolecular reactions and their variants. The kind of reaction that can occur depends on the type:

Monomolecular: possible reaction types are Decomposition and Redox (Reduction-Oxidation);

Bimolecular: possible reaction types are Decomposition, Redox (Reduction-Oxidation) and Synthesis.

In each case, the concept of *enthalpy* (a measure of total energy of a thermodynamic system, useful for describing energy transfer, as in chemical reactions) is used to decide if the reaction is reversible.

For further details of how these reaction types (Decomposition, Redox and Synthesis) are implemented for binary, real and other encodings, and for how the idea of enthalpy is used, see [9, 10].

In [10], ACROA was applied for classification rule discovery in data mining. Here, a database is the search space and ACROA was used as a search method for mining accurate and comprehensible classification rules. The performance of ACROA was compared to GA on two data sets and was found to be competitive with GA. The number of reaction species used was 50.

26.3 The CRO Algorithm

26.3.1 Potential and Kinetic Energy and the Buffer

All physical systems including chemical reactions seek to minimise their free-energy. As with simulated annealing (SA) (Chap. 23.1) and similar approaches, CRO uses minimisation of energy as a proxy for optimisation: the potential energy represents the objective function. Kinetic energy is energy that could be converted to potential energy, and so captures a tolerance to moving to states of higher potential energy, nominally a bad move, but possibly beneficial in escaping local minima (again as with SA). To enhance its physical realism, CRO limits chemical reactions to those that respect conservation of total energy:

$$PE_{\text{after}} \leq PE_{\text{before}} + KE_{\text{before}}. \tag{26.1}$$

As well as the molecules' potential energy and kinetic energy, there is a central buffer, which may be thought of as the energy associated with the container walls; this buffer stores energy, and in some reactions, kinetic energy may be drawn from this buffer if there is insufficient kinetic energy before the reaction to allow the reaction to take place, for example, in a decomposition reaction. Since this is usually the case with reactions where the products' total PE, PE_{after}, exceeds the reactants' total PE, PE_{before}, that is, with a less favourable solution after the reaction, we see that this mechanism allows for 'wrong-way' moves which — as in SA — may allow the algorithm to escape

local minima. The more total kinetic energy the reactants have, the more potential energy the products may have, and so the greater is the tolerance for ‘wrong-way’ moves, and so the greater the ability to escape from local maxima.

CRO uses a system parameter, $\mathrm{rate}_{\mathrm{loss}}^{KE}$, which controls the maximum percentage of kinetic energy lost in a given reaction. This lost energy is transferred to the central energy buffer, and can be used to support the decomposition operation (where one molecule breaks into two; see below). Over time, the molecules’ kinetic energy is transferred to the central buffer, reducing the amount of kinetic energy available in an individual reaction and so reducing the tolerance for ‘wrong-way’ moves. This reduces the molecules’ average kinetic energy as the algorithm proceeds; clearly, this has the same net effect as temperature reduction in SA. This, along with preferential selection of lower potential energy molecules, acts to reduce the potential energy of the molecules over successive iterations. This effect is the driving pressure in CRO to force convergence to lower potential energy and so to better objective function values.

Lam and Li (2010) [356] provide a thesaurus between chemical terms in CRO and mathematical optimisation (algorithmic) meaning, as in Table 26.1

Chemical Meaning	*Mathematical Meaning*
Molecular structure	Solution
Potential energy	Objective function value
Kinetic energy	Measure of tolerance of having worse solutions
Number of hits	Current total number of moves
Minimum structure	Current optimal solution
Minimum value	Current optimal function value
Minimum hit number	Number of moves when the current optimal solution is found

Table 26.1. CRO to mathematical optimisation thesaurus

26.3.2 Types of Collision and Reaction

A molecule may collide either:

- with another molecule; or
- with a wall of the container (this is called a *unimolecular* event).

A collision may be:

- ineffective, leading only to changes in motion of the molecule(s), that is, only physical changes; or

- effective,[2] leading to a genuine chemical reaction, in which both motion and chemical constitution are altered, giving new products.

These four possible combinations are modelled as four mechanisms or operations in CRO, called *elementary reactions*, for potentially altering the energy and/or internal molecular structure of one or more molecules:

i. on-wall ineffective collision (unimolecular ineffective),
ii. decomposition (unimolecular chemically effective),
iii. intermolecular ineffective collision (multimolecular ineffective), and
iv. synthesis (multimolecular chemically effective).

CRO allows conversion between potential energy and kinetic energy, within a molecule or among molecules, via these four elementary reactions (or steps).

Although the overall framework of CRO is fixed, the particular implementations of these four elementary reactions vary according to the problem type, although they always have the same high-level logical forms; thus, CRO may be applied to a range of problem types (see Sect. 26.4) and so is a *metaheuristic.*

Each elementary reaction may be thought of as generating new molecules (candidate solutions) which are neighbours of the old in a neighbourhood structure in the search space.[3] Neighbouring states are expected to have similar values of potential energy, that is, similar objective function values.

The two kinds of ineffective reactions are viewed as moving to nearby states in the search space and so are analogous to micro-mutations in Evolutionary Algorithms, while the two kinds of 'effective' chemical reactions are viewed as moving to further away (structurally different) states in the search space and so are analogous to crossover. In this way, the CRO algorithm at a high level is analogous to SA, though it has the possibility of combining features of two previous candidate solutions.

26.3.3 The High-Level CRO Algorithm

With these four kinds of elementary reaction introduced, we may now present the high-level CRO algorithm. Subsequently, we will give more details on the four elementary reactions, and so flesh out the CRO algorithm.

We assume a modelling exercise has been carried out to identify the objective function, the form of the search space, appropriate molecular structures and other properties (such as the particular forms of the four kinds of elementary reaction), and appropriate stopping criteria. All of these depend on the problem type. The stopping criterion may be defined based on one or

[2]This term is not actually used in [356] or other works by these authors, but seems an appropriate complement to 'ineffective'.

[3]In CRO, the search space is often referred to as the Potential Energy Space or PES.

more of an upper bound on CPU time used, the maximum number of iterations, an objective function value less than a predefined threshold obtained, the maximum number of iterations performed without improvements or any other relevant criteria.

The CRO run is then initialised. We choose values for CRO variables and control parameters, including the initial population size, N_{pop}.[4] Table 26.2, after [356], shows the symbols used in CRO.

When starting the CRO run, N_{pop} reactant molecules are metaphorically placed in a closed container and initialised by assigning molecular structures randomly. Thus, initially the molecules are distributed evenly over the search space to enhance the chances of exploring all important areas. Variations of this random initialisation are possible, where a proportion (such as 10%) of the population may be generated based on the modeller's domain knowledge; see, for example, [357].

In the actual run of iterations, the molecules collide randomly, with each other and the container walls. The time between collision events is not considered important: what is important is the changes in potential and kinetic energy of each molecule (and the buffer) in each event. Different types of collisions cause different types of elementary reactions depending on parameter values and other conditions. For each collision, a random number t is generated uniformly in the interval $[0, 1]$. If $t > p_{\text{MoleColl}}$, the collision will be unimolecular; otherwise, the collision will be intermolecular. Having decided upon the number of molecules to be involved in the reaction, we randomly select a suitable number of molecules from *Pop*.

Next, we decide upon the type of reaction among the chosen molecules, as ineffective (that is, on-wall or intermolecular ineffective collision as appropriate) or effective (decomposition or synthesis as appropriate, the former in the case of one molecule, the latter for more than one). The details of these are given later, and their implementation is generally problem-specific. The iterations repeat until one of the stopping criteria is met.

An ongoing record is kept throughout of which molecule has the molecular structure with the least potential energy found so far. The final solution obtained by CRO is the molecular structure with lowest potential energy found during the whole run of the algorithm, and this is output to the calling routine.

The pseudocode of CRO is given in Algorithm 26.1; there, and subsequently, E_{buffer} denotes the quantity of energy in the central energy buffer. The inputs to the algorithm are the problem-specific information: objective function f, constraints, and dimension N_{vars} of the problem. The output is the minimal solution and its objective function value. *Success* is a Boolean variable.

We now give more details on the four elementary reactions.

---

[4] The population size in CRO is not fixed, since it changes during the decomposition and synthesis events: it increases by 1 in the former, since a molecule splits into two, and decreases by 1 in the latter, since two molecules combine to give one.

Type	*Symbol*	*Algorithmic Meaning*	*Chemical Meaning*
Function	f	Objective function	Function defining search space S
	Nbh	Neighbour candidate generator	Neighbourhood structure on S
Variable	N_{vars}	Number of variables representing solution dimension of the problem	number of molecule characteristics
	Pop	Population (set) of solutions 2-D matrix where each row carries the values of a solution	Set of molecules
	PE	Vector of objective function values; $PE = f(Pop)$	Potential energy of all the molecules
	KE	Vector whose components measure solutions' tolerance to having worse objective function values afterward	Kinetic energy of all the molecules
Parameter	N_{pop}	Initial number of solutions maintained number of rows in Pop	Initial number of molecules in container
	$\text{rate}_{\text{loss}}^{KE}$	Percentage upper limit of reduction of KE in on-wall ineffective collisions	Percentage upper limit of KE lost to environment in on-wall ineffective collisions
	p_{MoleColl}	Fraction of all elementary reactions corresponding to intermolecular reactions	Same as the algorithmic meaning
	KE_{initial}	Initial value of KE assigned to each element in initialisation stage	KE of the initial set of molecules

Table 26.2. Symbols and parameters used in CRO

Algorithm 26.1: CRO Algorithm

Assign values to parameters N_{pop}, $\text{rate}_{\text{loss}}^{KE}$, p_{MoleColl} and KE_{initial};
Let Pop be the set of molecules $\{1, 2, \ldots, N_{\text{pop}}\}$;
for *each molecule* ω **do**
    Assign a random solution to the molecular structure ω;
    Assess potential energy $= f(\omega)$;
    Assign kinetic energy to ω using KE_{initial};
end
Let $E_{\text{buffer}} := 0$;
while *terminating condition not met* **do**
    Choose random t in interval $[0, 1]$;
    if $t > p_{MoleColl}$ **then**
        Randomly select molecule M from Pop;
        if *decomposition criterion met* **then**
            $(M_1, M_2, Success) = decompose(M, E_{\text{buffer}})$;
            if Success **then**
                Remove M from Pop;
                Add M_1, M_2 to Pop;
            end
        else
            $ineffCollOnWall(M, E_{\text{buffer}})$;
        end
    else
        Randomly select molecules M_1, M_2 from Pop;
        if *synthesis criterion met* **then**
            $(M', Success) = synthesise(M_1, M_2)$;
            if Success **then**
                Remove M_1, M_2 from Pop;
                Add M' to Pop;
            end
        else
            $interIneffColl(M_1, M_2)$;
        end
    end
    Check for new minimum solution
end
Output the overall minimum solution and its objective function value;

26.3.4 On-wall Ineffective Collision

An on-wall ineffective collision occurs when a molecule hits a container wall and bounces off it. Some (physical) molecular attributes associated with motion change in this collision. However, as the collision is regarded as not being very ‘vigorous’, the resultant molecular structure is not expected to be very different from the original one: only its motion has changed, not its chemical structure.

Suppose the current molecular structure is ω. The molecule will be altered to a new structure $\omega' = \text{Nbh}(\omega)$ in its neighbourhood of the search space in this collision. The change is energetically possible (that is, allowed) only if:

$$PE_\omega + KE_\omega \leq PE_{\omega'}. \tag{26.2}$$

Recalling that kinetic energy is lost to the buffer, we get $KE_{\omega'} = (PE_\omega + KE_\omega - PE_{\omega'}) \times q$ where $q \in [\text{rate}^{KE}_{\text{loss}}, 1]$, and $1 - q$ represents the proportion of KE lost to the environment when it hits the container wall.

If (26.2) does not hold, the change is prohibited and the molecule retains its original structure ω, including its potential energy and kinetic energy. The pseudocode of the on-wall ineffective collision is given in Algorithm 26.2. The inputs to the algorithm are a molecule M with its profile and the central energy buffer E_{buffer}. The outputs are the revised M and E_{buffer}.

Algorithm 26.2: Ineffective Collision on Wall

```
Let ω' = Nbh(ω);
Compute PE_ω' := f(ω');
if PE_ω + KE_ω ≥ PE_ω' then
    Select k randomly from interval [rate^KE_loss, 1];
    KE_ω' = (PE_ω + KE_ω − PE_ω') × k;
    Update E_buffer := E_buffer + (PE_ω + KE_ω − PE_ω') × (1 − k);
    Update profile of M by ω := ω', PE_ω := PE_ω' and KE_ω := KE_ω'
end
Output M and E_buffer;
```

26.3.5 Decomposition

In a decomposition event, a molecule hits a container wall and decomposes into two or more molecules (typically two is assumed in the CRO framework). The collision is ‘vigorous’: not only the (physical) molecular attributes associated with motion change in this collision; the resultant molecular structure is also very different from the original one: it has changed chemically. Denote the molecular structure of the original molecule by ω and those of the resultant

molecules by ω_1' and ω_2'. If the original reactant molecule has sufficient total energy (potential and kinetic energy) to meet the structural (potential energy) requirements of the resultant product molecules, that is,

$$PE_\omega + KE_\omega \geq PE_{\omega_1'} + PE_{\omega_2'}, \tag{26.3}$$

the change is energetically possible and is allowed. Let $E_1 = PE_\omega + KE_\omega - PE_{\omega_1'} - PE_{\omega_2'}$, the difference between the left-hand and right-hand sides of (26.3): it is the amount of energy available for kinetic energy, over and above the potential energy requirements of the two new molecules. To distribute this excess between ω_1' and ω_2', let k be a random number generated uniformly from the interval $[0, 1]$. Then we may let $KE_{\omega_1'} = E_1 \times k$ and $KE_{\omega_2'} = E_1 \times (1 - k)$.

However, it is unusual for (26.3) to hold, since neighbours typically have similar values of potential energy. For (26.3) to hold, we would need the kinetic energy of ω to be comparable to its potential energy. Yet this rarely occurs, because of the buffer effect: the kinetic energy of molecules tends to decrease in a sequence of on-wall ineffective collisions over the evolution of the chemical process. To make decomposition more energetically feasible, we allow the use of energy stored in the central buffer (E_{buffer}) to increase the chance of being able to cover $PE_{\omega_1'}$ and $PE_{\omega_2'}$. That is, should (26.3) not hold, we consider the alternative:

$$PE_\omega + KE_\omega + E_{\text{buffer}} \geq PE_{\omega_1'} + PE_{\omega_2'} \tag{26.4}$$

If (26.4) holds, we allow the reaction to occur, and calculate:

$$KE_{\omega_1'} = (E_1 + E_{\text{buffer}}) \times m_1 \times m_2 \tag{26.5}$$

and

$$KE_{\omega_2'} = (E_1 + E_{\text{buffer}} - KE_{\omega_1'}) \times m_3 \times m_4 \tag{26.6}$$

where m_1, m_2, m_3 and m_4 are random numbers independently uniformly generated from the interval $[0, 1]$.[5] Then E_{buffer} is updated to $E_1 + E_{\text{buffer}} - KE_{\omega_1'} - KE_{\omega_2'}$. If neither (26.3) or (26.4) holds, this decomposition operation does not occur, and the molecule keeps its original ω, potential energy and kinetic energy. The pseudocode of the decomposition is given in Algorithm 26.3. The inputs to the algorithm are a molecule M with profile ω and the central buffer energy E_{buffer}. The outputs are molecules M_1' and M_2', and E_{buffer}.

26.3.6 Intermolecular Ineffective Collision

An intermolecular ineffective collision occurs when two molecules collide with each other and then bounce apart without a chemical reaction.

[5] This approach of multiplication by two random numbers from $[0, 1]$ in both (26.5) and (26.6) is used in [356] to ensure that the energies $KE_{\omega_1'}$ and $KE_{\omega_2'}$ gain from the buffer are not too large, because E_{buffer} is usually large. There are other ways to achieve this.

Algorithm 26.3: Decomposition

Create new molecules M_1' and M_2';
Generate ω_1' and ω_2' by decomposing ω;
Compute $PE_{\omega_1'} := f(\omega_1')$ and $PE_{\omega_2'} := f(\omega_2')$;
Let $E_1 = PE_\omega + KE_\omega - PE_{\omega_1'} - PE_{\omega_2'}$;
Define a Boolean variable *Success*;
if $E_1 \geq 0$ **then**
  Success := **true**;
  Generate k randomly in interval $[0, 1]$;
  $KE_{\omega_1'} = E_1 \times k$;
  $KE_{\omega_2'} = E_1 \times (1 - k)$;
  Assign ω_1', $PE_{\omega_1'}$ and $KE_{\omega_1'}$ to the profile of M_1', and ω_2', $PE_{\omega_2'}$ and $KE_{\omega_2'}$ to the profile of M_2';
else if $E_1 + E_{\text{buffer}} \geq 0$ **then**
  Success := **true**;
  Generate m_1, m_2, m_3, m_4 independently and randomly in interval $[0, 1]$;
  $KE_{\omega_1'} = (E_1 + E_{\text{buffer}}) \times m_1 \times m_2$;
  $KE_{\omega_2'} = (E_1 + E_{\text{buffer}} - KE_{\omega_1'}) \times m_3 \times m_4$;
  Update $E_{\text{buffer}} := E_{\text{buffer}} + E_1 - KE_{\omega_1'} + KE_{\omega_2'}$;
  Assign ω_1', $PE_{\omega_1'}$ and $KE_{\omega_1'}$ to the profile of M_1', and ω_2', $PE_{\omega_2'}$ and $KE_{\omega_2'}$ to the profile of M_2';
else
  Success := **false**;
end
Output M_1', M_2' and E_{buffer};

As with an on-wall ineffective collision, some (physical) molecular attributes associated with motion change in this collision. However, as the collision is regarded as not being very ‘vigorous’, the resultant molecular structures are not expected to be very different from the original ones: only their motions have changed, not their chemical structures. That is, the products should be ‘close’ to the reactants in the search space.

The main difference between this operation and an on-wall ineffective collision is that this elementary reaction involves more than one molecule (two molecules are assumed in this framework) and no kinetic energy is transferred to the central energy buffer. Denote the original molecular structures by ω_1 and ω_2. We generate two new molecular structures ω_1' and ω_2' from the neighbourhoods of ω_1 and ω_2, respectively. The changes to the molecules are energetically possible only if:

$$PE_{\omega_1} + PE_{\omega_2} + KE_{\omega_1} + KE_{\omega_2} \geq PE_{\omega_1'} + PE_{\omega_2'}. \tag{26.7}$$

Let E_2 be the difference between the left-hand and right-hand sides of (26.7), $E_2 = PE_{\omega_1} + PE_{\omega_2} + KE_{\omega_1} + KE_{\omega_2} - PE_{\omega_1'} - PE_{\omega_2'}$. We let $KE_{\omega_1'} = E_2 \times k$

and $KE_{\omega_2'} = E_2 \times (1-k)$ where k is a random number uniformly generated from the interval [0, 1]. The molecules retain their original values of ω_1, ω_2, PE_{ω_1}, PE_{ω_2}, KE_{ω_1} and KE_{ω_2} if (26.7) does not hold.

An intermolecular ineffective collision allows the molecular structures to change to a larger extent than an on-wall ineffective collision, since two molecules are involved and the total of their kinetic energies is typically larger than the kinetic energy of a single molecule. The pseudocode of the intermolecular ineffective collision is given in Algorithm 26.4. The inputs to the algorithm are molecules M_1 and M_2 with their profiles. The outputs are the revised molecules M_1 and M_2.

Algorithm 26.4: Intermolecular Ineffective Collision

Let $\omega_1' = \text{Nbh}(\omega_1)$ and $\omega_2' = \text{Nbh}(\omega_2)$;
Compute $PE_{\omega_1'} := f(\omega_1')$ and $PE_{\omega_2'} := f(\omega_2')$;
Let $E_2 = PE_{\omega_1} + PE_{\omega_2} + KE_{\omega_1} + KE_{\omega_2} - PE_{\omega_1'} - PE_{\omega_2'}$;
if $E_2 \geq 0$ **then**
  Select k randomly from interval $[0, 1]$;
  $KE_{\omega_1'} := E_2 \times k$;
  $KE_{\omega_2'} := E_2 \times (1-k)$;
  Update profile of M_1 by $\omega_1 := \omega_1'$, $PE_{\omega_1} = PE_{\omega_1'}$ and $KE_{\omega_1} = KE_{\omega_1'}$, and profile of M_2 by $\omega_2 := \omega_2'$, $PE_{\omega_2} = PE_{\omega_2'}$ and $KE_{\omega_2} = KE_{\omega_2'}$;
end
Output M_1 and M_2;

26.3.7 Synthesis

A synthesis operation occurs when more than one molecule (assume two molecules) collide and chemically combine to give a single new molecule.

Denote by ω_1 and ω_2 the molecular structures of the two original molecules, and by ω' the molecular structure of the new molecule. We expect the product ω' to be quite different from the two reactants ω_1 and ω_2: synthesis is a "vigorous" reaction and may move us far in the search space. We may use any mechanism for synthesis which combines ω_1 and ω_2 in a reasonable way to form ω'. The product ω' is energetically possible only if:

$$PE_{\omega_1} + PE_{\omega_2} + KE_{\omega_1} + KE_{\omega_2} \geq PE_{\omega'}. \tag{26.8}$$

Then the amount of kinetic energy available for ω' is the excess of the left-hand side of (26.8) over the right-hand side: $KE_{\omega'} = PE_{\omega_1} + PE_{\omega_2} + KE_{\omega_1} + KE_{\omega_2} - PE_{\omega'}$. If (26.8) does not hold, then we retain ω_1, ω_2, PE_{ω_1}, PE_{ω_2}, KE_{ω_1}, and KE_{ω_2}, instead of synthesising ω', $PE_{\omega'}$ and $KE_{\omega'}$.

