A Comparison of Strength and Accuracy-Based Fitness in Learning Classifier Systems

Timothy Michael Douglas Kovacs

A thesis submitted
to The University of Birmingham
for the degree of Doctor of Philosophy

School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom

January 2002

Abstract

Classifier systems are an intriguing approach to a broad range of machine learning problems, based on automated generation and evaluation of condition/action rules. In reinforcement learning tasks they simultaneously address the two major problems of learning a policy and generalising over it (and related objects, such as value functions). Despite over 20 years of research, however, classifier systems have met with mixed success, for reasons which were often unclear. Finally, in 1995 Stewart Wilson claimed a long-awaited breakthrough with his XCS system, which differs from earlier classifier systems in a number of respects, the most significant of which is the way in which it calculates the value of rules for use by the rule generation system. Specifically, XCS (like most classifier systems) employs a genetic algorithm for rule generation, and the way in which it calculates rule fitness differs from earlier systems. Wilson described XCS as an accuracy-based classifier system and earlier systems as strength-based. The two differ in that in strength-based systems the fitness of a rule is proportional to the return (reward/payoff) it receives, whereas in XCS it is a function of the accuracy with which return is predicted. The difference is thus one of credit-assignment, that is, of how a rule's contribution to the system's performance is estimated. XCS is a Q-learning system; in fact, it is a proper generalisation of tabular Q-learning, in which rules aggregate states and actions. In XCS, as in other Q-learners, Q-values are used to weight action selection. In rule generation, a rule's weight is an inverse function of the error of its Q-value; thus XCS finds rules which minimise errors in the Q-function, and is able to accurately approximate it. Wilson argued XCS's superiority over earlier systems, and demonstrated improved performance on two tasks. However, the theoretical basis of XCS's results – and indeed those of earlier systems – needed development. Consequently, this thesis has been devoted to the development of theory to explain how and why classifier systems behave as they do. In particular, a theory underlying the difference between strength and accuracy-based fitness has been sought. In order to compare XCS with more traditional classifier systems, SB–XCS, XCS's strength-based twin, is introduced. SB–XCS is designed to

resemble XCS as much as possible, except that it uses the Q-value of a rule as its weight in both action selection and rule generation. It is shown that this minor algorithmic change has profound and subtle implications on the way in which SB–XCS operates, and on its capacity to adapt. Two types of non-sequential learning task are identified, distinguished by whether the reward function is biased or unbiased. (An unbiased reward function is constant over the action(s) in each state which maximise reward.) It is argued that although SB–XCS can be expected to adapt to unbiased reward functions, its ability to adapt to biased reward functions is limited. Further, it is claimed that SB–XCS will only adapt well to trivial sequential decision tasks. SB–XCS's difficulties are attributed to two forms of pathological relationships between rules, termed strong and fit overgeneral rules. The analysis of XCS and SB–XCS presented herein supports the study of accuracy-based fitness, and fitness sharing in strength-based systems as directions for future work.

Acknowledgements

My supervisor, Manfred Kerber, has always been particularly generous with his time and ideas. His intuition, patience, and thoroughness continue to surprise me. His talent as a proofreader and his expertise with LaTeX raised the standard of my work, but his example raised it further. Thank you Manfred.

I owe a great deal to Stewart Wilson, who has been a consistently invaluable source of information and encouragement since I was an MSc student. Through his own work he has, more than anyone, sustained the study of classifier systems in difficult times. But, in addition to this, his encouragement and assistance of others has meant a great deal, as many will attest.

Alwyn Barry has been a cheerful source of assistance and encouragement for many years, has shown an interest in both my research and my career, and has been good company in distant places. For all that I am indebted to him. I would also like to thank Larry Bull for his interest and efforts to help me, and for pointing out the importance of fitness sharing.

I am fortunate to have benefited from contact with a number of other classifier systems researchers who have made my studies much more stimulating, both through email and face to face. In particular, Larry Bull, Luis Hercog, Pier Luca Lanzi, Sonia Schulenburg and Wolfgang Stolzmann have helped make every conference a memorable one.

I would like to thank Riccardo Poli my internal examiner and thesis committee member for his kind words and advice over the years, for his careful consideration of this work and for his encouragement. Thanks also to my external examiner, Robert Smith, for his consideration and for making my viva less difficult than it might have been.

I am indebted to Aaron Sloman, my MSc co-supervisor and member of my thesis committee, as anyone who reads his work and mine might guess. I am also indebted to Ian Wright, my other MSc co-supervisor, who first steered me towards XCS.

It was my pleasure to share an office with Stuart Reynolds for four years, and such is his good nature, humour, and, it must be said, tolerance, that

I would welcome another four. He has also been a great source of technical assistance in various forms and on innumerable occasions.

On the subject of providing assistance, technical and otherwise, there can be few who excel as Axel Groβmann does. Most of his contemporaries at Birmingham have benefited from his expertise with computers and the culinary arts, just two of his skills.

I am grateful to Jeremy Wyatt for his patient efforts to teach me a little about reinforcement learning, but more so for a little poker, a few parties, and many pints.

Many others at Birmingham are due thanks; Peter Hancox and Achim Jung for making my PhD and teaching assistantship possible, Jonny Page for his surreal sanity checks, Adrian Hartley for getting me to the airport on time and for our excellent adventures, Marcos Quintana Hernandez for being excellent himself, John Woodward for his humour, Gavin Brown for being Gavin, Paola Maneggia for pancakes, Mathias Kegelmann for late discussions, Pawel Waszkiewicz for guitar lessons, Xin Yao for making sure I was not the only one working late at night, and Richard Pannell and Sammy Snow for many little things.

I would like to thank my parents for their support in this and earlier endeavours, without which this work would not have been possible. Finally, I would like to thank Qulsom, who has shared with me more of the ups and downs of this work than anyone, and for whose suffering I am sorry.

The description of XCS presented in this work is drawn from Stewart Wilson's published descriptions of it in [298, 304, 53], from his "NetQ" questions and answers on XCS to be found on his home page [314], and from extensive correspondence with him on the details of its implementation. Our valuable discussions with Alwyn Barry in 1998 helped bring out more details of Stewart's XCS implementation. This and the growing interest in XCS suggested the need for a new, more comprehensive description, which finally appeared in 2000 [53] thanks to the energy of Martin Butz. Needless to say, the mechanisms described herein are due entirely to Stewart Wilson, with the few exceptions duly noted.

Contents

1 Introduction **1**

1.1 Two Example Machine Learning Tasks 2
1.2 Types of Task 4
1.2.1 Supervised and Reinforcement Learning 4
1.2.2 Sequential and Non-Sequential Decision Tasks 4
1.3 Two Challenges for Classifier Systems 5
1.3.1 Problem 1: Learning a Policy from Reinforcement . . . 6
1.3.2 Problem 2: Generalisation 6
1.4 Solution Methods 6
1.4.1 Method 1: Reinforcement Learning Algorithms 6
1.4.2 Method 2: Evolutionary Algorithms 7
1.5 Learning Classifier Systems 7
1.5.1 The Tripartite LCS Structure 7
1.5.2 LCS = Policy Learning + Generalisation 8
1.5.3 Credit Assignment in Classifier Systems 9
1.5.4 Strength and Accuracy-based Classifier Systems 10
1.6 About the Thesis 10
1.6.1 Why Compare Strength and Accuracy? 11
1.6.2 Are LCS EC- or RL-based? 12
1.6.3 Moving in Design Space 15
1.7 Structure of the Thesis 17

2 Learning Classifier Systems **21**

2.1 Types of Classifier Systems 23
2.1.1 Michigan and Pittsburgh LCS 23
2.1.2 XCS and Traditional LCS? 24
2.2 Representing Rules 25
2.2.1 The Standard Ternary Language 25
2.2.2 Other Representations 26
2.2.3 Summary of Rule Representation 28
2.2.4 Notation for Rules 28

2.3 XCS 28
2.3.1 Wilson's Motivation for XCS 29
2.3.2 Overview of XCS 30
2.3.3 Wilson's Explore/Exploit Framework 33
2.3.4 The Performance System 35
2.3.5 The Credit Assignment System 38
2.3.6 The Rule Discovery System 45
2.4 SB–XCS 52
2.4.1 Specification of SB–XCS 53
2.4.2 Comparison of SB–XCS and Other Strength LCS . . . 56
2.5 Initial Tests of XCS and SB–XCS 57
2.5.1 The 6 Multiplexer 57
2.5.2 Woods2 61
2.6 Summary 66

3 How Strength and Accuracy Differ **69**
3.1 Thinking about Complex Systems 70
3.2 Holland's Rationale for CS-1 and his Later LCS 71
3.2.1 Schema Theory 71
3.2.2 The Bucket Brigade 72
3.2.3 Schema Theory + Bucket Brigade = Adaptation 73
3.3 Wilson's Rationale for XCS 73
3.3.1 A Bias towards Accurate Rules 73
3.3.2 A Bias towards General Rules 74
3.3.3 Complete Maps 77
3.3.4 Summary 78
3.4 A Rationale for SB–XCS 78
3.5 Analysis of Populations Evolved by XCS and SB–XCS 78
3.5.1 SB–XCS 79
3.5.2 XCS 81
3.6 Different Goals, Different Representations 83
3.6.1 Default Hierarchies 84
3.6.2 Partial and Best Action Maps 85
3.6.3 Complete Maps 86
3.6.4 What do XCS and SB–XCS Really Learn? 87
3.7 Complete and Partial Maps Compared 89
3.7.1 Advantages of Partial Maps 89
3.7.2 Disadvantages of Partial Maps 92
3.7.3 Complete Maps and Strength 99
3.7.4 Contrasting Complete and Partial Maps in RL Terminology 100

3.7.5 Summary of Comparison . . . 100
3.8 Ability to Express Generalisations . . . 100
3.8.1 Mapping Policies and Mapping Value Functions . . . 101
3.8.2 Adapting the Accuracy Criterion . . . 102
3.8.3 XCS-hard and SB–XCS-easy Functions . . . 103
3.8.4 Summary of Generalisation and Efficiency . . . 103
3.9 Summary . . . 104

4 What Should a Classifier System Learn? 105
4.1 Representing Boolean Functions . . . 107
4.1.1 Truth Tables . . . 107
4.1.2 On-sets and Off-sets . . . 108
4.1.3 Sigma Notation . . . 108
4.1.4 Disjunctive Normal Form . . . 108
4.1.5 Representing Functions with Sets of Rules . . . 109
4.2 How Should a Classifier System Represent a Solution? . . . 110
4.2.1 The Value of a Single Rule . . . 111
4.2.2 The Value of a Set of Rules . . . 111
4.2.3 Complete and Correct Representations . . . 112
4.2.4 Minimal Representations . . . 114
4.2.5 Non-overlapping Representations . . . 117
4.2.6 Optimal Rule Sets: [O]s . . . 120
4.2.7 Conflicting Rules . . . 120
4.2.8 Representation in XCS . . . 120
4.3 How Should We Measure Performance? . . . 121
4.3.1 Measures of Performance . . . 122
4.3.2 Measures of Population State . . . 123
4.3.3 Measuring Performance and Measuring State . . . 124
4.3.4 New Population State Metrics . . . 127
4.3.5 Testing XCS with %[PI] . . . 128
4.3.6 Testing XCS with %[m-DNF] . . . 130
4.3.7 Summary of Metrics and Properties . . . 131
4.4 Summary . . . 132

5 Prospects for Adaptation 135
5.1 Known Problems with Strength LCS . . . 137
5.2 Methodology for Rule Type Analysis . . . 139
5.3 Analysis of Rule Types . . . 141
5.3.1 Correct and Incorrect Actions . . . 141
5.3.2 Overgeneral Rules . . . 142
5.3.3 Strong Overgeneral Rules . . . 148

5.3.4 Fit Overgeneral Rules . . . 149
5.3.5 Parallel Definitions of Strength and Fitness . . . 149
5.4 When are Strong and Fit Overgenerals Possible? . . . 150
5.4.1 The Reward Function is Relevant . . . 151
5.4.2 Competition for Action Selection . . . 153
5.4.3 Competition for Reproduction . . . 154
5.5 Strong Overgenerals in XCS . . . 155
5.5.1 Biases between Actions do not Produce Strong Overgenerals . . . 156
5.5.2 Some Properties of Accuracy-Based Fitness . . . 157
5.6 Strong Overgenerals in SB–XCS . . . 159
5.6.1 When are Strong Overgenerals Impossible in SB–XCS? 160
5.6.2 What Makes Strong Overgenerals Possible in SB–XCS? 161
5.7 Fit Overgenerals and the Survival of Rules under the GA . . . 163
5.7.1 Comparison on an Unbiased Reward Function . . . 164
5.7.2 Comparison on a Biased Reward Function . . . 164
5.7.3 Discussion . . . 164
5.8 Designing Strong and Fit Overgenerals for XCS . . . 166
5.8.1 Biased Variance Functions . . . 166
5.8.2 Empirical Results . . . 168
5.8.3 Avoiding Fit Overgenerals . . . 168
5.8.4 SB–XCS and Biased Variance Functions . . . 170
5.9 Strong and Fit Undergeneral Rules . . . 170
5.10 Why Bias the Reward Function? . . . 171
5.10.1 Some State-actions are more Important than Others . . . 172
5.10.2 A Rule Allocation Bias can Focus Resources . . . 172
5.11 Rule Allocation Reconsidered . . . 173
5.11.1 Knowing What Not to Do . . . 174
5.11.2 Managing Exploration . . . 174
5.11.3 Complete and Partial Maps Revisited . . . 175
5.11.4 Alternatives to Biasing the Reward Function . . . 175
5.11.5 Can SB–XCS Avoid Strong and Fit Overgenerals? . . . 176
5.12 Sequential Tasks . . . 176
5.12.1 The Need to Pass Values Back . . . 178
5.12.2 The Need for Discounting . . . 179
5.12.3 How Q-functions become Biased . . . 179
5.12.4 Examples . . . 180
5.12.5 Woods2 Revisited . . . 181
5.12.6 When Will the Value Function be Unbiased? . . . 188
5.13 What Tasks can we Solve with SB–XCS? . . . 190
5.14 Extensions . . . 191

5.14.1 Fitness Sharing 191
5.14.2 Other Factors Contributing to Strong Overgenerals . . 192
5.14.3 Qualitative and Quantitative Approaches 193
5.15 Summary 194

6 Classifier Systems and Q-Learning **197**
6.1 Classifier Systems and Q-Learning 198
6.1.1 Q-Learning in Classifier Systems 198
6.1.2 Is it Really Q-Learning? 199
6.1.3 XCS is a Proper Generalisation of Tabular Q-Learning 200
6.1.4 Summary 201
6.2 The GA-View and RL-View Revisited 201
6.2.1 How SB–XCS Determines Policies 202
6.2.2 How XCS Determines Policies 203
6.2.3 Three Approaches to Determining a Policy 204
6.2.4 The GA-View and the RL-View 204
6.2.5 Combining Evolution and Q-learning 205
6.3 XCS is Closer to Tabular Q-Learning than to SB–XCS 208
6.4 Summary 208

7 Conclusion **209**
7.1 The Capacities of Various Types of LCS 209
7.2 Contributions 210
7.3 The Take-home Message 214
7.4 Open Problems and Future Work 216
7.4.1 Fitness Sharing and Strength-based Fitness 216
7.4.2 Further Study of Accuracy-based Fitness 217
7.5 Concluding Remarks 217
7.5.1 The Moral of the Story: The Need for a Complex Systems Design Methodology 218
7.5.2 Classifier Systems and Reinforcement Learning 219
7.5.3 The Future 219

APPENDICES **220**

A Evaluation of Macroclassifiers **221**

B Example XCS Cycle **223**
B.1 The Performance System Algorithm 224
B.2 The Credit Assignment Algorithm 226

B.3 The Rule Discovery Algorithm 229

C Learning from Reinforcement **233**
C.1 Three Learning Paradigms 234
C.1.1 Supervised Learning 234
C.1.2 Reinforcement Learning 235
C.1.3 Unsupervised Learning 237
C.2 The Explore/Exploit Dilemma: a Feature of RL 237
C.3 Sequential and Non-sequential Tasks 238
C.3.1 Immediate Reward and Long-term Value 239
C.3.2 Sequential Decisions Imply RL 240
C.3.3 Episodic and Continuing Tasks 240
C.4 The Agent's Goal: Maximising Return 241
C.4.1 Return and Reward 241
C.4.2 Sequential Formulations of Return 242
C.5 Formalising RL Tasks 243
C.5.1 Environment 243
C.5.2 Learning Agent 244
C.5.3 Agent-environment Interaction 245
C.6 Summary 246

D Generalisation Problems **247**
D.1 Why Generalise? 248
D.1.1 The Curse of Dimensionality 248
D.1.2 The Need for Generalisation 249
D.2 Generalisation in RL 249
D.2.1 Generalising Over Policies and Value Functions 250
D.3 State Aggregation 250
D.4 State Space and Generalisation Space 250
D.5 Summary 251

E Value Estimation Algorithms **253**
E.1 The Value of State-actions 254
E.2 Non-sequential RL: Estimating Reward Functions 255
E.2.1 The Value of State-actions in Non-sequential Tasks . . 255
E.3 Estimating Expectations with Sample Averages 256
E.3.1 Incremental Updates 256
E.3.2 A General Form of Incremental Update 257
E.3.3 Setting *StepSize* in Incremental Updates 258
E.3.4 A Prediction Algorithm for Non-sequential RL 258
E.4 Sequential RL: Estimating Long-term Value Functions 259

E.4.1 Long-term Value Functions . . . 259
E.4.2 The Value of State-actions in Sequential Tasks . . . 261
E.4.3 The Value of a Policy . . . 262
E.4.4 Estimating Values with Monte Carlo Methods . . . 262
E.4.5 Estimating Values with Temporal Difference Methods . 263
E.4.6 Russell and Norvig's Maze: A Sequential RL Task . . . 266
E.4.7 Summary of Sequential RL . . . 267
E.5 State Aggregation . . . 267
E.5.1 Fixed and Adaptive Aggregation Schemes . . . 267
E.5.2 The Value of Aggregations I: Return . . . 269
E.5.3 The Value of Aggregations II: Predictive Utility . . . 269
E.6 Storing Value Estimates . . . 270
E.6.1 Storing Estimates of Aggregations . . . 270
E.6.2 Sparse Estimators, Models and Search . . . 271
E.6.3 Function Approximators . . . 271
E.7 Summary . . . 272

F Generalised Policy Iteration Algorithms **273**
F.1 Policy Improvement . . . 274
F.2 Optimal Policies . . . 275
F.3 Generalised Policy Iteration . . . 275
F.3.1 How Well must we Evaluate a Policy? . . . 276
F.3.2 Convergence Properties of GPI Control Algorithms . . 277
F.3.3 Initialising Value Functions . . . 277
F.3.4 What Characterises GPI Algorithms? . . . 277
F.4 State-value Functions . . . 278
F.5 Summary . . . 278

G Evolutionary Algorithms **281**
G.1 Evolution . . . 282
G.2 Elements of EAs . . . 284
G.2.1 A Generic EA . . . 284
G.2.2 Population-based Search . . . 285
G.2.3 Fitness Functions . . . 285
G.2.4 Probabilistic Selection of Parents . . . 286
G.2.5 Genetic Operators . . . 287
G.2.6 Replacement . . . 288
G.3 EAs as Search . . . 288
G.3.1 Local and Global Optima . . . 289
G.4 The Generalisation Problem . . . 290
G.5 Niching and Mating Restriction . . . 290

G.5.1 Fitness Sharing . . . 290
G.5.2 Crowding . . . 291
G.5.3 Mating Restriction . . . 291
G.6 RL with EAs . . . 292
G.6.1 Non-associative RL with an EA . . . 292
G.6.2 Associative RL with an EA . . . 294
G.6.3 Sequential RL with an EA . . . 296
G.7 Comparing GPI and EA Methods for RL . . . 296
G.7.1 Similarities between GPI and EA Methods . . . 297
G.8 Summary . . . 298

H The Origins of Sarsa **299**
H.1 Modified Connectionist Q-learning . . . 300
H.2 ZCS's Implicit Bucket Brigade . . . 300
H.3 Who Invented Sarsa? . . . 301

I Notation **303**

References **307**

Index **336**

Errata **343**

List of Figures

1.1 Some types of credit assignment schemes. 9
1.2 How strength and accuracy-based systems use rewards. 11

2.1 The XCS performance system algorithm. 36
2.2 Schematic illustration of XCS. 37
2.3 The XCS credit assignment algorithm. 41
2.4 Illustration of the effects of the relative accuracy update. . . . 44
2.5 The XCS rule discovery algorithm. 48
2.6 SB–XCS with roulette and tournament selection. 55
2.7 Standard XCS parameter settings for the 6 multiplexer. 59
2.8 XCS and SB–XCS on the 6 multiplexer. 61
2.9 The layout of Woods2. 62
2.10 Standard XCS parameter settings for Woods2. 63
2.11 Performance of XCS and tabular Q-learning on Woods2. . . . 64
2.12 Population size of XCS and tabular Q-learning on Woods2. . . 65
2.13 SB–XCS on Woods2. 66

3.1 A subset of the rules evolved by SB–XCS for the 6 multiplexer. 80
3.2 A subset of the rules evolved by XCS for the 6 multiplexer. . . 82
3.3 Three representations of the 6 multiplexer. 85
3.4 Complete and partial map representations of a state transition. 86
3.5 %[O] for XCS and SB–XCS on the 6 multiplexer. 88
3.6 %[O] for XCS on the hidden parity suite. 88
3.7 Performance of XCS on the 37-bit multiplexer. 89
3.8 Population size of XCS on the 37-bit multiplexer. 90
3.9 A reward function for a non-sequential task with 4 actions. . . 91
3.10 Complete and partial map representations of a sequential task. 94
3.11 A partial map forming incomplete paths. 96
3.12 A partial map with interfering local optima. 97
3.13 A gridworld with interfering negative rewards. 98

4.1 Truth table for the 3 multiplexer. 109

4.2 The on-set of the 3 multiplexer. 109
4.3 The 3 multiplexer represented with classifiers. 109
4.4 Truth table for the 3-bit constant 0 function. 112
4.5 Four ways of representing a constant 0 function with rules. . . 112
4.6 Two sets of rules which misrepresent the constant 0 function. . 113
4.7 Minimal DNF rule set for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$. 117
4.8 Minimal non-overlapping DNF rule set for a function. 117
4.9 Part of a reward function. 118
4.10 Two overlapping rules which are consistent with figure 4.9. . . 119
4.11 Two [O]s for the same function. 120
4.12 XCS represents both correct and incorrect actions. 121
4.13 $\mathcal{P}$ and %[O] at comparing deletion schemes. 125
4.14 %[PI] found by XCS for the 6 multiplexer. 129
4.15 Prime implicants for the 6 multiplexer. 129
4.16 Union of the minimal DNF representations of $\Sigma(3, 5, 6, 8, 15)$. . 131
4.17 %[m-DNF] found by XCS for $\Sigma(3, 5, 6, 8, 15)$. 132
4.18 Properties to which various metrics are sensitive. 132

5.1 Reward function for a simple task. 142
5.2 All possible classifiers for the simple task in figure 5.1. 142
5.3 Reward functions with 1, 2 or 3 actions. 152
5.4 A reward function as a matrix indexed by state and action. . . 153
5.5 A minimal strong overgeneral task for XCS. 155
5.6 Biases between actions do not result in strong overgenerals. . . 156
5.7 An unbiased reward function. 160
5.8 A minimal strong overgeneral task for SB–XCS. 161
5.9 SB–XCS and XCS on the unbiased function from figure 5.7. . 163
5.10 SB–XCS and XCS on the biased function from figure 5.8. . . . 163
5.11 A task in which XCS should suffer from fit overgenerals. . . . 167
5.12 The biased variance function for the task in figure 5.11. 167
5.13 Fitness of two rules from figure 5.11 at different variance biases. 169
5.14 A task which supports strong undergeneral rules. 172
5.15 A simple sequential task. 179
5.16 The basic Woods2 pattern. 182
5.17 Woods2 with steps to food from each blank cell. 182
5.18 The most numerous rules evolved by SB–XCS for Woods2. . . 187
5.19 The mountain car task. 188
5.20 Value function for the mountain car task. 189

6.1 How certain systems generalise and find policies. 205

A.1 XCS with and without macroclassifiers on the 6 multiplexer. . 222

E.1 Algorithm for approximating a reward function. 259
E.2 Backing up values between state-action pairs. 261
E.3 A maze and its optimal policy. 266

F.1 A generic form of GPI control algorithm. 277

G.1 A generic evolutionary algorithm. 285
G.2 A roulette wheel with fitness-proportional slot sizes. 286
G.3 Single point crossover. 287
G.4 A fitness landscape. 289

Chapter 1

Introduction

The appeal of Machine Learning (ML) lies in the idea of computers teaching themselves to solve problems, rather than relying on humans to specify their every move. Relying on humans to hard-wire behaviour is limiting because of the obvious difficulties of anticipating any number of situations in advance, particularly in a changing world. But further, we often simply do not know *how* it is we do what we do, and so cannot specify it to a computer. And, of course, there are any number of problems we have trouble solving ourselves in the first place. So there are limits to the complexity of the problems we can address by building hard-wired solutions.