Normally, the product's kinetic energy, $KE_{\omega'}$, is large compared to KE_{ω_1} or KE_{ω_2}, because $PE_{\omega'}$ commonly has a value similar to that of PE_{ω_1} or PE_{ω_2}, and so there is usually a substantial excess of the left-hand side of (26.8) over the right-hand side. Because of this large kinetic energy, synthesis generally gives the product molecule M with ω' greater ability to escape from a local minimum during subsequent elementary reactions in which M is involved. Because of this excess kinetic energy effect, it is very rare to need to transfer energy from the buffer to allow synthesis to proceed; thus, this possibility of transference is not included in the synthesis operation. The pseudocode of the synthesis collision is given in Algorithm 26.5. The inputs to the algorithm are molecules M_1, M_2 with their profiles. The outputs are the single (sythesised) molecule M' and the Boolean variable *Success*.

Algorithm 26.5: Synthesis Collision

```
Create new molecule M';
Construct ω' from ω1 and ω2;
Compute PE_ω' := f(ω');
Define a Boolean variable Success;
if PE_ω1 + PE_ω2 + KE_ω1 + KE_ω2 ≥ PE_ω' then
    Success := true;
    KE_ω' := PE_ω1 + PE_ω2 + KE_ω1 + KE_ω2 − PE_ω';
    Assign ω', PE_ω' and KE_ω' to profile of M';
else
    Success := false;
end
Output M' and Success;
```

26.4 Applications of CRO

Given the relative recency of introduction of the CRO algorithm, there have been a limited number of applications of it.

In Lam and Li [356], the authors introduce CRO, and apply it to some classical problems: the quadratic assignment problem or QAP; the resource constrained project scheduling problem or RCPSP; and the channel assignment problem in wireless networks. On the quadratic assignment problem, CRO is compared to fast ant, improved SA and tabu search. All the heuristics are able to find the global minimum, but CRO outperforms the others in terms of mean and maximum costs found. On the resource constrained project scheduling problem, CRO finds the best makespan for 116 instances without using any special RCPSP heuristics, and this result is reasonable when compared to the 129 instances of the best-performing methods. On the channel

assignment problem in wireless networks, CRO is compared to tabu search and outperforms it on all instances examined. In [667], CRO is applied to the short adjacent repeat identification problem or SARIP, arising in mapping DNA. It is compared to the best method so far, Bayesian approach for short adjacent repeat detection, or BASARD. The four elementary reactions are equipped with the five moves used in BASARD. Since this is a sequence or order-based problem, the intermolecular reactions swap half of the starting positions and structures between the two original molecules, analogous to GA crossover for order-based problems. The algorithms are run on two synthetic cases, single and multiple segments, as well as real data. In each case, CRO on average finds better solutions than does BASARD, and is some 25 to 100 times faster in terms of computational time. In [357], CRO is applied to the fuzzy rule learning problem, with the cooperative rules (COR) approach for search space construction and selection of the most cooperative fuzzy rule set incorporated, giving the COR-CRO algorithm. Here, a molecular structure ω carries a vector of consequents for a given combination of antecedents. The elementary reactions are set up as follows:

- on-wall ineffective collision: one component of ω is randomly selected and modified to another one within its defined interval.
- decomposition: ω breaks into two pieces ω_1 and ω_2; each inherits half of the components of ω, randomly but exclusively; the other noninherited components are randomly generated.
- intermolecular ineffective collision between two molecules ω_1 and ω_2: either one or two components (according to whether the variation COR1 or COR2 is used) of each of ω_1 and ω_2 are modified.
- synthesis: if two molecules ω_1 and ω_2 are synthesised to get ω', each component of ω' is chosen randomly from the corresponding component of ω_1 and ω_2, analogously to uniform crossover in GA.

The approach is tested on two problems, three-dimensional surface modelling and electrical low-voltage line length estimation, against other approaches such as NIT, WM, WM-ALM, I-method, I-ALM, COR-SA, COR-BWAS, COR1-GA and COR2-GA, and others, depending on the problem. On each problem, COR-CRO appears to be competitive with the other approaches tested, though the results do not seem clear-cut.

The literature also includes some applications of chemical reaction optimisation to computer network and other problems, for example, task scheduling in grid computing [666] and population transition in peer-to-peer live streaming [358]. Further work on real-valued CRO is presented in [678].

26.5 Discussion of CRO

As in other metaheuristics, the aim in CRO is to explore only the most relevant and useful parts of the search space, in the sense that these parts are most

likely to contain a global minimum. Exploration is achieved through collisions among molecules and the container wall, namely, the four types of elementary reactions. [356] identifies two kinds of exploration: intensification and diversification. Given a molecule at a point in the search space, intensification explores the immediately surrounding area, analogous to micro-mutation in an evolutionary algorithm (EA). Diversification allows jumps to relatively distant points in the search space, analogous to a crossover or macro-mutation in an EA. In CRO, intensification is mainly achieved by the ineffective collision types, both on-wall and intermolecular, while diversification is mainly achieved by the operations decomposition and synthesis, which radically change molecular structure.

Exploitation of previously gained information is achieved in CRO by the gradual redistribution of kinetic energy from the molecules to the container walls. The rate at which this occurs may be modified during the run, thus altering the balance of exploration and exploitation.

In [356], Lam and Li state that, for certain problems at least (such as the Quadratic Assignment Problem or QAP), the ineffective elementary reactions act mainly as intensification operators (with some diversification), while the other two chemical elementary reactions act almost completely as diversification operators. As described by them, CRO thus differs somewhat from algorithms such as EAs, where the operators have more clearly defined roles, but is very close in spirit to SA (that is, SA with a number of neighbour generation operators).

From the description of the CRO algorithm, the main idea seems to be closely related to SA:

- The main chemical idea is the transformation of products to reactants, with the rest being related to conservation of energy and other physical ideas, and an exploration/exploitation approach very reminiscent of SA/quantum annealing.
- Lam and Li [356] state that the most distinguishable features of CRO are “the central energy buffer and the concept of energy exchange”. The ‘buffer’ is effectively an absorber of kinetic energy and its action is equivalent to reducing the temperature in SA, since temperature is just a measure of the molecules’ average kinetic energy.
- The KE_{initial} parameter controls average molecule starting kinetic energy, and so is a proxy for the initial temperature in SA.
- A solution is generated as a neighbour of the current solution, according to certain legal neighbourhood operations (the four CRO elementary reactions); this is the same Markov network paradigm used by SA and related approaches.
- CRO allows ‘bad’ moves as in SA, by allowing borrowing of energy from the buffer to give products with greater net energy than the reactants.

There are also similarities to EAs; for example:

- The operation 'On-Wall Ineffective Collision' is meant to change a molecular structure (solution) by a small amount, analagous to an EA mutation.
- Depending on the solution structure, the Decomposition operation is analogous to crossover of a given parent with a parent of random entries. This occurs, for example, in the Fuzzy Rule Learning application in [357].
- Again, depending on the solution structure, the Synthesis operation is analogous to crossover of two given parents to generate a single offspring. This occurs, for example, in the Fuzzy Rule Learning application in [357], where effectively uniform crossover is used.

However, although Lam et al. describe CRO as a populational algorithm [357], the amount of interaction between individuals appears to be limited, and only one or two molecules (depending on the operation chosen) are acted upon at each iteration. Also, the population size may increase (by decomposition) or decrease (by synthesis) and so is not fixed, as is usual in EAs.

There are also limited aspects of tabu search, since the outcome (products) of an operation is not allowed to be the same as the input (reactants), even in the 'Ineffective Collision' pair of operations.

26.5.1 Potential Future Avenues for Research

Further work needs to be done to establish the exact connection of CRO to SA, in particular to determine whether CRO is equivalent to SA equipped with several neighbour selection heuristics, allowing both micro and macro moves in the search space, and whether convergence results for SA can be carried over to CRO.

Lam and Li [356] have set up CRO so that "All the events are triggered by collisions (on wall or among molecules)". An extension of the method could allow for spontaneous decay (independent of collisions) of an unstable molecule into simple molecules, as occurs in nature.

Note that in the decomposition event of [356], "If the total energy of the original molecule is not enough to support the change, additional energy can be drawn from the central buffer", which may violate the Second Law of Thermodynamics, since the energy in the buffer may not be usable (insufficient thermal potential). This nonphysicality — adherence to the First Law (Conservation of Energy) but not the Second — is the case in all of their events and an algorithm more closely inspired by nature could be obtained by enforcing the Second Law also. However, it is not clear a priori whether such a revised algorithm would be more or less effective than CRO.

As mentioned previously, there are strong results for the asymptotic convergence of SA and SQA and related algorithms based on Markov Chain and/or Monte Carlo approaches. However, there do not appear to be equivalent results for CRO. This may be a reflection of the relative newness of CRO, or it may be that the connection to SA has not been formally made. A possible avenue for research here is to view the buffer in CRO as reducing available kinetic energy to molecules, thus decreasing the average KE and

thus the temperature. Then varying the buffer size systematically would be a proxy for temperature reduction. Applying the equivalent of a known asymptotically convergent cooling schedule (such as logarithmic cooling) to derive a buffer size modification schedule could then allow the results from SA to carry across.

Further possibilities along these lines would include allowing the parameter values $\text{rate}_{\text{loss}}^{KE}$ and p_{MoleColl} to vary according to systematic rules, and so control the proportion of ineffective and other operations and thus of exploration versus exploitation.

At a higher level, there are many aspects of chemistry as found in nature that are, to date, not incorporated in chemically inspired algorithms. These include the concepts of:

i. Individual atoms, each of a particular type of element, and what these could mean in algorithmic terms; some atoms may form few bonds (e.g., sodium, which has only one electron in a relatively high energy level) while others may form many (e.g., carbon, which may bond with up to four other atoms). In particular, the atomic number gives the number of electrons in an atom in its nonionised state; however, in practical terms, only the electrons in the outermost atomic orbitals determine the atom's chemical properties; how could this translate to an algorithm?
ii. Bonds among two or more atoms (covalent, ionic, metallic, etc.) and algorithmic analogies — what is meant by a bond in the problem domain, what is meant by valency, and what could be meant by a shared electron?
iii. Reaction rates — in chemistry these are described by differential equations in variables such as time, temperature, and concentrations of reactants and products; tuning these could allow more control of the exploration/exploitation balance.
iv. The extra complexity of organic chemistry, the chemistry of compounds of the carbon atom; here, enormous molecules (such as DNA) may be formed because of the ability of carbon atoms to bond with up to four other atoms; there are potential analogies to GP, with bond sites representing the attachment point of a subtree. Molecules could be mutated at specific locations based on the natural process of an acid or base unit being replaced by another acid or base; chemical potentials could be used to make certain sites/bonds more or less susceptible to breakup.
v. Could semantics, an active area of research in GP, be realised by the above chemical concepts such as atoms/molecules of different elements, different valencies, or different molecular structures?

It is likely that the meaning of such analogies, should they be successfully made, would be highly problem domain-specific. Investigation of the range of applicability of algorithmic analogies of these concepts may well reward further study.

26.6 Summary

In this chapter we have explored a new family of natural computing algorithms, chemically inspired algorithms. They are analogous to classical physically inspired algorithms such as Simulated Annealing, but employ particular neighbour generation routines that allow both small and large jumps in the search space, in a way reminiscent of Evolutionary Algorithms. Thus far, they lack the mathematical underpinning that Simulated Annealing, Simulated Quantum Annealing and similar algorithms have; but, as considered above, this may be achievable by analogy with Simulated Annealing. They have shown promise in a number of applications.

Part VIII

The Future of Natural Computing Algorithms

27

Looking Ahead

In this book we have described natural computing algorithms in terms of seven categories, namely, evolutionary computing (Part I), social computing (Part II), neurocomputing (Part III), immunocomputing (Part IV), developmental and grammatical computing (Part V), physical computing (Part VI), and other paradigms (Part VII). One cannot but be impressed by the rich tapestry of algorithms that have been developed thus far and we are confident that many more algorithms will be proposed in the coming years taking new sources of inspiration from the natural world.

27.1 Open Issues

While much progress has been made in advancing the field of natural computing over the last two decades, the field is far from mature and many open research avenues remain. Below we touch on a number of them.

27.1.1 Hybrid Algorithms

As noted in the first chapter of this book, the compartmentalisation of the natural computing paradigm into lineages, such as we have undertaken in this book, is somewhat artificial, as while it assists the construction of a structured exposition of each family of algorithms, it runs the risk of obfuscating the deep links and overlaps that exist in the natural world between the systems which have served as their inspiration.

One trend which we expect in the future is that of hybridisation of algorithms from the different forms of natural computing. It is quite common for practitioners to combine elements of these approaches when faced with challenging, real-world problems. These problems are often high-dimensional, dynamic and noisy, and their solution requires an ability to adapt on multiple time horizons. We expect future natural computing algorithms to combine many (perhaps indeed all) levels of adaptation as outlined in the POE model

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_27

(Sect. 1.1.1), resulting in truly complex and adaptive natural computing algorithms capable of the kind of problem solving potential realised in the natural world.

27.1.2 The Power and the Dangers of Metaphor

Computational processes abound in nature and natural phenomena have proven to be a rich vein of inspiration for the design of computational algorithms. Of course, as science advances and as we deepen our understanding of natural systems, it is reasonable to expect that new salient features will emerge which can be incorporated into novel natural computing algorithms. One just has to look at the rapid advances which have been made in molecular biology and genetics in the past 15 years to underscore this potential. We are at the dawn of the age of systems biology which seeks to understand biological systems as a whole, rather than through classical (silo) science perspectives. Understanding and modelling the importance of the interactions which occur between the different levels of the traditional 'omics' (e.g., genomics and proteomics) will develop our understanding of complex adaptive systems and potentially give rise to the development new, powerful, algorithms.

On the other hand, we need to recognise the limits of natural metaphors. For example, biological processes (for example, foraging strategies) occur in specific ecological niches or settings, and there is no reason to suppose a priori that a good strategy within one niche will necessarily be useful elsewhere. As noted by [587], we also need to be careful to avoid proliferating and overclaiming for 'novel' natural computing algorithms. There has been a tendency to introduce algorithms which although being derived from different natural processes (or organisms) appear to bear notable similarities in terms of their workings to previously developed algorithms. Employing greater rigour in the testing and theoretical analysis of new algorithms will help differentiate between "me too" algorithms and those which are truly useful and novel.

One of the weaker points of natural computing research to date has been the variability of the depth of theory underpinning many of the developed algorithms, especially for some of the most recently developed algorithms. The theoretical basis varies from strong convergence results for some algorithms, to purely experimental results on a limited set of examples ("toy problems": see Sect. 27.1.3). Advanced statistical and information theoretic methods should help us open the door to a stronger theoretical foundation for the entire field, in turn driving the creation of more efficient and effective natural computing algorithms.

27.1.3 Benchmarks and Scalability

Given the rich diversity of natural computing algorithms, it is challenging to compare and contrast their relative benefits and drawbacks. Often problems examined by researchers are small scale instances of a problem class and

these do not provide insight as to how well an algorithm will scale to real-world-sized problems. As has been demonstrated in the genetic algorithm (GA) community, it is often the case that canonical versions of the algorithms scale poorly, sometimes requiring exponentially increasing population sizes to tackle linearly increasing problem difficulty (order). Scalability studies of all new algorithms are required in order to assess their practical utility. Researchers also need to pay more attention to the relative utility of algorithms for different classes of problems, and identify those features which might best be combined to create efficient and effective algorithms for specific problem classes. The development of a set of robust benchmarks which proponents of new algorithms are required to test against may go some way towards building an understanding of the strengths and weaknesses of different algorithms.

27.1.4 Usability and Parameter-Free Algorithms

One practical issue in applying many natural computing algorithms is the number of parameters which must be set in order to apply the solver to a specific problem. Similarly, there can often be a myriad of design choices to be made for any one algorithm, with choices between alternative strategies for different steps within the algorithm. This complexity serves as a barrier to entry for practitioners. From an ease-of-use perspective, practitioners would prefer algorithms which will work "out-of-the-box", and as such are either parameter-free or which incorporate, so far as is possible, self-adapting settings.

27.1.5 Simulation and Knowledge Discovery

Although the focus of this book has been on the drawing of inspiration from the natural world for the design of computational algorithms, there is another side to natural computing, that of seeking to better understand the natural world itself. As we deepen our scientific understanding in various disciplines, our ability to design algorithms which can serve as faithful simulation models of the underlying systems will be enhanced. In turn, scientists may be able to use these algorithms to produce a deeper understanding of the natural world.

27.2 Concluding Remarks

The preceding discussion does not contain an exhaustive list of open issues in natural computing, but does serve to illustrate the breadth and depth of issues currently facing researchers in this community. It is safe to say that the natural computing paradigm is at an exciting phase in its history with an increasing rate of discovery in the natural systems which serve as their inspiration and an increasing maturity in terms of the rigor of testing being

applied to new algorithms. We hope that this book has served as a source of inspiration to you the reader, and that if you are not already an active researcher in this field or a practitioner of these methods, you will join us in the inspiring adventure that is natural computing.

References

1. Abbass H A (2001a) A monogenous MBO approach to satisfiability. In: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation (CIMCA 2001) Las Vegas, USA
2. Abbass H A (2001b) Marriage in honey-bee optimization (MBO): a haplometrosis polygynous swarming approach. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2001), pp 207–214, IEEE Press
3. Aeppli G, Rosenbaum T F (2005) Experiments on quantum annealing. In: Das A, Chakrabarti B (eds) Quantum Annealing and Related Optimization Methods, Lecture Notes in Physics 679, pp 159–169, Springer-Verlag
4. Aharonov D, van Dam W, Kempe J, Landau Z, Lloyd S, Regev O (2007) Adiabatic quantum computation is equivalent to standard quantum computation. SIAM J Comput 37(1): 166–194. Preprint at http://arxiv.org/quant-ph/0405098.
5. Ahn C, Ramakrishna R, Goldberg D (2004) Real-Coded Bayesian Optimization Algorithm: Bringing the Strength of BOA into the Continuous World. In: Proceedings of the 6th Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science 3102, pp 840–851, Springer
6. Aickelin U, Bentley P, Cayzer S, Kim J, McLeod J (2003) Danger theory: The link between AIS and IDS. In: Proceedings of the Second International Conference on Artificial Immune Systems (ICARIS 03), pp 147–155, Springer
7. Aickelin U, Cayzer S (2002) The Danger Theory and Its Application to Artificial Immune Systems. In: Proceedings of the 1st International Conference on Artificial Immune Systems, pp 141–148, Canterbury, UK.
8. Al Toufailia H, Couvillon M, Ratnieks F and Gruter C (2013) Honey bee waggle dance communication: signal meaning and signal noise affect dance follower behaviour. Behavioral Ecology and Sociobiology 67:549-556
9. Alatas B (2011) ACROA: Artificial chemical reaction optimization algorithm for global optimization. Expert Systems with Applications 38(10):13170–13180
10. Alatas B (2012) A novel chemistry based metaheuristic optimization method for mining of classification rules. Expert Systems with Applications 39:11080–11088
11. Alba E, Tomassini M (2002) Parallelism and Evolutionary Algorithms. IEEE Transactions on Evolutionary Computing 6(5):443–461

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8

12. Alpi A, Amrhein N, Bertl A et al. (2007) Plant neurobiology: no brain, no gain? Trends in Plant Science 12(4):135–136
13. Amarteifio S (2005) Interpreting a genotype-phenotype map with rich representations in XMLGE. MSc Thesis, University of Limerick
14. Amintoosi M, Fathy M, Mozayani N and Rahmani A (2007). A Fish School Clustering Algorithm: Applied to Student Sectioning Problem. In Proceedings of 2007 International Conference on Life System Modelling and Simulation (LSMS 2007), Watam Press
15. Angeline P (1996) Two Self-Adaptive Crossover Operators for Genetic Programming. In: Angeline P J, Kinnear K E Jr (eds) Advances in Genetic Programming 2, pp 89–110, MIT Press
16. Angeline P (1998) Using selection to improve particle swarm optimization. In: Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (CEC 1999), pp 84–89, IEEE Press
17. Anstey M, Rogers S, Ott S, Burrows M, Simpson S (2009) Serotonin Mediates Behavioral Gregarization Underlying Swarm Formation in Desert Locusts. Science 323(5914):627–630
18. Arita H, Fenton B (1997) Flight and echolocation in the ecology and evolution of bats. Tree 12(2):53–58
19. Arnett R (1985) American Insects: A Handbook of the Insects of America North of Mexico. New York: Van Nostrand Reinhold
20. Araujo L, Santamaria J (2010) Evolving natural language grammars without supervision. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010), pp 1–8, IEEE Press
21. Azad R M A (2003) A position independent representation for evolutionary automatic programming algorithms: the chorus system. PhD Thesis, University of Limerick
22. Bäck T (1996) Evolutionary Algorithms in Theory and Practice. Oxford University Press
23. Bak P, Sneppen K (1993) Punctuated equilibrium and criticality in a simple model of evolution. Phys Rev Lett 71(24):4083–4086
24. Bak P, Tang C, Wiesenfeld K (1987) Self-organized criticality: An explanation of the $1/f$ noise. Phys Rev Lett 59(4):381–384
25. Baker J (1987) Reducing bias and inefficiency in the selection algorithm. In: Grefenstette, J. (ed) Proceedings of the Second International Conference on Genetic Algorithms, pp 14–21, Hillsdale, N.J., Lawrence Erlbaum Associates
26. Baluja S (1994) Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. Technical Report CMU-CS-94–163, Pittsburgh, PA, Carnegie Mellon University
27. Baluja S, Caruana R (1995) Removing genetics from the standard genetic algorithm. In: Proceedings of Twelfth International Conference on Machine Learning, pp 38–46, San Mateo, CA: Morgan Kaufmann
28. Baluska F, Lev-Yadun S, Mancuso S (2010) Swarm intelligence in plant roots. Trends in Ecology & Evolution 25:682–683
29. Banks A, Vincent J and Phalp K (2009) Natural strategies for search. Natural Computing 8(3):547–570