We can sidestep these problems by, instead of building hard-wired machines, building learning machines which generate desired behaviour on their own. That is, by solving one problem (that of building a learning machine), we obtain a solution to many problems (those to which we apply our machine). This leaves us the task of specifying what the desired behaviour of the learning machine should be, but the assumption is that this should be easier than programming this behaviour. Of course, the problem of building learning machines which can handle complex tasks is itself a most difficult undertaking; it is generally easier to solve the problem at hand than to build a learning machine which is capable of solving it.

Nonetheless, ML promises rewards commensurate with its difficulties. It has already been suggested that learning machines promise to repay the effort invested in them, and to do things which humans cannot. But research into machine learning is not only a means of addressing engineering problems. It is also ultimately a part of the endeavour to understand intelligence and create machine intelligence, and so bears on philosophy, psychology, cognitive science, ethology and other fields. To sum up, ML involves very difficult problems, but holds tremendous promise, both in terms of extending the utility of computers and in the study of intelligence.

This thesis is concerned with Learning Classifier Systems (LCS), a form of machine learning system introduced in the 1970s by John Holland, which typically incorporates both Evolutionary Algorithms and Reinforcement Learning algorithms. Like ML in general, building successful LCS involves terribly difficult problems, but successful LCS hold tremendous promise. Put another way, if the goals of classifier systems research were met, many of the goals of machine learning would be met.

1.1 Two Example Machine Learning Tasks

This section introduces two example machine learning tasks and suggests a few ways in which we might approach them. In §1.5 we introduce a type of system which can be applied to these tasks.[1]

Example: Classifying Mushrooms

Consider the problem of enabling a computer to distinguish between poisonous and edible mushrooms. Human experts learn through experience how to distinguish the two, and it is difficult to find simple rules with good predictive power. Thus, it seems reasonable to apply machine learning to the problem of finding good, simple rules which distinguish the two cases. The standard approach is to train a machine learning system on examples of preclassified mushrooms, that is, a set of example mushrooms which have already been classified by a human expert. For example, we might use a database of descriptions of 100 poisonous and 100 edible mushrooms. Rather than present the computer with a raw image of a mushroom, we can extract certain information, e.g., concerning colour, size, shape and so on. The ML system can then teach itself how to classify the mushrooms, using the human expert's classifications for reference. Once the system has achieved good performance on this training set, we can test it on a set of previously unseen mushrooms (again, preclassified by a human expert) to see how well it generalises from the examples it has seen to new cases. If it performs well on this test, we may be willing to trust its judgement concerning mushrooms which have not been classified by a human expert. In other words, the ML system learns to duplicate a human expert's skill, and to generalise from its limited experience. One use for such a system is to evaluate mushrooms without the need to consult human experts, who may be few and far between. Another use is for the human expert to consider the decision-making rules found by

[1] In this work the symbol § will be used in denoting section numbers, e.g., section §1.2.1. Also, references to numbered equations will be enclosed in parentheses, e.g., equation (2.1).

the system, which may differ from those used by humans and which may offer some insight into the problem of mushroom classification.

A great variety of ML systems could conceivably be applied to this problem, but let us consider a system which, in teaching itself how to classify mushrooms, explicitly entertains many competing hypotheses and tests them against the available data. If the data supports a hypothesis, this suggests it should be retained (and perhaps used as the basis for new hypotheses). If the data does not support a hypothesis, this suggests it should be discarded. This approach seems fairly obvious and fundamental. (We might want to bias the system to be more likely to misclassify non-poisonous mushrooms as poisonous than the other way around, but we won't go into such details here.)

One issue in developing such a system is how to represent hypotheses. An obvious and fundamental approach is to use *condition-action rules*, such as these:

IF	small **AND** green	**THEN**	edible
IF	(small **AND** green) **OR** has-spots	**THEN**	poisonous
IF	small **AND** pink	**THEN**	poisonous
IF	small **AND** pink	**THEN**	edible

Example: Simulating a Frog

As another example, consider this problem: given sensors and effectors, provide a self-adapting control system for a simulated frog which learns only by trial and error, that is, from rewards and punishments from its simulated environment. The frog's objectives are to maximise the number of flies it eats, minimise its energy use, and to avoid being eaten itself.

This problem differs significantly from the mushroom classification task in that the system learns only from rewards and punishments, not from preclassified examples. Another difference is that in this case the learner *interacts* with a problem environment; the frog's actions influence the environment. As with the mushroom classification task, it may be difficult for humans to produce a system which effectively maximises the rewards and minimises the punishments the frog receives. To address the problem, we might again reasonably attempt to develop a machine learning system which entertains and evaluates multiple competing hypotheses, and which represents them with condition-action rules, for example:

IF	small **AND** dark **AND** buzzes	**THEN**	eat-it
IF	large **AND** white **AND** has-beak	**THEN**	hide

Classifier Systems

The broad approach suggested for both problems in this section is that taken by classifier systems, so-called because they learn condition-action rules called *classifiers*.

1.2 Types of Task

1.2.1 Supervised and Reinforcement Learning

The two examples we've just seen are meant to illustrate two kinds of task to which we can apply LCS. The mushroom classification task is meant to illustrate a *pattern classification* or *data mining* task, while the frog controller is an on-line *control* task.

Each task is suitable for a different learning paradigm, that is, a different model of interaction between the learning agent and the task. In the mushroom classification task we have a training set of pre-classified exemplars, making Supervised Learning (SL) feasible. For the frog controller task, however, we have only certain conditions which are good (eating flies) or bad (being eaten), and so the task must be modelled as a Reinforcement Learning (RL) task. Learning paradigms are covered in appendix C. Classifier systems are suitable for both paradigms, but this work focuses on RL.

1.2.2 Sequential and Non-Sequential Decision Tasks

The two tasks illustrate a second important distinction to be made in the space of tasks. The mushroom classification task is non-sequential, that is, the action taken by the learner on one time step does not influence what inputs it receives in the future. (The classification it gives at time t has no influence on which mushroom it sees at time $t + 1$.) In contrast, the frog simulation task is sequential; an action taken at time t may well influence what inputs are received in the future. For example, at time t the frog may choose between hopping into a pond and hopping into some tall grass, each of which will lead it through a different sequence of states in the future. Sequential tasks are in general more difficult because they require the learner to consider the long-term consequences of its actions. This work studies the application of classifier systems to both types of task.

1.3 Two Challenges for Classifier Systems

Our suggestion that the system entertain competing hypotheses in the form of condition-action rules is a rather vague specification. Among the major questions it leaves open are:

How do we choose between rules? The reader may have noticed that some of the rules in the mushroom classification example contradict each other. When this happens some kind of *conflict resolution* must be performed. Good conflict resolution relies on our ability to discriminate between good and bad rules, that is, to assess the utility of rules. Ideally, the means by which we do so should be independent of the domain in which the system operates; it is desirable to use the same methods for evaluating rules and resolving conflicts between them in all domains, rather than having to tailor them to each domain.

Where do rules come from? A classifier system needs some automated means of generating rules, and employs a *generate-and-test* approach. To optimise this process, rather than generate random rules, we would like to base new rules on existing high-quality rules. Thus we have two processes which must choose between rules; conflict resolution, and the selection of rules on which to base new rules. The generation of rules must be independent of human intervention, and, ideally, independent of the domain in which the system operates; to minimise human input it is desirable to use the same rule-discovery techniques in searching for good rules in all domains, rather than have to tailor them to each problem.

These two questions are major issues for LCS research. They have been the subject of much research over the years, and will be for many years to come. The answers to these questions can define vastly different systems, with vastly different characteristics.

Because this work is concerned with RL classifier systems, and because it will borrow much from existing work on RL, the two challenges above are recast below in terms of reinforcement learning. From this point on, we will assume tasks and learning agents which fall under the RL heading, unless otherwise noted.

1.3.1 Problem 1: Learning a Policy from Reinforcement

The challenge of choosing between conflicting rules is really the problem of learning a *policy* – a mapping from states to actions – which tells the learning agent how to behave. Consequently, this is sometimes known as the *control* problem. The goal of a learning agent is to find an *optimal policy* (§F.2), that is, one which, over the long run, maximises the rewards and minimises the punishments it receives.

One aspect of searching for good policies is the need to evaluate policies, or parts of policies, e.g., to evaluate how good it is to take an action a in a state s. This is also known as the *prediction* problem, since it can be seen as the task of predicting the outcome of behaviour. Methods of addressing the prediction problem are covered in appendix E, and the interaction of control and prediction is discussed in §F.3 and §G.7.1.

1.3.2 Problem 2: Generalisation

To scale up to large tasks learning methods must exploit regularities in the task, in their representation of it and their operation. Consequently, it is not enough to search for a good policy; to be effective, a learning system must exploit regularities in doing so. One way to exploit regularities is to generalise over (alias) aspects of the task. Appendix D discusses this subject in more depth.

1.4 Solution Methods

This work is concerned with two basic approaches to learning policies from reinforcement, and are introduced in the following sections. Then, in §1.5, classifier systems, which are hybrids of the two basic methods, are introduced.

1.4.1 Method 1: Reinforcement Learning Algorithms

The subfield of AI called Reinforcement Learning is concerned with a certain subset of the algorithms which can be used with the RL paradigm mentioned in §1.2.1. It is important to understand the distinction between the RL learning paradigm – a class of learning tasks – and so-called *RL algorithms*, one class of algorithms suited for such tasks. The distinction matters because RL algorithms are not the only methods applicable to these tasks; for example, the evolutionary algorithms of the following section are equally applicable.

RL algorithms learn policies by learning value functions (§E.4.1), which are estimates of the (long-term) utility of components of the policy. In short, RL algorithms iteratively evaluate the utility of the current policy (learn a value function for it) and then derive a better policy based on the value function (§F.3). RL algorithms are introduced in appendices E and F.

RL algorithms do not address the question of generalisation directly, but often make use of function approximators to provide generalisation. For example, a value function is often stored (approximated) using a neural network, rather than a look-up table. We will make frequent references to Q-learning ([286, 265], §E.4.5), the best-known RL algorithm.

1.4.2 Method 2: Evolutionary Algorithms

The subfield of AI called Evolutionary Computation (EC), introduced in appendix G, is applicable to both the problem of learning a policy and the problem of generalisation. EC comprises many forms of Evolutionary Algorithms (EAs), all of which, inspired by evolution in nature, employ simulated evolution of populations of candidate solutions. In short, this involves iteratively evaluating the current generation of solutions and producing the next generation by means of probabilistic transformations and selective pressure towards fitter members. EAs can be thought of as stochastic population-based search methods.

1.5 Learning Classifier Systems

Classifier systems originated and have been developed primarily within the field of Genetic Algorithms (see §G.2, and, e.g., [119, 94]), one of the stochastic search methods studied in the field of Evolutionary Computation. Classifier systems are traditionally hybrid Evolutionary/Reinforcement Learning systems, although some are Supervised Learning systems (e.g., [29]), or use non-evolutionary means of generating rules (e.g., [259]). Although such systems are significant, this work will focus on systems which are hybrids of EC and RL algorithms, and which learn from rewards.

1.5.1 The Tripartite LCS Structure

To return to the two challenges of §1.3, the questions of how rules are evaluated and generated are addressed, respectively, by a classifier system's *credit assignment system* and *rule discovery system.*

Rule Discovery System

The rule discovery system almost always consists primarily of a Genetic Algorithm, which may be supplemented by other mechanisms (see §2.3.6).

Credit Assignment System

A great many credit assignment algorithms exist, and different types will be briefly outlined in §1.5.3. Whatever method is used, the role of the credit assignment system is to observe the use of the rules as the system interacts with its problem environment, and the feedback the system receives from the environment, and update a numerical estimate of each rule's utility called its *strength*. (In the classifier systems for RL with which we are interested, feedback consists of rewards, that is, numbers.)

Production System

In addition to the two systems mentioned above, a classifier systems needs machinery which applies the right rules at the right time, that is, it is responsible for finding the rules which apply, or *match*, the current state of the problem. This machinery constitutes the third major component of a classifier system, the *production system*. We will see in more detail how these systems fit together to produce a working classifier systems in chapter 2, and will reconsider how they fit together in chapters 3 and 6.

1.5.2 LCS = Policy Learning + Generalisation

It is worth emphasising that classifier systems intrinsically address both the problem of learning policies and the problem of generalisation. The inherent capacity to generalise is one of the main features which distinguish LCS from other approaches to RL, and a motivation for interest in LCS. Of course other approaches to generalisation in RL exist; the most common is to use a neural network to approximate the value function of, e.g., a tabular Q-learner. In this case, generalisation is achieved by combining tabular Q-learning with a function approximator. The two processes are distinct. In a classifier system, however, we cannot separate generalisation and policy learning; a classifier system is a combination of both, and if either is missing we do not have a classifier system.

Although the potential to generalise is one reason for interest in LCS, it also makes the problems faced by classifier systems and study of them more difficult (§7.5.2).

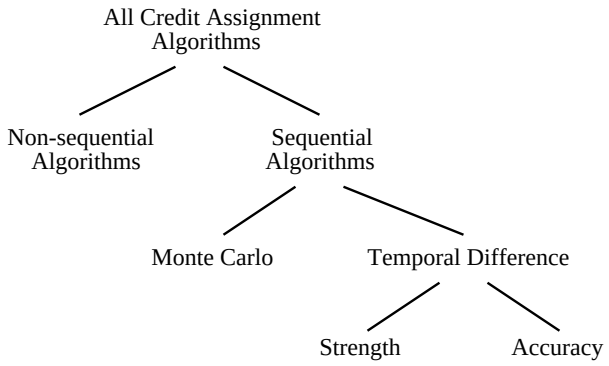


Figure 1.1: Some types of credit assignment schemes.

1.5.3 Credit Assignment in Classifier Systems

In sequential tasks, credit assignment is normally performed by some form of Temporal Difference algorithm (see [265], §E.4.5), from the field of Reinforcement Learning. Various versions of the bucket brigade algorithm (see [126, 94]) have most often been used, although Q-learning-like updates have become popular in recent years (see [297, 298], §E.4.5).

However, not all LCS use Temporal Difference algorithms; some have used what are referred to as epochal schemes in the LCS literature and Monte Carlo updates (§E.4.4) in the RL literature (following the usage of Sutton and Barto [265]). Such LCS include CS-1 [127] and RUDI [100]. Monte Carlo updates have two disadvantages compared to Temporal Difference updates. First, it is difficult to apply Monte Carlo updates in tasks which do not divide naturally into episodes, that is, which periodically terminate and restart. Such tasks are termed episodic, as opposed to continuing tasks which do not terminate. (See §C.3.3.) Second, Temporal Difference methods take advantage of the sequential structure of a task in a way which Monte Carlo methods do not, and so often outperform them. On the other hand, Monte Carlo methods have some advantage on tasks which are not well-modelled as Markov processes (see §C.5). There is, however, relatively little known about how the two compare [265].

Some of the different types of credit assignment schemes are illustrated in figure 1.1, which includes strength-based and accuracy-based forms of Temporal Difference schemes, to be introduced in the following section. This work is primarily concerned with strength and accuracy-based schemes, and has little to say about Monte Carlo updates.

1.5.4 Strength and Accuracy-based Classifier Systems

It is difficult to generate new rules efficiently without being able to determine the quality of existing rules, meaning rule evaluation is an issue in rule discovery. For example, the typical approach of using a genetic algorithm requires an estimate of the value of each rule called its *fitness*.

Since rules are evaluated by the credit assignment system for conflict resolution, an obvious approach is to use the same evaluation in rule discovery, and this is just the approach used by *strength-based* classifier systems. That is, these systems use a rule's strength in both action selection and rule discovery.

In contrast, in Wilson's XCS classifier system [298], a rule's strength is used only in action selection. For reproduction, the utility (fitness) of a rule is a different value, although it is ultimately derived from the strength of a rule. More specifically, in XCS, a rule's fitness is a function of the accuracy with which it predicts the reward it will receive, which is why XCS is called an *accuracy-based* classifier system. The difference is illustrated in figure 1.2, in which arrows indicate the flow of information from its source (rewards) to its use in the processes of action selection and rule discovery. We will see that this simple difference has profound implications for the system. We will take XCS as a representative of a class of related possible accuracy-based systems, and, where possible, consider the class of such systems and not just XCS.

A major aim of this work is to explain the differences between the older strength-based LCS and the newer XCS, and the significance of these differences. We will make extensive comparisons between strength and accuracy-based systems, both theoretically and empirically. In order to do so, a system called SB–XCS (Strength-based XCS) will be introduced in chapter 2.

1.6 About the Thesis

This work arises in response to a number of questions:

- How do strength and accuracy compare?
- Is a classifier system GA-based or RL-based?
- How do classifier systems relate to Q-learning?
- How do different types of classifier system relate to each other?

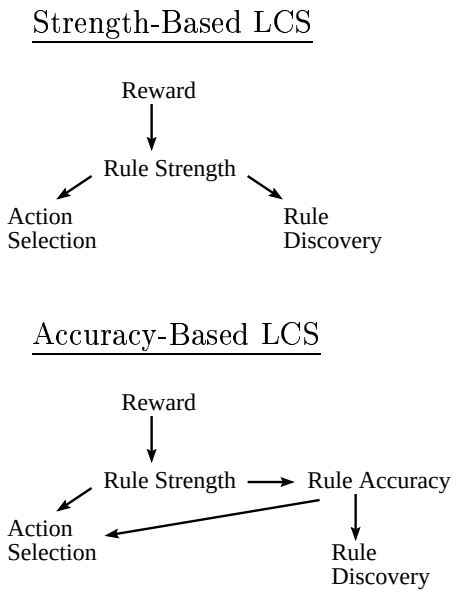


Figure 1.2: How strength and accuracy-based systems use rewards to weight rules in action selection and rule discovery.

The goals of this work, then, are to compare two broad types of classifier system: strength-based and accuracy-based, and to improve our understanding of how different learning systems relate to each other, that is, how they 'fit together'. The following section explains the need for comparison between strength and accuracy, while §1.6.2 explains the confusion over the question of whether LCS are GA-based or RL-based. Finally, §1.6.3 casts this work as an exploration of design space, and explains the necessity of this kind of work.

1.6.1 Why Compare Strength and Accuracy?

The creator of XCS, Stewart W. Wilson, motivates the switch to accuracy-based fitness in [298], but this is the only discussion in the literature and a comparison of the two approaches has been lacking. A better understanding of the two approaches is important for a number of reasons.

- The question of how to calculate rule fitness has been the subject of much study over the years. Fitness definition is a fundamental issue since a classifier system will not adapt well if fitness is not defined in a suitable way. Thus, understanding of how strength and accuracy compare is important for the development of the field of classifier systems.

- There is evidence that traditional strength-based fitness is unsuitable for some sequential tasks [298, 55] (but see [47]).
- There is a growing body of evidence that accuracy-based fitness *is* suitable for sequential tasks (e.g., [298, 165, 11, 12, 13, 14]).
- XCS has generated considerable interest and has become a major focus of classifier systems research [149].
- It has been suggested that XCS's complete mapping from inputs and actions to reward predictions is advantageous [298].
- Later in this work it will be suggested that XCS also has important advantages in handling the explore/exploit dilemma (§C.2) and value propagation (§3.7.2) in reinforcement learning, and what we have called strong and fit overgeneral rules (see chapter 5).
- It has been suggested that accuracy-based fitness shows better generalisation than strength (and consequently requires smaller population sizes) [298, 146, 304].
- However, it has also been suggested that accuracy may require *larger* population sizes than strength (see chapter 3).

Most of these points suggest an advantage of accuracy-based fitness, and a comparison is needed to ascertain whether this important new direction in LCS research lives up to these expectations.

1.6.2 Are LCS Based on Evolutionary Computation or Reinforcement Learning?

What a classifier system is seems contentious, to the extent that discussion of this issue dominated the First International Workshop on Learning Classifier Systems (IWLCS-92). As Robert Smith, paraphrasing Lashon Booker, reported:

> "The LCS is usually described as a *method*: a set of algorithmic details that define a way to solve a class of problems. However, in many ways the LCS is more of an *approach*: a set of conceptual details that define a certain direction for developing methods. Therefore, the defining issues for the LCS are not necessarily algorithmic, but conceptual. The central problem addressed by the workshop's discussions was to clarify these defining, conceptual issues." [242] p. 2.

That conceptual issues remain a concern for classifier systems is indicated by the inclusion of a series of 11 short essays under the title "What is a Learning Classifier System?" in a recent publication [125]. Although I was unaware of this diversity of opinion when I first began working with them in 1996, I soon became concerned that I was unsure of the difference between a genetic algorithm and a classifier system. I had seen the LCS described as a combination of a production system, rule discovery system and credit assignment system. I reasoned that since the rule discovery system typically *is* a genetic algorithm, the LCS must be something more, since it has two additional components. However, I decided the credit assignment system was just what we call the fitness function of a genetic algorithm. Granted, credit assignment in an LCS was more complex than the examples of function optimisation with a GA which I had seen, but it was still a kind of fitness function. This left the production system as the real difference between the LCS and GA. But since the production system is conceptually straightforward – its task is simply to apply the rules when appropriate – a classifier system seemed to be just a way of applying a GA to certain kinds of problems. Certainly we need to wrap the GA up with a little machinery (the production system and a special kind of fitness function) to interface it with the problem, and perhaps the GA needs a little help in the form of operators like covering (see, e.g., [120, 31, 298], §2.3.6), but the LCS seemed to be essentially a GA.[2]

This is a view which I still think is consistent with Holland's intentions, and those of many others. Classifier systems have, after all, been described as *Genetics-Based* Machine Learning (GBML) systems [94].