30. Banzhaf W (1994) Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Proceedings of Parallel Problem Solving from Nature III (PPSN III), Lecture Notes in Computer Science 866, pp 322–332, Springer
31. Banzhaf W, Nordin P, Keller R E, Francone F D (1998) Genetic Programming – An Introduction: On the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann
32. Banzhaf W (2003) Artificial Regulatory Networks and Genetic Programming. In: Genetic Programming - Theory and Applications, pp 43–61, Kluwer Academic Publishers
33. Banzhaf W (2004) On Evolutionary Design, Embodiment and Artificial Regulatory Networks. In Lecture Notes in Artificial Intelligence 3139, Embodied Artificial Intelligence, pp 284–292, Springer
34. Bartholdi J, Seeley T, Tovey C, Vande Vate J (1993) The pattern and effectiveness of forager allocation among flower patches by honey bee colonies. Journal of Theoretical Biology 160:23–40
35. Barto A G, Sutton, R S, Anderson C W (1983) Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man and Cybernetics 13:834–846
36. Basak A, Pal S, Das S, Abraham A, Snasel V (2010) A Modified Invasive Weed Optimization Algorithm for Time-Modulated Linear Antenna Array Synthesis. In Proceedings of the IEEE World Congress on Computational Intelligence (WCCI 2010), pp 372–379, IEEE Press
37. Bassler B (2002) Small Talk: Cell-to-Cell Communication in Bacteria. Cell 109:421–424
38. Bastos Filho C, de Lima Neto F, Lins A, Nascimento A and Lima M (2008) A Novel Search Algorithm Based on Fish School Behavior. In Proceedings of IEEE International Conference on Systems, Man and Cybernetics (SMC 2008), pp 2646–2651, IEEE Press
39. Bastos Filho C, de Lima Neto F, Sousa M, Pontes M and Madeiro S (2009) On the Influence of the Swimming Operators in the Fish School Search Algorithm. In Proceedings of IEEE International Conference on Systems, Man and Cybernetics (SMC 2009), pp 5012–5017, IEEE Press
40. Battaglia D, Stella L (2006) Optimization through quantum annealing: theory and some applications. Contemporary Physics 47(4):195–208
41. Baykasoglu A, Ozbakir L, Tapkan P (2007) Artificial Bee Colony Algorithm and Its Application to Generalized Assignment Problem. In: Swarm Intelligence, Focus on Ant and Particle Swarm Optimization, Chan, F. T. S. and Tiwari, M. K. (eds), pp 113–144, InTech Education and Publishing, Vienna
42. Beadle L, Johnson C G (2008) Semantically Driven Crossover in Genetic Programming. In: Proceedings of the IEEE World Congress on Computational Intelligence (CEC 2008), pp 111–116, IEEE Press
43. Beadle L, Johnson C G (2009) Semantic Analysis of Program Initialisation in Genetic Programming. Genetic Programming and Evolvable Machines 10(3):307–337
44. Beadle L, Johnson C G (2009) Semantically Driven Mutation in Genetic Programming. In: Proceedings of the 2009 IEEE Congress on Evolutionary Computation (CEC 2009), pp 1336–1342, IEEE Press

45. Beekman M, Fathke R, Seeley T (2006) How does an informed minority of scouts guide a honeybee swarm as it flies to its new home? Animal Behavior 71(1):161–171
46. Beer C, Hendtlass T, Montgomery J (2012) Improving Exploration in Ant Colony Optimisation with Antennation. In Proceedings of the IEEE World Congress on Computational Intelligence 2012 (WCCI 2012), pp 2926–2933, IEEE Press
47. Bell, J.S. (1964). On the Einstein-Podolsky-Rosen Paradox. Physics 1: 195–200.
48. Bellanta J, Kadlec J (1985) Introduction to immunology. In: Bellanti J (ed) Immunology: Basic Processes, pp 1–15, Philadelphia: W.B. Saunders
49. Bellman R E (1961) Adaptive Control Processes. Princeton, NJ: Princeton University Press
50. Benioff P (1980) The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines. Journal of Statistical Physics 22: 563–591.
51. Berdahl A, Torney C, Ioannou C, Faria J and Couzin I (2013) Emergent Sensing of Complex Environments by Mobile Animal Groups. Science 339(6119):574–576
52. Berg H, Brown D (1972) Chemotaxis in Escherichia coli analysed by three-dimensional tracking. Nature 239:500–504
53. Beyer H-G (2001) The Theory of Evolutionary Strategies. Springer
54. Beyer H-G, Schwefel H-P (2002) Evolution strategies: A comprehensive introduction. Natural Computing 1:3–52
55. Bilchev G, Parmee I (1995) The ant colony metaphor for searching continuous design spaces. In: Proceedings of AISB Workshop on Evolutionary Computing. Lecture Notes in Computer Science 993, pp 25–39, Springer-Verlag.
56. Birbil S I, Fang S C (2003) An electromagnetism-like mechanism for global optimization. Journal of Global Optimization 25:263–282
57. Birchenhall C, Kastrinos N, Metcalfe S (1997) Genetic Algorithms in Evolutionary Modelling. Journal of Evolutionary Modelling 7(4):375–393
58. Bishop C M (1995) Neural Networks for Pattern Recognition. Oxford: Oxford University Press
59. Bishop C M (2007) Pattern Recognition and Machine Learning. New York: Springer-Verlag
60. Blackwell T M, Bentley P J (2002) Dynamic search with charged swarms. In: Spector et al. (eds) Proceedings of the 4th Genetic and Evolutionary Computation Conference (GECCO 2002), pp 19–26, Morgan Kaufmann
61. Blackwell T (2003) Swarms in Dynamic Environments. In: E. Cantú-Paz et al. (eds) Proceedings of the 5th Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science 2723, pp 1–12, Springer-Verlag
62. Blackwell T M, Branke J (2004) Multi-swarm optimization in dynamic environments. In: Proceedings of EvoSTOC Workshop 2004, Lecture Notes in Computer Science 3005, pp 489–500, Springer
63. Boettcher S (1999) Extremal optimization of graph partitioning at the percolation threshold. Journal of Physics A: Mathematical and General 32(28):5201–5211

64. Boettcher S, Percus A G (1999) Extremal optimization: Methods derived from co-evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), pp 825–832, Morgan Kaufmann
65. Boettcher S, Percus A G (2000) Nature's way of optimizing. Artif Intell 119(1–2):275–286
66. Boettcher S, Percus A G (2001) Optimization with extremal dynamics. Phys Rev Lett 86(23):5211–5214
67. Boettcher S, Percus A G (2002) Extremal optimization: an evolutionary local-search algorithm. CoRR cs.NE/0209030
68. Bonabeau E, Dorigo M, Theraulaz G (1999) Swarm Intelligence: From Natural to Artificial Systems. Oxford: Oxford University Press
69. Bora T, Coelho L, Lebensztajn L (2012) Bat-inspired Optimization Approach for the Brushless DC Wheel Motor Problem. IEEE Transactions on Magnetics 48(2):947–950
70. Born M, Fock V A (1928) Beweis des Adiabatensatzes. Zeitschrift für Physik A Hadrons and Nuclei 51(3–4): 165-180.
71. Boser B, Guyon I, Vapnik V (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the 4th Workshop on Computational Learning Theory, pp 144–152, ACM Press
72. Branke J et al. (eds) (2010) Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO 2010). 7–11 July, Portland, OR, USA, ACM Press
73. Brenner E, Stahlberg R, Mancuso S, Vivanco J, Baluska F, Van Volkenburgh E (2006) Plant neurobiology: an integrated view of plant signaling. Trends in Plant Science 11(8):413–419
74. Brits R, Engelbrecht A, van den Bergh F (2002) A niching particle swarm optimizer. In: Proceedings of the Fourth Asia-Pacific Conference on Simulated Evolution and Learning (SEAL 2002), 2:692–696, IEEE Press
75. Brabazon A, O'Neill M (2006) Biologically Inspired Algorithms for Financial Modelling. Springer
76. Bremermann H (1974) Chemotaxis and optimization. Journal of the Franklin Institute 297(5):397–404
77. Brooke J, Bitko D, Rosenbaum T F, Aeppli G (1999) Quantum annealing of a disordered magnet. Science 284(5415):779–781
78. Brown D, Berg H (1974) Temporal Stimulation of Chemotaxis in Escherichia coli. Proc. Nat. Acad. Sci. (USA) 71(4):1388–1392
79. Brownlee J (2007) Clonal Selection Algorithms. Centre for Information Technology Research (CITR), Technical Report 070209A, pp 1–13, Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology
80. Brownlee J (2005) Artificial Immune Recognition System (AIRS): A Review and Analysis. Centre for Intelligent Systems and Complex Processes (CISCP), Technical Report 1-02, pp 1–39, Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology
81. Bryson A E, Ho Y-C (1969) Applied optimal control: optimization, estimation, and control. Blaisdell Publishing Company, Xerox College Publishing
82. Bullnheimer B, Hartl R, Strauss C (1999) A new rank-based version of the Ant-System: A computational study. Central European Journal for Operations Research and Economics 7(1):25–38

83. Byrne J (2012) Approaches to Evolutionary Architectural Design Exploration Using Grammatical Evolution. PhD Thesis. University College Dublin
84. Byrne J, O'Neill M, McDermott J, Brabazon A (2009) Structural and Nodal Mutation in Grammatical Evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009), pp 1881–1882, ACM Press
85. Byrne J, McDermott J, Galvan-Lopez E, O'Neill M (2010) Implementing an Intuitive Mutation Operator for Interactive Evolutionary 3D Design. In: Proceedings of the 2010 IEEE World Congress on Computational Intelligence (WCCI 2010), pp 2919–2925, IEEE Press
86. Byrne J, McDermott J, O'Neill M, Brabazon A (2010) An Analysis of the Behaviour of Mutation in Grammatical Evolution. In: Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010), Lecture Notes in Computer Science 6021, pp 14–25, Springer
87. Byrne J, Fenton M, McDermott J, Hemberg E, O'Neill M, McNally C, Shotton E (2011) Combining Structural Analysis and Multi-Objective Criteria for Evolutionary Architectural Design. In: Proceedings of EvoMUSART 2011 the 9th European Event on Evolutionary and Biologically Inspired Music, Sound, Art and Design, Lecture Notes in Computer Science 6625, pp 200–209, Springer
88. Byrne J, Hemberg E, O'Neill M, Brabazon A (2012) A Local Search Interface for Interactive Evolutionary Architectural Design. In: Proceedings of EvoMUSART 2012 European Conference on Evolutionary and Biologically Inspired Music, Sound, Art and Design. Lecture Notes in Computer Science 7247, pp 23–34, Springer
89. Cahill J, McNickle G (2011) The Behavioral Ecology of Nutrient Foraging by Plants. Annual Review of Ecology, Evolution and Systematics 42:289–311
90. Cai W, Yang W, Chen X (2008) A Global Optimization Algorithm Based on Plant Growth Theory: Plant Growth Optimization. In: Proceedings of 2008 International Conference on Intelligent Computation Technology and Automation (ICICTA 2008), pp 1194–1199, IEEE Press
91. Calixto M (2009) Quantum computation and cryptography: an overview. Natural Computing 8(4): 663–679.
92. Campbell D (1969) Variation and Selective Retention in Socio-Cultural Evolution. General Systems 14:69–85
93. Campelo F, Guimaraes F, Igarshi H (2005) A Clonal Selection Algorithm for Oprimization in Electromagnetics. IEEE Transactions on Magnetics 41(5):1736–1739
94. Carpenter G, Grossberg S (1987) ART 2: Self-organization of stable category recognition codes for analog input patterns. Applied Optics 26(23):4919–4930
95. Carpenter G, Grossberg S, Reynolds J (1991) ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. Neural Networks 4:565–588
96. Cavalcanti-Junior G, Bastos-Filho C and Lima-Neto F (2012) Volitive Clan PSO - An Approach for Dynamic Optimization Combining Particle Swarm Optimization and Fish School Search. In Theory and New Applications of Swarm Intelligence R Parpinelli (ed), pp 69–86, Intech
97. Cerny V (1982) A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. Technical report, Comenius University, Bratislava, Czechoslovakia

98. Cerny V (1985) Thermodynamical Approach to the Travelling Salesman Problem: An Efficient Simulation Algorithm. Journal of Optimization Theory and Applications 45:41–51
99. Chakrabarti B K, Dutta A, Sen P (1996) Quantum Ising Phases and Transitions in Transverse Ising Models. Springer-Verlag
100. Chamovitz D (2012a) What a plant smells. Scientific American, May 2012, pp 48–51
101. Chamovitz D (2012b) What a Plant Knows. New York: Scientific American Books.
102. Chen J (2002) A heuristic approach to efficient production of detector sets for an artificial immune algorithm-based bankruptcy prediction system. In Proceedings of the Congress on Evolutionary Computation 2002 (CEC 2002), 1:932–937, New Jersey: IEEE Press
103. Chen S (2009) Locust Swarms - A new multi-optima search technique. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2009), pp 1745–1752, IEEE Press
104. Chen S and Vargas Y (2010) Improving the performance of particle swarms through dimension reductions - A case study with locust swarms. In: Proceedings of IEEE Congress on Evolutionary Computation 2010 (CEC 2010), pp 1–8, IEEE Press
105. Chena L, Aihara K (1995) Chaotic simulated annealing by a neural network model with transient chaos. Neural Networks 8(6):915–930
106. Chester D L (1990) Why Two Hidden Layers are Better than One. In: Proceedings of IJCNN 1990, Washington, DC, 1:265–268, Lawrence Erlbaum
107. Chomsky N (1956) Three models for the description of language. IRE Transactions on Information Theory 2(3):113–124
108. Chomsky N (1957) Syntactic Structures. The Hague, Netherlands: Mouton
109. Chomsky N (1975) Reflections on Language. New York: Pantheon Books
110. Chong C, Low M, Sivakumar A, Gay K (2006) A Bee Colony Optimization Algorithm to Job Shop Scheduling. In: Proceedings of the 2006 Winter Simulation Conference (WinterSim 2006), pp 1954–1961, New Jersey: IEEE Press
111. Ciszack M, Comparini D, Mazzolai B, Baluska F, Arecchi F, Vicsek T, Mancuso S (2012) Swarming behavior in plant roots. PLOS One 7(1) e29759
112. Cleary R (2005) Extending grammatical evolution with attribute grammars: an application to knapsack problems. MSc Thesis. University of Limerick
113. Cleary R, O'Neill M (2005) An Attribute grammar decoder for the 01 Multi-Constrained Knapsack Problem. In: Proceedings of the European Conference on Evolutionary Combinatorial Optimisation (EvoCOP 2005), Lecture Notes in Computer Science 3488, pp 34–45, Springer
114. Clerc M (1999) The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In: Proceedings of the Congress of Evolutionary Computation 1999 (CEC 1999), pp 1951–1957, IEEE Press
115. Cobb H (1990) An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report 6760, Naval Research Laboratory, Washington, D.C.
116. Coello Coello C, Lamont G, and Van Veldhuizen D (2007) Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd ed. Springer
117. Cortes C and Vapnik V (1995) Support vector networks. Machine Learning 20:273–297

118. Cover T (1965) Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. IEEE Transactions in Electron. Comput. EC-14:326–334
119. Cramer N L (1985) A representation for the adaptive generation of simple sequential programs. In: Proceedings of the International Conference on Genetic Algorithms and Their Applications, pp 183–187, Lawrence Erlbaum and Associates
120. Crist E (2004) Can an insect speak? The case of the honeybee dance language. Social Studies of Science (Sage Publications) 34(1):7–43
121. Cristianini N, Shawe-Taylor J (2000) An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press
122. Cui W, Brabazon A, O'Neill M (2011) Dynamic Trade Execution: A Grammatical Evolution Approach. International Journal of Financial Markets and Derivatives 2(1/2):4–31
123. Curry R, Heywood M (2009) One-class Genetic Learning. In: Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009), Lecture Notes in Computer Science 5481, pp 1–12, Springer
124. Cutello V, Nicosia G, Pavone M (2004) Exploring the capability of immune algorithms: A characterization of hypermutation operators. In: Proceedings of the Third International Conference on Artificial Immune Systems (ICARIS04), pp 263–276, Springer.
125. Cybenko G (1989) Approximation by superpositions of a sigmoidal function. Math. Control Signal Systems 2:303–314
126. da Cruz A, Vellasco M, Pacheco M (2006) Quantum-inspired evolutionary algorithm for numerical optimization. In Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 9180–9187, IEEE Press
127. Darwin C (1859) On the Origin of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life. (reprinted 1985) London: Penguin Books
128. Darwin, C (1880) The Power of Movement in Plants. London: John Murray
129. Das A, Chakrabarti B (eds) (2005) Quantum Annealing and Related Optimization Methods. Lecture Notes in Physics 679, Springer
130. Das A, Chakrabarti B (2008) Quantum annealing and analog quantum computation. Reviews of Modern Physics 80(3):1061–1081
131. Das S, Suganthan P N (2011) Differential Evolution: A survey of the state-of-art. IEEE Transactions on Evolutionary Computation 15(1):4–31
132. Das S, Konar A, Chakraborty U (2005) Two Improved Differential Evolution Schemes for Faster Global Search. In: Proceedings of the 7th Genetic and Evolutionary Computation Conference (GECCO 2005), pp 991–998, ACM Press
133. Dasgupta S, Das S, Abraham A, Biswas A (2009) Adaptive Computational Chemotaxis in Bacterial Foraging Optimization: An Analysis. IEEE Transactions on Evolutionary Computation 13(4):919–941
134. Dawkins R (1976) The Selfish Gene. Oxford University Press
135. Deb K (2001) Multi-Objective Optimization using Evolutionary Algorithms. Wiley
136. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2):182–197

137. Debels D, De Reyck B, Leus R, Vanhoucke M (2006) A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. European Journal of Operational Research 169(2):638–653
138. Deboeck G, Kohonen T (eds) (1998) Visual Explorations in Finance with self-organizing maps. Berlin: Springer-Verlag
139. De Castro L, Timmis J (eds) (2002) Artificial Immune Systems: A New Computational Intelligence Approach. Springer
140. De Castro L, Von Zuben F (2000) The clonal selection algorithm with engineering applications. In: Workshop Proceedings of the 2nd Genetic and Evolutionary Computation Conference (GECCO 2000), pp 36–37
141. De Castro L, Von Zuben F (2001) aiNET: An artificial immune network for data analysis. In: Abbass H, Sarker R, Newton C (eds) Data Mining: A Heuristic Approach, pp 231–259, Idea Group Publishing
142. De Castro L, Von Zuben F (2002) Learning and optimization using the clonal selection principle. IEEE Transactions on Evolutionary Computation 6(3):239–251
143. Dechmann D, Kranstauber B, Gibbs D, Wikelski M (2010) Group Hunting - A Reason for Sociality in Molossid Bats. PLoS One 5(2):e9012
144. De Jong K (2006) Evolutionary Computation: A unified approach. Cambridge, MA: MIT Press
145. de la Maza M, Tidor B (1993) An Analysis of Selection Procedures with Particular Attention Paid to Proportional and Boltzmann Selection. In: Forrest S (ed) Proceedings of the 5th International Conference on Genetic Algorithms, pp 124–131, Morgan Kaufmann
146. Dempsey I (2007) Grammatical Evolution in Dynamic Environments. PhD Thesis. University College Dublin
147. Dempsey I, O'Neill M, Brabazon A (2004) Grammatical constant creation. In: Proceedings of the 6th Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science 3103, pp 447–458, Springer
148. Dempsey I, O'Neill M, Brabazon A (2005) Meta-grammar constant creation. In: Proceedings of the 7th Genetic and Evolutionary Computation Conference (GECCO 2005), pp 1665–1672, ACM Press
149. Dempsey I, O'Neill M, Brabazon A (2007) Constant Creation with Grammatical Evolution. International Journal of Innovative Computing and Applications 1(1):23–38
150. Dempsey I, O'Neill M, Brabazon A (2009) Foundations in Grammatical Evolution for Dynamic Environments. Springer
151. Dempster A, Laird N, Rubin D (1977) Maximum Likelihood from Incomplete Data via the EM algorithm. Journal of the Royal Statistical Society, Series B 39(1):1–38
152. Deneubourg J, Gross S, Franks N, Sendova-Franks A, Detrain C, Chretie, L (1991) The dynamics of collective sorting robot-like ants and ant-like robots. In: Meyer J, Wilson S (eds) Proceedings of 1st Conference on Simulation of Adaptive Behavior: From Animals to Animats (SAB 90), pp 356–365, MIT Press
153. DeRosier D (1998) The Turn of the Screw: The Bacterial Flagellar Motor. Cell 93:17–20