Less-genetic Classifier Systems

The view of an LCS as essentially a GA is somewhat extreme, and other less extreme views exist. In addition to the GA, a classifier system may contain rule discovery mechanisms such as covering, triggered chaining [213], bridging [122, 210], and corporate linkage [315, 273]. In such systems the GA is just one component of the rule discovery system, although perhaps an important one. However, some LCS emphasise the use of non-genetic operators more heavily than others, and in some cases the GA is even considered a 'background' operator [14].

[2] A more detailed account of the differences between the two would be desirable, but must be deferred to another work.

Non-genetic Classifier Systems

That an LCS is essentially a GA is flatly contradicted by the considerable recent work on LCS which use alternative rule discovery systems. In hindsight, there seems no justification for insisting on the use of GAs as opposed to other evolutionary algorithms. Alternatives were suggested some time ago [297, 277, 278, 243] and some recent work has indeed used Genetic Programming rather than Genetic Algorithms [164, 4]. What's more, a significant amount of recent work has been on systems which contain no evolutionary algorithms [254, 256, 255, 257, 258, 48, 49, 259, 260, 261, 51]. If we accept such systems as classifier systems (as is the norm, e.g., work on such systems has appeared at the International Workshop on Learning Classifier Systems – IWLCS), we are dealing with a much broader concept than that of a GA and some wrapping. Unfortunately, discussion of this trend lies outside the scope of this thesis.

Linking Classifier Systems and Mainstream RL

A second trend which breaks from the view of an LCS as a GA-based system is that which seeks to link LCS and mainstream reinforcement learning. RL has made great strides since the introduction of LCS, and it is clear that (most) LCS are RL systems, that they address many of the same issues addressed by other RL systems, and that there is much to be gained from integrating LCS with mainstream RL. The need to bridge LCS and mainstream RL appeared to be the consensus during the discussion at IWLCS-99.

The GA-view and RL-view

This leaves us with two contradictory views of what a classifier system really is, what we might call the *GA-view* – that the LCS is essentially the application of a GA to a problem – and the *RL-view*; that the LCS is a kind of RL system, i.e., a Q-learning-like system in which the GA is (or may be) a component, but in which many of the interesting issues are to do with credit assignment. The two views place different emphasis on different subsystems: according to the GA-view, the GA, and issues relating to it, are of primary importance, while the RL-view places greater importance on credit assignment. The existence of two alternative views begs an important question: does an LCS solve problems using evolutionary means, or does it solve them in the way non-evolutionary RL systems do?

One aim of this work is to recognise and publicise the existence of these alternative views, since they seem under-recognised, particularly in the literature. Another aim is to clarify these views, and to justify the RL-view

of (some) LCS. Significantly, the RL-view focuses on XCS, which, it will be argued, differs fundamentally from Holland's LCS, to the extent that it more closely resembles mainstream RL systems, such as tabular or neural network-based Q-learners.[3]

In order to make this RL-view clear, and accessible to the many in the LCS community who are not well versed in RL, this work goes to considerable lengths introducing RL in appendices C, E and F. Those familiar with RL may wish to skip over these sections, but they are highly recommended to those unfamiliar with the subject. Without a good, basic understanding of RL, the distinction between the GA and RL views of LCS is likely to be unclear. What's more, the future of LCS research (for sequential tasks) seems heavily grounded in this view; the most important issues to be addressed in LCS research are those which are and will be addressed in RL. This is not to marginalise evolutionary approaches to RL, but to say that they too will benefit from understanding of non-evolutionary approaches. The text by Sutton and Barto [265] offers a far more complete introduction to RL, and should be required reading for anyone wishing to apply LCS to RL problems.

1.6.3 Moving in Design Space

This section briefly discusses ways in which progress can be made in a scientific field, and the contribution this thesis makes to the study of LCS.

The Design-based Approach

The literature contains a great number of designs for classifier systems, and will surely see new ones introduced in the future. One can think of these systems as points in the space of possible designs – design space – and the space of requirements for our designs – niche space. These are concepts from the *design-based* approach, which involves taking the role of an engineer who is trying to design a system that meets certain requirements, and is inspired by software engineering and conceptual analysis in philosophy [236, 237, 238, 239]. It involves analysing alternative sets of requirements, designs and implementations in an attempt to establish the nature of their relationships. It allows a high level functional comparison of systems, both natural and artificial, despite differences in origin or implementation. This comparison seeks to identify which aspects of a system are essential for given functions and which are not. Note that this approach does not require a full

[3]Despite this, XCS originated and has been studied exclusively within the LCS community, and is by far most strongly integrated with the LCS literature. Much better integration with mainstream RL awaits.

understanding of the requirements or the available tools at the outset, nor does it assume that there is a single correct design to be found.

The succession of designs for classifier systems can be seen as a search in design space, a search for optimised designs. How does search in design space progress? Ideally, one would make as big a jump as necessary to reach, all at once, an optimum design for one's requirements. But even if the optimum design was found immediately, how would one know one had found the optimum? Without some formal method which can be used to prove the optimality of a design, one can only continue to search for better designs (i.e., consider other designs). (In the study of classifier systems (or Artificial Intelligence more generally), the requirements of our systems are not fully understood, and so the set of requirements an optimal design would meet is not defined. Chapter 4, however, does at least address the issue of the requirements of a classifier system.)

An Analogy with Evolutionary Computation

Search in design space can be thought of in terms of the familiar idea of an evolutionary algorithm moving across a fitness landscape (§G.3). Movement consists of iteratively generating and evaluating new designs. Because a field consists of a population of researchers, many of them generating and evaluating different designs, a community of researchers conducts a parallel, interacting search – effectively an evolutionary algorithm.

Interleaving Analysis and Invention

How well must one evaluate a design before one can move on? There is a form of explore/exploit tradeoff (§C.2) at work; the better one understands the current area of design space, the better one can direct oneself to a preferable region. The cost of gaining deeper understanding is time and effort. As with other explore/exploit problems, the optimal tradeoff between evaluating designs and generating new ones is difficult to achieve.

In Evolutionary Computation one typically only needs to know the approximate fitness of an individual to plausibly generate improved ones. Similarly, in searching design space, one may only need to evaluate a design incompletely before moving on to consider others. For example, if a design fails to meet a vital requirement, it can be rejected without exploring how it meets other requirements.

However, in cases where designs are not evaluated deeply, one is likely to make smaller improvements on them than one might otherwise. That is, in making an informed jump in design space, the size of jump which

can be made is limited by how well one understands that space; limited understanding restricts the points to which one can jump. Small jumps may consign us to hill-climbing; approaching a locally optimal design. Bigger jumps can escape local optima, but making bigger jumps – in the direction intended – requires greater understanding of the surrounding space.

XCS as a Jump in Design Space

XCS is a big jump in the design space of classifier systems; even the shift to accuracy-based fitness by itself constitutes a big jump. Though the differences between the specification of XCS and its strength-based twin SB–XCS are quite minor, the resulting systems operate in a very different way. (But then changing a few genes in an animal can have a huge effect.) In fact, it is argued later (chapter 6) that XCS has more in common with tabular Q-learners than with older classifier systems, or even with its twin SB–XCS.

The Contribution of this Thesis

So far, XCS has been shown to be worthy of further consideration. However, it is not yet well understood, and neither are earlier classifier systems. This work addresses the difficult problem of showing that strength-based LCS are indeed unsuitable for many tasks, and why this is so. We will see considerable evidence for this argument, particularly in §2.5 and chapter 5.

I have pursued the study of strength and accuracy presented here because I believed it was the best way to make progress with classifier systems. Rather than extend XCS, or create a new system, I wanted to understand – in detail – the differences between XCS and other LCS, and between LCS and other reinforcement learning systems.

1.7 Structure of the Thesis

- Chapter 2 begins with a discussion of the range of classifier systems in the literature, and the difficulties of classifying them given our limited understanding of how they compare. It then reviews representations used by classifier systems, in particular the standard ternary language. Finally, it introduces two systems, XCS and SB–XCS, in detail and presents an initial comparison of them.

- Chapter 3 studies the differences between XCS and SB–XCS in greater detail, and considers rationales for why they should adapt to a task. It

then examines representational differences between the two, and evaluates the alternative representations.

- Chapter 4 considers what features the representation of a solution should have, and examines metrics for them.

- Chapter 5 analyses what types of rules are possible and under what conditions they can be expected to occur. Competition between selfish rules is shown to produce various forms of rules which are detrimental to the performance of the system as a whole, namely overgeneral, strong overgeneral and fit overgeneral rules. It is shown that XCS does not produce such rules in the circumstances in which SB–XCS does. However, it is also shown that there are circumstances in which XCS will produce such rules. The prospects for adaptation of the two systems to sequential and non-sequential tasks are evaluated.

- Chapter 6 considers the relationship between classifier systems and Q-learning, and revisits the issue of the GA-view and RL-view from §1.6.2.

- Chapter 7, the conclusion, outlines a model of the capacities of various types of classifier systems, lists the contributions of the thesis, discusses open problems and future work, and ends with some remarks on the history and state of the field.

- Appendix A examines XCS with and without the use of macroclassifiers (§2.3.2) and finds in favour of their use.

- Appendix B walks through a step of the XCS algorithm in order to illustrate the process.

- Appendices C and D introduce the Reinforcement Learning and Generalisation problems respectively.

- Appendix E introduces methods for solving the prediction problem, while F introduces RL methods which are based on policy improvement and appendix G introduces Evolutionary Computation.

- Appendix H points out that Wilson developed a version of the Sarsa update (§E.4.5) contemporaneously with Rummery and Niranjan.

- Finally, appendix I lists the notation used in this work.

How to Read this Thesis

Readers wishing to cover any of the introductions to Reinforcement Learning problems (appendix C), Generalisation problems (appendix D), prediction methods (appendix E), RL algorithms (appendix F) and Evolutionary Computation (appendix G) may wish to do so before proceeding to chapter 2. The material in these appendices should help clarify chapters 2–7, although familiarity with this material should not be strictly necessary.

Chapter 2

Learning Classifier Systems

Certainly the notion of a classifier system is appealing — how could roving bands of mating, reproducing, fighting, and dying rules fail to captivate our collective imagination?

Stewart W. Wilson and David E. Goldberg [315] p. 254.

Classifier systems are a quagmire – a glorious, wondrous, and inviting quagmire, but a quagmire nonetheless.

David E. Goldberg, quoted in [96] p. 1.

Classifier systems are stuck.

Stewart W. Wilson circa 1996, quoted by John H. Holmes [131].

Chapter Outline

The general idea of a classifier system and its tripartite structure have already been introduced in §1.5. We begin this chapter by briefly outlining various types of classifier systems and reviewing the representation of rules in them. Following this we proceed to describe two classifier systems, accuracy-based XCS and its strength-based twin SB–XCS, in detail sufficient to allow implementation. We make a brief empirical comparison of them in §2.5, and will conduct more extensive comparisons and experiments in later chapters. Throughout the thesis these two systems will be used as references in our reasoning about strength and accuracy-based systems.

Classifier systems were invented by John Holland in the 1970s. The first published work on LCS appeared in 1976 [120]; the first report of an implemented system, Cognitive System 1 (CS-1), appeared in 1978 [127]. Classifier systems developed from Holland's earlier work on adaptive systems [116, 117], schemata processors [118], the broadcast language [119], and genetic algorithms [119]. See [94] chapter 7 for a short LCS prehistory.

In 1996 Cribbs and Smith [61] suggested a classifier systems renaissance was underway, with new representations, credit assignment algorithms and rule discovery methods being considered. The five years since then have seen further developments in these and other areas, prompting Goldberg [125] to refer to a classifier systems renaissance again in 2000. Analysis of LCS publication trends [149], the emergence of an annual LCS workshop, and anecdotal evidence all support the suggestion of a renaissance.

Classifier systems are suitable for both reinforcement and supervised learning tasks (see §C.1). Most LCS research has been concerned with reinforcement learning tasks (e.g., [127, 30, 293, 298]), although some of the most impressive results have been in supervised learning (or essentially supervised learning) tasks (e.g., [28, 133, 223]). In addition, LCS have been used for machine discovery [245, 246] and to automate the building of simulations of complex systems [232]. Application areas include the study of multi-agent systems (e.g., [101, 103, 102, 191, 42, 87, 230, 231, 54, 43, 216, 18, 44, 83, 175, 113, 115, 114, 136]), economics (e.g., [176, 169, 240, 284,

291, 185, 86, 194, 82, 43, 44, 18, 168, 170, 226, 85]), autonomous robotics (e.g., [79, 77, 68, 56, 104, 80, 57, 25, 84, 75, 105, 106, 78, 58, 193, 227, 73, 46, 45, 71, 61, 76, 70, 280, 27, 195, 260, 109, 143]), and cognitive modelling (e.g., [215, 229, 110]). Recently there has been particular interest in applying classifier systems to data mining (e.g., [133, 223]).

For introductory material on LCS see [123, 39, 94, 40, 180, 8, 186, 112], of which Goldberg's 1989 book [94] provides the most gentle introduction. Online resources on LCS include the LCS Web [15], the LCS Mailing List [132], the LCS Bibliography [152], and the Electronic Archive of LCS publications [153].

Broad surveys of LCS research are relatively few. The 1989 paper by Wilson and Goldberg [315] is now somewhat dated, but still valuable. Following on from this, Lanzi and Riolo survey LCS research from 1989-1999 in [166]. Midway between these two lies Fogarty, Carse and Bull's 1994 survey [88]. Chapter 2 of [14] provides historical information on LCS research. Surveys of particular subjects within the LCS field include Fogarty, Ireson and Bull's 1995 survey of industrial and commercial LCS applications [89], Wilson's 1999 survey of the state of XCS research [312], Barry's surveys of work on hierarchical LCS [10, 14] and Bonarini's survey of work on Fuzzy LCS [26].

2.1 Types of Classifier Systems

This section briefly reviews two dimensions along which we can divide the space of classifier systems, and discusses some of the difficulties in doing so. For further details on the range of systems in the literature the reader is referred to chapter 2 of [14], which reviews a number of influential classifier systems.

2.1.1 Michigan and Pittsburgh LCS

In *Pittsburgh*-style (Pitt) classifier systems (e.g., LS-1 [250, 251, 252]) the GA operates on chromosomes which are complete solutions (entire sets of rules), whereas in the more common *Michigan*-style LCS chromosomes are partial solutions (individual rules) (see, e.g., [94]). Some hybrid Pittsburgh-Michigan systems have been used (e.g., [100, 92, 137]).

Although this difference may seem minor, Michigan and Pittsburgh systems are really quite different approaches to learning (as §6.2 will attempt to explain). Wilcox [291] suggests that although Pitt LCS are capable of solving more complex problems than (strength-based) Michigan LCS, they

require much greater computational resources since each chromosome is an entire set of rules.

This work is concerned almost exclusively with Michigan-style systems, but the analysis of strength-based LCS in chapter 5 reveals fundamental problems which are consistent with the limitations suggested by Wilcox.

One claim of this work is that the XCS system does not suffer the limitations of strength-based LCS, and yet enjoys the computational advantage of other Michigan LCS.

2.1.2 XCS and Traditional LCS?

The preceding section distinguished between Michigan and Pittsburgh-style LCS, while this section classifies LCS in another way; according to their credit assignment scheme. Types of credit assignment schemes for Reinforcement Learning systems were introduced in §1.5.3 and §1.5.4, but here we see that the distinction between strength and accuracy-based systems outlined there is not so straightforward.

Wilson introduced a distinction between his accuracy-based XCS system and "traditional strength-based systems" [298], but it is unclear, however, precisely how to make this distinction. Effectively it has referred to XCS on one hand and an ill-defined class of earlier systems on the other.

One reason this partitioning of the space of classifier systems is problematic is that XCS is only one of many conceivable future accuracy-based systems. More troublingly, Frey and Slate's system is accuracy-based [91]. Should we include it in references to "accuracy-based systems"? Regrettably, as the author has not analysed it he cannot make any statements concerning its relationship to other systems.

Additionally, some primarily strength-based systems have incorporated accuracy. GOFER-1 [33] included an accuracy component in its fitness calculation, Smith and Cribbs's system [248] had variance-based fitness, and SAMUEL found a role for accuracy in deletion and mutation [101]. SAMUEL also incorporated accuracy into action selection, as did Goldberg's variance-sensitive bidding method [95]. Whether these systems should be classified as strength-based, accuracy-based, or as a third category of hybrid systems is unclear.

A further problem lies in classifying Booker's endogenous-fitness LCS [36, 35, 37] whose fitness calculations may defy the strength/accuracy dichotomy, while ACS [259] with its rule-generation heuristics certainly does.

SB–XCS: A New Strength-based LCS

Given the broad range of strength-based systems, and the imprecision of the definition of such systems, it was decided to introduce a strength-based system called SB–XCS (§2.4) for comparison with XCS. Comparing XCS to a specific strength-based system, rather than attempting to reason about the capacities of a range of strength-based systems, seems the course least likely to lead to confusion and overgeneralisation regarding the capacities of strength LCS.

2.2 Representing Rules

This section reviews some of the ways in which rules, and especially rule conditions, have been represented in classifier systems. In chapter 4 we will review the representation of problem solutions using *sets* of rules.

2.2.1 The Standard Ternary Language

A number of representations for the condition-action rules called *classifiers* have been used with LCS, in particular a number of variations based on binary and ternary strings. Using what we will call the *standard ternary LCS language* each rule has a single condition and a single action. Conditions are fixed length strings from $\{0, 1, \#\}$, while rule actions and environmental inputs are fixed length strings from $\{0, 1\}$. This is the representation we will use with XCS and SB–XCS, the two LCS used in this work. It has also been used with many other LCS, including Animat (denoted simply as *) [292], BOOLE [293], SCS [94], NEWBOOLE [29], ZCS [297], and EpiCS [128].

Conditions can Generalise over Inputs

A rule's condition c is said to *match* an environmental input m if for each character m_i the character in the corresponding position c_i is identical or the wildcard (#). For example, the condition $00\#$ matches two inputs: 000 and 001. The wildcard is the means by which rules generalise over environmental states; the more #s a rule contains the more general it is. The most general 3-bit rule condition is ###, and the most specific are those without any #s, e.g., 000, 001, 010 and so on. Since actions do not contain wildcards the system cannot generalise over them.

Denoting Rules

To denote the condition and action of a rule we will write it in the form *condition* $\rightarrow$ *action*. For example, the rule $000 \rightarrow 0$ says "If the input is 000 then take action 0".

Limitations of the Descriptive Power of Single Classifier Conditions

Classifiers define regions within the state $\times$ action space $\mathcal{S} \times \mathcal{A}$, but, using the ternary language, are not able to describe arbitrary regions. For example, the only condition which matches both 01 and 10 is ##. However, this condition also necessarily matches 00 and 11, which may be undesirable. There is no single condition which matches only 01 and 10.

Similarly, no single rule can express the union of $00 \rightarrow 0$ and $00 \rightarrow 1$ since rule actions do not contains wildcards (#s) and so rules cannot generalise over actions.

More complex languages Other representations have different limitations on the generalisations they can express. Some representations allow us to express a greater variety of generalisations using a single rule, but this comes at the cost of making the rules more complex. One approach, for example, is to allow a single rule to make use of conjunctions of conditions. Although this allows us to represent the same generalisations with fewer rules, the rules are more complex, which may tend to offset any advantage derived from there being fewer of them.

Representation and Bias The representation used by a learning system is an important source of bias. Although some representations are superior for use with a given task than others, no one representation is ideal for all tasks.

2.2.2 Other Representations

Here we briefly review some of the other representations used with classifier systems.

More Complex Ternary Languages

Extensions to the standard ternary language described earlier include the use of multiple conditions (which are logically ANDed together) in a single rule (e.g., [93, 126, 39]), negation of conditions (e.g., [250, 126, 39]), and the

use of pass-through characters (e.g., [90, 126, 212, 259]). Systems like LS-1 [250] (see also [94] p. 270) use even more complex languages based on ternary strings.

Non-Ternary Languages

Although most work with LCS has made use of the ternary languages just described, a significant body of work makes use of alternative schemes. Booker [34] describes alternative encodings using binary and ternary coding schemes which are better suited for encoding ordinal and nominal attributes than the earlier ternary languages.

Wilson [309, 310] employs a real-valued version of XCS, in which, instead of binary input strings, each attribute can take a value within a given range. Each attribute in a rule condition is represented by an upper and lower value, and that attribute is matched if the input falls within this range.

A few researchers, following the suggestion of Wilson [297], have replaced the usual genetic algorithm with other search engines. Lanzi investigated the use of variable-length conditions [163], and genetic programming [164], as have Ahluwalia and Bull [4]. (Genetic programming [60, 155, 156, 157, 158, 159, 160] is concerned with the evolution of algorithms. Applied to LCS, genetic programming allows the evolution of conditions – algorithms – which compute whether they match the current input, and what action to take.)

In 1991 Valenzuela-Rendón [282] described a fuzzy LCS; an LCS which used fuzzy logic in rule conditions. Since then fuzzy LCS have been the subject of considerable research (see [26] for a survey).

Representing Models of the World

A special form of representation, which we mention only briefly, is one which models the state transition dynamics of the environment in which the system operates. Such systems can simulate the outcome of a sequence of actions, allowing them to explicitly plan ahead, as proposed by Samuel [221] many years ago. This approach has been investigated in LCS by a number of researchers [32, 124, 214, 298, 259].

Sets of Rules

Some LCS have been modified to encourage the formation of sets of rules which constitute special ways of representing information about a task. Default hierarchies ([126, 94, 247], §3.6.1) consist of general default rules which more specific exception rules override. Corporations [315, 272] are sets of rules which are activated sequentially.

2.2.3 Summary of Rule Representation

It should be emphasised again that the classifier system is a very broad paradigm, and that the variety of representations reviewed here only scratches the surface of what is possible. The study of representation in LCS is beyond the scope of this work and remains an area for future research.

2.2.4 Notation for Rules

In addition to the condition/action notation for rules introduced on page 26, we will also occasionally (mainly in chapters 4 and 5) make use of the following:

- A Boolean target function f is a total function on a binary bit string, that is $f : \{0,1\}^n \rightarrow \{0,1\}$.
- Classifiers are constant partial functions, that is, they map some subset of the domain of f to either 0 or 1. Classifiers are constant because, using the standard ternary language, they always advocate the same action regardless of their input.

As a shorthand, and to approximate Sutton and Barto's reinforcement learning notation [265], we define $\mathcal{S}$ = domain and $\mathcal{A}$ = range when dealing with classifiers and target functions. That is, a task's state is an element of $\mathcal{S}(f)$ and a classifier system's action is an element of $\mathcal{A}(f)$, where f is a target function. The states matched by a classifier c are elements of $\mathcal{S}(c)$, and the action advocated by c is $\mathcal{A}(c)$.