154. Deutsch D (1985) Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. Proceedings of the Royal Society (London) A400: 97–117.
155. Devert A, Bredeche N, Schoenauer M (2007) Robust Multi-Cellular Developmental Design. In: Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO 2007), pp 982–989, ACM Press
156. de Villiers J, Barnard E (1992) Backpropagation Neural Networks with One and Two Hidden Layers. IEEE Transactions on Neural Networks 4(1):136–141
157. Dickmans D, Schmidhuber J, Winklhofer A (1987) Der genetische Algorithmus: Eine Implementierung in Prolog. Institut fur Informatik, Technische Universität München, Fortgeschrittenenpraktikum, http://www.idsia.ch/juergen/geneticprogramming.html
158. Dieks D (1982) Communication by EPR Devices. Physics Letters A 92: 271–272.
159. Diez L, Le Borgne H, Lejeune P, Detrain C (2013) Who brings out the dead? Necrophoresis in the red ant, *Myrmica ruba*. Animal Behaviour 80:1259–1264
160. Diosan L, Oltean M (2009) Evolutionary Design of Evolutionary Algorithms. Genetic Programming and Evolvable Machines 10:263–306
161. Dittmer H (1937) A quantitative study of the roots and root hairs of a winter rye plant. American Journal of Botany 25:417–420
162. Diwold K, Beekman M, Middendorf M (2010) Bee nest site selection as an optimization process. In: Proceedings of ALife XII Conference, pp 626–633, MIT Press
163. Diwold K, Himmelbach D, Meier R, Baldauf C, Middendorf M (2011) Bonding as a Swarm: Applying Bee Nest-Site Selection Behavior to Protein Docking. In: Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO 2011), pp 93–100, ACM Press
164. Dorigo M (1992) Optimization, Learning and Natural Algorithms. PhD Thesis. Politecnico di Milano
165. Dorigo M, DiCaro G (1999) Ant colony optimization: a new meta-heuristic. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 1999), pp 1470–1477, IEEE Press
166. Dorigo M and Gambardella L (1997) Ant Colony System: A cooperative Learning Approach to the Travelling Salesman Problem. IEEE Transactions on Evolutionary Computation 1:53–66
167. Dorigo M and Gambardella L (1997) Ant colonies for the travelling salesman problem. BioSystems 43:73–81
168. Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics 26(1):29–41
169. Dorigo M, Stützle T (2004) Ant Colony Optimization, Cambridge, Massachusetts, MIT Press
170. Duda R, Hart P and Stork D (2001) Pattern Classification, 2nd edn. NY: Wiley
171. Dunham B, Fridshal D, North J H (1963) Design by Natural Selection. Synthese 15:254–259
172. Dunn J (1973) A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. Journal of Cybernetics 3:32–57
173. Eberhart R, Dobbins R, Simpson P (1996) Computational Intelligence PC Tools, Boston, MA: Academic Press

174. Edmonds B (1998) Meta-Genetic Programming: Co-evolving the Operators of Variation. CPM Report 98–32, Centre for Policy Modelling, Manchester Metropolitan University, UK
175. Eiben A, Smith J (2015) Introduction to Evolutionary Computing. Second edition. Berlin: Springer-Verlag
176. Elman J (1990) Finding structures in time. Cognitive Science 14:179–211
177. Engelbrecht A (2005) Fundamentals of Computational Swarm Intelligence. Chichester, UK: Wiley
178. Engelbrecht A (2012) Particle Swarm Optimization: Velocity Initialization. In: Proceedings of the IEEE World Congress on Computational Intelligence 2012 (WCCI 2012), pp 70–77, IEEE Press
179. Escuela G, Ochoa G, Krasnogor N (2005) Evolving L-Systems to Capture Protein Structure Native Conformations. In: Proceedings of the 8th European Conference on Genetic Programming (EuroGP 2008), Lecture Notes in Computer Science 3447. pp 74–84, Springer
180. Fagan D (2014) An Analysis of Genotype-Phenotype Mapping in Grammatical Evolution. PhD Thesis, University College Dublin.
181. Fagan D, Hemberg E, O'Neill M, McGarraghy S (2013) Understanding Expansion Order and Phenotypic Connectivity in πGE. In: Proceedings of the 16th European Conference on Genetic Programming (EuroGP 2013), Lecture Notes in Computer Science 7831, pp 37-48, Springer.
182. Fagan D, Nicolau M, O'Neill M, Galvan-Lopez E, Brabazon A (2010) Investigating Mapping Order in πGE. In: Proceedings of the IEEE Congress on Evolutionary Computation 2010 (WCCI 2010), pp 3058–3064, IEEE Press
183. Fagan D, O'Neill M, Galvan-Lopez E, Brabazon A, McGarraghy S (2010) An analysis of Genotype-Phenotype Maps in Grammatical Evolution. In: Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010), Lecture Notes in Computer Science 6021. pp 62–73, Springer
184. Fagan D, Nicolau M, Hemberg E, O'Neill M, Brabazon A, McGarraghy S (2011) Investigation of the Performance of Different Mapping Orders for GE on the Max Problem. In: Proceedings of the 14th European Conference on Genetic Programming (EuroGP 2011), Lecture Notes in Computer Science 6621. pp 286–297, Springer.
185. Fagan D, Hemberg E, O'Neill M, McGarraghy S (2012) Fitness Reactive Mutation in Grammatical Evolution. In: Proceedings of the 18th International Conference on Soft Computing (Mendel 2012), Brno, Czech Republic
186. Fahlman S (1988) Faster-learning variations on back-propagation: An empirical study. Sejnowski T J, Hinton G E, Touretzky D S (eds) 1988 Connectionist Models Summer School, pp 38–51, San Mateo, CA: Morgan Kaufmann
187. Federici D, Downing K (2006) Evolution and Development of a Multicellular Organism: Scalability, Resilience, and Neutral Complexification. Artificial Life 12(3):381–409
188. Fenton B (2013) Questions, ideas and tools: lessons from bat echolocation. Animal Behaviour 85:869–879
189. Feynman R (1982) Simulating Physics with Computers. International Journal of Theoretical Physics 21 (6&7): 467–488.
190. Finnila A, Gomez M, Sebenik C, Stenson C, Doll J (1994) Quantum annealing: A new method for minimizing multidimensional functions. Chem. Phys. Lett. 219:343–348

191. Firn R (2004) Plant Intelligence: an Alternative Point of View. Annals of Botany 93:345–351
192. Fogel D (ed) (1998) Evolutionary Computation: The Fossil Record, IEEE Press
193. Fogel D (2000) Evolutionary Computation: Towards a New Philosophy of Machine Intelligence, IEEE Press
194. Fogel L, Owens A, Walsh M (1966) Artificial Intelligence through Simulated Evolution, New York: Wiley
195. Fogel L J (1962) Autonomous automata. Industrial Research 4:14–19
196. Fonseca C, Fleming P (1993) Genetic Algorithms for Multiobjective Optimization. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp 416–423, San Mateo: Morgan Kaufmann
197. Formato R A (2007) Central force optimization: a new metaheuristic with applications in applied electromagnetics. Progress in Electromagnetics Research 77:425–491
198. Formato R A (2008) Central force optimization: a new nature inspired computational framework for multidimensional search and optimization. Studies in Computational Intelligence 129:221–238
199. Forrest S, Perelson A, Allen L, Cherukuri R (1994) Self-nonself discrimination in a computer. In: Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, pp 202–212, IEEE Press
200. Foster J A (2001) Evolutionary computation. Nature Genetics Reviews 2:428–436
201. Freeland S, Hurst L (2004) Evolution encoded. Scientific American, April 2004, pp 84–91
202. Freitas A, Timmis J (2007) Revisiting the Foundations of Artificial Immune Systems for Data Mining. IEEE Transactions on Evolutionary Computation 11(4):521–540
203. Friedberg R M (1958) A Learning Machine: Part 1. IBM J. Research and Development 2(1):2–13
204. Friedberg R M, Dunham B, North J H (1959) A learning machine: Part 2. IBM J. Research and Development 3:282–287
205. Fromm J, Lautner S (2007) Electrical signals and their physiological significance. Plant, Cell and Environment 30:249–257
206. Galvan-Lopez E, Fagan D, Murphy E, Swafford J M, Agapitos A, O'Neill M, Brabazon A (2010) Comparing the Performance of the Evolvable PiGrammatical Evolution Genotype-Phenotype Map to Grammatical Evolution in the Dynamic Ms. Pac-Man Environment. In: Proceedings of the IEEE Congress on Evolutionary Computation 2010 (WCCI 2010), pp 1587–1594, IEEE Press
207. Gandomi A, Yang X-S, Alavi A, Talatahari S (2013) Bat algorithm for constrained optimization tasks. Neural Computing and Applications 22:1239–1255
208. Garavaglia S (2002) A quantum-inspired self-organizing map (QISOM). In: Proceedings of 2002 IEEE International Joint Conference on Neural Networks (IJCNN 2002), pp 1779–1784, IEEE Press
209. Geman S, Geman D (1984) Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. IEEE Transactions on Pattern Analysis and Machine Intelligence 6:721–741
210. Gilbert S F (2006) Developmental Biology (8th edn). Sinauer Associates
211. Goldberg D (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Boston: Addison-Wesley

212. Goldberg D (2002) The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers
213. Goldberg D, Richardson J (1987) Genetic algorithms with sharing for multimodal function optimization. In: Proceedings of the second international conference on genetic algorithms, pp 41–49, L. Erlbaum Associates Inc.
214. Gonzalez F, Dasgupta D (2003) Anomaly detection using real-valued negative selection. Genetic Programming and Evolvable Machines 4(4):383–403
215. Gonzalez L, Cannady J (2004) A self-adaptive negative selection approach for anomaly detection. In: Proceedings of the IEEE Congress on Evolutionary Computation 2004 (CEC 2004), pp 1561–1568, New Jersey: IEEE Press
216. Gordon T G W, Bentley P J (2005) Bias and Scalability in Evolutionary Development. In: Proceedings of the 7th Genetic and Evolutionary Computation Conference (GECCO 2005), pp 83–90, ACM Press
217. Grahl J, Bosman P, Rothlauf F (2006) The correlation-triggered adaptive variance scaling IDEA. In: Proceedings of the 8th Genetic and Evolutionary Computation Conference (GECCO 2006), pp 397–404, ACM Press
218. Granovskiy B, Latty T, Duncan M, Sumpter D and Beekman M (2012) How dancing honey bees keep track of changes: the role of inspector bees. Behavioral Ecology, published online 16 February 2012, http://beheco.oxfordjournals.org/content/early/2012/02/16/beheco.ars002.full.pdf+html
219. Grefenstette J (1992) Genetic algorithms for changing environments. In: Proceedings of Parallel Problem Solving from Nature II (PPSN II), pp 137–144, Elsevier Science Inc.
220. Greensmith J, Aickelin U, Cayzer S (2008) Detecting Danger: The Dendritic Cell Algorithm. HP Laboratories Report HPL-2008-200, http://ima.ac.uk/papers/cayzer2008.pdf, accessed 27 July 2013
221. Greensmith J, Feyereisl J, Aickelin U (2008) The DCA: SOMe Comparison: A comparative study between two biologically-inspired algorithms. Evolutionary Intelligence 1(2):85–112
222. Greensmith J, Twycross J, Aickelin U (2006) Dendritic Cells for Anomaly Detection. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2006) pp 664–671, IEEE Press
223. Greensmith J, Aickelin U, Cayzer S (2005) Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection. In: Proceedings of 4th International Conference on Artificial Immune Systems (ICARIS 2005), Lecture Notes in Computer Science 3627, pp 153–167, Springer
224. Griffin D (1944) Echolocation by blind men, bats and radar. Science 100:589–590
225. Griffin D (1958) Listening in the Dark. Yale University Press, New Haven, CT
226. Grime J, Mackey J (2002) The role of plasticity in resource capture by plants. Evolutionary Ecology 16:299–307
227. Grossberg S (1976) Adaptive pattern classification and universal recoding, II: Feedback, expectation, olfaction and illusions. Biological Cybernetics 23:187–2l2
228. Grossberg S (1980) How does a brain build a cognitive code? Psychological Review 1:1–51
229. Grover L K (1996) A fast quantum mechanical algorithm for database search. In: Proceedings of the 28th ACM Symposium on the Theory of Computing, Piscataway, NJ, Nov 1994. IEEE Press. 212–219.

230. Grover L K (1997) Quantum computers can search arbitrarily large databases by a single query. Phys. Rev. Lett. 79: 4709–4712.
231. Gruau F (1994) Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. PhD Thesis. L'École Normale Supérieure de Lyon, l'Université Claude Bernard Lyon 1
232. Grünbaum D, Viscido S and Parrish J (2004) Extracting interactive control algorithms from group dynamics of schooling fish. Coop Control Lecture Notes in Control and Information Sciences (LNCIS 309), pp 103–117, Springer
233. Gruter C, Segers F and Ratnieks F (2013) Social learning strategies in honeybee foragers: do the costs of using private information affect the use of social information? Animal Behaviour 85(6):1143–1449 http://dx.doi.org/10.1016/j.anbehav.2013.03.041
234. Gu F, Greensmith J, Aickelin U (2013) Theoretical formulation and analysis of the deterministic dendritic cell algorithm. Biosystems 111(2):127–135
235. Gu F, Feyereisl J, Oates R, Reps J, Greensmith J, Aickelin U (2011) Quiet in class: classification, noise and the dendritic cell algorithm. In: Proceedings of the 10th International Conference on Artificial Immune Systems (ICARIS 2011), pp 173–186, Springer
236. Gudise V, Venayagamoorthy G (2003) Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In: Proceedings of the 2003 IEEE Swarm Intelligence Symposium (SIS '03), pp 110–117, IEEE Press
237. Guney K, Durmus A, Basbug S (2009) A Plant Growth Simulation Algorithm for Pattern Nulling of Linear Antenna Arrays by Amplitude Control. Progress in Electromagnetics Research B 17:69–84
238. Gurney K (1997) An introduction to Neural Networks. London: University College London Press
239. Hajela P, Li, C-Y (1992) Genetic search strategies in multicriterion optimal design. Structural Optimization 4(2):99–107
240. Han K-H, Kim J-H (2002) Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. IEEE Transactions on Evolutionary Computation 6(6):580–593
241. Han K-H, Kim J-H (2004) Quantum-inspired evolutionary algorithms with a new termination criterion, H_ε gate and two-phase scheme. IEEE Transactions on Evolutionary Computation 8(3):156–169
242. Hancock P (1992) Genetic algorithms and permutation problems: a comparison of recombination operators for neural net structures. In: Proceedings of International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92), pp 108–122, IEEE Press
243. Handl J, Knowles J, Dorigo M (2006) Ant-based Clustering and Topographic Mapping. Artificial Life 12(1):35–62
244. Hansen N, Ostermeier A (1996) Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In: Proceedings of the IEEE International Conference on Evolutionary Computation (CEC 1996), pp 312–317, IEEE Press
245. Hansen N, Ostermeier A (2001) Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation 9(2):159–195
246. Hansen N, Ostermeier A, Gawelczyk A (1995) On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating

set adaptation. In: Eshelman L J (ed) Proceedings of the Sixth International Conference on Genetic Algorithms, pp 57–64, San Mateo, CA: Morgan Kaufmann

247. Hara A, Ichimura T, Fujita N, Takahama T (2006) Effective Diversification of Ant-based Search Using Colony Fission and Extinction. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 3773–3780, IEEE Press
248. Harano K-i, Mitsuhata-Asai A, Konishi T, Suzuki T and Sasaki M (2013) Honeybee foragers adjust crop contents before leaving the hive: Effects of distance to food source, food type, and informational state. Behavioral Ecology and Sociobiology, published online 8 May 2013, DOI 10.1007/s00265-013-1542-5
249. Harding S, Banzhaf W (2008) Artificial Development. In: Wurtz, R. (ed) Organic Computing, pp 201–220, Springer
250. Harding S, Miller J F, Banzhaf W (2007) Self-Modifying Cartesian Genetic Programming. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO 2007), pp 1021–1028, ACM Press
251. Harding S, Miller J F, Banzhaf W (2009) Evolution, Development and Learning using Self-Modifying Cartesian Genetic Programming. In: Proceedings of the 11th annual conference on Genetic and evolutionary computation (GECCO 2009), pp 699–706, ACM Press
252. Harik G, Lobo F, Goldberg D (1998) The compact genetic algorithm. In: Proceedings of the International Conference on Evolutionary Computation (CEC 1998), pp 523–528, New Jersey: IEEE Press
253. Harper R, Blair A (2005) A Structure Preserving Crossover in Grammatical Evolution. In: Proceedings of the 2005 IEEE International Conference on Evolutionary Computation (CEC 2005), pp 2537–2544, IEEE Press
254. Harper R, Blair A (2006) A Self-selecting Crossover Operator. In: Proceedings of the 2006 IEEE International Conference on Evolutionary Computation (CEC 2006), pp 5569–5576, IEEE Press
255. Harper R, Blair A (2006) Dynamically Defined Functions in Grammatical Evolution. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 9188–9195, IEEE Press
256. Hart E, Timmis J (2008) Application areas of AIS: The past, the present and the future. Applied Soft Computing 8(1):191–201
257. Hart W, Krasnogor N, Smith J (eds) (2004) Recent Advances in Memetic Algorithms. Berlin: Springer-Verlag
258. Hastings W K (1970) Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57(1):97–109
259. He D, Qu L and Guo X (2009) Artificial Fish-school Algorithm for Integer Programming. In Proceedings of IEEE International Conference on Information Engineering and Computer Science (ICIECS 2009), pp 1–4, IEEE Press
260. Heil M and Ton J (2008) Long-distance signalling in plant defence. Trends in Plant Science 13(6):264–272
261. Hemberg E (2010) An Exploration of Grammars in Grammatical Evolution. PhD Thesis. University College Dublin
262. Hemberg E, McPhee N, O'Neill M, Brabazon, A (2008) Pre-, In- and Postfix grammars for Symbolic Regression in Grammatical Evolution. In: Proceedings of the IEEE Workshop and Summer School on Evolutionary Comput-

ing, pp 18–22, IEEE Press (also available from http://ncra.ucd.ie/papers/HembergMcPhee_etal.pdf)
263. Hemberg E, O'Neill M, Brabazon A (2009) An investigation into automatically defined function representations in Grammatical Evolution. In: Proceedings of Mendel 2009, 15th International Conference on Soft Computing (also available from http://ncra.ucd.ie/papers/mendel2009ADF.pdf)
264. Hemberg E, Ho L, O'Neill M, Claussen H (2011) A Symbolic Regression Approach To Manage Femtocell Coverage Using Grammatical Genetic Programming. In: Proceedings of the Genetic and Evolutionary Computation SRM Workshop at GECCO 2011, pp 639–646, ACM Press
265. Hemberg E, Ho L, O'Neill M, Claussen H (2012) Evolving Femtocell Algorithms with Dynamic and Stationary Training Scenarios. In: Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN XII), Lecture Notes in Computer Science 7492, pp 518-527, Springer
266. Hemberg E, Ho L, O'Neill M, Claussen H (2012) A Comparison of Grammatical Genetic Programming Grammars for Controlling Femtocell Network Coverage. Genetic Programming and Evolvable Machines 14(1):65–93
267. Hendtlass T (2001) A Combined Swarm Differential Evolution Algorithm for Optimization Problems. In: Proceedings of the Fourteenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Lecture Notes in Computer Science 2070, pp 11–18, Springer
268. Hendtlass T (2004) TSP Optimisation Using Multi Tour Ants. In: Proceedings of 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2004), Lecture Notes in Computer Science 3029, pp 523–532, Springer
269. Hendtlass T (2005) WoSP: a multi-optima particle swarm algorithm. In: Proceedings of the IEEE Congress on Evolutionary Computation 2005 (CEC 2005), pp 727–734, IEEE Press
270. Hendtlass T (2007) TSP optimisation using multi tour ants. In: Proceedings of IEA/AIE, Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Lecture Notes in Computer Science 3029, pp 523–532, Springer
271. Higashi N, Iba H (2003) Particle swarm optimization with Gaussian mutation. In: Proceedings of the 2003 IEEE Swarm Intelligence Symposium (SIS '03) pp 72–79, IEEE Press
272. Higashitani M, Ishigame A, Yasuda K (2006) Particle Swarm Optimization Considering the Concept of Predator-Prey Behavior. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 1541–1544, IEEE Press
273. Hirvensalo, M. (2002). Quantum computing — Facts and Folklore. Natural Computing 1: 135–155.
274. Hoai N X, McKay R I, Abbass H A (2003) Tree adjoining grammars, language bias, and genetic programming. In: Proceedings of 6th European Conference on Genetic Programming (EuroGP 2003), Lecture Notes in Computer Science 2610, pp 340–349, Springer
275. Hoai N X, McKay R I, Essam D (2002) Some Experimental Results with Tree Adjunct Grammar Guided Genetic Programming. In: Proceedings of the 5th European Conference on Genetic Programming (EuroGP 2002), Lecture Notes in Computer Science 2278, pp 228–237, Springer

276. Hoai N X, McKay R I, Essam D (2006) Representation and Structural Difficulty in Genetic Programming. IEEE Transactions on Evolutionary Computation 10(2):157–166
277. Hoai N X, McKay R I, Essam D, Hoang T H (2005) Genetic Transposition in Tree-Adjoining Grammar Guided Genetic Programming: The Duplication Operator. In: Proceedings of the 8th European Conference on Genetic Programming (EuroGP 2005), Lecture Notes in Computer Science 3447, pp 108–119, Springer
278. Hoang T H, McKay R I, Essam D, Hoai N X (2011) On Synergistic Interactions Between Evolution, Development and Layered Learning. IEEE Transactions on Evolutionary Computation 15(3):287–312
279. Hochreiter S, Mozer M (2000) An electric approach to independent component analysis. In: Proceedings of the 2nd International Workshop on Independent Component Analysis and Signal Separation, Helsinki, pp 45–50
280. Hofmeyer S, Forrest S (2000) Architecture for an artificial immune system. Evolutionary Computation 8(4):443–473
281. Holland J H (1975) Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Michigan: University of Michigan Press
282. Hölldobler B and Wilson E (1994) Journey to the Ants: A story of scientific exploration. Belknap Press/Harvard University Press
283. Hordijk W (1995) A Measure of Landscapes. Santa Fe Institute Working Paper 95–05–04, Santa Fe: Santa Fe Institute
284. Hornby G, Pollack J (2001) Evolving L-Systems to Generate Virtual Creatures. Computers and Graphics 25(6):1041–1048
285. Hornik K, Stinchcombe M, White H (1990) Multi-layered feedforward neural networks are universal approximators. Neural Networks 2:359–366
286. Howley T, Madden M (2005) The genetic kernel support vector machine: Description and evaluation. Artificial Intelligence Review 24(3–4):379–395
287. Hu X, Shi Y, Eberhart R (2004) Recent advances in particle swarm. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2004), pp 90–97, IEEE Press
288. Hugosson J, Hemberg E, Brabazon A, O'Neill M (2010) Genotype Representations in Grammatical Evolution. Applied Soft Computing 10(1):36–43
289. Ingber L (1989) Very fast simulated re-annealing. Mathl Comput Modelling 12(8):967–973
290. Ingber L (1993) Simulated annealing: Practice versus theory. Mathl Comput Modelling 18(11):29–57
291. Ishida Y (2004) Immunity-Based Systems: A Design Perspective. Berlin: Springer
292. Jacob C (1994) Genetic L-System Programming. In: Proceedings of the 3rd International Conference on Parallel Problem Solving from Nature (PPSN III), Lecture Notes in Computer Science 866, pp 334–343, Springer
293. Jacob C (1995) Genetic L-System Programming: Breeding and Evolving Artificial Flowers with Mathematica. In: Proceedings of the First International Mathematica Symposium (IMS'95), pp 215–222, Computational Mechanics Publisher
294. Jacob C, Burleigh I (2005) Genetic Programming Inside a Cell. In: Yu T, Riolo R, Worzel B (eds) Genetic Programming Theory and Practice III, pp 191–206, Springer