2.3 XCS

XCS is a classifier system introduced by Stewart Wilson in 1995 [298] whose primary distinguishing feature is the way in which it calculates the fitness of a rule for use in rule discovery. Most classifier systems use a rule's strength as its fitness, that is, they use the same value to weight a rule in both action selection and reproduction, the two great challenges introduced on page 5. (Recall the figure on page 11 which shows how strength and accuracy-based systems use reward.) The first of Wilson's great insights with XCS was that the two values differ; that the value of a rule in selecting actions is not the same as its value in generating new rules. His second insight was that fitness could be based entirely on the accuracy of a rule's reward prediction, not on its magnitude, and that this would allow the formation of complete maps of

the value function, as mainstream reinforcement learning systems do. XCS has a number of other features which differ from other LCS, but this work is concerned primarily with its most significant one, its fitness calculation. The consequences of this novel approach to fitness, some anticipated by Wilson, others not, are the subject of this thesis.

XCS has many features in common with Wilson's earlier ZCS [297], which was intended as a minimalist classifier system whose mechanisms would be more easily understood than those of the traditional LCS. XCS also owes a debt to Booker's GOFER-1 system [33] from which it inherited the niche GA, a deletion scheme which encourages the distribution of rules throughout the environment, and an interest in rule accuracy ([298] p. 32).

In the following, we first outline Wilson's motivations for XCS in §2.3.1, provide an overview of XCS in §2.3.2, and then proceed to a more detailed discussion of XCS's component systems: §2.3.4 covers the performance system, §2.3.5 covers the credit assignment system, and finally §2.3.6 covers the rule discovery system. Each of these sections presents an algorithm for the relevant part of the system, following which a detailed description is given. The algorithms are in English, and omit minor details such as the update of time-keeping variables. It is hoped that they will provide a concise, comprehensible description of the major features of the XCS algorithm not easily obtained from more detailed, formal presentations, such as actual source code, while at the same time adding to the rigour of the textual description. The language of mathematics has been adopted for many key components, as experience has shown the difficulty of conveying this material in its absence. The specification in this chapter is supplemented by a detailed walk-through of XCS's activity on a single time step in appendix B. Recently, an excellent lower-level specification has become available [53], as has a formal specification using ISO VDM-SL [14].

Wilson's terminology and notation have been retained in order to reduce confusion, and the list of notation in appendix I reflects this. Page numbers refer to the electronic version of [298] which is available on the net [314].

2.3.1 Wilson's Motivation for XCS

With XCS, Wilson set out to build a classifier system that would evolve general but accurate rules, and which would map the problem space completely, in the manner of mainstream reinforcement learners, rather than just parts of it, in the manner of other classifier systems [298].

Although complete maps of the problem space are a stated goal of XCS, Wilson never motivated them other than to say they should help the system in exploring alternative actions ([298] p. 5). We will consider complete maps

in detail in chapter 3.

Wilson emphasised the need for accurate generalisations because he perceived that in strength-based systems there was nothing to inhibit overgeneral rules, and nothing to encourage accurate, general rules:

> " ... under payoff-based [i.e., strength-based] fitness, there appears to be no clear tendency, or, indeed, theoretical reason, for accurate generalizations to evolve." [298] p. 4.

We will examine overgeneral rules in detail in chapter 5, but for now, let us simply say that an overgeneral rule is one which is too general; from which a superior, more specific rule could be produced.

XCS contains mechanisms, notably the niche GA, designed to encourage general rules, but at the same time, its accuracy-based fitness is designed to ensure that only accurate rules survive. We will consider the rationale behind the XCS approach in §3.3.

2.3.2 Overview of XCS

Features of XCS

As mentioned previously, the most significant feature of XCS is its accuracy-based fitness. Among its other interesting features, XCS uses a form of restricted mating called a niche GA which focuses genetic search and provides a strong generalisation pressure (p. 47), a technique called subsumption deletion which provides another bias towards general rules (p. 52), deletion schemes and a GA invocation scheme which balance rule allocation between niches (p. 49), and macroclassifiers which decrease run times and provide interesting statistics (§2.3.2). Most of these features could be incorporated in strength-based LCS, and some have been, e.g., in SB–XCS. The aspects of XCS which depend on its accuracy-based fitness should be clear, but in any case will be highlighted in §2.4 when we specify how strength-based SB–XCS differs from XCS.

Interacting with the Environment

XCS is designed for reinforcement learning, so it interacts with its environment following the RL paradigm; on each time step it 1) receives input from the task environment, 2) sends an action to the environment, and 3) receives a numeric reward (see §C.1 and §C.5). Because XCS uses the standard ternary language of §2.2.1 to represent its rules, the input from the environment and

XCS's actions are represented using binary strings. (In both cases, a single fixed-length string.)

XCS has no internal memory, and thus the environmental inputs constitute the only information it has about the current state of the world.

The Population of Rules

The population of rules is denoted [P] and has a fixed size limit denoted N. Once this limit is reached, the addition of new rules requires the deletion of the same number of old rules. The population may be initialised randomly, although the standard approach is to use an empty initial population and to use the covering technique of §2.3.6 to generate initial rules.

Rule Parameters

Classifiers are fundamentally condition-action rules, but a classifier system needs to associate additional information with each rule. XCS maintains many statistics on each rule, most importantly:

Prediction: p, an estimate of the reward to be received if the rule's action is taken.

Prediction Error: ε, an estimate of the error in the predictions the rule has recently made.

Fitness: F, an estimate of the utility of the rule which is used in rule discovery. Simplifying somewhat, the lower a rule's prediction error, the higher its fitness.

We will see how XCS updates these estimates in §2.3.5.

Macroclassifiers

A *macroclassifier* is simply a classifier with a *numerosity* parameter which indicates the number of identical virtual classifiers it represents. That is, rather than store and process n classifiers with the same condition and action, XCS simply stores one macroclassifier with numerosity n, which improves run-time. Statistics based on macroclassifiers and their numerosity are also useful.

Instead of directly inserting newly generated classifiers into the population, XCS checks to see if there is an existing classifier with the same condition and action as the new one. If so, the new classifier is discarded and the existing classifier has its numerosity incremented by one. Similarly,

when a classifier is selected for deletion it is actually only deleted if it has a numerosity of one. Otherwise, its numerosity is decremented by one.

As a result, XCS needs to be implemented in such a way as to take a classifier's numerosity into account and treat it as an equivalent number of microclassifiers in all relevant cases. For example, in calculating the probability of selecting classifiers for reproduction, the system must treat a classifier with a numerosity of 5 as 5 times more likely to be selected than one of the equivalent microclassifiers.[1]

The following table shows a population of *micro*classifiers:

Rule	Condition	Action	Prediction	Prediction Error	Accuracy
a	##0011	1	200.0	0.0	1.0
b	##0011	1	200.0	0.0	1.0
c	##0011	0	100.0	0.0	1.0
d	001110	1	100.0	0.0	1.0
e	10##01	0	501.1	0.0011	0.75
f	10##01	0	500.8	0.0008	0.80
g	10##01	0	500.3	0.0003	0.95

The next table shows the equivalent population implemented with macroclassifiers (note the additional numerosity parameter):

Rule	Cond.	Action	Prediction	Pred. Error	Acc.	Numerosity
a	##0011	1	200.0	0.0	1.0	2
c	##0011	0	100.0	0.0	1.0	1
d	001110	1	100.0	0.0	1.0	1
g	10##01	0	500.3	0.0003	0.95	3

In the figure above, a has absorbed b, which had the same condition and action. Similarly, g has absorbed e and f. Note that g absorbs the numerosity of e and f, but ignores their prediction, prediction error and accuracy. In practice, rules are absorbed by older, better evaluated rules, and it seems safe to ignore the parameters of the newer rules.

As a fitter population evolves, the rule discovery component tends to produce more and more copies of the same highly fit condition/action pairs, so the proportion of the population consisting of classifiers with a numerosity

[1] Technical note: Because XCS's fitness calculation already takes numerosity into account (see p. 45), the fitness of a classifier is *not* multiplied by its numerosity in determining its probability of selection.

greater than one tends to grow. As a result, there can be a significant reduction in the size of the population, resulting in a improvement of processing speed.

In keeping with [298], the term "macroclassifier" will be reserved for those situations in which it makes explanation clearer, and normally macroclassifiers will simply be referred to as classifiers. However, it should be understood that all classifiers in XCS have a numerosity field and are thus macroclassifiers.

Wilson reports informally [298] that there is no apparent difference between a system which uses macroclassifiers and one which does not, other than the speed of operation. Appendix A reports the results of a comparison of XCS with and without the use of macroclassifiers.

A Note on Variations of XCS

Replicating Wilson's XCS precisely has proved surprisingly difficult; small discrepancies between the author's implementation and Wilson's were occasionally uncovered over the course of over four years, despite considerable effort by both to minimise the differences. Similarly, small differences between these two versions and that of Alwyn Barry were occasionally uncovered. Fortunately, experience suggests that slight variations in the algorithm tend to have little effect on performance. In this sense XCS is robust: many minor variations make little real difference to the system. In fact, XCS has always been presented as a framework in which alternative methods for action selection, rule reproduction, exploration and so on can be used, rather than as a prescription for a unique system to be strictly adhered to [298].

2.3.3 Wilson's Explore/Exploit Framework

Wilson used an experimental framework with XCS in [298] which will be referred to as *Wilson's pure explore/exploit framework*. This scheme is not an intrinsic part of XCS – other approaches can be used – but to date it has been the standard approach and it is used here. It is detailed in this section because it influences the operation of XCS, which will be detailed in subsequent sections.

Exploration and Exploitation

In Wilson's scheme, on each time step XCS performs either a *pure explore trial* or a *pure exploit trial*. In explore trials, the system selects an action at random from those advocated by the matching rules; that is, it performs an

unbiased exploration of the available options. On such trials no attempt is made to exploit the system's current knowledge of how to maximise rewards. In exploit trials, in contrast, the system deterministically selects the action which is most highly recommended by the matching rules. That is, its behaviour is maximally biased towards exploitation of its current knowledge – there is no exploration involved in action selection on such trials. We refer to this as the *pure* explore/exploit scheme because the two cases represent the extremes of exploratory and exploitative behaviour.

In non-sequential tasks XCS either 1) chooses the explore or exploit mode with equal probability ([298] p. 16, [53] p. 10[2], or 2) alternates between the two ([304] p. 670). The difference is insignificant and my implementation has always taken the latter approach. In sequential tasks, the action selection mode is determined at the start of the episode and maintained throughout all its time steps ([298] p. 24).

When to Learn and When to Evaluate

The idea behind this scheme is to separate learning and evaluation into distinct modes, but to interleave them. Ideally, learning (credit assignment and rule discovery) should only occur on explore trials, and only exploit trials should be used to monitor performance. A consequence is that exploration has no direct effect on the statistics generated. To see this, consider the alternative of having only one type of trial, in which the system combines both exploration and exploitation. It could, for example, take the most highly recommended action 90% of the time, and some other action at random otherwise. We can then consider all trials in monitoring performance, but even if the system learns the correct action for each input, performance statistics will reach only about 90% because they are influenced by exploratory behaviour.

Although the idea is to separate learning and evaluation, covering is always operative, as it is needed to ensure the system has at least one matching rule. Additionally, in [298] Wilson allowed updates to rules (but not GA invocations) in sequential tasks in order to allow the system to escape from loops through a process called *fatigue* (p. 25). Fatigue is not mentioned in the later description of XCS [53], but it will be discussed further on page 47. The later description is more vague than [298] about action selection, and, with respect to allocation of learning says only: "As an aside, it appears better to perform the GA only on exploration steps, especially if most steps are exploitative." [53] p. 10.

[2]Page references are to the electronic version of [53].

Analysis of the Framework

One interesting feature of this scheme is that exploratory trials are not included in graphs, which has the advantage that it shows more clearly how good the agent's policy currently is. Note, however, that we need not use pure explore/exploit trials to achieve this effect. We could, for example, obtain it using ϵ-greedy action selection. A simple alternative means of assessing the policy without the confounding effects of exploration is to halt learning at a certain point and evaluate the system in pure exploit mode for some time.

Another interesting feature of Wilson's scheme is the very high amount of exploration which occurs during learning, given that learning only occurs during pure explore trials. (This is significant because, informally, XCS appears to need high exploration rates in order to avoid forgetting what it has learnt.) Because learning only occurs when actions are selected randomly, off-policy updates (p. 265) must be used, although in non-sequential tasks there is no distinction between on-policy and off-policy updates.

Wilson's scheme is less than fully satisfactory in sequential tasks as it handles exploration very inefficiently, tends to get stuck in loops in deterministic sequential environments, and follows a random policy, which makes it likely that it will learn slowly about the optimal policy. Although the scheme is more suitable for non-sequential tasks, note that the practice of using separate training and test data sets, or cross-validation, is the norm in the machine learning literature [184].

Wilson's approach is used here despite these problems as it is the standard approach with XCS, and so allows comparison with existing results, and because it is adequate for our purposes.

2.3.4 The Performance System

This section details XCS's performance system, which is responsible for handling its interactions with the task environment in which it operates. This system determines which rules match the current input, considers their recommendations, selects an action and triggers the other systems as necessary.

XCS is a stimulus-response classifier system, that is, it has no message list and thus no internal memory. Environmental inputs are matched directly against the rules in the population, and the action selected by XCS is sent directly to the environment.

1. Obtain the single input string from the environment.
2. Form the match set [M] of classifiers in [P] whose conditions match the input string (p. 36).
3. Calculate the system prediction for each action using the rules in [M] in order to fill the prediction array (p. 36).
4. Select an action based on the system predictions (p. 37).
5. Send the selected action to the environment and receive a reward.
6. Form the action set [A] of classifiers in [M] which advocated the action selected in 4 above.
7. Call the credit assignment algorithm (figure 2.3).

Figure 2.1: The XCS performance system algorithm.

The XCS Performance System Algorithm

This section presents the basic performance system algorithm, and subsequent sections discuss each step in more detail. Appendix A presents a detailed walk-through of an XCS cycle in order to clarify the process. Please refer to the schematic diagram of XCS in figure 2.2 while considering the performance system algorithm shown in figure 2.1.

The Match Set and Prediction Array

Operation is as follows. A *match set* [M] is formed from those classifiers in the general population [P] which match the system's input. (See §2.2.1 for the matching algorithm.) Next, a *system prediction* $P(a_i)$ is computed for each action a_i in [M] using a fitness-weighted average of the predictions of classifiers advocating a_i ([298] p. 7). That is, [M] can be partitioned into a number of mutually exclusive sets according to the action advocated by each rule. System prediction is defined as:

$$P(a_i) = \frac{\sum_{c \in [\mathrm{M}]_{a_i}} F_c \cdot p_c}{\sum_{c \in [\mathrm{M}]_{a_i}} F_c} \tag{2.1}$$

where $[\mathrm{M}]_{a_i}$ is the subset of [M] advocating action a_i, and p_c is the prediction of rule c.[3] This value is the system's estimate of the return it will receive if it takes action a_i.

[3]Note that (2.1) does not consider a rule's numerosity, because fitness already takes numerosity into account (p. 45).

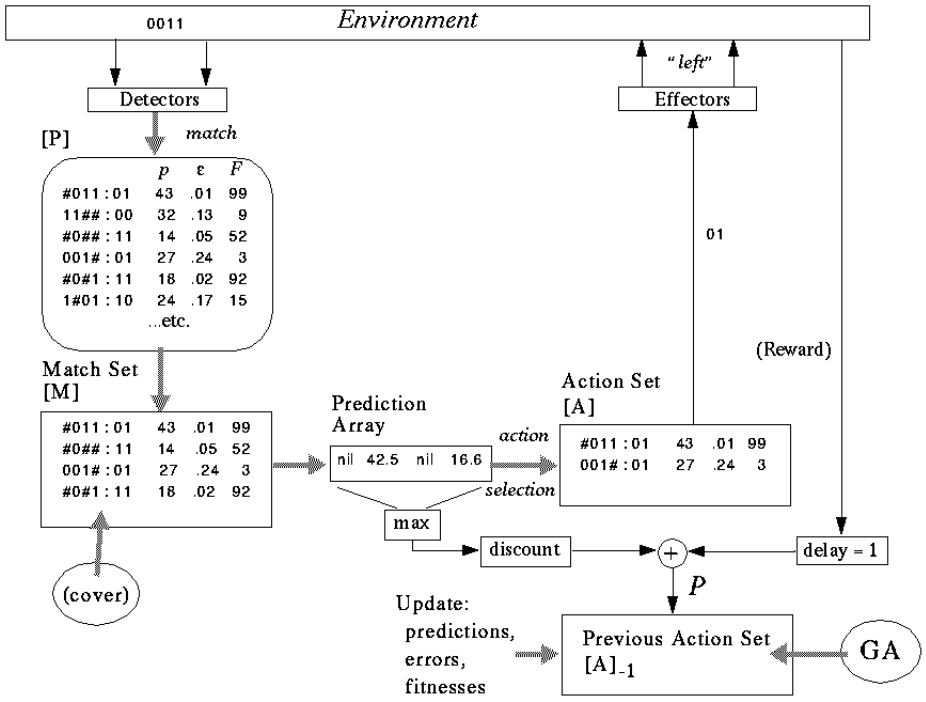


Figure 2.2: This schematic illustration of XCS appears courtesy of Stewart W. Wilson. Note that the GA sometimes operates in $[A]_{-1}$ and sometimes in [A], as detailed in figure 2.3.

The system prediction for each advocated action is placed in a *prediction array* in preparation for action selection. Some actions may not be advocated by any classifier in the current [M] and will have a void prediction, meaning they cannot be selected. If no action is advocated (i.e., [M] is empty), then a random classifier is generated through *covering* (see section §2.3.6), which guarantees that [M] contains at least one rule, and that at least one action is advocated.

Action Selection

The system next selects an action from those with non-null predictions in the prediction array and forms an *action set* [A] of classifiers in [M] advocating the selected action. This set is customarily referred to as *the* action set, although we note that [M] may contain many actions sets; it contains as many [A]s as there are different actions advocated by its members.

Once an action has been selected it is sent to the environment and a reward r_t is returned.

Unadvocated Actions Note that although some actions may have no prediction, XCS tends very strongly to produce a prediction for each possible action, thanks to its accuracy-based fitness (see §3.6.3). This more complete mapping of the alternatives, relative to strength-based LCS, allows more sophisticated action selection. (See the discussions of fatigue in §2.3.6, and action selection in §6.2.1 for more on the effects of actions without predictions.)

Other Action Selection Schemes Many other action selection methods are possible, including some which are much more sophisticated (see, e.g., [265]). For non-sequential tasks it seems desirable to adopt the standard practice in machine learning of using separate training and testing data sets, and cross-validation (see, e.g., [184]). However, Wilson's pure explore/exploit approach has been used with XCS by convention, and it is used here to allow comparisons with earlier work (as mentioned on page 35).

Experience-Weighting of System Prediction

Wilson reports [300] that he has used a modified method of calculating the prediction array which is not mentioned in either of the XCS papers [298, 304]. In this new version, if a classifier's experience is less than a threshold (e.g., 20), its fitness is considered as $1/16^{th}$ of its actual value for the purpose of calculating the system prediction. This has the effect of reducing the influence of relatively untested classifiers which may have unreasonably high fitness values. This technique has the advantage of having no effect if all classifiers involved are inexperienced since they are all then subjected to the same devaluation. However, this heuristic has yet to be formally evaluated and we do not consider it part of the standard XCS specification.

2.3.5 The Credit Assignment System

This section details XCS's credit assignment system, which is responsible for updating the parameters associated with each rule. Most importantly, this system updates each rule's reward prediction and fitness on the basis of the rewards received from the environment.

The most important feature of credit assignment in XCS, and of XCS generally, is that fitness is based on the accuracy of reward prediction. In this section we give the full details of how fitness is calculated in XCS. Although

most classifier systems employ some form of bucket brigade algorithm to update strength/fitness estimates, we will see that XCS employs a more Q-learning-like update.

The MAM Technique

The MAM technique ("Moyenne Adaptive Modifiée", or Modified Adaptive Method) was introduced in [283] as a means of speeding up the estimation of various classifier parameters based on information obtained on successive time steps. Using this technique, a parameter is updated using one method early on and a second method later. The reasoning is that the first method can be used to quickly get a rough approximation of the true value of the variable, while the second method can make more conservative adjustments and refine the estimate. More specifically, for the first n updates, we simply use the average of the sample values seen so far. For the $n^{th}+1$ and subsequent updates we use another, more conservative, update.

We know how many times a rule's parameters have been updated because each classifier includes an *experience* parameter which is a count of the number of times it has occurred in the action set (and thus the number of times it has been updated).

Wilson states that using the MAM technique the value of a parameter should be the sample average while a rule has been updated less than $1/\beta$ times. After this it should be updated using the Widrow-Hoff technique ([298] p. 8). If we use the incremental update for the sample average (equation (E.6) from §E.3.1) the complete prediction update is:

$$\begin{aligned}&\text{if experience} < 1/\beta \text{ then}\\ &\qquad p_j \leftarrow p_j + \tfrac{1}{n+1}\Big[r_t - p_j\Big]\\ &\text{else}\\ &\qquad p_j \leftarrow p_j + \beta\Big[r_t - p_j\Big]\end{aligned}$$

However, as we note in §E.3.1, the Widrow-Hoff delta rule is in the same form as the incremental average. In fact, they differ only in the step-size term, as we can see above. Consequently, we can rewrite the update above simply as:

$$p_j \leftarrow p_j + \alpha\Big[r_t - p_j\Big]$$

(which is identical to (2.4)) and note that:

if experience $< 1/\beta$ then
$\alpha = \frac{1}{n+1}$
else
$\alpha = \beta$

where we use α to denote the step size term to avoid using β as both the step size and the constant to which experience is compared.

In other words, the MAM technique is simply a particular way of adjusting the learning rate term in the delta rule over a number of time steps.

The Credit Assignment Algorithm

Figure 2.3 presents the algorithm which determines when rules are updated, checks to see if the GA should be triggered and attempts subsumption deletion. This algorithm is executed following the production system algorithm (figure 2.1) on each time step.

Effectively only steps 7 to 10 apply to non-sequential tasks, but the algorithm accommodates both sequential and non-sequential tasks by assuming that in non-sequential tasks $[A]_{-1}$ is always empty (so the test at step 1 always fails) and that each trial is a separate episode (so the test at step 6 always succeeds).

Note that for the non-sequential case updates and GA invocations occur exclusively in [A] and never in $[A]_{-1}$. In contrast, in sequential tasks, updates and GA invocations only occur in $[A]_{-1}$, except during the last trial of an episode, during which both $[A]_{-1}$ and [A] are updated. If a sequential task is completed on the first cycle the update occurs as for a non-sequential task ([298] p. 8). Updates occur in $[A]_{-1}$ on sequential tasks because they make use of the prediction array on the *following* time step (see equation (2.3)).

This algorithm does not appear explicitly in Wilson's publications, and is not obvious from published descriptions of XCS. Wilson has, however, confirmed its form [301].

Sequential and Non-Sequential Updates

A rule's prediction is updated somewhat differently for sequential and non-sequential tasks ([298] p. 8), but a unified notation for the two cases can be obtained using a value P to represent the target (see §E.3.2) towards which the prediction moves.

When Rewards Occur On page §C.4.1 it is explained that Sutton and Barto's RL notation [265] assumes the immediate reward for acting at time

```
1.  If the previous time step's action set [A]_{-1} is not empty then
2.      Update [A]_{-1} (p. 42).
3.      Do action set subsumption in [A]_{-1} (p. 52).
4.      If conditions for GA invocation in [A]_{-1} are met (p. 49) then
5.          Call the rule discovery algorithm in [A]_{-1} (p. 47).
6.  If the episode has just ended then
7.      Update [A].
8.      Do action set subsumption in [A].
9.      If conditions for GA invocation in [A] are met then
10.         Call the rule discovery algorithm in [A] (p. 47).
11. else
12.     [A]_{-1} ← [A] (in sequential mode only).
```

Figure 2.3: The XCS credit assignment algorithm.

t is received at time $t+1$. Wilson, however, assumed reward was received at time t, and in dealing with XCS we adopt Wilson's convention.

For non-sequential tasks the predictions of classifiers in [A] are updated towards:

$$P = r_t \tag{2.2}$$

where r_t is the immediate reward (§C.3.1) returned by the environment. The full prediction update, equation (2.4), will be presented shortly.

For sequential tasks, rules in the previous time step's action set $[A]_{-1}$ are updated towards the sum of the previous time step's reward and the discounted maximum of the current time step's prediction array:

$$P = r_{t-1} + \gamma \max_i P(a_i) \tag{2.3}$$

where $0 \leq \gamma \leq 1$ is the *discount rate* (§C.4.2) which weights the contribution of the next time step to the value of P, and $P(a_i)$ is the system prediction for action a_i (see p. 36). Note that (2.3) reduces to (2.2) when $\gamma = 0$.

Parameter Update Order

The order in which parameters are updated has been the subject of some experimentation. In the original XCS paper, Wilson states ([298] p. 8) that the order is: 1) Fitness, 2) Prediction Error, and 3) Prediction. MAM updates (p. 39) were used in all three cases.

In subsequent work [300], Wilson experimented with the order: 1) Prediction Error (with MAM), 2) Prediction (with MAM) and 3) Fitness (without MAM), which is the order the author has always used. This is a conservative order as the fitness begins at a low value and rises slowly, which results from updating the prediction error first (typically resulting in a large error on the first update as a default value is used for the as-yet unupdated prediction) and of not using the faster MAM technique for updating the fitness parameter. Informally, this order appears to work better than the first on tasks with longer input strings, where fitness is established slowly.

Finally, in [53] (p. 11) the order is given as: 1) Prediction, 2) Prediction Error, and 3) Fitness, with MAM updates used only for the first two. This order is perhaps the most intuitive one, and we will adopt it for the description of XCS in the following section. The differences between the orders are not great.

XCS Parameter Updates

Using the last update order discussed in the previous section, the reinforcement process for each classifier in the set being updated (i.e., [A] or $[A]_{-1}$) is:

- **Update Prediction**

$$p_j \leftarrow p_j + \beta(P - p_j) \tag{2.4}$$

where p_j is the prediction of rule j, and $0 < \beta \leq 1$ is a value controlling the learning rate. (In XCS work to date β has always been a constant (apart from the effect of the MAM technique on page 39), though in most RL work it is declined.)

Note that expanding P according to (2.3) yields the 1-step Q-learning update ((E.17), p. 265). (α and β are simply different notations for the step size term.) The only difference between the Q-learning in XCS and a tabular system is that in XCS, the system predictions (i.e., Q-values) are the product of many estimates (many classifiers), whereas in a tabular system each Q-value is maintained by a single estimate (see [298] p. 21–22, §6.1.1).

Outside this chapter, prediction will for simplicity normally be referred to as strength, since this is the traditional term for this value, and the one used with the SB–XCS system introduced in §2.4. See §5.3.5 for additional notes on terminology.

• **Update Prediction Error**

$$\varepsilon_j \leftarrow \varepsilon_j + \beta\left(|\overbrace{P - p_j}^{Error}| - \varepsilon_j\right) \tag{2.5}$$

where ε_j is the prediction error of rule j. Although not specified in [298], the error was originally divided by the *payoff range* (the difference between the highest and lowest possible rewards), in order to normalise the error to a value between 0 and 1 [302]. This normalisation has since been removed as an unnecessary complication [53, 313].[4]

• **Update Fitness** This has three steps:

1. Calculate Accuracy

$$\kappa_j = \begin{cases} 1 & \text{if } \varepsilon_j < \varepsilon_o \\ \alpha(\varepsilon_j/\varepsilon_o)^{-v} & \text{otherwise} \end{cases} \tag{2.6}$$

where $0 < \varepsilon_o$ is the *accuracy criterion*, a constant controlling the tolerance for prediction error. Any rules with $\varepsilon < \varepsilon_o$ are considered to be equally (and fully) accurate.

The *accuracy falloff rate* $0 < \alpha < 1$ and *accuracy exponent* $0 < v$ are constants controlling the rate of decline in accuracy when ε_o is exceeded.

The Original Accuracy Calculation Originally ([298] p. 10), the accuracy calculation was given as:

$$\kappa_j = \begin{cases} \exp[(\ln\ \alpha)(\varepsilon_j - \varepsilon_o)/\varepsilon_o] & \text{if } \varepsilon_j > \varepsilon_o \\ 1 & \text{otherwise} \end{cases} \tag{2.7}$$

Later, the update was changed to (2.6) in order to improve discrimination of small errors [308]. As Wilson explains:

> "The reason for the change is that for small differences in error, the [original] exponential method produces a smaller and smaller difference in accuracy. I don't think this is desirable. For

[4] We note that in any case the payoff range bounds the error only in non-sequential mode; in sequential mode the error $|P - p_j|$ could exceed 1. The effect of the error exceeding 1 has not been investigated, but seems unlikely to disrupt the system as larger errors simply lead to lesser accuracies.

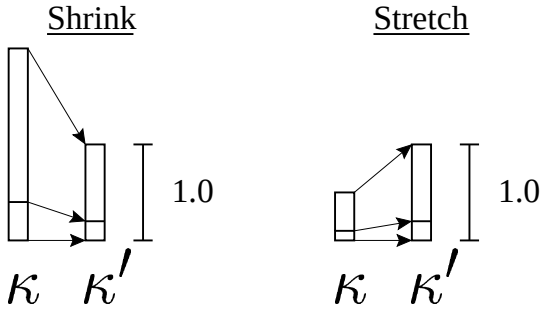


Figure 2.4: Illustration of the effects of the relative accuracy update.

> states with small predictions, and thus small errors in the first place, you lose the ability to differentiate predictions.
>
> In contrast, suppose you decide that if, say, two errors differ by a factor of two, you want the accuracies to differ by a factor that is independent of the absolute sizes of the errors. The [new] power function does this: if two errors differ by a factor r, then the ratio of accuracies is r^{-n}." [311]

2. Calculate Relative Accuracy Once the accuracy of all rules in [A] has been updated we calculate each rule's relative accuracy κ':

$$\kappa'_j = \frac{\kappa_j \cdot numerosity(j)}{\sum_{x \in [A]} \kappa_x \cdot numerosity(x)} \tag{2.8}$$

(2.8) causes the relative accuracies of the rules in an action set sum to 1. Whether a rule's relative accuracy is less than, equal to, or greater than its accuracy depends on whether the sum of the accuracies in its action set is less than, equal to or greater than 1. Two effects of the relative accuracy update are illustrated in figure 2.4. The left side shows that if the sum of the accuracies of the rules in [A] exceeds 1, the relative accuracies of the rules are less than their accuracies. Conversely, relative accuracies are greater than accuracies if the sum of the accuracies in [A] is less than 1.

One problem with the relative accuracy update is that if [A] contains only a single rule, it will have a relative accuracy of 1 regardless of its accuracy. The same problem occurs when [A] consists of multiple inaccurate rules, although the effect is less drastic since they share the total relative accuracy.

3. Update Fitness Finally, each rule's fitness is updated towards its relative accuracy:

$$F_j \leftarrow F_j + \beta(\kappa'_j - F_j) \tag{2.9}$$

Notes on Numerosity and Fitness Note that a rule's share of the relative accuracy in [A] is weighted by its numerosity. Because fitness is updated towards relative accuracy, this means that fitness too is weighted by numerosity. Consequently, in using fitness, e.g., to construct a roulette wheel with which to select parents, we do not explicitly take numerosity into account, because fitness already reflects numerosity. For this reason, in XCS we *never* multiply fitness by numerosity.

Note also that weighting relative accuracy by numerosity implements fitness sharing (§G.5.1), a technique which distributes a population (of rules, in this case) more evenly through space. This form of fitness sharing differs from that in other systems, e.g., ZCS [297]. First, XCS shares accuracy-based, rather than strength-based fitness; it shares information about the accuracy of rules than their strengths. Second, XCS shares fitness within the set the GA operates in (normally [A]), whereas in ZCS the GA and fitness sharing are panmictic (although a niche GA could be added to ZCS).

Summary of Updates The XCS updates treat the strength of a rule as a prediction of the reward to be received, and maintain an estimate of the error ε_j in each rule's prediction. An accuracy score κ_j is calculated based on the error as follows. If the error is below the accuracy criterion threshold ε_o the rule is fully accurate (has an accuracy of 1), otherwise its accuracy drops off quickly. The accuracy values in the action set [A] are then converted to relative accuracies (the κ'_j update), and finally each rule's fitness F_j is updated towards its relative accuracy.

To simplify, in XCS fitness is an inverse function of the error in reward prediction, with errors below ε_o being ignored entirely.

2.3.6 The Rule Discovery System

The rule discovery system is the third and final of XCS's major subsystems. It is responsible for the generation of new rules and the removal of old rules. Rules may be created in one of three ways: 1) as part of a random initial population, 2) by covering, and 3) by the genetic algorithm. Each of these is discussed in turn, along with the selection of rules for deletion, subset deletion, and the initialisation of newly created rules.

Random Initial Populations

One approach to initialising the rule population in XCS is to fill it with random rules at the outset. N rules are generated, each with a random condition and random action. Each bit in a rule's condition is a # with probability $P_{\#}$, otherwise it is equally likely to be a 0 or 1. This produces a random initial population consisting of more and less general hypotheses. In practice, however, covering has almost always been used instead.

Covering

The alternative to creating a random initial population is to start with an empty initial population and create matching rules as needed through covering (as first suggested by Holland [120]). When a classifier is created through covering, its condition is a copy of the current environmental input and it is given a randomly chosen action. Each character in the condition is then mutated with probability $P_{\#}$ into a #. This procedure allows the newly generated rule to generalise, but guarantees that it matches the current input. The covering classifier is then inserted into [P] and if the population size limit is exceeded, a classifier is deleted using the normal method. When covering is used to initialise a population, it normally occurs only a few times at the beginning of the run. This is sufficient to start the system, after which the genetic algorithm takes over. See [52] for discussion of cases in which covering may continue indefinitely.

Triggering Covering Covering is triggered when less than θ_{mna} rules match the current input.[5] In [298] $\theta_{mna} = 1$, but more recently Wilson has adopted the practice of setting it to the number of possible actions [313]. This produces better covering of the input/action space more quickly, although on some tasks the effects are small.

Action Set Covering The author has experimented with a simpler, more thorough approach, which is simply to produce a rule by covering when the *action set* (not the match set) contains fewer than θ_{mna} rules, where typically $\theta_{mna} = 1$. This guarantees that all actions are advocated by some rule, and has produced a considerable improvement in XCS's initial performance on sequential decision tasks (specifically, the Woods1 and Woods2 tasks described in §2.5.2).

[5] mna stands for Minimum Number of Actions.

Fatigue In [298] covering was used for a second purpose, which was to break XCS out of loops in which it may get stuck in sequential mode ([298] p. 10). Such loops are mainly the consequence of using deterministic action selection in exploit mode, combined with environments with deterministic state transitions.

If XCS is stuck the discount rate γ in (2.3) steadily reduces the predictions of the rules active in the loop, which we could say are being fatigued by their repeated activation. To detect and correct this situation, covering was triggered if the total prediction of [M] is less than ϕ times the mean prediction of [P]. Inserting a new classifier via covering is usually sufficient to break the system out of the loop.

Wilson reports that this form of covering seems not to be needed when θ_{mna} is raised so that the first form more thoroughly covers the input-action space [313]. From this we can infer that loops occurred mainly due to a lack of choice when only a subset of actions were available. The deprecation of fatigue is desirable as it was an unsatisfactory method of breaking out of loops. One problem is that the true prediction of a rule could be such that it would trigger covering, another is that fatigue results in a considerable distortion of predictions, which must then be relearnt on subsequent trials.

However, experiments presented in §2.5.2 show that even using action set covering, which is more effective than match set covering, XCS still occasionally becomes stuck in loops during a brief initial phase. To deal with this two simpler and less problematic solutions than fatigue have been used: timing the system out after a fixed number of steps, and non-deterministic action selection.

The Niche Genetic Algorithm

Most LCS employ a *panmictic* GA, that is, all classifiers are eligible for selection as parents. XCS, however, employs a niche GA, in which only a subset of classifiers participate in the process of selecting parents. In other words, the niche GA is a form of restricted mating scheme ([8], §G.5.3). The niche GA was first introduced by Booker [30, 31] as a way of focusing genetic search. Booker reasoned that since the rules in a match set were related (in that they apply to the same input), crossover among them would be more productive than crossover between unrelated rules. In XCS the GA originally operated upon [M] [298] but was moved to [A] when Wilson realised this was more effective [304]. The GA algorithm incorporating subsumption deletion (§2.3.6) appears in figure 2.5.

```
1.  Reset GA counters of rules in [A] (or [A]_{-1}).
2.  Select two parents P1 and P2 from [A] (or [A]_{-1}).
3.  Cross P1 & P2 with probability χ otherwise clone to obtain C1 & C2.
4.  Mutate each bit in C1 and C2 with probability μ.
5.  Initialise parameters of C1 and C2.
6.  Delete rules as needed.
7.  If C1 subsumed by P1 or P2 then
8.      Increment numerosity of subsuming parent.
9.  else
10.     Insert C1 into [P].
11. If C2 subsumed by P1 or P2 then
12.     Increment numerosity of subsuming parent.
13. else
14.     Insert C2 into [P].
```

Figure 2.5: The XCS rule discovery algorithm.

Selection The probability of a classifier being selected for reproduction is proportional to its fitness ([298] p. 8). In practice [306] Wilson and others have used roulette wheel selection (§G.2.4).

Genetic Operators Copies of the parent classifiers are generated and then transformed using standard genetic algorithm operators. One point crossover (§G.2.5) occurs with probability χ per pair of chromosomes (i.e., per pair of bitstrings) and point mutation (§G.2.5) occurs with probability μ per allele (i.e., per bit) ([298] p. 8). Crossover only occurs in the conditions, but mutation occurs in both the condition and action [53]. (Other forms of crossover have been used with XCS (e.g., uniform crossover [154]), with apparently little effect.)

Deletion If the population size (in *micro*classifiers) exceeds the limit N, sufficient classifiers are deleted to make room for the new ones ([298] p. 8).

Alternative Mutation Schemes

A feature of mutation which is not mentioned in the original XCS paper, and did not emerge until [307], is that Wilson does not allow 0 and 1 to mutate into each other. The algorithm for this biased mutation scheme is:

```
For each character, with probability μ
    If the character is # then
        Mutate it equiprobably to 0 or 1.
    else
        With probability μ_g mutate it to #.
```

The rationale behind disabling mutations between 0 and 1 is that in XCS the genetic algorithm is intended to search for the appropriate level of generality in conditions. Since mutations between 0 and 1 do not affect generality they would appear not to contribute to this search. The parameter μ_g controls the bias towards generality; higher values of μ_g make mutation towards #s more likely.

Unlike Wilson, the author has always simply mutated characters into each other with equal probability. A comparison (not shown) of the two approaches on the 6 multiplexer of §2.5.1 found that equiprobable mutation produced better performance and %[O] (§4.2.6) curves at the expense of a larger population size [305].

Clearly equiprobable mutation is less constrained in terms of the rules it can produce than Wilson's biased mutation. One hypothesis is that its advantage derives from a need, at least when an empty initial population is used, for genetic operators which help cover the input/action space quickly. This theory remains untested.

Butz and Wilson [53] (p. 15) describe a third mutation scheme:

```
For each character, with probability μ
    If the character is # then
        Mutate it to the corresponding character in the current input string.
    else
        Mutate it to #.
```

The author has not evaluated this third scheme, and the results reported with XCS in this work used equiprobable mutation exclusively.

Triggering the Niche GA

GA invocations are controlled by a scheme which aims to allocate them to different parts of the rule population (different [A]s) approximately equally.

> " ... the rate of incidence of the GA is controlled with the aim of allocating classifier resources approximately equally to the different match sets (such an allocation being consistent with the

> purpose of forming a relatively complete mapping). This cannot in general be achieved if the GA simply occurs with a certain probability in each match set. Depending on the environment, some match sets (niches) may occur much more often than others. Instead, the GA is run in a match set if the number of time-steps since the last GA in that match set exceeds a threshold. As a result, the rate of reproduction per match set per unit time is approximately constant – except in the most rarely occurring match sets." [298] p. 9.

To implement this scheme, each rule contains a counter which, when the rule is created, is set to the current time step. When [A] is formed, the mean difference between this counter and the current time step is computed, and the GA is triggered in that [A] if this value exceeds a threshold θ_{GA}. When the GA is triggered, the counters for all rules in [A] are then set to the current time step ([298] p. 9). In other words, the GA is triggered in [A] if:

$$\frac{\sum_{x \in [\mathrm{A}]} \left(t - x_{\mathrm{GA}}\right)}{|[\mathrm{A}]|} > \theta_{GA} \tag{2.10}$$

where t is the current time step, x_{GA} is the time step on which rule x was last in an action set in which the GA was invoked (or the time step of its creation if it has never been in such an action set), and $|[\mathrm{A}]|$ is the size of the action set.

Deletion of Rules

In the following quotation from [298], Wilson discusses two methods of selecting classifiers for deletion, which we label t1 and t2 for techniques 1 and 2.

t1. "Every classifier keeps an estimate of the size of the match sets in which it occurs. The estimate is updated every time the classifier takes part in an [M], using the MAM technique with rate β. A classifier's deletion probability is set proportional to the match set size estimate, which tends to make all match sets have about the same size, so that classifier resources are allocated more or less equally to all niches (match sets)."[6] [298] p. 9.

[6] An estimate of the size of [A] instead of [M] is kept if the GA operates in [A].

t2. "A classifier's deletion probability is as in (1), except if its fitness is less than a small fraction δ of the population mean fitness. Then the probability from (1) is multiplied by the mean fitness divided by the classifier's fitness. If for example δ is 0.1, the result is to delete such low-fitness classifiers with a probability 10 times that of the others." [298] p. 9.

A third scheme t3, a hybrid of the first two, was introduced in [147], and has since been adopted as the standard approach in XCS [53]. In this scheme, the deletion probability of a rule depends on how much experience it has. Rules with less than an experience threshold θ_{del} are treated as in t1, while those exceeding the threshold are treated as in t2. This scheme protects newly evaluated rules from the deletion penalty of t2 until they have been well evaluated, at which point useful rules should have gained enough fitness to escape the low fitness penalty.

Note that because XCS uses macroclassifiers, when a classifier with numerosity > 1 is deleted its numerosity is simply decremented by 1. Only when a classifier's numerosity reaches 0, i.e., when no more copies exist, is the data structure representing the classifier recycled.

Note also that all rules in [P] are eligible for deletion, i.e., selection for deletion is panmictic, although selection for reproduction is not.

Classifier Parameter Initialisation

Each of the three means of introducing new rules has its own method of determining the rule's initial settings. The aim in setting initial values is to make the best guess possible as to the true parameter value in order to improve system performance, however, the precise initialisation appears to have little effect on the system since XCS quickly update parameters away from their default values. The only significant rule of thumb is to avoid giving new rules large initial fitness, which would allow them to influence action selection and reproduction before they are well evaluated.

The Initial Population. If an initial population is used, the parameters are set to constant default values. Wilson has used the following fairly arbitrary values: 10 for prediction, 0.0 for prediction error and 0.01 for fitness [303]. (Wilson denotes these values p_I, ε_I, F_I.)

Covering. Classifiers created through covering are initialised as follows: initial prediction and prediction error are set to the population means and initial fitness is set to 0.1 times the mean fitness of [P] [303].

Genetic Algorithm. If crossover occurred, prediction is the mean of the parents' predictions. Otherwise, prediction is the same as the parent's prediction. Prediction error is always set to 0.25 times the mean error of [P] and fitness is set to 0.1 times the mean fitness of [P] [303].

Subsumption Deletion

Subsumption deletion was introduced in [304] as a way of biasing the system towards more general (but still accurate) classifiers. There are two forms of subsumption deletion [53]. In *GA subsumption*, when new rules are generated by the GA, they are checked against their parents. A parent subsumes its offspring if it is 1) experienced (has some minimum experience θ_{sub}), ii) accurate (has an accuracy of 1.0) and iii) logically subsumes the offspring (the inputs it matches are a superset of the inputs matched by the offspring). If these conditions are met, the offspring is discarded and the numerosity of the parent is incremented by 1.

> "...(subsumption deletion) may be viewed genetically as a kind of directed mutation. In effect, for parents "known" to be accurate, the GA is constrained to generate and evaluate only offspring that are more general than the parents." [304] p. 669.

Action set subsumption is a second form of subsumption deletion which occurs each time an action set is updated. The most general of the experienced and accurate rules in [A] is found, and any rules which it subsumes are discarded. The numerosity of the subsuming rule is incremented by the numerosity of the subsumed rules.

Wilson showed that the combined use of both forms of subsumption deletion can be quite effective in reducing the size of the population [304].

2.4 SB–XCS

In order to compare strength and accuracy-based fitness, SB–XCS, a strength-based LCS which differs as little as possible from accuracy-based XCS, was defined. SB–XCS was used rather than some other strength-based LCS because the similarity of XCS and SB–XCS allows us to isolate the effects of the fitness calculation on performance in a way we cannot with any other strength LCS. Other strength-based systems differ from XCS in many ways, such as how rule strength is calculated and how actions are selected, in addition to the obvious difference in fitness calculation.[7]

[7] SB–XCS was earlier called Goliath [150].

SB–XCS is not simply a straw man for XCS to outperform. It is a functional LCS, and is capable of finding optimal solutions to some tasks. SB–XCS is competitive with, for example, Wilson's Boole [293] and ZCS [297] systems, at least on the 6 multiplexer task we will use in §2.5.

SB–XCS's value is that we can study when and why it fails, and can attribute any difference between its performance and that of XCS to the difference in fitness calculation. Consequently we are more interested in comparing the qualitative performance on the two than their quantitative performance, and few attempts have been made to optimise SB–XCS.

2.4.1 Specification of SB–XCS

The following modifications were made to XCS to produce SB–XCS:

Rule Fitness

To update rule strength, SB–XCS uses the same Q-learning update XCS uses to calculate prediction, i.e., equation (2.2) for non-sequential tasks and equation (2.3) for sequential tasks. That is, the strength of a rule in SB–XCS is identical to the prediction of a rule in XCS; only the name differs. However, whereas XCS goes on to calculate prediction error and other parameters involved in fitness calculation, SB–XCS simply uses strength as the rule's fitness. This simplification of the XCS updates is in fact the only major difference between XCS and SB–XCS; a number of other modifications are necessary to make SB–XCS functional, but all are minor.

Note 1. Since (2.3) is the tabular Q-learning update, SB–XCS uses Q-values for both action selection and as rule fitness in the GA.