295. Janecek A and Tan Y (2011) Feeding the Fish - Weight Update Strategies for the Fish School Search Algorithm. In Proceedings of the Second International Conference on Swarm Intelligence (ICSI 2011), pp 553–562, Springer
296. Janson S, Middendorf M, Beekman M (2007) Searching for a new home – scouting behavior of honeybee swarms. Behavioral Ecology 18(2):384–392
297. Jerne N (1974) Towards a Network Theory of the Immune System. Ann. Immunol. (Inst. Pasteur), 125C, pp 373–389
298. Jerne N (1985) The Generative Grammar of the Immune System. EMBO J. 4:847–852
299. Ji Z, Dasgupta D (2004) Augmented negative selection algorithm with variable-coverage detectors. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2004), pp 1081–1088, IEEE Press
300. Ji Z, Dasgupta D (2007) Revisiting Negative Selection Algorithms. Evolutionary Computation 15(2):223–251
301. Jiao L, Li Y (2005) Quantum-inspired immune clonal optimization. In: Proceedings of 2005 IEEE International Conference on Neural Networks and Brain (ICNN&B 2005), pp 461–466, IEEE Press
302. Jin Y, Branke J (2005) Evolutionary Optimization in Uncertain Environments: A Survey. IEEE Transactions on Evolutionary Computation 9(3):303–317
303. Johnson C (2003) Artificial Immune System Programming for Symbolic Regression. In: Proceedings of 6th European Conference on Genetic Programming (EuroGP 2003), Lecture Notes in Computer Science 2610, pp 345–353, Berlin: Springer-Verlag
304. Johnson S (1967) Hierarchical Clustering Schemes. Psychometrika 2:241–254
305. Jones T (1995) Evolutionary Algorithms, Fitness Landscapes and Search. PhD Thesis. Department of Computer Science, University of New Mexico
306. Jones G (2005) Echolocation. Current Biology 15(13)R484–R488
307. Jones J, Dangl J (2006) The plant immune system. Nature 444:323–329
308. Jones G, Teeling E (2006) The evolution of echolocation in bats. Trends in Ecology and Evolution 21(3):149–156
309. Joshi A K, Levy L S, Takahashi M (1975) Tree Adjunct Grammars. Journal of Computer and System Sciences 10(1):136–163
310. Joshi A K, Schabes Y (1991) Tree-Adjoining Grammars and Lexicalised Grammars. Technical Report MS-CIS-91–22, Department of Computer and Information Science, University of Pennsylvania, March 1991 http://repository.upenn.edu/cgi/viewcontent.cgi?article=1463&context=cis_reports&sei-redir=1
311. Joshi A K, Schabes Y (1997) Tree Adjoining Grammars. In: Rozenberg G, Salomaa A (eds) Handbook of Formal Languages, Beyond Words, pp 69–123, Springer-Verlag
312. Kantschik W, Banzhaf W (2002). Linear-graph GP—A new GP Structure. In: Proceedings of the 4th European Conference on Genetic Programming (EuroGP 2002), Lecture Notes in Computer Science 2278, pp 83–92, Springer
313. Kantschik W, Dittrich P, Brameier M, Banzhaf W (1999) Meta-evolution in graph GP. In: Proceedings of the European Conference on Genetic Programming (EuroGP 1999), Lecture Notes in Computer Science 1598, pp 15–28, Springer
314. Karaboga D (2005) An idea based on honeybee swarm for numerical optimization. Technical Report TR06, Engineering Faculty, Computer Engineering

Department, Erciyes University, http://mf.erciyes.edu.tr/abc/pub/tr06_2005.pdf
315. Karaboga D, Akay B (2009) A survey: algorithms simulating bee intelligence. Artificial Intelligence Review 31:61–85
316. Karaboga D, Basturk B (2007) A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization 39:459–471
317. Karaboga D, Ozturk C (2011) A novel clustering approach: Artificial Bee Colony (ABC) algorithm. Applied Soft Computing 11:652–657
318. Kawamura H, Yamamoto M, Suzuki K, Ohuchi A (2000) Multiple Ant Colonies Algorithm Based on Colony Level Interactions. Transactions of the Institute of Electronics, Information and Communication Engineers E83-A:722–741
319. Keijzer M, Babovic V, Ryan C, O'Neill M, Cattolico M (2001) Adaptive Logic Programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pp 42–49, Morgan Kaufmann
320. Keijzer M, O'Neill M, Ryan C, Cattolico M (2002) Grammatical Evolution Rules: The mod and the Bucket Rule. In: Proceedings of the 5th European Conference (EuroGP 2002), Lecture Notes in Computer Science 2278, pp 123–130, Springer-Verlag
321. Keijzer M et al. (eds) (2008) Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO 2008). 12–16 July, Atlanta, GA, USA, ACM Press
322. Keller R E, Banzhaf W (1996) Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes. In: Proceedings of the First Annual Conference on Genetic Programming, pp 116–122, MIT Press
323. Keller R E, Banzhaf W (1999) The Evolution of Genetic Code in Genetic Programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), 2:1077–1082, Morgan Kaufmann
324. Keller R E, Banzhaf W (2001) Evolution of Genetic Code on a Hard Problem. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp 50–56, Morgan Kaufmann
325. Kelsey J, Timmis J (2003) Immune Inspired Somatic Contiguous Hypermutation for Function Optimisation. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2005, pp 207–218, Springer.
326. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks, pp 1942–1948, IEEE Press
327. Kennedy J, Eberhart R (1997) A discrete binary version of the particle swarm algorithm. In: Proceedings of the IEEE Conference on Systems, Man and Cybernetics, pp 4104–4109, IEEE Press
328. Kennedy J, Eberhart R, Shi Y (2001) Swarm Intelligence. San Mateo, California: Morgan Kaufmann
329. Khan N, Goldberg D, Pelikan M (2002) Multiobjective bayesian optimization algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pp 684, San Mateo, California: Morgan Kaufmann
330. Kirkpatrick S, Gelatt C, Vecchi M (1983) Optimization by simulated annealing. Science 220(4598):671–680

331. Kitano H (1990) Designing neural networks using genetic algorithms with graph generation. Complex Systems 4:461–476
332. Kohonen T (1982) Self-organized formation of topologically correct feature maps. Biological Cybernetics 43:59–69
333. Kohonen T (1990) The Self-organizing map. Proceedings of the IEEE 78(9):1464–1480
334. Kohonen T (1998) The SOM Methodology. In: Deboeck G, Kohonen, T (eds) Visual Explorations in Finance with Self-organizing Maps, pp 159–167, Berlin: Springer-Verlag
335. Kokai G, Toth Z, Vanyi R (1999) Modelling Blood Vessels of the Eye with Parametric L-Systems using Evolutionary Algorithms. In: Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making, Lecture Notes in Computer Science 1620, pp 433–443, Springer-Verlag
336. Konak A, Coit D, Smith A (2006) Multi-objective optimization using genetic algorithms: A tutorial. Reliability Engineering and System Safety 91:992–1007
337. Kong X, Chen Y-L, Xie W, Wu X (2012) A Novel Paddy Field Algorithm Based on Pattern Search Method. In: Proceedings of the IEEE International Conference on Information and Automation, pp 686–690, IEEE Press
338. Korosec P, Jurij S (2009) A Stigmergy-Based Algorithm for Continuous Optimization Tested on Real-Life-Like Environment. In: Proceedings of 2nd European Workshop on Bio-inspired Algorithms for Continuous Parameter Optimisation (EvoNum), Lecture Notes in Computer Science 5484, pp 675–684, Springer
339. Koza J (1989) Hierarchical genetic algorithms operating on populations of computer programs. In: Proceedings of the 11th International Joint Conference on Artificial Intelligence, pp 768–774, Morgan Kaufmann
340. Koza J R (1992) Genetic Programming. Massachusetts: MIT Press
341. Koza J R (1994) Genetic Programming II. Massachusetts: MIT Press
342. Koza J R, Andre D, Bennett (III) F H, Keane M (1999) Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann
343. Koza J R, Keane M, Streeter M J, Mydlowec W, Yu J, Lanza, G (2003) Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers
344. Koza J R (2010) Human-competitive results produced by genetic programming. Genetic Programming and Evolvable Machines 11(3–4):251–284
345. Krasnogor N (2002) Studies on the Theory and Design Space of Memetic Algorithms. PhD thesis. Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England
346. Krasnogor N, Smith J (2005) A Tutorial for Competent Memetic Algorithms: Model, Taxonomy, and Design Issues. IEEE Transactions on Evolutionary Computation 9(5):474–488
347. Krasnogor N et al. (eds) (2011) Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO 2011). 12–16 July, Dublin, Ireland, ACM Press
348. Krawiec K (2011) Semantically embedded genetic programming: automated design of abstract program representations. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO 2011), pp 1379–1386, ACM Press

349. Krawiec K, Lichocki P (2009) Approximating geometric crossover in semantic space. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO 2009), pp 987–994, ACM Press
350. Krawiec K, Wieloch B (2009) Analysis of Semantic Modularity for Genetic Programming. Foundations of Computing and Decision Sciences 34(4):265–285
351. Krishnanand K, Ghose D (2005) Detection of Multiple Source Locations using a Glowworm Metaphor with Applications to Collective Robotics. In: Proceedings of 2005 IEEE Swarm Intelligence Symposium, pp 84–91, IEEE Press
352. Krishnanand K, Ghose D (2006) Glowworm swarm based optimization algorithm for multimodal functions with collective robotics applications. Multiagent and Grid Systems 2(3):209–222
353. Krishnanand K, Ghose D (2006) Theoretical Foundations for Multiple Rendezvous of Glowworm-inspired Mobile Agents with Variable Local-decision domains. In: Proceedings of 2006 American Control Conference, pp 3588–3593, IEEE Press
354. Langdon W B, Gustafson S (2005) Genetic programming and evolvable machines: five years of reviews. Genetic Programming and Evolvable Machines 6(2):221–228
355. Langdon W B, Gustafson S, Koza J R (2004) The Genetic Programming Bibliography. http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html
356. Lam A Y S, Li V O K (2010) Chemical-reaction Inspired Metaheuristic for Optimization. IEEE Transactions on Evolutionary Computation 14(3):381–399
357. Lam A Y S, Li V O K, Wei Z (2012) Chemical Reaction Optimization for the Fuzzy Rule Learning Problem. In: Proceedings of the 2012 IEEE Congress on Evolutionary Computation (WCCI 2012), pp 1–8, IEEE Press
358. Lam A Y S, Xu J, Li V O K (2010) Chemical reaction optimization for population transition in peer-to-peer live streaming. In: Proceedings of 2010 IEEE Congress on Evolutionary Computation WCCI 2010), 1–8, IEEE Press
359. Langdon W B, Poli R (2002) Foundations of Genetic Programming, Springer-Verlag
360. Larrañaga P (2010) Probabilistic Graphical Models and Evolutionary Computation. In: World Congress on Computational Intelligence 2010, Plenary and Invited Lectures (WCCI 2010), pp 23–54, IEEE Press
361. Larrañaga P, Lozano J (eds) (2001) Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Kluwer
362. Lay N, Bate I (2007) Applying artificial immune systems to real-time embedded systems. In: Proceedings of the Congress on Evolutionary Computation 2007 (CEC 2007), pp 3743–3750, New Jersey: IEEE Press
363. Lee C-D, Chen Y-J, Huang H-C, Hwang R-C, Yu G-R (2004) The nonstationary signal prediction by using quantum NN. In: Proceedings of 2004 IEEE International Conference on Systems, Man and Cybernetics, pp 3291–3295, IEEE Press
364. Leier A, Kuo P D, Banzhaf W, Burrage K (2006) Evolving Noisy Oscillatory Dynamics in Genetic Regulatory Networks. In: Proceedings of the 9th European Conference on Genetic Programming (EuroGP 2006), Lecture Notes in Computer Science 3905, pp 290–299, Springer

365. Levenik J (1995) Metabits: Generic Endogenous Crossover Control. In: Proceedings of the Sixth International Conference on Genetic Algorithms 1995, Eshelman L J (ed), pp 88–95, Morgan Kaufmann
366. Lewin B (2000) Genes VII. Oxford University Press
367. Li X, Shao Z and Qian J (2002) An optimizing method based on autonomous animats: fish swarm algorithm. Systems Engineering Theory and Practice 22(11):32–38 (in Chinese)
368. Li Y, Zhang Y, Zhao R, Jiao L (2004) The immune quantum-inspired evolutionary algorithm. In: Proceedings of 2004 IEEE International Conference on Systems, Man and Cybernetics, pp 3301–3305, IEEE Press
369. Lin S (1965) Computer solution of the traveling salesman problem. Bell System Technical Journal 44:2245–2269
370. Lindenmayer A (1968) Mathematical Models for Cellular Interaction in Development, Parts I and II. Journal of Theoretical Biology 18:280–315
371. Lindenmayer A, Rozenberg G (1979) Parallel Generation of Maps: Developmental Systems for Cell Layers. In: Graph Grammars and their Application to Computer Science: First International Workshop. Lecture Notes in Computer Science 73, pp 301–316, Springer-Verlag
372. Lindsay W (1876) Mind in Plants. The British Journal of Psychiatry 21:513–532
373. Lins A, Bastos-Filho C, Nascimento D, Oliveira Junior M and Lima-Neto F (2012) Analysis of the Performance of the Fish School Search Algorithm Running in Graphic Processing Units. In Theory and New Applications of Swarm Intelligence, R. Parpinelli (ed) pp 17–32, Intech
374. Liu B, Abbass H, McKay B (2002) Density-based heuristic for rule discovery with ant-miner. In: Proc. 6th Australisia-Japan Joint Workshop in Intell. Evol. Syst., pp 180–184
375. Liu B, Abbass H, McKay B (2003) Classification rule discovery with ant colony optimization. In: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology (IAT 2003), pp 83–88, IEEE Press
376. Liu R, Wang X, Li Y (2012) Multi-objective Invasive Weed Optimization Algorithm for Clustering. In: Proceedings of 2012 IEEE World Congress on Computational Intelligence (WCCI 2012), pp 1556–1563, IEEE Press
377. Liu Y, Passino K (2002) Biomimicry of Social Foraging Bacteria for Distributed Optimization: Models, Principles, and Emergent Behaviors. Journal of Optimization Theory and Applications 115(3):603–628
378. Liu Y, Qin Z, He X (2004) Supervisor-Student Model in Particle Swarm Optimization. In: Proceedings of 2004 IEEE Congress on Evolutionary Computation (CEC 2004), pp 542–547, IEEE Press
379. Lones M A, Tyrrell A M (2001) Enzyme Genetic Programming. In: Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001), pp 1183–1190, IEEE Press
380. Lones M A, Tyrrell A M (2002) Biomimetic Representation with Genetic Programming Enzyme. Genetic Programming and Evolvable Machines 3(2):193–217
381. Lones M A, Tyrrell A M (2004) Modelling biological evolvability: Implicit context and variation filtering in enzyme genetic programming. BioSystems 76(1–3):229–238

382. Lumer E, Faieta B (1994) Diversity and adaptation in populations of clustering ants. In: Proceedings of Third International Conference on Simulation of Adaptive Behavior, pp 501–508, Cambridge, MA, MIT Press
383. MacNab R, Koshland D (1972) The Gradient-Sensing Mechanism in Bacterial Chemotaxis. Proc. Nat. Acad. Sci. (USA) 69(9):2509–2512
384. MacQueen J (1967) Some Methods for classification and Analysis of Multivariate Observations. In: Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, 1:281–297, Berkeley, University of California Press
385. Mahfoud S (1995) Niching Methods for Genetic Algorithms. PhD Thesis. Department of General Engineering, University of Illinois at Urbana-Champaign
386. Mahfoud S, Goldberg D (1995) Parallel recombinative simulated annealing: A genetic algorithm. Parallel Computing 21(1):1–28
387. Manevitz L, Yousef M (2007) One class document classification via neural networks. Neurocomputing 70(7–9):1466–1481
388. Marijuán P C (1995) Enzymes, artificial cells and the nature of biological information. BioSystems 35(2–3):167–170
389. Marinakis Y, Marinaki M (2011) A Honey Bee Mating Optimization Algorithm for the Open Vehicle Routing Problem. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2011), pp 101–108, ACM Press
390. Martens D, De Backer M, Haesen R, Vanthienen J, Snoeck M, Baesens B (2007) Classification with Ant Colony Optimization. IEEE Transactions on Evolutionary Computation 11(5):651–665
391. Martin M, Chopard B, Albuquerque, P (2002) Formation of an ant cemetery: swarm intelligence or statistical accident? Future Generation Computer Systems 18(7):951–959
392. Martonak R, Santoro G, Tosatti E (2004) Quantum annealing of the traveling salesman problem. Physical Review E 70(057701):1–4
393. Matzinger P (1994) Tolerance, danger and the extended family. Annual Review of Immunology 12:991–1045
394. Matzinger P (2002) The Danger Model: a renewed sense of self. Science 296(5566):301–305
395. Mavrovouniotis M, Yang S (2010) Ant Colony Optimization with Immigrant Schemes in Dynamic Environments. In: Proceedings of the 11th Parallel Problem Solving from Nature (PPSN XI), Lecture Notes in Computer Science 6239, pp 371–380, Springer
396. McDermott J (2012) Graph Grammars as a Representation for Interactive Evolutionary 3D Design. In: Proceedings of EvoMUSART 2012 European Conference on Evolutionary and Biologically Inspired Music, Sound, Art and Design, Lecture Notes in Computer Science 7247, pp 199–210, Springer
397. McDermott J, Byrne J, Swafford J M, O'Neill M, Brabazon A (2010) Higher-order functions in aesthetic EC encodings. In: Proceedings of the 12th Annual Congress on Evolutionary Computation (CEC 2010), pp 3018–3025, IEEE Press
398. McDermott J, O'Reilly U-M (2011) An executable graph representation for evolutionary generative music. In: Proceedings of the 13th annual conference on Genetic and Evolutionary Computation (GECCO 2011), pp 403–410, ACM

399. McDermott J, Swafford J M, Hemberg M, Byrne J, Hemberg E, Fenton M, McNally C, Shotton E, O'Neill M (2012) An Assessment of String-Rewriting Grammars for Evolutionary Architectural Design. Environment and Planning B: Planning and Design, 39(4):713–731
400. McDermott J, White D R, Luke S, Manzoni L, Castelli M, Vanneschi L, Jaskowski W, Krawiec K, Harper R, De Jong K, O'Reilly U-M (2012). Genetic Programming Needs Better Benchmarks. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2012), pp 791–798, ACM
401. McGarraghy S, Phelan M (2011). Generating Supply Chain ordering policies using Quantum Inspired Genetic Algorithms and Grammatical Evolution. In: Minis I, Zeimpekis V, Dounias G, Ampazis N (eds) Supply Chain Optimization, Design & Management: Advances & Intelligent methods. pp 125–154, Hershey, PA: IGI Global
402. McGarraghy S, Williams P (2010). Extension of Self Organising Swarm to include Acceleration, with applications to an Insurance industry problem. Technical Report, University College Dublin: Centre for Business Analytics.
403. McKay R I, Nguyen X H, Whigham P A, Shan Y, O'Neill M (2010) Grammar-based Genetic Programming: a survey. Genetic Programming and Evolvable Machines 11(3–4):365–396
404. McKey D (1974) Adaptive patterns in alkaloid physiology. American Naturalist 108:305–320
405. Medzhitov R, Janeway C (1997) Innate immunity. New England Journal of Medicine 343(5):338–344
406. Mehrabian A, Lucas C (2006) A novel numerical optimization algorithm inspired from weed colonization. Ecological Informatics 1:355–366
407. Mehrabian A R, Yousefi-Koma A (2007) Optimal positioning of piezoelectric actuators on a smart fin using bio-inspired algorithms. Aerospace Science and Technology 11:174–182
408. Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E (1953) Equations of state calculations by fast computing machines. Journal of Chemical Physics 21(6):1087–1092
409. Michalewicz Z, Schmidt M, Michalewicz M, Chiriac C (2007) Adaptive Business Intelligence. Berlin: Springer
410. Middendorf M, Reischle F, Schmeck H (2002) Multi colony ant algorithms. Journal of Heuristics 8:305–320
411. Middleton A A (2004) Improved extremal optimization for the Ising spin glass. Phys Rev E 69(5):055701
412. Miller J F (ed) (2011) Cartesian Genetic Programming. Springer
413. Miller J F, Thomson P (2000) Cartesian genetic programming. In: Proceedings of the European Conference on Genetic Programming (EuroGP 2000), Lecture Notes in Computer Science 1802, pp 121–132, Springer
414. Minsky M L, Papert S A (1969) Perceptrons. Cambridge, MA: MIT Press
415. Mishra S (2004) A Hybrid Adaptive Bacterial Foraging and Feedback Linearization Scheme based on D-STATCOM. In: Proceedings of the 2004 International Conference on Power System Technology (POWERCON 2004), pp 275–280, IEEE Press
416. Mishra S (2005) A Hybrid Least Square-Fuzzy Bacterial Foraging Strategy for Harmonic Estimation. IEEE Transactions on Evolutionary Computation 9(1):61–73