Note 2. XCS factors a rule's numerosity into its fitness using the relative accuracy update (as we noted on p. 45). Since SB–XCS does not use this update, its strength/fitness does not take numerosity into account. Consequently, we must explicitly factor numerosity into fitness, unlike in XCS.

Note 3. XCS's relative accuracy update provides fitness sharing. Since SB–XCS does not use this update, it does not implement fitness sharing. SB–XCS could, of course, be modified to add fitness sharing.

System Prediction and System Strength Calculations

In XCS, a rule's contribution to the system prediction for its action is a function of both its prediction and fitness (equation (2.1), p. 36). In SB–XCS, however, a rule's fitness is its strength, so there is no separate fitness parameter to factor into the calculation. We do, however, need to explicitly factor numerosity into the prediction, as explained in note 2 above.

Removing fitness from equation (2.1) and factoring in numerosity we obtain the *System Strength*:

$$S(a_i) = \sum_{c \in [\mathrm{M}]_{a_i}} p_c \cdot numerosity(c) \tag{2.11}$$

In preparation for action selection, SB–XCS constructs a system strength array using (2.11), just as XCS constructs a system prediction array using (2.1). Note, however, that the two differ in that the system strength (2.11) for an action is *not* a prediction of the reward to be received for taking it. For example, suppose that in a given state action 1 receives a reward of 1000, and that the only matching macroclassifier advocating action 1 has strength 1000 and numerosity 2. The system strength for action 1 is $P(a_1) = 1000 \cdot 2 = 2000$, twice the actual reward since there are two copies of the rule.

In order to estimate the return for an action, we must divide the system prediction by the numerosity of the rules which advocate it. For this purpose, we define the *System Prediction* in SB–XCS as:

$$\begin{aligned} P(a_i) &= \frac{S(a_i)}{\sum\limits_{c \in [\mathrm{M}]_{a_i}} numerosity(c)} \\ &= \frac{\sum\limits_{c \in [\mathrm{M}]_{a_i}} p_c \cdot numerosity(c)}{\sum\limits_{c \in [\mathrm{M}]_{a_i}} numerosity(c)} \end{aligned} \tag{2.12}$$

The system prediction is needed to calculate the target for the Q-update in sequential tasks (2.3).

In summary, whereas XCS uses system prediction for both action selection and the Q-update, SB–XCS uses system strength for the former and system prediction only for the latter.

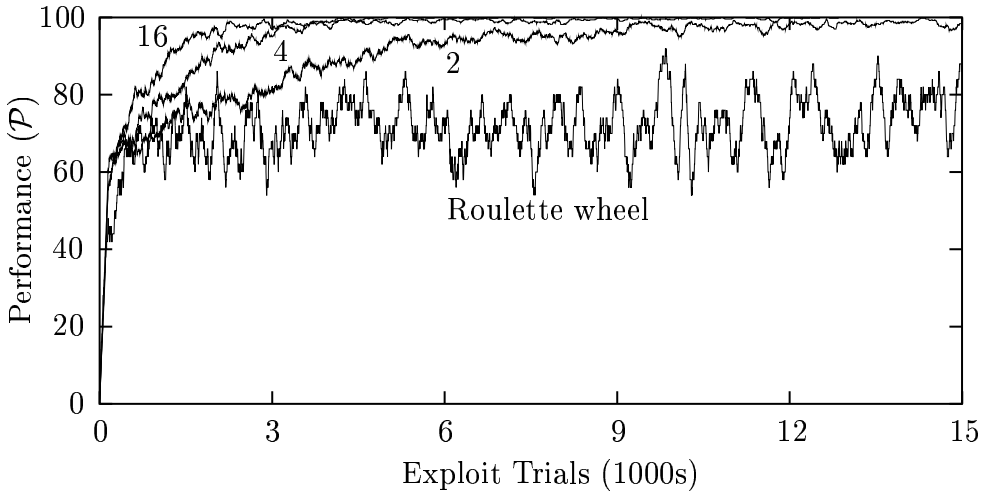


Figure 2.6: SB–XCS with roulette and tournament selection on the 6 multiplexer.

Subsumption Deletion

SB–XCS does not use either form of subsumption deletion, since these techniques make reference to a rule's accuracy. Although it might be possible to develop useful forms of subsumption deletion for strength-based systems, we will not investigate this here.

Selection for Reproduction

In preliminary tests with the 6 multiplexer function (to be introduced in §2.5.1) it was found that SB–XCS's rule population was swamped with overgeneral rules, and that the rules with the greatest numerosity were those with fully general conditions. Reasoning that the generalisation pressure due to the niche GA was overwhelming the selective pressure in reproduction, the roulette wheel selection inherited from XCS was replaced with tournament selection (see [8]), in which the degree of selective pressure is parameterised by the size of the tournament. Figure 2.6 shows SB–XCS's performance using roulette wheel selection and tournament selection with tournament sizes of 2, 4, and 16. Curves are averages of 10 runs, and a full explanation of the experimental procedure (including the performance metric $\mathcal{P}$) is given in §2.5.1.

It should be noted that in both roulette and tournament selection, SB–XCS factors a rule's numerosity into its probability of selection.

Selection for Deletion

Using tournament selection SB–XCS evolved accurate, general rules for the 6 multiplexer, and achieved good performance. It was noted, however, that numerous overgeneral rules survived in the population, despite the selective pressure against them. For simplicity the preliminary tests employed random deletion, and, in order to introduce further pressure against overgeneral rules, this was replaced with tournament selection. The rule with the lowest strength of those in the tournament was deleted. This modification reduced the number of overgenerals in SB–XCS's population and improved performance on the 6 multiplexer slightly.

Summary

These minor differences make the implementation of SB–XCS very close to that of XCS, which allows us to easily convert an implementation of XCS into SB–XCS. More importantly, the near identity of the algorithms allows us to attribute differences in their capacities to the difference in the fitness calculation. We will see that these minor differences – mainly the change in fitness calculation – have a major effect on how the system operates and on its capacities.

2.4.2 Comparison of SB–XCS and Other Strength LCS

SB–XCS makes no attempt to encourage the formation or survival of default hierarchies, although it is not unique among strength LCS in this respect. (Wilson's ZCS [297] and Boole [293] systems, for example, do not encourage them either.) The resemblance of SB–XCS and Wilson's ZCS is particularly strong, as indicated in the following section.

SB–XCS and ZCS

ZCS was intended as a minimalist strength-based LCS, but was in many ways a precursor to XCS, which accounts for its similarity to SB–XCS. Nonetheless, the two differ in a number of ways:

- Most significantly, SB–XCS does not employ fitness sharing, whereas ZCS does.
- ZCS employs a different strength update rule ([297] p. 6). (See appendix H for ZCS's update.)

- ZCS does not employ a niche GA ([297] p. 7), though Wilson suggests a niche GA as an extension ([297] p. 21).[8]
- ZCS employs a deletion scheme in which deletion probability is proportional to the inverse of a rule's strength ([297] p. 7), which is similar but not identical to SB–XCS's tournament deletion.
- ZCS employs a tax intended to encourage it to choose the same action consistently in a given state ([297] p. 6).
- In ZCS, GA invocation occurs with a fixed probability on each cycle ([297] p. 7).
- In ZCS, parents give half their strength to their offspring when they are created ([297] p. 7).

2.5 Initial Tests of XCS and SB–XCS

In this section we briefly apply XCS and SB–XCS to two problems from the LCS literature: the 6 multiplexer (a non-sequential task), and Woods2 (a sequential task), which provide simple abstractions of the types of tasks illustrated, respectively, by the mushroom classification and frog simulation tasks in chapter 1.

Neither the 6 multiplexer nor Woods2 is very difficult, but they are the primary tasks to which XCS (which is still relatively new) has been applied in the literature, and, in any case, they provide a basic test of SB–XCS and an opportunity to compare it to XCS.

Wilson's explore/exploit framework (§2.3.3) is used with both tasks. Although it is less than fully satisfactory for sequential tasks in general, it has been used with XCS and Woods2 in the past [298]. The framework's inefficient exploration should not be a major problem as the number of steps to the goal tend to be relatively small, and the problem of getting stuck in loops is dealt with by simply ending episodes if they reach 100 time steps.

2.5.1 The 6 Multiplexer

The first task to which we apply the two systems is the venerable 6 multiplexer function, the most widely used test in the LCS literature [293, 228, 294, 33, 94, 29, 285, 172, 66, 98, 99, 65, 298, 128, 144, 146, 304, 62, 147,

[8] Page numbers refer to the electronic version of [297] which is available on the net.

14, 50, 52, 151]. It has also been used with other machine learning systems including neural networks [16, 7, 139, 29, 244], perceptrons [295], decision trees [205, 192], and the GPAC algorithm [190]. See [172] for a review of some of the earlier work using the multiplexer.

Definition

The 6 multiplexer is one of a family of Boolean multiplexer functions defined for strings of length $L = k + 2^k$ where k is an integer > 0. The series begins $L = 3, 6, 11, 20, 37, 70, 135, 264, 521 \ldots$. The first k bits are used to encode an address into the remaining 2^k bits, and the value of the function is the value of the addressed bit. In the 6 multiplexer ($k = 2, L = 6$), the input to the system consists of a string of six binary digits, of which the first $k = 2$ bits (the address) represent an index into the remaining $2^k = 4$ bits (the data). E.g., the value of 101101 is 0 as the first two bits 10 represent the index 2 (in base ten) which is the third bit following the address. Similarly, the value of 001000 is 1 as the 0th bit after the address is indexed.

Use as an RL Task

To use the 6 multiplexer as a test, on each time step we generate a random binary string of 6 digits which we present as input to the LCS. The LCS responds with either a 0 or 1, and receives a high reward (1000) if its output is that of the multiplexer function on the same string, and a low reward (0) otherwise.

Measuring Performance

In Wilson's explore/exploit framework training and testing are interleaved, and the performance of the learner is evaluated on-line, that is, as it is learning, rather than after it has been trained. Each time the system is presented with a new input we take the opportunity of evaluating its response and updating our statistics regarding its performance.

Wilson defines a measure of performance which he refers to simply as "performance" [298], but which we'll refer to as $\mathcal{P}$ in order to distinguish it from the more general notion of performance. $\mathcal{P}$ is defined as a moving average of the proportion of the last n trials in which the system has responded with the correct action, where n is customarily 50. That is, on each time step, we determine the proportion of the last n time steps on which the LCS has taken the correct action. The $\mathcal{P}$ curve is scaled so that when the system has acted correctly on all of the last 50 time steps it reaches the top of the

Parameter		Value	See Page
Subsumption threshold	θ_{sub}	20	52
GA threshold	θ_{GA}	25	50
t3 deletion threshold	θ_{del}	25	51
Covering threshold	θ_{mna}	1	46
Low-fitness deletion threshold	δ	0.1	51
Population size limit	N	400	31
Learning rate	β	0.2	42
Accuracy falloff rate	α	0.1	43
Accuracy criterion	ε_o	0.01	43
Crossover rate	χ	0.8	48
Mutation rate	μ	0.04	48
Hash probability	$P_\#$	0.33	46

Figure 2.7: Standard XCS parameter settings for the 6 multiplexer.

figure, and when it has acted incorrectly on all these time steps it reaches the bottom of the figure.

In addition to $\mathcal{P}$, we'll monitor the number of *macro*classifiers in the population on each time step, which gives us an indication of the diversity in the rule population. In the following tests, this value is divided by 1000 in order to display it simultaneously with the $\mathcal{P}$ curve.

The macroclassifier curve initially starts at 0 since we start the LCS with an empty rule population and use covering (§2.3.6) to generate initial rules. This curve can at most reach the population size limit, which would occur when each rule in the population had a unique condition/action pair.

Further consideration is given to metrics for use with classifier systems in chapter 4.

Parameter Settings

The standard XCS parameter settings for the 6 multiplexer used since [298, 304] were used with XCS. See figure 2.7. SB–XCS used the relevant subset of these settings, and used tournament sizes of 32 for selecting parents and 4 for selecting rules to delete. These two values were chosen after a brief comparison of alternative settings – no serious attempt was made to optimise them.

For this test, neither system used action set covering, instead using the original match set covering (§2.3.6). XCS was in fact tested twice, once with

neither form of subsumption, and once with GA subsumption only.[9] SB–XCS used neither form of subsumption deletion since it is incapable of doing so (p. 55).

Results and Discussion

Figure 2.8 shows the $\mathcal{P}$ and population size curves for XCS and SB–XCS on the 6 multiplexer, averaged over 10 runs. The number of exploit trials is shown on the X-axis; recall from page 34 that we alternate explore and exploit trials so that the LCS has actually seen twice as many inputs as indicated, although it has only used the indicated number for learning.

Performance: $\mathcal{P}$ We can see that the $\mathcal{P}$ curves converge stably to the top of the figure in all three cases, suggesting the entire input/output mapping was successfully learnt. Although SB–XCS takes somewhat longer than XCS to initially reach the top, and even longer to stabilise there, we are not concerned with slight differences in performance here. What does concern us is that they both solve the problem, and do so in roughly the same amount of time.

Macroclassifiers SB–XCS's macroclassifiers curve is the most steady of the three, showing a very gradual downward trend following the initial steep increase. With GA subsumption, XCS's curve shows a much steeper decline following the initial increase, and towards the end it has a much smaller population size than SB–XCS. Without subsumption, however, XCS's curve declines only very gradually, and its population size eventually almost exactly equals SB–XCS's.

These results suggest that while XCS with GA subsumption quickly converges on a relatively small number of useful rules, in the other two cases the population contains a large number of redundant (overly specific) and overgeneral rules. Inspection of the populations evolved by SB–XCS and XCS with subsumption bears this out. (The evolved populations are shown in §3.5.)

Conclusions There are two points to note. First, XCS's somewhat better $\mathcal{P}$ may be due to the larger population it allows itself in the early phase of the test. Second, XCS's ability to represent the solution with few rules is due to subsumption deletion rather than to its accuracy-based fitness.

[9] In experiments in later chapters GA subsumption will be used and action set subsumption will not.

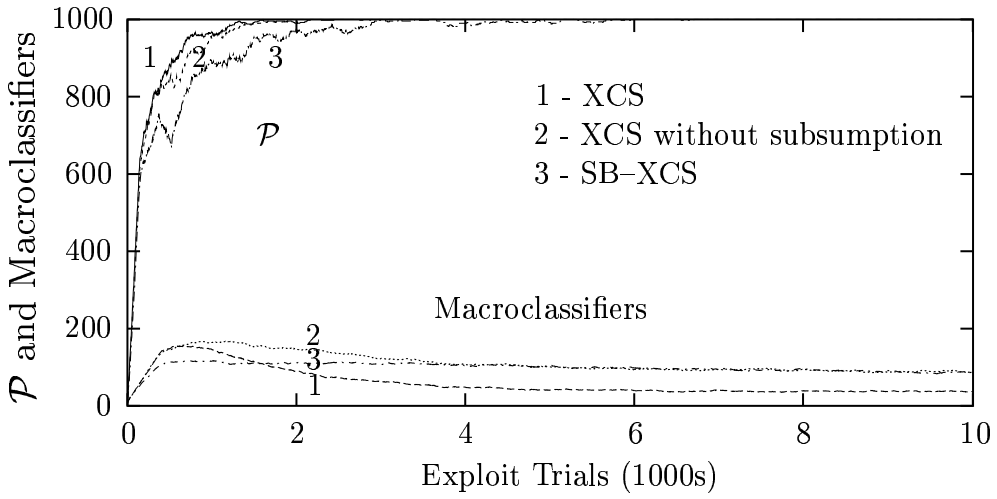


Figure 2.8: XCS and SB–XCS on the 6 multiplexer.

No major attempt was made to optimise SB–XCS, and improvements to it are likely possible. However, the results of this test indicate that it is a functional classifier system, and in fact compares favourably with other strength-based LCS on this task. Its performance on the 6 multiplexer is much superior to that of SCS ([94] p. 256), and superior to Wilson's BOOLE [293], which reached approximately 94% performance after a little more than 5000 time steps.

Bonelli et al.'s NEWBOOLE [29] outperformed SB–XCS on this task, but NEWBOOLE is a supervised learning system, so it faces an easier task than SB–XCS does (see §5.13). Non-LCS approaches, e.g., C4 [205] and GPAC [190], have been shown to solve this task more quickly than either XCS or SB–XCS (see [172] for a survey).

2.5.2 Woods2

Woods2 is a Markov sequential decision task (see §C.5.1) introduced in [298], designed to allow generalisation over parts of the input space. The environment consists of a two-dimensional gridworld containing two types of rock (Q and O), two types of food (F and G) and empty cells (.). Its top and bottom and left and right edges are connected (i.e., it is toroidal).

The classifier system is used as the control system for an animat which acts in this environment, and whose goal is to reach food in as few steps as possible. On each time step, input to the classifier system consists of a

```
.............................
.QQF..QQF..OQF..QQG..OQG..QQF.
.OOO..QOO..OQO..OOQ..QQO..QQQ.
.OOQ..OQQ..OQQ..QQO..OOO..QQO.
.............................
.............................
.QOF..QOG..QOF..OOF..OOG..QOG.
.QQO..QOO..OOO..OQO..QQO..QOO.
.QQQ..OOO..OQO..QOQ..QOQ..OQO.
.............................
.............................
.QOG..QOF..OOG..OQF..OOG..OOF.
.OOQ..OQQ..QQO..OQQ..QQO..OQQ.
.QQO..OOO..OQO..OOQ..OQQ..QQQ.
.............................
```

Figure 2.9: The layout of Woods2.

24-bit string representing the contents of the 8 cells immediately adjacent to the animat, starting with the cell to the North and moving clockwise. Each cell is coded using 3 bits as follows: F = 110, G = 111, O = 010, Q = 011, and empty cells = 000. The agent can move by 1 cell in one of 8 directions, with action 0 taking it to the North, 1 to the North East, and so on clockwise through action 7 to the North West. Actions are coded as 3-bit binary strings.

State transitions are deterministic and the animat's actions always have the intended effect, except that attempts to move into rock result in no change in position. An episode consists of placing the animat in a randomly chosen blank space (.) and allowing it to move until it reaches food, at which point the episode ends. Woods2 is shown in figure 2.9.

The animat receives a reward of 1000 when it takes an action which brings it to a food state (F or G), and 0 at all other times. Wilson discusses Woods2 and XCS's performance in it at some length in [298].

How Hard is Woods2?

The animat is only ever at most 3 steps from food, so long sequences of actions are not needed to solve the problem; although it is a sequential task, the sequences which must be learnt are quite modest! Acting randomly, the mean number of steps to reach food is 27, while acting optimally the mean is approximately 1.7 steps.

Parameter		Value	See Page
Subsumption threshold	θ_{sub}	20	52
GA threshold	θ_{GA}	25	50
t3 deletion threshold	θ_{del}	25	51
Covering threshold	θ_{mna}	1	46
Low-fitness deletion threshold	δ	0.1	51
Population size limit	N	800	31
Learning rate	β	0.2	42
Accuracy falloff rate	α	0.1	43
Accuracy exponent	v	5	43
Accuracy criterion	ε_o	0.01	43
Crossover rate	χ	0.5	48
Mutation rate	μ	0.01	48
Hash probability	$P_\#$	0.5	46

Figure 2.10: Standard XCS parameter settings for Woods2.

In order to provide a baseline for comparison, a tabular Q-learning system was also tested on Woods2. (See §6.1.3 for details of how the Q-learner was implemented.) Results for all three appear later in this section. We return to the subject of Woods2's difficulty in §5.12.5.

Measuring Performance

Following Wilson [298], for Woods2 we monitor the number of macroclassifiers in the population on each time step and record a moving average of the number of time steps taken to reach food during the last 50 exploit episodes.

Parameter Settings

The settings for XCS in Woods2 [301] from the original XCS paper [298] were used for all three systems; XCS, SB–XCS and tabular Q-learning, although the last only uses a small subset of them. See figure 2.10. In addition, SB–XCS used tournament sizes of 16 for reproduction and 4 for deletion. XCS and SB–XCS both used action set covering (§2.3.6), although more covering and less improvement result when it is used with SB–XCS than with XCS. XCS used GA subsumption but did not use action set subsumption (§2.3.6) as this was found to reduce performance on this task.

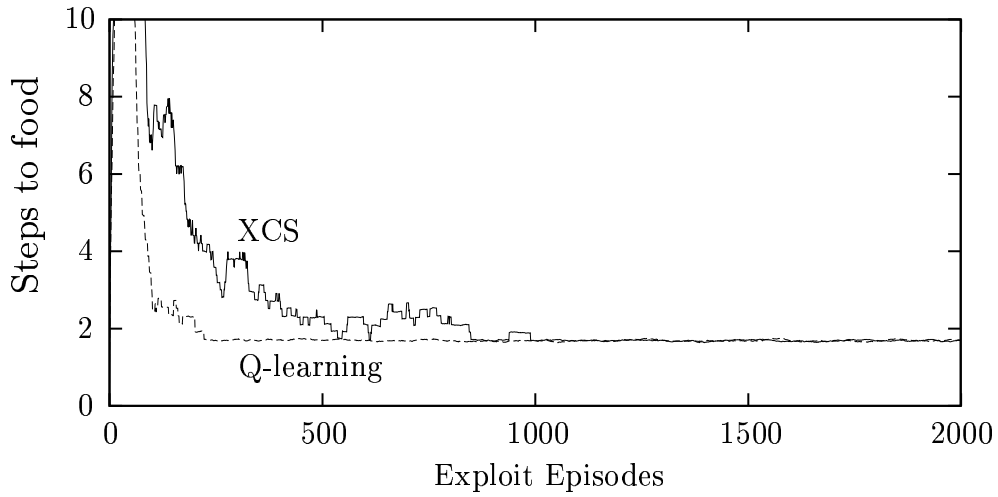


Figure 2.11: Performance of XCS and tabular Q-learning on Woods2.

Results and Discussion

XCS and Q-learning Figure 2.11 shows the mean steps to food for XCS and tabular Q-learning, averaged over 10 runs. Both converge to optimal performance (≈ 1.7 steps), although XCS takes longer to do so. Figure 2.12 shows the number of macroclassifiers in XCS's population, and the number of state-action pairs for which the Q-learner estimated Q-values. (Because the input space was immense (2^{24} syntactically possible strings), the Q-learner only allocated state-action pairs as they were needed, accounting for the initial rise in the curve.) XCS is quite effective at generalising in this task, and after 2000 episodes has less than a third as many macroclassifiers as the tabular Q-learner has state-actions.

SB–XCS It was found that SB–XCS's performance was very poor on Woods2. However, after some investigation this was attributed to the use of Wilson's explore/exploit framework. On exploit episodes, when the most highly advocated action is always selected, the system often gets stuck in loops and only ends the episode when it times out after 100 time steps. Because the steps-to-food curve is a moving average of the last 50 exploit episodes, and because results are averaged over 10 runs, this makes performance appear very poor, although the system in fact often behaves optimally for considerable periods.

Although XCS and the tabular Q-learner also often timed out after 100 time steps, they did so only at the outset so later performance was not

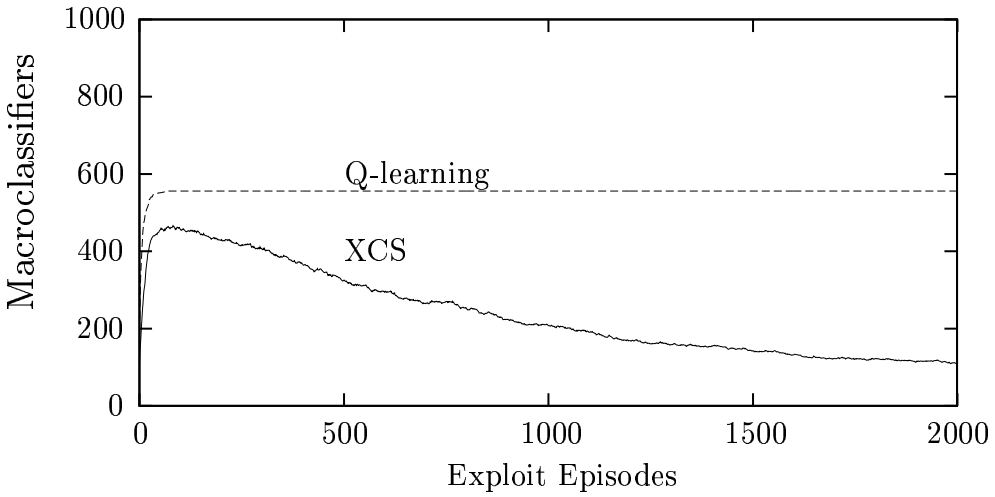


Figure 2.12: Population size of XCS and tabular Q-learning on Woods2.

affected by this problem. Because SB–XCS continued to time out throughout the run, its performance appeared worse than it really was.

To avoid the confounding effect of loops, Wilson's framework was abandoned and much better results were obtained using ϵ-greedy action selection, in which the system selects a random action with probability ϵ and the most highly advocated action otherwise. For the first 10,000 episodes ϵ was 0.1, during which the occasional random actions allowed SB–XCS to break out of loops and avoid timing out (apart from during after the initial period of adaptation). As shown in figure 2.13, SB–XCS achieved an average of approximately 3 steps to food, which is just how ZCS performed on the related Woods1 task [297], and a little less than twice the optimal ≈ 1.7 steps to food.

Unfortunately, the occasional random actions increase SB–XCS's average steps to food and so slightly complicate the assessment of the quality of its policy. However, we can estimate that since the random actions only added, on average, 1 step to every 10 (since $\epsilon = 0.1$), and since SB–XCS averaged approximately 3 steps to food, the contribution of occasional random actions should be approximately 1/3 of a step per episode.

In order to evaluate the policy learnt by SB–XCS without the effect of exploration, after 10,000 time steps ϵ was set to 0, meaning it never acted randomly. Although SB–XCS was still able to complete episodes in a few steps on some occasions, it also sometimes got stuck in loops, with episodes timing out after 100 time steps. With $\epsilon = 0$ the average steps to food jumped

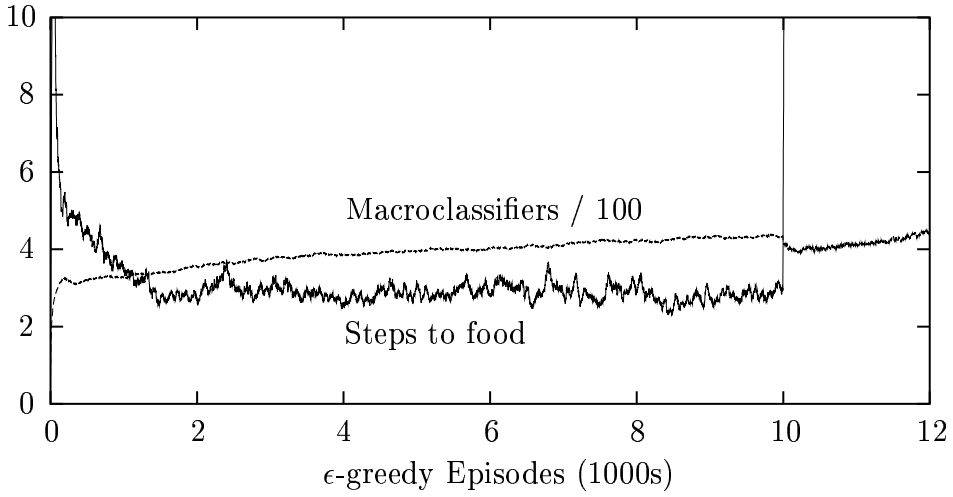


Figure 2.13: SB–XCS on Woods2 with $\epsilon = 0.1$ for 10,000 episodes and $\epsilon = 0$ thereafter.

to approximately 28, as a result of averaging episodes which timed out with those on which SB–XCS was more successful. This indicates that SB–XCS was relying on the occasional random actions to get it out of loops, and that it had not learnt a suitable policy over the entire environment.

SB–XCS was also evaluated on Woods1, in which its performance was almost identical to that in Woods2. It would be interesting to see if ZCS behaved the same way in Woods1 in the absence of the non-determinism afforded by its roulette action selection.

2.6 Chapter Summary

This chapter has given detailed specifications of an accuracy-based classifier system called XCS and a strength-based system called SB–XCS, which are as similar as possible while making use of the two different approaches to fitness calculation.

Empirical evaluation of the two found that although both can solve the non-sequential 6 multiplexer task, only XCS was able to achieve optimal performance on the sequential Woods2 task. Results such as these have long plagued classifier systems research. Why was SB–XCS much more able to adapt to one task than the other? That is, what is it about Woods2 that SB–XCS finds difficult? What part or parts of SB–XCS have this difficulty?

Unfortunately there is no broad theory which predicts when a given clas-

sifier system will be able to adapt to a given task. In the absence of theory, the only way to find out how (and if) a classifier system will solve a given task is to try it out. To provide such theory, we need an understanding of what types of task and what types of classifier system are possible, and how they interact. This work addresses both questions to some extent. Although we have already distinguished strength and accuracy-based classifier systems, and categorised the tasks to which we might apply them as either sequential or non-sequential, we need to go deeper to provide a predictive theory.

In the next chapter we will analyse XCS and SB–XCS in some detail in an attempt to understand the differences between them. Ultimately, we will try to explain why SB–XCS adapted poorly to Woods2, although this will take most of the thesis. We will also return to the 6 multiplexer and in §3.5 analyse the populations evolved by the two systems.

Chapter 3

How Strength and Accuracy Differ

Chapter Outline

We begin this chapter with a short discussion of the difficulties of working with and understanding complex systems such as classifier systems, in which we suggest that many models or paradigms of a complex system may need to be considered before we arrive at the most suitable ones. Following this we consider a number of models of classifier systems. We review the rationale behind Holland's classifier systems and XCS, as presented by their authors. We do not engage in detailed discussions of their ideas, but rather attempt to state in simple terms why they expect these systems to learn. The main point of this part of the chapter will be that strength and accuracy-based systems differ fundamentally in how they solve problems.

In the second half of the chapter we consider the differences in how XCS and SB–XCS represent solutions. We begin in §3.5 with inspection of the solutions evolved by the two for the 6 multiplexer task. Next, we distinguish the representations towards which the two tend (§3.6), and compare them (§3.7). Finally, in §3.8 we compare the ability of the two to express fit generalisations over their inputs.

3.1 Thinking about Complex Systems

Classifier systems are complex systems. It is easy to build complex systems and not fully understand them, because their components often interact in unforeseen ways, yielding unforeseen behaviours.[1] Even systems which are simple to describe can show complex behaviour. In the terminology of Kolmogorov complexity, we could say such systems have low algorithmic complexity, but their behaviour has high structural complexity [171].

Coming to Understand Complex Systems

In trying to understand a complex subject we build some form of mental model of it. Often our models are approximations and correspond more or less well to reality. Often it is possible to produce quite different models of the same subject, and sometimes more than one of them will fit reality reasonably well. Sometimes two different models both fit quite well, as in the case of the forward and backward views of eligibility traces in temporal difference learning [265].

We update our approximate models as we learn more about a subject, and so our thinking about the subject typically changes over time. To make things easier we seem to often rely initially on analogies with more familiar subjects. For example, neural networks were originally conceived as biological metaphors and, among other things, ways of modelling biological processes. Now, however, the biological analogy is less in favour and they tend to be described as statistical algorithms.

As another example, the bucket brigade was conceived using an economic metaphor, with agents (rules) buying and selling information (messages) [123]. Interest in this metaphor seems to have waned as people became more concerned with producing working systems than with sustaining the metaphor. Now we recognise that bucket brigades are instances of the more general class of temporal difference methods, and tend to use these methods without giving much thought to economics.

Interestingly, in both cases we've moved from analogies with familiar subjects (economics, neurons), to a deeper understanding of the subject and

[1] This suggests any design methodology for complex systems should take into account the difficulty of getting them to behave as intended. We should certainly not design complete complex systems on paper and expect them to function as intended! Rather, we should take the animat path [293], building complexity incrementally and evaluating our systems at each step. Extensive evaluation is necessary because of the particularly complex design and niche spaces [236, 237, 238, 239] of such systems.

its own intricacies. Perhaps the same will happen to the biological metaphors in evolutionary computation. EC is, of course, only broadly based on the principles by which biological systems operate.

This chapter is primarily concerned with explaining the broad principles of how two very different types of classifier systems work, and with some of the implications of these differences. We proceed by presenting in the next two sections rather general and vague descriptions of models of adaptation in Holland's LCS and Wilson's XCS.

3.2 Holland's Rationale for CS-1 and his Later LCS

Cognitive System 1 (CS-1) [127], implemented in 1978, was the first classifier system. CS-1 was clearly built around Holland's earlier creation, the Genetic Algorithm, learning process', a credit assignment system, which serves to provide a fitness function for the GA.

3.2.1 Schema Theory

Holland's rationale for the adaptation of CS-1 (and subsequent LCS) was the effectiveness of the GA [127, 123, 126, 40], which he formalised with his schema theory, first formulated in [119]. Holland's schema theory applies to GAs in general and it is notable that he never specialised it to take into account the special conditions found in LCS.

We might describe Holland's schema theory as an appeal to mathematical intuition which suggests that rules with above-average fitness will propagate while those with below-average fitness will die out. The schema theory has been the subject of both considerable criticism (see [182]) and considerable development (see [160] and, e.g., [198, 200, 199, 201, 197, 196, 178]). Regrettably, discussion of the schema theory and the controversy surrounding it is far beyond the scope this work. So too is discussion of many of Holland's other ideas relating to LCS, such as tags, rule coupling, and the details of the bucket brigade (see, e.g., [126]) although none of these seems essential to the basic concept of a classifier system.

The Effect of the Genetic Algorithm

Although interpretation of the schema theory remains controversial and beyond our scope, we can nonetheless say something about the effect of the GA. The GA involves two processes: reproduction based on selection, and

genetic manipulation of selected individuals. Without making claims about the efficiency of the GA, its capacity to solve given problems, or converge on optima, we can say that it *tends* from one generation to the next to produce individuals with higher fitness. That is, of course, a very vague statement, and subject to all sorts of exceptions, but it is sufficient for the purposes of this chapter. That is, in seeking a rationale for the design of an LCS, we can rely on the GA as a plausible general-purpose optimisation device, which will *tend* to provide us with fitter rules, subject to whatever definition of fitness we give it.

3.2.2 The Bucket Brigade

The GA reproduces above-average rules preferentially, but for the LCS to adapt it must assign higher fitness to more desirable rules. This is the task addressed by the credit assignment system. For credit assignment CS-1 employed an *epochal algorithm* which is not fully specified in [127], and which found very limited use in LCS because Holland later introduced the *Bucket Brigade* as a replacement for it [126]. (Grefenstette [100] provides an empirical comparison of the Bucket Brigade with the Profit Sharing Plan (PSP), a modified version of CS-1's scheme.)

Various forms of Bucket Brigade have been employed (see, e.g., [126, 94, 294, 247]), and the strength update (2.4) adopted for use with SB–XCS can be seen as a simplified form of Bucket Brigade.

The Bucket Brigade is generally described using an economic metaphor, in which rules buy from and sell to each other (see, e.g., [126] p. 72). Regardless of whether we think in terms of the economic metaphor or not, the intention is that the Bucket Brigade will assign higher strengths to better rules.

The strength of a rule is meant to correlate with the performance of the system as a result of the rule's use. More precisely, Holland sees a rule's strength as an estimate of the probability that the rule will advocate an appropriate action. It is perhaps easiest to consider these issues in the context of the task of assigning inputs to classes:

> "... the operation of the basic mechanisms for strength revision will, under a broad range of conditions, tend to refine strength values to be proportional to the rule's probability of success [at correctly categorising its input]." [126] p. 52.

In other words, the more often a rule takes the correct action, the greater its strength will be. In the simplest case a rule which is always correct would have maximal strength, and a rule which is always incorrect would

have minimal strength, although this is not implied by Holland's statement above.

3.2.3 Schema Theory + Bucket Brigade = Adaptation

If the bucket brigade assigns higher strength to more successful rules, and the GA reproduces these rules preferentially, we have a rationale for the adaptation of the classifier system.[2] This statement of Holland's rationale has been vague and oversimplified, but is sufficient for the purpose of contrasting it with Wilson's rationale for XCS.

3.3 Wilson's Rationale for XCS

Rather than the Bucket Brigade, XCS employs a related algorithm called Q-learning to evaluate rules. With this substitution, the basic rationale for Holland's LCS presented in the last section applies equally to XCS; we could argue that XCS should adapt to tasks thanks to the combined action of Q-learning and schema theory.

XCS, however, operates on different principles from Holland's LCS, and Wilson emphasises these in [298], in which, in fact, schema theory is never mentioned, the power of the GA being taken for granted. Rather, Wilson's rationale for XCS focuses on the following. First, a clear emphasis on two complementary pressures: a pressure towards accurate rules, and a pressure towards general rules. Second, XCS explicitly seeks to evolve complete maps of the input/action space. We discuss each of these in turn.

3.3.1 A Bias towards Accurate Rules

XCS is very strongly biased towards reproducing accurate rules, thanks to its accuracy-based fitness calculation (§2.3.5) which severely penalises rules whose prediction errors exceed some threshold (see equation (2.5)).

In hindsight a bias towards accuracy is clearly an important property for an LCS. Although a number of classifier systems have incorporated some bias towards accuracy, it has generally been under-emphasised. To see the importance of accuracy, consider that the performance of an LCS depends heavily on its ability to find accurate rules. With a population of overgeneral rules, an LCS will necessarily act incorrectly, and its performance will depend

[2] In fact, this makes the assumption that reproducing successful rules will yield a successful population, which as we'll see in chapter 5 is not always the case.

on how overgeneral its rules are. (Unless the system contains functional default hierarchies which compensate for the overgenerality of some rules.)

3.3.2 A Bias towards General Rules

Wilson's rationale for XCS's success in finding accurate, general rules is expressed in his *generalisation hypothesis*:

> "Consider two classifiers C1 and C2 having the same action, where C2's condition is a generalization of C1's. That is, C2's condition can be generated from C1's by changing one or more of C1's specified (1 or 0) alleles to don't cares (#). Suppose that C1 and C2 are equally accurate in that their values of ε [prediction error] are the same. Whenever C1 and C2 occur in the same action set, their fitness values will be updated by the same amounts. However, since C2 is a generalization of C1, it will tend to occur in more match sets than C1. Since the GA occurs in match sets,[3] C2 would have more reproductive opportunities and thus its number of exemplars would tend to grow with respect to C1's (or, in macroclassifier terms, the ratio of C2's numerosity to C1's would increase). Consequently, when C1 and C2 next meet in the same action set, a larger fraction of the constant fitness update amount would be "steered" toward exemplars of C2, resulting through the GA in yet more exemplars of C2 relative to C1. Eventually, it was hypothesized, C2 would displace C1 from the population." [298] p. 13.

We can express this argument more succinctly as follows:

> Given two equally accurate rules, one of which subsumes the other, the more general will gain more numerosity because it participates in more reproductive events. Eventually the more general should displace the less general from the population.

The generalisation hypothesis is thus a reference to the niche GA, and its bias towards more general rules, which is indeed the source of generalisation pressure in XCS. Recall that if the GA operates in the action set, only rules in the current action set are eligible to participate in the stochastic selection of parents. It follows that rules which occur in more action sets – i.e., more general rules – will have more opportunities to reproduce. We can think of

[3] In [298] the GA occurred in [M]; it was moved to [A] in [304].

the niche GA as effectively producing an implicit fitness bias towards more general rules; that is, such rules effectively have a higher fitness.

As Wilson noted ([298] p. 14), for the niche GA's purposes (and so those of the generalisation hypothesis) generality is defined as the number of inputs a rule matches, rather than its syntactic generality (number of #s a rule's condition contains).

Interpretation and Extension of the Generalisation Hypothesis

Although the generalisation hypothesis refers strictly to rules in a subsumptive relationship, it has often been used as a rationale for XCS's entire generalisation ability (e.g., [304]). Has the generalisation hypothesis been used to justify unwarranted claims?

Happily, the generalisation bias applies to *all* rules, not just those in the subsumption relationship to which the generalisation hypothesis explicitly refers. Consequently, we can attempt to restate the generalisation hypothesis in slightly more general terms:

> The more general of two equally accurate rules will gain more numerosity because it participates in more reproductive events.

Note, however, that in this case we cannot state that the more general rule will displace the less general from the population, since this will only occur if the two compete for reproduction. For example, if the two never occur in the same action set, the more general cannot drive the less general from the population.

Clearly the generalisation bias does not apply only to equally accurate rules; generality confers its benefits even on less fit rules, although its effect is tempered by fitness. We could, therefore, attempt to extend the generalisation hypothesis further to deal with all rules, and the schema theory approach we will see is one way to do so.

The Generalisation Pressure of the Niche GA

How strong is the generalisation pressure introduced by the niche GA? Let's make some simplifying assumptions. The niche GA confers its generality bias based on a rule's effective generality, that is, the number of inputs a rule matches. It is, however, easier to deal with a rule's formal generality, the number of #s in its conditions. Let's assume inputs occur with equal probability, in which case the two coincide in the limit, and we can use the formal generality of a rule to determine the expected proportion of reproductive events in which it occurs.

To avoid complications due to allocation of reproductive events by equation (2.10), let's assume $\theta_{GA} = 1$, i.e., that the GA operates on each time step. Let's also assume that either the GA operates in [M], in which case a rule participates in a reproductive event each time it matches, or that the GA operates in [A] and that actions are chosen at random, so that the proportion of reproductive events a rule participates in does not depend on its action. Given these assumptions, the expected proportion of reproductive events in which a rule participates is:

$$\frac{2^g}{2^l} \tag{3.1}$$

where g is the formal generality and l is the length of the rule's condition. (Recall that we assume uniform probability for all input strings.)

For example, a fully specific rule ($g = 0$) of length $l = 6$ is eligible for selection in $2^0/2^6 = 1/64$ of all reproductive events, while a fully general rule ($g = 6$) is eligible in all events. That is, a fully general 6-bit rule enjoys 64 times as many reproductive opportunities as a fully specific 6-bit rule, regardless of their respective fitness.

Equation (3.1) makes it clear that the niche GA's bias towards general rules can be very strong, and that, because the denominator in (3.1) increases with l, the degree of bias increases with the length of the condition. In general, a fully general l-bit rule participates in 2^l more selective trials than a fully specific l-bit rule.

Equation (3.1) suggests that, for longer strings, a classifier system must be able to make very fine distinctions between overgeneral and not-overgeneral rules in order to counteract the pressure towards generalisation. That is, the system must assign sufficiently higher fitness to not-overgeneral rules, and must apply sufficient selective pressure towards such rules. Otherwise, the population risks being taken over by overgenerals.

A Schema Theorem for a Niche GA

Equation (3.1) can be added to an LCS schema theory as an extra term which accounts for the niche GA. A sketch of one approach follows, in which we treat the reproduction of classifiers as schema theory treats the reproduction of schemata. If we drop the terms which account for the effects of genetic operators in a schema equation, we have a form of reproductive schema growth equation, which we can express for classifiers in the notation of [94] as:

$$E\Big[m(h,t+1)\Big] = m(h,t)\frac{f(h,t)}{\bar{f}(t)} \cdot \frac{2^g}{2^l} \tag{3.2}$$

where $m(h,t)$ is the numerosity of rule h at time t, $f(h,t)$ is the fitness of h at time t, and $\bar{f}(t)$ is the mean fitness of rules which participate in the reproductive event at time t. (I.e., with the GA in [A], $\bar{f}(t)$ is the mean fitness of rules in [A].) Note that the incorporation of (3.1) means (3.2) only applies when inputs occur equiprobably and $\theta_{GA} = 1$.

The XCS Optimality Hypothesis

The outcome of the process described by the generalisation hypothesis is that the more general rule should displace the less general rule from the population. This argument was followed to its logical conclusion in [144] which introduced the *XCS Optimality Hypothesis*, which suggests that XCS will evolve optimally general rules for each action set. (Optimally general rules are those which cannot be made any more general without becoming inaccurate.) To date, empirical evidence supports this hypothesis for the case of Boolean functions (see [144, 146, 147, 154], §3.6.4, §4.3.5, §4.3.6).

3.3.3 Complete Maps

In the original XCS paper [298] Wilson refers many times to XCS's ability and tendency to form complete maps (which suggests he values them), but he does not explain the value of complete maps other than to suggest they can help prevent the system from converging to suboptimal actions:

> "A second source of inspiration came from reinforcement learning [263], which emphasizes the formation of relatively complete mappings X × A ⇒ P from the product set of situations and actions to payoffs. In contrast, the general classifier system philosophy (see, e.g., [126]) attempts more pragmatically to discover the best rule in each niche without worrying too much about knowing the payoff consequences of every possible action. However, should a suboptimal rule be converged upon as a consequence of incomplete exploration, it may be difficult for the standard system to discover and switch to a better one. If, on the other hand – as in reinforcement learning – the system were oriented toward learning relatively complete maps of the consequences of each action in each niche, then determining the most remunerative action would be straightforward." [298] p. 5.

We will consider the value of complete maps in §3.7 and §5.11.

3.3.4 Summary

XCS is clearly based on the principle that rule fitness should be biased towards general but accurate rules, which it accomplishes thanks to its accuracy-based fitness and niche GA. XCS's fitness bias towards general but accurate rules, combined with the adaptive power of the GA, suggests XCS will indeed find accurate, general rules. Empirical evidence with Boolean functions supports this.

3.4 A Rationale for SB–XCS

Although it will later be argued (§6.3) that SB–XCS is fundamentally very different from XCS, they are algorithmically very similar, and we can explain SB–XCS's behaviour by reference to pressures towards accurate and general rules, as we did in XCS.

Let's consider the strength of SB–XCS's rules in a non-sequential task in which it receives a high reward r_h for acting correctly and a low reward r_l otherwise, and in which inputs occur equiprobably. (This is the situation we considered in the 6 multiplexer task in §2.5.1.) In this case, a rule's strength under equation (2.4) is a function of r_h and r_l, and of the distribution of updates towards r_h and r_l. That is, strength is a function of the number of inputs to which a rule responds correctly; more accurate rules will have higher strengths.

Since in SB–XCS a rule's fitness is its strength, the above suggests that in such tasks SB–XCS will reproduce more accurate rules preferentially. That is, *in such tasks* it has an accuracy bias, despite being a strength-based LCS. Since SB–XCS makes use of a niche GA, it also has a bias towards general rules. These two biases explain SB–XCS's success in finding accurate, general rules *in this type of task*. We will see how SB–XCS fares on more complex tasks in chapter 5. Thus we have both empirical evidence (§2.5) and a theoretical explanation for the ability of a strength-based system to find accurate, general rules on this type of task.