417. Montana D (1994) Strongly Typed Genetic Programming. Bolt, Beranek and Newman Inc., Technical Report No. 7866
418. Montana D, Davis L (1989) Training feedforward neural networks using genetic algorithms. In: Proceedings of the 11th International Joint Conference on Artificial Intelligence, pp 762–767, Morgan Kaufmann
419. Moraglio A, Krawiec K, Johnson C G. (2012) Geometric Semantic Genetic Programming. In: Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN XII), Lecture Notes in Computer Science 7491, pp 21-31, Springer.
420. Morrison R (2004) Designing Evolutionary Algorithms for Dynamic Environments. Berlin: Springer
421. Moscato P (1989) On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Caltech concurrent computation program report 826, California Institute of Technology
422. Moscato P (1999) Memetic Algorithms: A short introduction. In: Corne D, Dorigo M, Glover F (eds) New Ideas in Optimization, pp 219–234, Maidenhead, UK: McGraw-Hill
423. Mühlenbein H, Paaß G (1996) From recombination of genes to the estimation of distributions (i). Binary parameters. In: Proceedings of Parallel Problem Solving from Nature IV (PPSN IV), Lecture Notes in Computer Science 1411, pp 178–187, Berlin: Springer
424. Murphy E (2014) An Exploration of Tree-Adjoining Grammars for Grammatical Evolution. PhD Thesis, University College Dublin.
425. Murphy E, Hemberg E, Nicolau M, O'Neill M, Brabazon A (2012) Grammar Bias and Initialisation in Grammar Based Genetic Programming. In: Proceedings of the 15th European Conference on Genetic Programming (EuroGP 2012), Lecture Notes in Computer Science 7244, pp 85-96, Springer.
426. Murphy E, Nicolau M, Hemberg E, O'Neill M, Brabazon A (2012) Differential Gene Expression with Tree-Adjunct Grammars. In: Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN XII), Lecture Notes in Computer Science 7491, pp 377-386, Springer.
427. Murphy E, O'Neill M, Galvan-Lopez E, Brabazon A (2010) Tree-adjunct Grammatical Evolution. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010), pp 4449–4456, IEEE Press
428. Murphy J E (2011) Applications of Evolutionary Computation to Quadrupedal Animal Animation. PhD Thesis. University College Dublin
429. Murphy J E, O'Neill M, Carr H (2009) Exploring Grammatical Evolution for Horse Gait Optimisation. In: Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009), Lecture Notes in Computer Science 5481, pp 183–194, Springer
430. Musilek P, Lau A, Reformat M, Wyard-Scott L (2006) Immune Programming. Information Sciences 176(8):972–1002
431. Nakamura R, Pereira L, Costa K, Rodrigues D, Papa J, Yang X-S (2012) BBA: A Binary Bat Algorithm for Feature Selection. In: Proceedings of XXV SIBGRAPI Conference on Graphics, Patterns and Images, pp 291–297, IEEE Press
432. Nakrani S, Tovey C (2004) On Honey Bees and Dynamic Server Allocation in Internet Hosting Centres. Adaptive Behavior 12(3–4):223–240

433. Narayanan A, Moore M (1996) Quantum-inspired genetic algorithms. In: Proceedings of IEEE International Conference on Evolutionary Computation (CEC 1996), pp 61–66, IEEE Press
434. Negrevergne C., Mahesh T. S., Ryan C. A., Ditty M., Cyr-Racine F., Power W., Boulant N., Havel T., Cory D. G. and Laflamme R. (2006). Benchmarking Quantum Control Methods on a 12-Qubit System. Physical Review Letters 96(17): 170501.
435. Nelles O (2001) Nonlinear System Identification. Berlin: Springer
436. Neshat M, Sepidnam G, Sargolzaei M and Najaran Toosi A (2012) Artificial fish swarm algorithm: a survey of the state-of-the-art, hybridization, combinatorial and indicative applications. Artificial Intelligence Review, available online 6 May 2012 (DOI 10.1007/s10462-012-9342-2)
437. Nguyen Q U (2011) Examining Semantic Diversity and Semantic Locality of Operators in Genetic Programming. PhD Thesis. University College Dublin
438. Nguyen Q U, Nguyen T H, Nguyen X H, O'Neill M (2010) Improving the Generalisation Ability of Genetic Programming with Semantic Similarity based Crossover. In: Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010), Lecture Notes in Computer Science 6021, pp 184–195, Springer
439. Nguyen Q U, Nguyen X H, O'Neill M, McKay B (2010) The Role of Syntactic and Semantic Locality of Crossover in Genetic Programming. In: Proceedings of the 11th International Conference on Parallel Problem Solving From Nature (PPSN 2010), Lecture Notes in Computer Science 6239, pp 533–542, Springer
440. Nguyen Q U, Nguyen X H, O'Neill M, McKay R I, Galvan-Lopez E (2011) Semantically-based Crossover in Genetic Programming: Application to Real-valued Symbolic Regression. Genetic Programming and Evolvable Machines 12(2):91–119
441. Nguyen Q U, Nguyen X H, O'Neill M, McKay R I, Dao N.P. (2013) On the Roles of Semantic Locality of Crossover in Genetic Programming. Information Sciences 235:195-213.
442. Nicolau M (2004) Automatic Grammar Complexity Reduction in Grammatical Evolution. In: Proceedings of GECCO 2004 Workshop. Also available from http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/WGEW002.pdf
443. Nicolau M (2006) Genetic Algorithms using Grammatical Evolution. PhD Thesis. University of Limerick
444. Nicolau M, Costelloe D (2011) Using Grammatical Evolution to Parameterise Interactive 3D Image Generation. In: Proceedings of 9th European Event on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART 2011), Lecture Notes in Computer Science 6625, pp 366–375, Springer
445. Nicolau M, Dempsey I (2006) Introducing Grammar Based Extensions for Grammatical Evolution. In: Proceedings of IEEE International Conference on Evolutionary Computation (CEC 2006), pp 648–655, IEEE Press
446. Nicolau M, O'Neill M, Brabazon A (2012) Termination in Grammatical Evolution: Grammar Design, Wrapping, and Tails. In: Proceedings of the 2012 IEEE Congress on Evolutionary Computation (WCCI 2012), pp 1–8, IEEE Press
447. Nicolau M, O'Neill M, Brabazon A (2012) Applying Genetic Regulatory Networks to Index Trading. In: Proceedings of 12th International Conference on

Parallel Problem Solving from Nature (PPSN XII), Lecture Notes in Computer Science 7492, pp 428–437, Springer

448. Nicolau M, Saunders M, O'Neill M, Brabazon A, Osborne B (2012) Evolving Interpolating Models of Net Ecosystem CO2 Exchange Using Grammatical Evolution. In: Proceedings of the 15th European Conference on Genetic Programming (EuroGP 2012), Lecture Notes in Computer Science 7244, pp 134–145, Springer
449. Nicolau M, Schoenauer M, Banzhaf W (2010) Evolving Genes to Balance a Pole. In: Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010), Lecture Notes in Computer Science 6021, pp 196–207, Springer
450. Nikolaev N, Iba H, Slavov V (1999) Inductive genetic programming with immune network dynamics. In: Spector L, Langdon W B, O'Reilly U-M, Angeline, P (eds) Advances in Genetic Programming 3, pp 355–376, MIT Press
451. Niknam T, Azizipanah-Abarghooee R, Zare M, Bahmani-Firouzi B (2013) Reserve Constrained Dynamic Environmental/Economic Dispatch: A New Multiobjective Self-Adaptive Learning Bat Algorithm. IEEE Systems Journal 7(4):763–776. Digital Object Identifier: 10.1109/JSYST.2012.2225732
452. Ning Z, Ong Y S, Wong K, Lim M (2003) Choice of Memes in Memetic Algorithm. In: Proceedings of 2nd International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS 2003), Special Session on Optimization using Genetic, Evolutionary, Social and Behavioral Algorithms, December 15–18, 2003, Singapore
453. Noman N, Iba H (2005) Enhancing Differential Evolution Performance with Local Search for High Dimensional Function Optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005), pp 967–974, ACM Press
454. Nordin P (1997) Evolutionary Program Induction of Binary Machine Code and its Applications. PhD Thesis. Technische Universität Dortmund
455. Novoplansky A (2002) Developmental plasticity in plants: implications of noncognitive behavior. Evolutionary Ecology 16:177–188
456. Olague G, Puente C (2006a) The Honeybee Search Algorithm for Three-Dimensional Reconstruction. In: Proceedings of EvoIASP 2006, Lecture Notes in Computer Science 3907, pp 427–437, Springer
457. Olague G, Puente C (2006b) Parisian Evolution with Honeybees for Three-dimensional Reconstruction. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO 2006), pp 191–198, New York: ACM Press
458. O'Neill L (2004) TLRs: Professor Mechnikov, Sit on your hat. Trends in Immunology 25(12):687–693
459. O'Neill L (2005) Immunity's Early warning system. Scientific American, Jan 2005, pp 38–45
460. O'Neill M (2001) Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. PhD thesis. University of Limerick, Ireland
461. O'Neill M, Brabazon A (2004) Grammatical swarm. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science 3120, pp 163–174, Springer-Verlag

462. O'Neill M, Brabazon A (2005) mGGA: The meta-Grammar genetic algorithm. In: Proceedings of the European Conference on Genetic Programming (EuroGP 2005), Lecture Notes in Computer Science 3447, pp 311–320, Springer
463. O'Neill M, Brabazon A (2006) Grammatical Swarm: The Generation of Programs by Social Programming. Natural Computing 5(4):443–462
464. O'Neill M, Brabazon A (2006) Self-Organizing Swarm (SOSwarm): A Particle Swarm Algorithm for Unsupervised Learning. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2006), pp 2649–2654, IEEE Press
465. O'Neill M, Brabazon A (2006) Grammatical Differential Evolution. In: Proceedings of the 2006 International Conference on Artificial Intelligence (ICAI'06), Vol 1, pp 408–413, CSEA Press
466. O'Neill M, Brabazon A (2008) Self-organising Swarm (SOSwarm). Soft Computing 12(11):1073–1080
467. O'Neill M, Brabazon A (2008) Evolving a Logo Design Using Lindenmayer Systems, Postscript and Grammatical Evolution. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008), pp 3788–3794, IEEE Press
468. O'Neill M, Ryan C (1999) Automatic Generation of Caching Algorithms. In: Proceedings of EUROGEN 1999, Short Course on Evolutionary Algorithms in Engineering and Computer Science, pp 127–134, Wiley
469. O'Neill M, Ryan C (2000) Grammar based function definition in Grammatical Evolution. In: Proceedings of the 2nd Genetic and Evolutionary Computation Conference (GECCO 2000), pp 485–490, Morgan Kaufmann
470. O'Neill M, Ryan C (2001) Grammatical evolution. IEEE Trans. Evolutionary Computation 5(4):349–358
471. O'Neill M, Ryan C (eds) (2002). Grammatical Evolution Workshop 2002. In: Barry A (ed) Proceedings of the Workshops of the Genetic and Evolutionary Computation Conference GECCO 2002, New York, NY, USA, July 2002
472. O'Neill M, Ryan C (2003) Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Boston: Kluwer Academic Publishers
473. O'Neill M, Ryan C (eds) (2003) Grammatical Evolution Workshop 2003. In: Barry A (ed) Proceedings of the Workshops of the Genetic and Evolutionary Computation Conference GECCO 2003, Chicago, IL, USA, July 2003
474. O'Neill M, Ryan C (2004) Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code. In: Proceedings of the European Conference on Genetic Programming (EuroGP 2004), Lecture Notes in Computer Science 3003, pp 138–149, Springer
475. O'Neill M, Ryan C (eds) (2004b) Grammatical Evolution Workshop 2004. In: Poli R et al. (eds) Proceedings of the Workshops of the Genetic and Evolutionary Computation Conference GECCO 2004, Seattle, WA, USA, June 2004
476. O'Neill M, Brabazon A, Adley C (2004) The automatic generation of programs for classification problems with grammatical swarm. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2004), Vol 1 pp 104–110, IEEE Press
477. O'Neill M, Brabazon A, Nicolau M, McGarraghy S, Keenan P (2004) πGrammatical Evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science 3103, pp 617–629, Springer-Verlag

478. O'Neill M, Brabazon A, Hemberg E (2008) Subtree Deactivation Control with Grammatical Genetic Programming. In: Proceedings of the IEEE World Congress on Computational Intelligence (WCCI 2008), pp 3768–3744, IEEE Press
479. O'Neill M, Cleary R, Nikolov N (2004) Solving Knapsack problems with attribute grammars. In: Poli R et al. (eds) Proceedings of the Grammatical Evolution Workshop, Genetic and Evolutionary Computation Conference GECCO 2004, Seattle, WA, USA, June 2004
480. O'Neill M, Hemberg E, Gilligan C, Bartley E, McDermott J, Brabazon A (2008) GEVA: Grammatical Evolution in Java. SIGEVOlution 3(2):17–23
481. O'Neill M, McDermott J, Swafford J M, Byrne J, Hemberg E, Shotton E, McNally C, Brabazon A, Hemberg M (2010) Evolutionary Design using Grammatical Evolution and Shape Grammars: Designing a Shelter. International Journal of Design Engineering 3(1):4–24
482. O'Neill M, Ryan C, Keijzer M, Cattolico M (2003) Crossover in Grammatical Evolution. Genetic Programming and Evolvable Machines 4(1):67–93
483. O'Neill M, Ryan C, Nicolau M (2001) Grammar Defined Introns: An Investigation into Grammars, Introns, and Bias in Grammatical Evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp 97–103, Morgan Kaufmann
484. O'Neill M, Vanneschi L, Gustafson S, Banzhaf W (2010) Open Issues in Genetic Programming. Genetic Programming and Evolvable Machines 11(3–4):339–363
485. O'Reilly U-M, Hemberg M (2007) Integrating generative growth and evolutionary computation for form exploration. Genetic Programming and Evolvable Machines 8(2):163–186
486. Ong Y-S, Keane A (2004) Meta-Lamarckian Learning in Memetic Algorithms. IEEE Transactions on Evolutionary Computation 8(2):99–110
487. Ong Y-S, Lim M-H, Chen X (2010) Memetic Computation - Past, Present and Future. IEEE Computational Intelligence Magazine 5(2):24–31
488. Ong Y-S, Lim M-H, Zhu N, Wong K-W (2006) Classification of adaptive memetic algorithms: a comparative study. IEEE Transactions on Systems, Man, and Cybernetics Part B 36(1):141–152
489. Ortega A, de la Cruz M, Alfonseca M (2007) Christiansen Grammar Evolution: Grammatical Evolution With Semantics. IEEE Transactions on Evolutionary Computation 11(1):77–90
490. Ostermeier A, Gawelczyk A, Hansen N (1994) A Derandomized Approach to Self Adaptation of Evolution Strategies. Evolutionary Computation 2(4):369–380
491. Pampara G, Engelbrecht A (2006) Binary Differential Evolution. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (WCCI 2006), pp 6764–6770, IEEE Press
492. Pampara G, Engelbrecht A (2011) Binary Artificial Bee Colony Optimization. In: Proceedings of 2011 IEEE Swarm Intelligence Symposium (SiS 2011), pp 170–177, IEEE Press
493. Pampara G, Franken N, Engelbrecht A (2005) Combining Particle Swarm Optimisation with Angle Modulation to Solve Binary Problems. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), pp 89–96, IEEE Press

494. Parpinelli R, Lopes H, Freitas A (2002) Data mining with an ant colony optimization algorithm. IEEE Transactions on Evolutionary Computing 6(4):321–332
495. Parrish J, Viscido S and Grunbaum D (2002) Self-organized Fish Schools: An Examination of Emergent Properties. Biol. Bull. 202:296–305
496. Passino K (2000) Distributed Optimization and Control Using Only a Germ of Intelligence. In: Proceedings of the 2000 IEEE International Symposium on Intelligent Control, pp 5–13, IEEE Press
497. Passino K (2002) Biomimicry of Bacterial Foraging for Distributed Optimization and Control. IEEE Control Systems Magazine 22(3):52–67
498. Passino K, Seeley T (2006) Modeling and analysis of nest-site selection by honeybee swarms: the speed and accuracy trade-off. Behavioral Ecology and Sociobiology 59:427–442
499. Paton R, Vlachos C, Wu Q, Saunders J (2006) Simulated bacterially-inspired problem solving — the bacterial domain. Natural Computing, 5:43–65
500. Pelikan M (2005) Hierarchical Bayesian Optimization Algorithm: Toward a New Generation of Evolutionary Algorithms. Springer: Berlin
501. Pelikan M, Goldberg D, Cantú-Paz E (1998a) Linkage Problem, Distribution Estimation, and Bayesian Networks. IlliGAL Report No.98013, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL
502. Pelikan M, Goldberg D, Cantú-Paz E (1998b) BOA: The Bayesian Optimization Algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), Vol 1 pp 13–17, Morgan Kaufmann
503. Pelikan M, Goldberg D, Cantú-Paz E (2000) Linkage Problem, Distribution Estimation and Bayesian Networks. Evolutionary Computation 8(3):311–340
504. Pelikan M, Sastry K, Cantú-Paz E (eds) (2006) Scalable Optimization via Probabilistic Modeling. Berlin: Springer
505. Pennisi E (2012) Genomics. ENCODE project writes eulogy for junk DNA. Science 337(6099):1159–1161
506. Perez D, Nicolau M, O'Neill M, Brabazon A (2011) Evolving Behaviour Trees for the Mario Bros Game Using Grammatical Evolution. In: Proceedings of 3rd European Event on Bio-inspired Algorithms in Games (EvoGAMES 2011), Lecture Notes in Computer Science 6624, pp 121–130, Springer
507. Perez D, Nicolau M, O'Neill M, Brabazon A (2011) Reactiveness and Navigation in Computer Games: Different Needs, Different Approaches. In: Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games (CIG 2011), pp 273–280, IEEE Press
508. Pham D, Ghanbarzadeh A, Koc E, Otri S, Rahim S, Zaidi M (2006) The Bees Algorithm – A novel tool for complex optimisation problems. In: Proceedings of International Production Machines and Systems (IPROMS 2006), pp 454–459, Elsevier
509. Platel M, Schliebs S, Kasabov N (2009) Quantum-inspired Evolutionary Algorithm: A multimodal EDA. IEEE Transactions on Evolutionary Computation 13(6):1218–1232
510. Pincus M (1970) A Monte Carlo method for the approximate solution of certain types of constrained optimization problems. Operations Research 18:1225–1228
511. Pinker S (1995) The language instinct: the new science of language and the mind. London: Penguin Books

512. Poli R, Di Chio C, Langdon W B (2005) Exploring Extended Particle Swarms: A Genetic Programming Approach. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005), Beyer H-G et al. (eds), pp 169–176, ACM
513. Poli R, Langdon W B, Holland O (2005) Extending Particle Swarm Optimisation via Genetic Progamming. In: Proceedings of the European Genetic Programming Conference (EuroGP 2005), Keijzer M et al. (eds), pp 291–300, Springer
514. Poli R, Langdon W B, McPhee N F (2008) A Field Guide to Genetic Programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk
515. Poli R, Stephens C (2004) Constrained Molecular Dynamics as a Search and Optimization Tool. In: Proceedings of the 7th European Conference (EuroGP 2004), pp 150–161, Springer-Verlag
516. Porowski J, Da M, McGarraghy S (2009) Quantum annealing in traveling salesman related problems. Technical Report, University College Dublin
517. Pourtakdoust S, Nobahari H (2004) An Extension of Ant Colony System to Continuous Optimization Problems. In: Proceedings of the 4th International Workshop on Ant Colony Optimization and Swarm Intelligence, pp 294–301, Springer
518. Powell M (1987) Radial basis functions for multivariable interpolation: a review. In: Mason J, Cox M (eds) Algorithms for Approximation. Oxford: Clarendon Press
519. Premaratne U, Samarabandu J, Sidhu, T (2009) A New Biologically Inspired Optimization Algorithm. In: Proceedings of the Fourth International Conference on Industrial and Information Systems (ICIIS 2009), pp 279–284, IEEE Press
520. Price K (1999) An introduction to differential evolution. In: Corne D, Dorigo M, Glover F (eds) New Ideas in Optimization, pp 79–108, London: McGraw-Hill
521. Price K, Storn R, Lampinen J (2005) Differential Evolution: A Practical Approach to Global Optimization. Berlin: Springer
522. Principe J, Fisher I, Xu D (2000) Information theoretic learning. In: Haykin S (ed) Unsupervised adaptive filtering. New York: Wiley
523. Prusinkiewicz P, Lindenmayer A (1990) The Algorithmic Beauty of Plants. Springer-Verlag
524. Quijano N, Passino K, Andrews B (2006) Foraging Theory for Multizone Temperature Control. IEEE Computational Intelligence Magazine, November, 2006, pp 18–27, IEEE Press
525. Raidl G, Gottlieb J (2005) Empirical Analysis of Locality, Heritability and Heuristic Bias. Evolutionary Computation 13(4):441–476
526. Raidl G et al. (eds) (2009) Proceedings of the 11th annual conference on Genetic and evolutionary computation (GECCO 2009). 8–12 July, Montreal, Canada, ACM Press
527. Rashedi E, Nezamabadi-pour H, Saryazdi S (2009) GSA: A Gravitational Search Algorithm. Information Sciences 179:2232–2248
528. Rashedi E, Nezamabadi-pour H, Saryazdi S (2010) BGSA: binary gravitational search algorithm. Natural Computing 9(3):727–745
529. Ray T (1995) An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life. In: Artificial Life: An Overview, pp 179–209, MIT Press

530. Rechenberg I (1965) Cybernetic Solution Path of an Experimental Problem. Royal Aircraft Establishment, Farnborough, Library Translation No. 1122, August
531. Rechenberg I (1973) Evolutionsstrategie: Optimierung technisher Systeme nach Prinzipien der biologischen Evolution, Stuttgart: Frommann-Holzboog Verlag
532. Reynolds C (1987) Flocks, Herds and Schools: A distributed behavioral model. In Proceedings of the 14th annual conference on computer graphics and interactive techniques (SIGGRAPH 1987), pp 25–34
533. Robinson E, Jackson D, Holcombe M, Ratnieks F (2005) Insect communication: 'No entry' signal in ant foraging. Nature 438:442
534. Robinson R (1973) Counting labeled acyclic digraphs. In: Harary F (ed) New Directions in the Theory of Graphs, pp 239–273, New York: Academic Press
535. Robson S, Traniello J (1998) Resource Assessment, Recruitment Behavior, and Organization of Cooperative Prey Retrieval in the Ant *Formica schaufussi* (Hymenoptera: Formicidae). Journal of Insect Behavior 11(1):1–22
536. Rodriguez F, Harkins S, Slifka M, Whitton L (2002) Immunodominance in Virus-Induced CD8+ T-Cell Responses Is Dramatically Modified by DNA Immunization and Is Regulated by Gamma Interferon. Journal of Virology 76(9):4251–4259
537. Rodriguez J, Garcia-Tuñón I, Taboada J, Basteiro F (2007) Broadband HF antenna matching network design using a real coded genetic algorithm. IEEE Trans. Ant. Propag. 55(3):611–618
538. Rosenberg R (1967) Simulation of Genetic Populations With Biochemical Properties. PhD Thesis. University of Michigan
539. Rosenblatt F (1962) Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanism. Spartan
540. Rothlauf F (2002) Representations for Genetic and Evolutionary Algorithms. Physica-Verlag
541. Rothlauf F, Goldberg D (2003) Redundant Representations in Evolutionary Computation. Evolutionary Computation 11(4):381–415
542. Rothlauf F, Oetzel M (2006). On the Locality of Grammatical Evolution. In: Proceedings of the European Genetic Programming Conference (EuroGP 2006), Collet P et al. (eds), pp 320–330, Springer
543. Rudolph S, Alber R (2002) An Evolutionary Approach to the Inverse Problem in Rule-based Design Representations. In: Proceedings of the 7th International Conference on Artificial Intelligence in Design (AID'02), pp 329–350, Kluwer Academic Publishers
544. Rui T, Fong S, Yang X-S, Deb S (2012) Nature-inspired Clustering Algorithms for Web Intelligence Data. In: Proceedings of the IEEE, WIC, ACM International Conference on Web Intelligence and Intelligent Agent Technology, pp 147–153, IEEE Press
545. Rumelhart D E, Hinton G E, Williams, R J (1986) Learning representations by back-propagating errors. Nature 323(6088):533–536
546. Ruta D, Gabrys B (2009) A framework for machine learning based on dynamic physical fields. Natural Computing 8(2):219–237
547. Ryan C, Collins J J, O'Neill M (1998) Grammatical evolution: evolving programs for an arbitrary language. In: Proceedings of the First European Workshop on Genetic Programming, Lecture Notes in Computer Science 1391, pp 83–95, Springer

548. Salhi A, Fraga, E (2011) Nature-inspired Optimisation Approaches and the New Plant Propagation Algorithm. In: Proceedings of 2011 International Conference on Numerical Analysis and Optimization (ICeMATH 2011), pp K2–1:K2–8
549. Santoro G E, Tosatti E (2006). Optimization using quantum mechanics: quantum annealing through adiabatic evolution. J Phys A: Math Gen 39:R393-R431
550. Santoro G E, Tosatti E (2007) Computing: Quantum to classical and back. Nature Physics 3:593–594
551. Sarle W S (1994) Neural Networks and Statistical Models. In: Proceedings of the Nineteenth Annual SAS Users Group International Conference, pp 1538–1550, Cary, NC, SAS Institute Inc. (online at ftp://ftp.sas.com/pub/neural/neural1.ps)
552. Sastry K, Goldberg D E (2003) Probabilistic model building and competent genetic programming. In: Riolo R L, Worzel B (eds) Genetic Programming Theory and Practice, pp 205–220, Kluwer
553. Sata T, Hagiwara M (1997) Bee System: Finding Solutions by a Concentrated Search. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, pp 3954–3959, IEEE Press
554. Schaffer J (1985) Multiple objective optimization with vector evaluated genetic algorithms. In: Proceedings of the 1st International Conference on Genetic Algorithms (ICGA 1985), pp 93–100, Lawrence Erlbaum Associates
555. Schaffer J, Morishima A (1987) An Adaptive Crossover Distribution Mechanism for Genetic Algorithms. In: Grefenstette J J (ed) Proceedings of the Second International Conference on Genetic Algorithms 1987 (ICGA 1987), pp 36–40, Lawrence Erlbaum Associates
556. Schalkoff R (1992) Pattern Recognition - Statistical, Structural and Neural Approaches, New York: Wiley
557. Schechter M and Laflorencie N (2006). Quantum spin glass and the dipolar interaction. *Physical Review Letters*, **97**(13):137204.
558. Schölkopf B, Platt J C, Shawe-Taylor J, Smola A J, Williamson R C (2001) Estimating the Support of a High-Dimensional Distribution. Neural Computation 13(7):1443–1471
559. Schölkopf B, Smola A J (2002) Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond. Cambridge, MA: MIT Press
560. Schwefel H-P (1965) Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik. Diploma Thesis. Technical University of Berlin
561. Schwefel H-P (1981) Numerical Optimization of Computer Models. Chichester, UK: Wiley
562. Schwefel H-P (1995) Evolution and Optimum Seeking. New York: Wiley
563. Seeley T (1995) The Wisdom of the Hive. Cambridge, MA: Harvard University Press
564. Seeley T, Mikheyev A, Pagano G (2000) Dancing bees tune both duration and rate of waggle-run production in relation to nectar-source profitability. Journal of Comparative Physiology A 186:813–819
565. Seeley T, Morse R, Visscher P (1979) The natural history of the flight of honey bee swarms. PSYCHE 86(2–3):103–113
566. Seeley T, Visscher P, Passino K (2006) Group Decision Making in Honey Bee Swarms. American Scientist 94(3):220–229

567. Senthilnath J, Omkar S N, Mani V (2011) Clustering using firefly algorithm: performance study. Swarm and Evolutionary Computation 1(3):164–171
568. Sepehri Rad, H S, Lucas C (2007) A recommender system based on invasive weed optimization algorithm. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007), pp 4297–4303, IEEE Press
569. Settles M, Nathan P, Soule T (2005). Breeding swarms: A new approach to recurrent neural network training. In: Beyer et al. (eds) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005), 1:185–192, ACM Press
570. Shaker N, Nicolau M, Yannakakis G N, Togelius J, O'Neill M (2012) Evolving Levels for Super Mario Bros Using Grammatical Evolution. In Proceedings of IEEE Conference on Computational Intelligence and Games (CIG 2012), pp 304-311, IEEE Press
571. Shaker N, Yannakakis G N, Togelius J, Nicolau M, O'Neill M (2012). Evolving Personalized Content for Super Mario Bros Using Grammatical Evolution. In: Proceedings of Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-12), pp 75-80, AAAI
572. Shan Y, McKay R I, Baxter R, Abbass H, Essam D, Nguyen H. (2004). Grammar model-based program evolution. In: Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004), pp 478–485, IEEE Press
573. Shao J, McDermott J, O'Neill M, Brabazon A (2010) Jive: A Generative, Interactive, Virtual, Evolutionary Music System. In: Proceedings of the 8th European Event on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART 2010), Lecture Notes in Computer Science 6025, pp 341–350, Springer
574. Shor P W (1994)Algorithms for quantum computation: Discrete logarithms and factoring. In: Proceedings of the 28th ACM Symposium on the Theory of Computing, Piscataway, NJ, Nov 1994. IEEE Press. 124–134.
575. Shor P W (1997) Quantum computing. SIAM J. Comput. 26: 1484.
576. Shor P W (1998) Quantum computing. Documenta Mathematica (Extra Volume) ICM(I):467–486
577. Shudo E, Iwasa Y (2001) Inducible defense against pathogens and parasites: optimal choice among multiple options. J. Theoretical Biology 209:233–247
578. Silva A, Neves A, Costa E (2002) An empirical comparision of particle swarm and predator prey optimisation. In: Proceedings of AICS 2002, Lecture Notes in Artificial Intelligence (2464), pp 103–110, Springer
579. Simoes L, Cruz C, Ribeiro R, Correia L, Seidl T, Ampatzis C, Izzo D (2011) Path Planning Strategies Inspired by Swarm Behaviour of Plant Root Apexes. Technical Report 09/6401 of European Space Agency, Advanced Concepts Team. Ariadna Final Report. http://www.esa.int/gsp/ACT/doc/ARI/ARI\%20Study\%20Report/ACT-RPT-BIO-ARI-09--6401-PathPlanningInspiredByRoots.pdf
580. Simpson M, Sayler G, Fleming J, Sanseverino J, Cox, C (2004) The Device Science of Whole Cells as Components in Microscale and Nanoscale Systems In: Amos M (ed) Cellular Computing, Oxford University Press
581. Sipper M, Sanchez E, Mange D, Tomassini M, Perez-Uribe A, Stauffer A (1997) A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. IEEE Transactions on Evolutionary Computation 1(1):83–97

582. Smith J, Fogarty, T (1996) Recombination Strategy Adaptation via Evolution of Gene Linkage. In: Proceedings of 1996 IEEE International Conference on Evolutionary Computation, pp 826–831, IEEE Press
583. Socha K, Dorigo M (2008) Ant colony optimization for continuous domains. European Journal of Operational Research, 185(3):1155–1173
584. Somma R D, Batista C D, Ortiz G (2007a) Quantum approach to classical statistical mechanics. Physical Review Letters 99(3):030603
585. Somma R D, Boixo S, Barnum H (2007b) Quantum simulated annealing. http://arxiv.org/quant-ph/0712.1008v1
586. Soule T et al. (eds) (2012) Proceedings of the 14th annual conference on Genetic and evolutionary computation (GECCO 2012). 7–11 July, Philadelphia, USA, ACM Press
587. Sörensen K (2013) Metaheuristics — the metaphor exposed. International Transactions in Operational Research, published online 8 Feb 2013, doi:10.1111/itor.1200
588. Spector L, Stoffel K (1996) Ontogenetic Programming. In: Proceedings of the First Annual Conference Genetic Programming 1996, pp 394–399, MIT Press
589. Spencer H (1864) The Principles of Biology, Volume 1. London and Edinburgh: Williams and Norgate
590. Srinivas N, Deb K (1994) Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation 2(3)221–248
591. Stanley K, Miikkulainen R (2002a) Evolving neural networks through augmenting topologies. Evolutionary Computation 10(2):99–127
592. Stanley K, Miikkulainen R (2002b) Efficient evolution of neural network topologies. In: Proceedings of the 2002 IEEE Congress on Evolutionary Computation (CEC 2002), pp 1757–1762, IEEE Press
593. Stanley K, Miikkulainen R (2002c) Efficient Reinforcement Learning Through Evolving Neural Network Topologies. In: Proceedings of the 2002 Genetic and Evolutionary Computation Conference (GECCO 2002), pp 569–577, Morgan Kaufmann
594. Stephens D, Krebs J (1986) Foraging Theory. Princeton, New Jersey: Princeton University Press
595. Stibor T, Oates R, Kendall G, Garibaldi J (2009) Geometrical insights into the dendritic cell algorithm. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO '09), pp 1275–1282, ACM Press
596. Stibor T (2009) Foundations of r-contiguous matching in negative selection for anomaly detection. Natural Computing 8:613–641
597. Stibor T, Timmis J, Eckert C (2005) A comparative study of real-valued negative selection to statistical anomaly detection techniques. In: Proceedings of the 4th International Conference on Artificial Immune Systems, Lecture Notes in Computer Science 3627, pp 262–275, Berlin: Springer
598. Stöcker S (2009) Models for tuna formation. Mathematical Biosciences 156:167–190
599. Storn R (1999) System design by constraint adaptation and differential evolution. IEEE Transactions on Evolutionary Computation 3:22–34
600. Storn R, Price K (1995) Differential evolution: a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95–012: International Computer Science Institute, Berkeley

601. Storn R, Price K (1997) Differential evolution: a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization 11:341–359
602. Struik P, Yin X, Meinke H (2008) Perspective Plant neurobiology and green plant intelligence: science, metaphors and nonsense. Journal of the Science of Food and Agriculture 88:363–370
603. Stützle T (1998) Parallelization strategies for ant colony optimization. In: Proceedings of Parallel Problem Solving from Nature (PPSN 1998), Lecture Notes in Computer Science 1498, pp 722–741, Springer
604. Stützle T, Hoos H (2000) *MAX-MIN* Ant System. Future Generation Computer Systems 16(8):889–914
605. Sullivan K, Luke S (2007) Evolving Kernels for Support Vector Machine Classification. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007), pp 1702–1707, New York: ACM Press
606. Sumpter D, Krause J, James R, Couzin I and Ward A (2008) Consensus Decision Making by Fish. Current Biology 18:1773–1777
607. Swafford J M (2013) Analyzing the Discovery and Use of Modules in Grammatical Evolution. PhD Thesis, University College Dublin.
608. Swafford J M, Hemberg E, O'Neill M, Brabazon A (2012) Analyzing Module Usage in Grammatical Evolution. In: Proceedings of the 12th Int. Conf. on Parallel Problem Solving from Nature (PPSN XII), pp 347-356, Springer
609. Swafford J M, Nicolau M, Hemberg E, O'Neill M, Brabazon A (2012) Comparing Methods for Module Identification in Grammatical Evolution. In: Proceedings Annual Conference on Genetic and Evolutionary Computation (GECCO 2012), pp 823-830, ACM
610. Swafford J M, O'Neill M, Nicolau M, Brabazon A (2011) Exploring Grammatical Modification with Modules in Grammatical Evolution. In: Proceedings of the 14th European Conference on Genetic Programming (EuroGP 2011), Lecture Notes in Computer Science 6621, pp 310–321, Springer
611. Swafford J M, Hemberg E, O'Neill M, Nicolau M, Brabazon A (2011) A Non-Destructive Grammar Modification Approach to Modularity in Grammatical Evolution. In: Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO 2011), pp 1411–1418, ACM
612. Swafford J M, O'Neill M (2010) An Examination on the Modularity of Grammars in Grammatical Evolutionary Design. In: Proceedings of the IEEE Congress on Evolutionary Computation 2010 (WCCI 2010), pp 1027–1034, IEEE Press
613. Symmons P, Cressman K (2001) United Nations Food and Agricultural Organization (FAO) Desert Locust Guidelines (2001 edition). Chapter 1 - Biology and behaviour, http://www.fao.org/ag/locusts/common/ecg/347_en_DLG1e.pdf
614. Tan S, Ting K, Teng S (2006) Reproducing the results of ant-based clustering without using ants. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 6224–6231, IEEE Press
615. Tang W, Wu Q, Saunders J (2006) Bacterial foraging algorithms for dynamic environments. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 6224–6231, IEEE Press
616. Tavakolian R, Charkari N (2011) Novel Hybrid Clustering Optimization Algorithms Based on Plant Growth Simulation Algorithm. Journal of Advanced Computer Science and Technology Research 1:84–95

617. Tavares J, Pereira F (2010) Evolving Strategies for Updating Pheromone Trails: A Case Study with the TSP. In: Proceedings of the 11th Conference on Parallel Problem Solving from Nature (PPSN XI), Lecture Notes in Computer Science 6239, pp 523–532, Springer
618. Teller A, Veloso M (1995) PADO: A new learning architecture for object recognition. In: Symbolic Visual Learning, pp 81–116, Oxford University Press
619. Tereshko V, Lee T (2002) How Information-Mapping Patterns Determine Foraging Behavior of a Honey Bee Colony. Open Systems and Information Dynamics 9:181–193
620. Textor J (2012) Efficient Negative Selection Algorithms by Sampling and Approximate Counting. In: Proceedings of the 12th Parallel Problem Solving from Nature 2012 (PPSN 2012), pp 32–41, Springer
621. Thierens D (1999) Scalability problems of simple genetic algorithms. Evolutionary Computation 7(4):331–352
622. Thierens D, Bosman P (2001) Multi-objective mixture-based iterated density estimation evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp 663–670, San Mateo: Morgan Kaufmann
623. Thierens D et al. (eds) (2007) Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO 2007). 7–11 July, London, UK, ACM Press
624. Togelius J, Karakovskiy S, Koutnik J, Schmidhuber J (2009) Super Mario Evolution. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp 156–161, IEEE Press (also available from http://julian.togelius.com/Togelius2009Super.pdf)
625. Tong L, Wang C, Wang W, Su W (2005) A global optimization bionics algorithm for solving integer programming-plant growth simulation algorithm. Systems Engineering - Theory and Practice 25(1):76–85
626. Tong L, Zhong-tuo W (2008) Application of Plant Growth Simulation Algorithm on Solving Facility Location Problem. Systems Engineering - Theory & Practice 28(12):107–115
627. Torkkola K, Campbell W (2000) Mutual information in learning feature transformations. In: Proceedings of the 17th International Conference on Machine Learning, pp 1015–1022, Morgan Kaufmann
628. Torkkola K (2001) Nonlinear feature transforms using maximum mutual information. Proceedings of 2001 International Joint Conference on Neural Networks (IJCNN 2001), pp 2756–2761, IEEE Press
629. Trefzer M A, Kuyucu T, Miller J F, Tyrrell A M (2010) Image compression of natural images using artificial gene regulatory networks. In: Proceedings of the Annual Genetic and Evolutionary Computation Conference (GECCO 2010), pp 595–602, ACM
630. Trewavas A 2003) Aspects of Plant Intelligence (Invited Review). Annals of Botany 92:1–20
631. Trewavas A (2004) Aspects of Plant Intelligence: an Answer to Firn. Annals of Botany 93:353–357
632. Trewavas A (2005) Green Plants as intelligent organisms. Trends in Plant Science 10(9):413–419
633. Trewavas A (2007) Response to Alpi et al.:Plant neuro-biology - all metaphors have value. Trends in Plant Science 12(4):231–233

634. Trojanowski K (2007) B-Cell Algorithms as a Parallel Approach to Optimization of Moving Peaks Benchmark Tasks. In: Proceedings of 6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM'07), pp 143–148, IEEE Press
635. Tsai X-Y, Chen Y-J, Huang H-C, Chuang S-J, Hwang R-C (2005) Quantum NN vs NN in Signal Recognition. In: Proceedings of the Third International Conference on Information Technology and Applications (ICITA 05), pp 308–312, IEEE Press
636. Turing A M (1948) Intelligent Machines, Reprinted in Ince D.C. (ed) (1992) Mechanical Intelligence: Collected Works of A. M. Turing, pp 21–23, North-Holland
637. Turing A M (1950) Computing Machinery and Intelligence. Mind 59(236):433–460. Reprinted in Ince D.C. (ed) (1992) Mechanical Intelligence: Collected Works of A. M. Turing, North-Holland
638. Turing A M (1952) The Chemical Basis of Morphogenesis. Philosophical Transactions of the Royal Society B 237:37–72
639. Tyrrell A, Jin Y (eds) (2011) Special issue on Evolving Developmental Systems. IEEE Transactions on Evolutionary Computation 15(3)
640. van den Bergh F (2005) An Analysis of Particle Swarm Optimizers. PhD Thesis. Department of Computer Science, University of Pretoria, South Africa
641. Vandersypen L M K, Steffen M, Breyta G, Yannoni C S, Sherwood M H, Chuang I L (2001) Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. Nature 414: 883–887.
642. Vapnik V, Golowich S, Smola A (1997) Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing. In: Mozer M, Jordan M, Petsche T (eds) Neural Information Processing Systems, Vol. 9, Cambridge, MA: MIT Press
643. Viana F, Kotinda G, Rade D, Steffan V (2006) Can Ants Design Mechanical Engineering Systems. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), pp 3173–3179, IEEE Press
644. von Frisch K (1967) The Dance Language and Orientation of Bees. Harvard University Press
645. von Frisch K, Lindauer M (1996) The "Language" and Orientation of the Honey Bee, In: Houck L, Drickamer L (eds) Foundations of Animal Behavior: Classic papers with Commentaries, pp 539–552, University of Chicago Press
646. Wang C, Cheng H Z, Yao L Z (2008) Reactive Power Optimization by Plant Growth Simulation Algorithm. In: Proceedings of Third International Conference on Electric Utility regulation and Restructuring and Power Technologies (DRPT 2008), pp 771–774, IEEE Press
647. Wang L (2005) Support Vector Machines: Theory and Applications, Berlin: Springer
648. Wang Y, Feng X Y, Huang Y X, Pu D B, Zhou W G, Liang Y C, Zhou C G (2007) A novel quantum swarm evolutionary algorithm and its applications. Neurocomputing 70(4–6):633–640
649. Ward J (1963) Hierarchical Grouping to optimize an objective function. Journal of the American Statistical Association 58(301):236–244
650. Watkins A, Timmis J, Boggess L (2005) Artificial Immune Recognition System (AIRS): An Immune-Inspired Supervised Learning Algorithm. Genetic Programming and Evolvable Machines 5(3):291–317

651. Watkins A, Boggess L (2002a) A resource limited artificial immune classifier. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2002), pp 926–931, IEEE Press
652. Watkins A, Boggess L (2002b) A New Classifier Based on Resource Limited Artificial Immune Systems. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2002), pp 1546–1551, IEEE Press
653. Watkins A, Timmis J (2002c) Artificial Immune Recognition System (AIRS): Revisions and Refinements. In: Proceedings of the First International Conference on Artificial Immune Systems (ICARIS 2002), pp 173–181, University of Kent at Canterbury
654. Wedde H, Farooq M, Zhang Y (2004) BeeHive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior. In: Proceedings of ANTS 2004, Lecture Notes in Computer Science 3172, pp 83–94, Berlin: Springer-Verlag
655. Werbos P J (1974) Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis. Harvard University
656. Whigham P A (1996) Grammatical bias for evolutionary learning. PhD Thesis. University of New South Wales, Australian Defence Force Academy
657. Whitley D, Dominic S, Das R, Anderson C W (1993) Genetic reinforcement learning for neurocontrol problems. Machine Learning 13(2–3):259–284
658. Wilkinson G (1992) Information transfer at evening bat colonies. Animal Behaviour 44(3):501–518
659. Wilkinson G, Boughman J (1998) Social calls coordinate foraging in greater spear-nosed bats. Animal Behaviour 55:337–350
660. Wilson G C, McIntyre A, Heywood M I (2004) Resource review: three open source systems for evolving programs—Lilgp, ECJ and Grammatical Evolution. Genetic Programming and Evolvable Machines 5(1):103–105
661. Wolpert D, Macready W (1995) No Free Lunch Theorems for Search. Santa Fe Institute Working Paper 95–02–010, Santa Fe: Santa Fe Institute
662. Wong M L, Leung K S (2000) Data Mining Using Grammar Based Genetic Programming and Applications. Kluwer Academic Publishers
663. Wootters W K, Zurek W H (1982) A Single Quantum Cannot be Cloned. Nature 299: 802-803
664. Wray M, Klein B and Seeley T (2011) Honey bees use social information in waggle dances more fully when foraging errors are more costly. Behavioral Ecology, published online 1 October 2011, http://beheco.oxfordjournals.org/content/early/2011/09/30/beheco.arr165.full.pdf+html
665. Wu S, Banzhaf W (2008) Combatting financial fraud: A co-evolutionary anomaly detection approach. In: Proceedings of the Annual Genetic and Evolutionary Computation Conference (GECCO 2008), pp 1673–1680, ACM Press
666. Xu J, Lam A Y S, Li V O K (2011) Stock portfolio selection using chemical reaction optimization. In: Proceedings of the International Conference on Operations Research and Financial Engineering (ICORFE 2011), Paris, France, pp 458-463, World Academy of Science Engineering & Technology
667. Xu J, Lam A Y S, Li V O K, Li Q, Fan X (2012) Short adjacent repeat identification based on Chemical Reaction Optimization. In: Proceedings of the 2012 IEEE Congress on Evolutionary Computation (WCCI 2012), 1–8, IEEE Press

668. Yang S (2005) Memory-enhanced Univariate Marginal Distribution Algorithms for Dynamic Optimization Problems. In: Proceedings of 2005 IEEE Congress on Evolutionary Computation (CEC 2005), pp 2560–2567, IEEE Press
669. Yang S, Wang M, Jiao L (2004a) A Quantum Particle Swarm Optimization. In: Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004), pp 320–324, IEEE Press
670. Yang S, Wang M, Jiao L (2004b) A novel quantum evolutionary algorithm and its application. In: Proceedings of 2004 IEEE Congress on Evolutionary Computation (CEC 2004), pp 820–826, IEEE Press
671. Yang S, Wang M, Jiao L (2004c) A genetic algorithm based on quantum chromosome. In: Proceedings of IEEE International Conference on Signal Processing (ICSP 04), pp 1622–1625, IEEE Press
672. Yang X-S (2005) Engineering Optimization via Nature-Inspired Virtual Bee Algorithms. In: Mira J, Álvarez, J (eds) Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: First International Work-Conference on the Interplay Between Natural and Artificial Computation (IWINAC 2005), Lecture Notes in Computer Science 3562, pp 317–323, Berlin: Springer
673. Yang X-S (2008) Nature-inspired metaheuristic algorithms. Luniver Press
674. Yang X-S (2010) A New Metaheuristic Bat-Inspired Algorithm. In: Proceedings of Fourth International Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO 2010), pp 65–74, Springer
675. Yang X-S, Gandomi A (2012) Bat Algorithm: A Novel Approach for Global Engineering Optimization. Engineering Computations 29(5):464–483
676. Yang X-S (2013) Bat Algorithm: Literature Review and Applications. International Journal of Bio-Inspired Computation 5(3):141–149
677. Yao X (1999) Evolving artifical neural networks. Proceedings of the IEEE 87(9):1423–1447
678. Yu J J Q, Lam A Y S, Li V O K (2012) Real-coded chemical reaction optimization with different perturbation functions. In: Proceedings of the 2012 IEEE Congress on Evolutionary Computation (WCCI 2012), pp 1–8, IEEE Press
679. Zhang J, Sanderson A (2009) JADE: Adaptive differential evolution with optional external archive. IEEE Transactions on Evolutionary Computation 13(5):945–958
680. Zhao, Z, Cui Z, Zeng J, Yue X (2011). Artificial Plant Optimization Algorithm for Constrained Optimization Problems. In: Proceedings of 2011 Second International Conference on Innovations in Bio-inspired Computing and Applications, pp 120–123, IEEE Press
681. Zhou Y and Liu B (2009) Two Novel Swarm Intelligence Clustering Analysis Methods. In: Proceedings of the 5th International Conference on Natural Computation, pp 497–501, IEEE Press
682. Zitzler E, Laumanns M, Bleuler S (2004) A tutorial on evolutionary multiobjective optimization. Metaheuristics for Multiobjective Optimisation 535:3–38
683. Zitzler E, Laumanns M, Thiele L (2001). SPEA2: Improving the strength Pareto evolutionary algorithm. In: Proceedings of Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001), pp 95–100, International Center for Numerical Methods in Engineering (CIMNE)

684. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Transactions on Evolutionary Computation 3(4):257–271

Index

$(1+1)$-ES, 74
$(\mu+\lambda)$-ES, 75
(μ,λ)-ES, 75
$\mathcal{MAX}$-$\mathcal{MIN}$, 152
πGE, 379
k-exchange neighbourhood, 159
1/5 success rule, 76

activation function, 226, 230
adaptive immune system, 301–309
adaptive resonance theory, 276
adaptive simulated annealing, 421
adiabatic, 411
aerotaxis, 188
affinity, 306
affinity maturation, 306, 320
AIC, 70
amino acid, 358
angle modulation, 90
annealing, 411
  quantum, 411
ant algorithms, 142
  ant clustering algorithms, 160
  ant foraging, 142
  continuous optimisation, 155
  multiple ant colonies, 157
ant colony system (ACS), 152
ant miner, 167
ant multitour systems, 153
ant system (AS), 147
antennation, 141
antibody, 304, 305, 320, 321, 323
antigen, 304, 305, 320
antigen-presenting cell, 309
apoptosis, 315
architecture-altering operation, 111
arity, 98
artificial chemical reaction optimisation algorithm, 482
artificial immune recognition system (AIRS), 325
artificial immune system (AIS), 10, 199, 301
artificial neural network, 10, 221–259
artificial neural networks, 222
artificial recognition ball, 325
asynchronous update, 120
atomic swarm, 133
attentional subsystem, 277
auto-catalytic, 143
auto-immune, 307
autoinducer, 187
automatically defined function (ADF), 106–108, 111
automatically defined iteration (ADI), 109–111
automatically defined loop (ADL), 109–111
automatically defined recursion (ADR), 111
automatically defined storage (ADS), 108, 111
Avogadro's number, 398

B cell, 304, 305, 320
B cell algorithm, 322

© Springer-Verlag Berlin Heidelberg 2015

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8

backpropagation, 228, 235, 240, 241, 245, 284
Backus–Naur form (BNF), 340
bacterial foraging, 187
bacterial foraging optimisation algorithm
  dynamic enviroments, 198
bacterial foraging optimisation algorithm (BFOA), 190, 198
base vector, 88
basis function, 223
bat algorithm, 206, 208
bayesian optimisation algorithm (BOA), 66
Bayesian–Dirichlet metrics, 70
bee colony algorithm, 175
bee nest algorithm, 182
bee nest site selection, 180
  optimisation algorithm (BNSO), 181, 182
bee system algorithm, 174
best matching unit (BMU), 265
bias node, 225
binary encoding, 27
binary gravitational search algorithm, 431, 434
binary PSO, 137–138
binary valued quantum inspired evolutionary algorithm, 440
bioluminesence, 201
black box, 259
Black–Scholes(–Merton) equation, 406
blast cell, 306
bloat, 105
Boltzmann factor, 400, 419
brood sorting, 160
bubble neighbourhood function, 270
building block, 59

cell membrane, 304
cellular differentiation, 7
cellular EA (cEA), 48, 49
cellular encoding, 297
cellular immunity, 304
cemetery formation, 161
central force optimisation, 429
charged particle swarm optimisation, 132
chemical reaction optimisation, 481
chemically inspired algorithms, 479
chemotaxis, 188, 189
chromosome, 282
classical mechanics, 393, 395
classification, 301, 311
clonal expansion and selection, 306
CLONALG algorithm, 320
closure, 98, 99
clustering, 261, 262
  clustering algorithm, 142, 266
codebook vector, 267
codon, 359
colony fission, 180
committee decision, 244
commutator, 405, 407
compact genetic algorithm (cGA), 65
comparison layer, 277
competent genetic algorithm, 59
competent genetic programming, 59
competing conventions problem, 288
compound, 480
computing
  quantum adiabatic, 410
connection gene, 290
connection matrix, 285, 286
connection topology, 223
conservative force, 396
constrained molecular dynamics (CMD), 426
constrained optimisation, 50
constriction coefficient, 124
construction graph, 145
context layer, 245, 246
context-free grammar, 339
context-sensitive grammar, 339
contiguous somatic hypermutation, 322
continuous ant colony system (CACS), 155
continuous univariate marginal distribution algorithm (UMDAc), 64
cooling schedule, 420
costimulated, 306
Cover's theorem, 253
crossover, 22, 35, 59
  in differential evolution, 83–87
  in genetic programming, 102
crowding operators, 40
Curie temperature, 411

cytokines, 303, 306

damage associated molecular patterns (DAMPS), 309
dance attrition, 180
dance floor, 173
dance language, 172
danger theory, 309
decoherence, 405, 409
degenerate genetic code, 361
dendrites, 222
dendritic cell algorithm, 309, 315
dendritic cells, 309
derivation tree, 338, 346
developmental computing, 10, 335
developmental plasticity, 469
developmental tree-adjoining grammar-based genetic programming, 375
difference vector, 84
differential cellular elongation, 458
differential evolution (DE), 83, 85, 86, 372
 with random scale factor, 89
 with time varying scale factor, 89
differential gene expression, 386
digital quantum computers, 408
direct encoding, 282
discrete recombination, 79
distributed EA (dEA), 48
distributed learning, 142
diversity, 40, 48, 88, 295, 361
DNA, 306, 357, 358
dodder plant, 461
DTAG3P, 379

early-stopping, 239
echolocation, 206
efficient frontier, 54
element, 479
elitism, 40
elitist ant system (EAS), 151
Elman network, 245
emergent, 142
encoding, 22
energy, 395
entanglement, 404, 409
enthalpy, 483
ephemeral random constants (ERCs), 99
epigenesis, 59
epitope, 305
equivalent convention problem, 288
ergodic, 402, 422
 strongly, 402
 weakly, 402
ergodicity, 402, 422
 strong, 402
 weak, 402
error surface, 235
estimation of distribution algorithms (EDAs), 61, 450
Euclidean distance, 164, 273
evolution strategies, 73
evolutionary algorithm, 281
exons, 358
exploitation, 31, 150
exploration, 31, 150
exponential cooling, 420
exponential crossover, 88
extremal optimisation algorithm, 436

fear threshold, 131
feedforward, 242, 245
feeding buzz, 207
field theory, 394
 quantum, 395
first law of thermodynamics, 400
fish school algorithm, 211
fish school search, 212
fish school search algorithm, 214
fitness, 4, 22, 32, 119
fitness function, 22
fitness imitation, 30
fitness inheritance, 30
fitness proportionate selection, 32
fitness sharing, 46, 295
fittest individual refinement (FIR), 90
fixed line sweep, 50
fixed-length encoding, 96
floor function, 90
food foraging strategy, 143
foraging for light, 456
force, 396
free grammars, 339
frustration, 412
full method, 100

function approximation problem, 221
function set, 98

gbest, 88, 119, 120, 124
gene, 96
gene-overlapping, 366
general relativity, 394
generalisation, 237, 251
generation, 22
generation gap, 39
generational replacement, 39
genetic algorithm (GA), 22, 59
genetic programming (GP), 59, 95, 112, 345, 357, 373
genetic regulatory network, 383, 386–388
genotype, 21
geotropism, 457, 458
global recombination, 79
glow worm swarm algorithm (GWSA), 201
gradient-descent, 240
grammar, 10, 112, 297, 330, 337, 357
  free, 339
grammar-based genetic programming, 345, 357
grammar-guided genetic programming, 346
grammatical computing (GC), 10, 335, 336, 343, 356, 381
grammatical evolution (GE), 357–373, 379
grammatical swarm, 371
gravitational search algorithm, 431
Gray code, 27, 28
greedy search, 4
grow method, 100

Hamiltonian, 396, 397, 400, 405, 407, 410, 412, 418, 422, 424
Hamming cliffs, 27
Hamming distance, 269
Heaviside step function, 230
helper T cell, 306, 307
heritability, 29
heuristic bias, 29
hidden layer, 225, 228, 241, 246
hierarchical BOA, 71
Hilbert space, 402
hill climbing algorithm, 4
honeybee dance language, 171
honeybee foraging, 172
honeybee mating optimisation algorithm, 184, 185
honeybee optimisation algorithm, 173
humoral immunity, 304, 307
hydrotropism, 457, 459

immune programming, 330
immune system, 301–309
immune system grammar, 330
immunodominance, 305
immunodominant, 305
immunoglobulin, 307
indirect encoding, 282
initial weight vector, 241
innate immune system, 302
innovation number, 290
input space, 245
intermediate recombination, 79
internal state space, 245
introns, 358
invasive weed optimisation algorithm (IWO), 464
Ising spin glasses, 412
island model, 48, 134, 136

jump connection network, 242

k nearest neighbour, 165
k nearest neighbours algorithm, 311
kernel function, 249, 256
Kolmogorov's theorem, 228

Lagrangian, 397
lbest, 124
learning algorithm, 223, 281
learning rate, 236, 240, 265
learning vector quantisation (LVQ), 272
leukocyte, 302, 305
likeness function, 226
linear ranking, 33
linkage learning, 59
Lisp S-expressions, 96
local recombination, 79
locality, 28
locality preserving, 263
lock in, 144

locust swarm algorithm, 216, 217
locusts, 215
logistic function, 230
lymph node, 306
lymphocyte, 302, 304, 305

magnetotaxis, 188
major histocompatibility complex, 304
mapping, 341
marriage in honeybees optimisation algorithm, 184
maximally separating hyperplane, 256
mean squared error, 239
mechanics
  classical, 393, 395
  quantum, 394, 402
  relativistic quantum, 394
  statistical, 395, 400
memetic algorithms, 58
memetic computation, 58
memory, 22, 119, 143
memory in ant foraging algorithms, 147
meristem tissue, 471
messy GA (mGA), 59
Metropolis(–Hastings) algorithm, 401, 413, 418
migration, 48
minimum description length, 70
mixing number, 78
molecular biology, 357
molecule, 480
momentum, 240
momentum coefficient, 240
monopodial, 471
morphogenesis, 335
multilayer perceptron (MLP), 228, 243, 244, 297
multimodal, 216
multiobjective optimisation, 53
multiple swarms, 134
mutation, 22, 25, 35
  in differential evolution, 84
mutation step size, 76

natural computing, 1
NEAT, 289
necrosis, 315
negative selection algorithm, 301, 310, 312
neighbourhood function, 265
neuroevolution, 281
neuroevolution of augmenting topologies, 289
neurons, 221
neutral evolution, 361
nondominated, 54
nondominated sorting genetic algorithm (NSGA), 57
nonlinear modulation index, 466
nonself, 301, 302, 310
nonsynonymous redundancy, 288
nonterminal, 340

objective function, 29
onemax, 23
ontogeny, 7
optimisation, 142, 301
orienting subsystem, 277
otoliths, 459
out-of-sample, 164, 239
overfitting, 237, 242
overtraining, 237

paddy field algorithm, 467
panmictic, 48
parasitic plants, 461
paratope, 305
Pareto frontier, 54, 55
Pareto optimal, 54
Pareto ranking, 56
Pareto set, 54
particle, 118, 120
particle swarm algorithm, 88
particle swarm optimisation (PSO), 118–140, 272
pathogen, 301, 302, 304, 305
pathogen-associated molecular patterns, 302
pbest, 119, 120
penalty function, 52
peptide, 304
permutation problem, 288
phagocyte, 302
phenotype, 21, 361
phenotypic plasticity, 469
pheromone, 143, 144
pheromone evaporation, 146
pheromone matrix, 146

pheromone trail, 143
phloem, 462
photosynthesis, 456
phototaxis, 188
phototropism, 457, 458
phylogeny, 7
physical computing (PC), 437
physical field inspired algorithms, 429
plant algorithms, 455
plant growth simulation algorithm, 469, 471, 474
plant immune systems, 303
plant neurobiology, 462
plant propogation algorithms, 464
POE model, 7
point mutation, 104
Poisson bracket, 397
population based incremental learning (PBIL), 62
positive feedback, 144
predator–prey, 130
prefix notation, 96
premature convergence, 33, 151
primary response, 308
principal component analysis, 267
production rule, 340, 341, 377
protein, 304, 305, 357

quantum adiabatic computing (AQC), 410
quantum annealing, 414
  simulated, 422
quantum annealing (QA), 411
quantum binary PSO, 450
quantum chromosome, 439, 441
quantum field theory, 395
quantum gate, 443
quantum information, 409
quantum inspired evolutionary algorithms, 439, 440
quantum mechanics, 394, 402
  relativistic, 394
quantum mutation, 445
quantum observation step, 442
quantum spin glass, 414
quantum swarm, 450
quantum tunnelling, 406, 414, 422
qubit, 407, 409, 439
qubit representation, 439, 440
quenching, 411
quickprop, 283
quorum sensing, 181, 187

r-contiguous bits, 312
radial basis function, 247
radial basis function network (RBFN), 246, 249
radicle, 459
ramped-half-and-half, 101
random immigrants strategy, 47
rank-based ant system (AS_{rank}), 151
rank-based selection, 33
real-valued clonal selection algorithm, 323
real-valued quantum inspired evolutionary algorithm, 446
receptor, 304, 306, 307
recognition layer, 277
recruitment dance, 172
recurrent network, 244, 245, 286
recuts, 243
regular grammar, 339
regularisation, 239
relativistic quantum mechanics, 394
relativity
  general, 394
  special, 394
repair operator, 52
replacement strategy, 39
restricted mating strategy, 46
restricted replacement strategy, 46
result-producing branch (RPB), 106, 107
ribonucleic acid (RNA), 358
root foraging, 459
root-swarm behaviour, 475
round dance, 173

schema, 35
schema theorem, 112
Schrödinger equation, 403
search engine, 361, 371
second law of thermodynamics, 400
secondary response, 308
selection, 22
selection pressure, 31, 32, 34
self, 310
self-organisation, 117, 142

self-organised criticality, 436
self-organising map (SOM), 164, 262, 263, 266
self-organising swarm (SoSwarm), 272
sentry particles, 130
sentry strategy, 45
sessile, 457
sharing function, 296
sigmoid transformation, 138
simulated annealing, 398, 417
simulated quantum annealing, 422
simulated quenching, 420
single point crossover, 35, 36
social insects, 142
social learning, 117
social models, 9
social programming, 371
somatic hypermutation, 306
special relativity, 394
spin glasses, 411
sporulation, 188
squashing function, 230
stability/plasticity dilemma, 276
stacking, 243, 244
start symbol, 340, 341
statistical mechanics, 395, 400
statoliths, 458
steady state, 40
stigmergy, 141
stomata, 457
strategy parameter, 75
strawberry plant algorithm, 468, 470
strength Pareto evolutionary algorithm (SPEA), 57
stridulation, 141
strong ergodicity, 402
strong locality, 28
strongly ergodic, 402
strongly typed GP, 112
structured population GA, 48
subtree mutation, 104
sufficiency, 98, 99
sum of the squared errors, 239
superior chromosome, 174
superposition of states, 403
supervised learning, 223, 234, 261, 272
support vector machines (SVMs), 252
support vectors, 255
Suzuki-Trotter formalism, 415
Suzuki-Trotter formula, 407
switching objective approach, 55
symplectic structure, 405
sympodial, 471
synapses, 222
synonymous redundancy, 288
syntax tree, 96, 100, 102, 357
systems biology, 502

T cell, 304, 305, 310
TAG3P, 377
TAG3P+, 377
TAGE, 379, 380
taxis, 188
terminal, 340
terminal set, 98
test data, 239
thermodynamic equilibrium, 399
thermodynamics, 395, 398
thermotaxis, 188
thigmotropism, 457, 459
thymus, 307
tolerogenesis, 307, 310
topology preserving, 263
tournament selection, 34
trace, 406
training algorithm, 264
training data, 239, 259
training method, 223
transfer function, 226, 229, 230
transpiration, 457
transverse Ising spin glass, 414
travelling salesman problem (TSP), 148, 417, 419
tree-adjoining grammar (TAG), 375–378, 380
tree-adjunct grammar, 377
tremble dance, 173
Trotter dimension, 415
truncation selection, 34
two point crossover, 35, 36

uncertainty principle, 405
uniform choice, 50
uniform crossover, 36, 37
univariate marginal distribution algorithm (UMDA), 63
universal approximator, 228
unsupervised learning, 164, 223, 261

validation data, 239
variable-length, 97, 357
variable-size detectors, 313
vector evaluated GA (VEGA), 55
velocity clamping, 123
velocity update, 120, 124, 137
Venus flytrap, 462
very fast (simulated) reannealing, 421
vigilance parameter, 277, 279
volatile organic compound, 461

waggle dance, 173
wavefunction, 402
weak ergodicity, 402
weak locality, 29
weakly ergodic, 402
weight-based genetic algorithm (WBGA), 55
wrapping operator, 366

xylem, 462