3.5 Analysis of Populations Evolved by XCS and SB–XCS

In this section we return to the 6 multiplexer experiment from §2.5.1 and inspect sample rule populations evolved by XCS and SB–XCS. Although

both systems solved the problem, and did so given a similar number of inputs, their evolved populations differ, and the differences are revealing.

3.5.1 SB–XCS

Figure 3.1 shows an extract of 26 rules from a population evolved by SB–XCS in the experiment on page 61. The rules have been sorted in order of descending numerosity, and, to save space, only those with 3 or more copies are shown. The total number of macroclassifiers in the population was 64. What can we learn from inspecting this population of rules?

The Solution

Perhaps the first thing we might notice is that the population is split between rules with high numerosity (numerosity ≥ 25) and rules with low numerosity (numerosity ≤ 7). A horizontal line has been inserted between the two sets in order to highlight the division. Let's consider the high numerosity rules. All have high experience, reflecting both their generality and the length of time they have been in use. All have strengths of 1000, and, given their high experience, we can expect this to be an accurate estimate of the reward they will receive on use.

Clearly SB–XCS has found a number of good, general rules, produced many copies of them, and retained them in the population. In fact, these particular 8 rules constitute the unique minimal representation of the 6 multiplexer in the standard ternary language (discounting the use of default hierarchies, which SB–XCS does not support §3.6.1). So SB–XCS has not only solved the problem, but found a representation of it which is optimal in that it consists of a minimal number of rules. Following an earlier convention [144], let us refer to this set of 8 rules as SB–XCS's *optimal solution* for the 6 multiplexer, and denote the set [O].

Subsumed Rules

If the first 8 rules suffice to solve the problem, what about the rest of the population? The remainder consists mainly of rules which also have a strength of 1000, some of which have considerable experience, and some of which do not. Inspection reveals that these rules are more specific versions of the first 8. Since each is subsumed by some more general, but not overgeneral, rule, these rules are redundant. Although they do not have as much numerosity as the first 8 rules, some of them have considerable numerosity, suggesting

Rule	Strength	Numerosity	Experience
000###→0	1000	43	726
10##1#→1	1000	40	900
01#0##→0	1000	38	865
10##0#→0	1000	37	833
11###1→1	1000	33	954
11###0→0	1000	33	675
001###→1	1000	32	971
01#1##→1	1000	25	898
11#1#1→1	1000	7	112
111##0→0	1000	6	362
1###1#→1	645	5	12
0011##→1	1000	4	88
100#0#→0	1000	4	40
##1###→1	668	4	288
##0###→0	640	4	96
1####1→0	401	4	39
0110##→0	1000	3	54
0#11##→1	1000	3	83
0#####→1	508	3	253
#0#10#→0	569	3	61
000#1#→0	1000	3	29
000##1→0	1000	3	24
11##01→1	1000	3	27
0010##→1	1000	3	13
0111##→1	1000	3	8
10#10#→0	1000	3	5

Figure 3.1: A subset of the rules evolved by SB–XCS for the 6 multiplexer. Those above the horizontal line have high numerosity while those below have low numerosity.

that SB–XCS might benefit from the introduction of pressure against such subsumed rules.

Overgeneral Rules

In addition, the population contains a number of rules whose strength is neither 1000 nor 0, which tells us they are overgeneral. The strength of these rules is alternately updated towards 1000 (when they take the correct ac-

tion) and 0 (when they take the incorrect action), and so lies somewhere between the two values. Such rules are unreliable and of little or no value to the system. It would seem that SB–XCS would benefit from the introduction of further pressure against them, in order to reduce the population size. Nonetheless, SB–XCS has succeeded in finding the set of rules which represent the problem minimally, and has learnt to select the correct action for each input.

3.5.2 XCS

Figure 3.2 shows an extract of 21 rules from a population evolved by XCS for the 6 multiplexer in the experiment on page 61. As before, the rules are sorted in order of numerosity and to save space only those with 2 or more copies are shown, except for the last rule, which is shown despite its numerosity of 1 because it is of special interest. The total number of macroclassifiers in the population was 44.

The Solution

Like SB–XCS's population, XCS's population is split between high and low numerosity rules, and again the two groups are separated by a horizontal line. Two points are worth noting. First, the difference in numerosity between the last member of the top group and the first member of the bottom group is greater in XCS than in SB–XCS, suggesting that XCS is better able to distinguish elements of the solution. Second, XCS has allocated high numerosity to 16 rules while SB–XCS allocated high numerosity to only 8.

Consideration of the 16 high numerosity rules reveals that they consist of the 8 rules in SB–XCS's solution and their complements, that is, the rule with the same condition but the other action. These additional 8 rules all have a prediction of 0, since they all advocate the incorrect action for the inputs they match. Why has XCS allocated high numerosity to these consistently incorrect rules? Precisely because they are consistently incorrect, that is, they accurately predict the reward they will receive, and so have low prediction errors and high prediction accuracy and fitness (as the reader can verify from consulting figure 3.2). To XCS it is irrelevant that these rules advocate the incorrect action and should never be used in action selection; they are accurate and so have high fitness regardless of their utility in action selection.

Following convention, let us refer to these 16 rules as XCS's optimal solution [O] to the 6 multiplexer. (Note that [O] differs depending on whether

Rule	Prediction	Accuracy	Fit.	Numerosity	Exp.
01#0##→0	1000	1.0	0.981	27	839
10##1#→1	1000	1.0	0.986	26	937
01#1##→0	0	1.0	0.998	26	759
001###→1	1000	1.0	0.980	25	851
000###→0	1000	1.0	0.991	25	841
10##0#→1	0	1.0	0.976	24	917
11###0→0	1000	1.0	0.949	24	857
01#0##→1	0	1.0	0.984	24	842
000###→1	0	1.0	0.986	24	735
11###0→1	0	1.0	0.994	23	889
01#1##→1	1000	1.0	0.999	23	740
11###1→0	0	1.0	0.990	21	795
10##1#→0	0	1.0	0.956	20	958
10##0#→0	1000	1.0	0.934	19	913
11###1→1	1000	1.0	0.989	18	823
001###→0	0	1.0	0.999	18	715
00#0##→1	591	0.0	0.004	3	15
11#1##→0	600	0.0	0.005	2	5
11####→1	460	0.0	0.000	2	23
#01###→1	804	0.0	0.000	2	19
1###00→0	1000	1.0	0.120	1	118

Figure 3.2: A subset of the rules evolved by XCS for the 6 multiplexer. Those above the horizontal line have high numerosity while those below have low numerosity.

we refer to XCS or to SB–XCS.) We will define [O] in chapter 4, and we will investigate the difference in the two representations in §3.6.

Subsumed Rules

There is a notable lack of subsumed rules in the population evolved by XCS, suggesting it is effective at removing them. Of course, many of the rules not shown were subsumed, but they all have numerosity of only 1.

XCS's superior ability to rid its population of subsumed rules is likely due mainly to its use of GA subsumption, since without GA subsumption XCS's macroclassifier curve converged to approximately the same level as SB–XCS's (see page 61). There may, however, be a small effect due to the deletion scheme (t3) XCS used, in which the deletion probability of a rule

depends partly on the number of rules in the action sets in which it participates (§2.3.6). This places pressure on subsumed rules, since the rules which subsume them necessarily occur in their action sets, mutually increasing their odds of deletion. SB–XCS could easily be modified to use a similar deletion scheme and if so would likely have somewhat fewer subsumed rules in its population.

Overgeneral Rules

In addition to the high numerosity rules, XCS's population contains some overgeneral rules (whose predictions are somewhere between 1000 and 0). As figure 3.2 shows, these rules have low fitness and low numerosity. Furthermore, they have low experience despite their generality, indicating they have been generated relatively recently. From this we can conclude that these rules represent (inaccurate) hypotheses which XCS has recently generated and which it will likely delete in the near future.

An Extra Rule

The last rule shown in figure 3.2, $1\#\#\#0\,0 \rightarrow 0$, is an interesting case. It does not belong to the optimal population, nor is it subsumed by any of them. Neither is it overgeneral.

This is an accurate, general rule. In fact, it is just as accurate and general as the rules in [O]. Why is such a good rule only barely represented in the population, with a numerosity of only 1? Why has XCS given this good, general rule a fitness of only 0.12, when elements of [O] have fitness > 0.9? The answer is that this rule overlaps with elements of [O], and so competes with them for reproductive trials, and suffers as a result. We will revisit this subject in chapter 4.

3.6 Different Goals, Different Representations

Although the difference in fitness calculation between XCS and SB–XCS may seem minor – after all, most of their machinery is still identical – it has profound implications. One is the form of the *covering map* of classifiers the system maintains, that is, the set of state-actions which are matched (mapped to a strength) by some rule.

When we inspected the rule populations evolved by XCS and SB–XCS for the 6 multiplexer we saw that they converged on two different representations (§3.5). This section explains in more detail that strength and accuracy-based

systems tend towards different representations of their environment and suggests this difference indicates that the systems have different representational goals. We'll see consequences of these different inferred goals in much of the rest of the thesis; in §3.7, and in chapters 4, 5 and 6.

Like any RL system, a classifier system needs a policy which tells it how to act in any state it encounters. This means that, for each state, an LCS must contain at least one rule which matches that state. Otherwise, how can it know what to do? Different classifier systems represent their policies, and information related to them, in different ways. The following sections investigate three ways.

3.6.1 Default Hierarchies

Default hierarchies [211, 294, 94, 241, 247, 72] are sets of rules in which more specific rules provide exceptions to more general default rules. Figure 3.3 shows a default hierarchy for the 6 multiplexer, in which the default rule $\#\#\#\#\#\#\rightarrow 1$ matches all inputs, but is overruled by the other more specific rules in half of these cases. This hierarchy is notable as it is the one which represents this function with the fewest rules.

Default hierarchies were not part of Holland's original LCS formulation [119, 127], appearing only later [93, 123]. Default hierarchies have been praised as a means of increasing the number of solutions to a problem without increasing the size of the search space, and as a way of allowing LCS to adapt gradually to a problem. Consequently, they have been seen by many as an important part of the representational capacity of classifier systems, although they seem never to have been considered central to the idea of LCS. However, despite much early work, the problems of encouraging their formation and survival remain unsolved, and they have attracted little attention in recent years. (But see [125] p. 6.)

Because XCS evaluates the accuracy of each rule individually, default rules, which on their own are sometimes incorrect, are assigned low accuracy and hence low fitness. Consequently, XCS does not support default hierarchies. Although SB–XCS does not have this obstacle, it does not encourage them and indeed should not support them as it does not bias action selection towards more specific rules, as default hierarchies require.

Because of difficulties with default hierarchies, and the lack of support for them in XCS and SB–XCS, we mention them only in passing and now proceed to consider other representations.

Default Hierarchy	Best Action Map	Complete Map
000###→0	000###→0	000###→0
01#0##→0	001###→1	001###→0
10##0#→0	01#0##→0	01#0##→0
11###0→0	01#1##→1	01#1##→0
######→1	10##0#→0	10##0#→0
	10##1#→1	10##1#→0
	11###0→0	11###0→0
	11###1→1	11###1→0
		000###→1
		001###→1
		01#0##→1
		01#1##→1
		10##0#→1
		10##1#→1
		11###0→1
		11###1→1

Figure 3.3: Three representations of the 6 multiplexer.

3.6.2 Partial and Best Action Maps

Strength-based systems – which reproduce higher-strength rules – tend to allocate more rules to states with higher values, and to higher-valued actions within a state. In effect they attempt to find rules which advocate the best (i.e. highest valued) action for each state and in the extreme case each state would only have a single action (its best) advocated by some rule(s). Let's call this extreme case a *best action map*. More formally, in the notation of page 28, a set of rules C constitutes a best action map for some function f iff:

$$\forall s \in \mathcal{S}(f)\ \exists c \in C \text{ such that } s \in \mathcal{S}(c)$$

$$\text{and where } \forall c \in C,\ \forall s' \in \mathcal{S}(c)\ f(s') = c(s')$$

Figure 3.3 shows a best action map for the 6 multiplexer. It should be familiar, since it is SB–XCS's [O] for this function from §3.5, and, in fact, the solution SB–XCS evolved (see figure 3.1). Other best action maps can represent this function; a larger one consists of 32 rules each of which is fully specific, i.e., has no #s. The right side of figure 3.4 shows a best action map

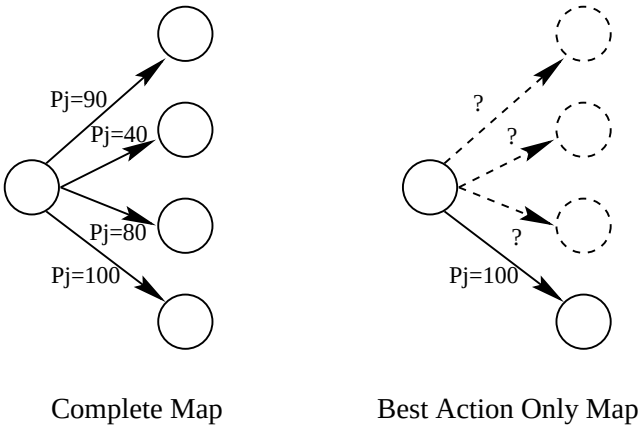


Figure 3.4: Complete and partial map representations of a state transition. In the best action map, only the highest valued transition is represented.

for a fragment of a sequential task – note that of all the transitions from the current state, only the state-action with the highest prediction ($P_j = 100$) is represented. Further examples of best action maps in sequential tasks are shown in §3.7.2 and on the right of figure E.3.

In practice strength-based systems only tend towards best action maps – often for a given state there is more than one action advocated. We could say they maintain *partial maps*. (See, for example, SB–XCS's population in figure 3.1.) The important point is that there tend to be states with (typically low-valued) actions unadvocated by any classifier.

Since the action selection mechanism can only make an informed choice between advocated actions, the tendency towards a best action map means the rule allocation mechanism has a hand in action selection. Best action maps are in a sense ideal as they represent the solution with a minimal number of rules (without resorting to default hierarchies), but this does not imply partial maps are an ideal representation for finding solutions, as we'll see in sections 3.7 and 5.11.

3.6.3 Complete Maps

In contrast to the partial maps of strength-based systems, the combination of XCS's accuracy-based fitness and various mechanisms which promote diversity result in a strong tendency to find a population of rules such that each action in each state is advocated by at least one rule. Wilson calls this a *complete map*, and has noted that it resembles the representation of mainstream reinforcement learning systems more closely than does a partial map

([298] p. 5). More formally, a set of rules C constitutes a complete map for some function f iff:

$$\forall s \in \mathcal{S}(f), \forall a \in \mathcal{A}(f)\ \exists c \in C \text{ such that } s \in \mathcal{S}(c), a \in \mathcal{A}(c)$$

Figure 3.3 shows a complete map for the 6 multiplexer, which is like the best action map in that figure except that for each condition there is a rule for each of the two actions. This set of rules should look familiar as it is XCS's [O] for this function from §3.5, and, in fact, it is the solution which XCS evolved (see figure 3.2). Examples of complete maps in sequential tasks are shown on the left of figures 3.4 and E.3.

When all actions are advocated, it is left entirely to the action selection mechanism to decide which action to take; the rule allocation mechanism is dissociated from action selection. This is consistent with the role of the GA in XCS, which is only to search for useful generalisations over states. We'll return to this point in §6.2.

3.6.4 What do XCS and SB–XCS Really Learn?

In the preceding sections it was argued that XCS and SB–XCS will respectively tend towards complete and partial maps, and the sample populations inspected in §3.5 support this. But how strong is this tendency?

Figure 3.5 was generated during evaluation of XCS (using GA subsumption) and SB–XCS on the 6 multiplexer in §2.5.1 and shows the percentage of the optimal solution found by each system on the current time step (denoted %[O]). A %[O] of 0 indicates that no rules from this set were present in the rule population, while %[O] of 100 indicates that all were present.

The curves are averages of 10 runs. That both systems converge to 100% of [O] indicates that *in all runs* the complete optimal set of rules is present in the rule population. This suggests XCS and SB–XCS reliably find their respective optimal solutions to the 6 multiplexer. Note that XCS finds all 16 rules in its [O] more quickly than SB–XCS finds the 8 rules in its [O].

Although the results in figure 3.5 were averaged over only 10 runs, XCS's ability to find [O] is highly reliable. Figure 3.6 shows XCS's %[O] for a set of 7 functions from the hidden parity test suite [154], each averaged over 1000 runs. Parameter settings were the standard ones for XCS from page 59, except that uniform crossover was used and the population size limit was 2000. That XCS found the complete [O] in all 7000 runs is indicated by the convergence of all curves to %100.

XCS is also able to find the complete [O] for much larger Boolean functions, for example the 37-bit multiplexer, for which [O] consists of 128 rules.

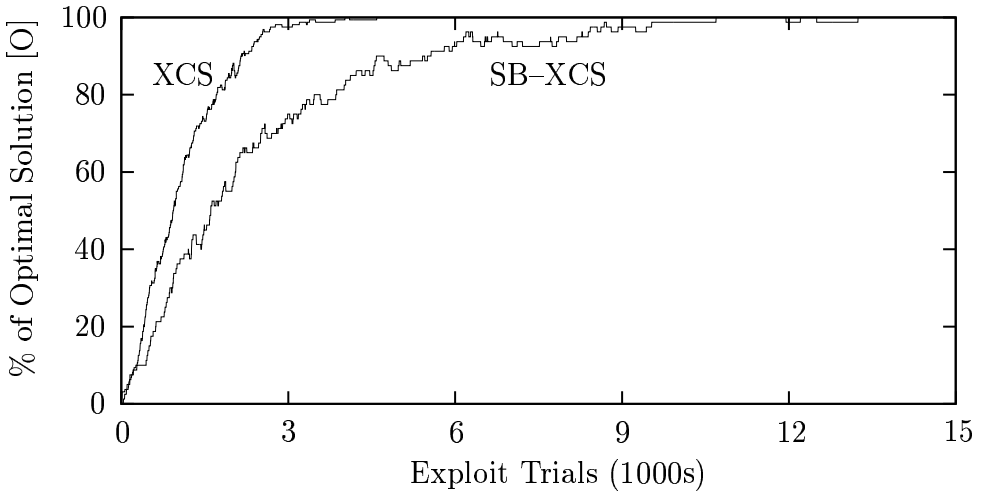


Figure 3.5: Proportion of optimal solutions found by XCS and SB–XCS on the 6 multiplexer.

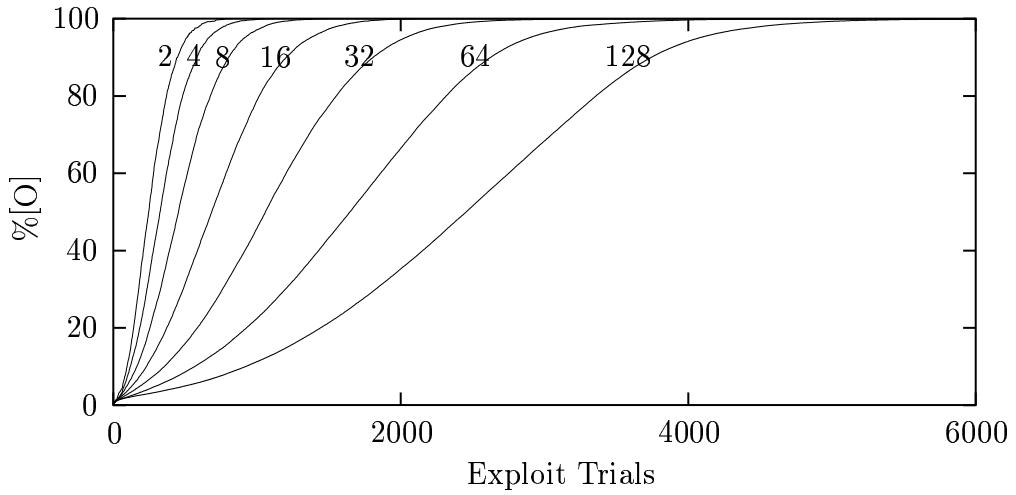


Figure 3.6: Proportion of optimal solutions found by XCS on the 6-bit hidden parity suite. The number beside each curve indicates the number of rules in the [O] for that function.

XCS's %[O] and $\mathcal{P}$ for this test are shown in figure 3.7 and the population size in macroclassifiers is shown in figure 3.8. Parameter settings were as on page 59, except for the following: $\theta_{sub} = 100$, $\theta_{GA} = 80$, $P_{\#} = 0.66$, and uniform crossover was used. Curves were averaged over 10 runs.

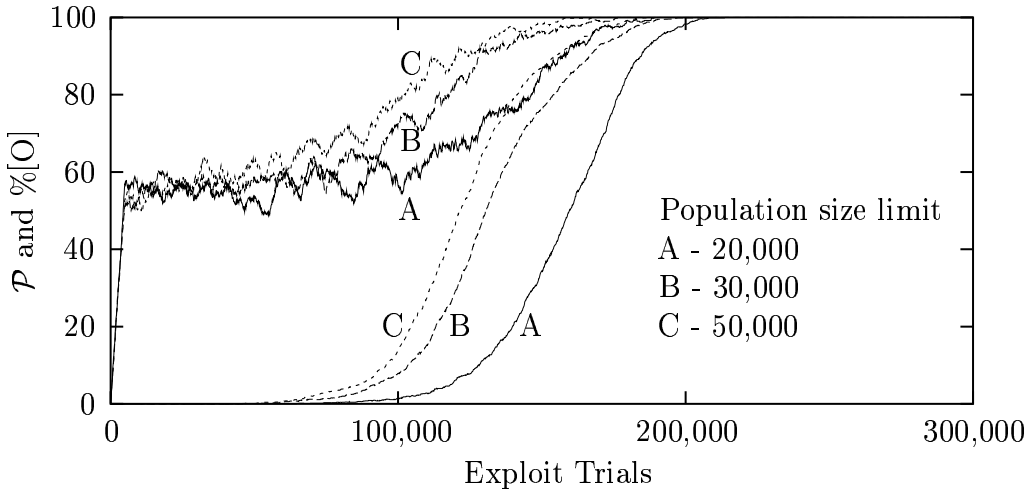


Figure 3.7: $\mathcal{P}$ and %[O] for XCS on the 37-bit multiplexer for three population size limits.

In both figures, XCS was run with three different population size limits (20,000, 30,000 and 50,000 rules) to examine the effect of population size. As with other test functions performance improves with population size, although it is a case of diminishing returns. At the same time, the computational load on the system increases greatly as the population size limit increases.

With the help of a special form of reward function XCS has also learnt the 70-bit multiplexer [52].

3.7 Complete and Partial Maps Compared

In this section we weigh the pros and cons of complete and partial maps. Little is known about how they compare – the literature has nothing to say on the subject, apart from the speculation in ([298] p. 5) that complete maps may help avoid convergence to suboptimal actions, and my speculations in [148]. This is a complex subject which will require considerable effort to unravel, and this section is of necessity a basic treatment.

3.7.1 Advantages of Partial Maps

Partial maps use fewer rules than complete maps to represent the same input/output function, which may prove advantageous in the following ways.

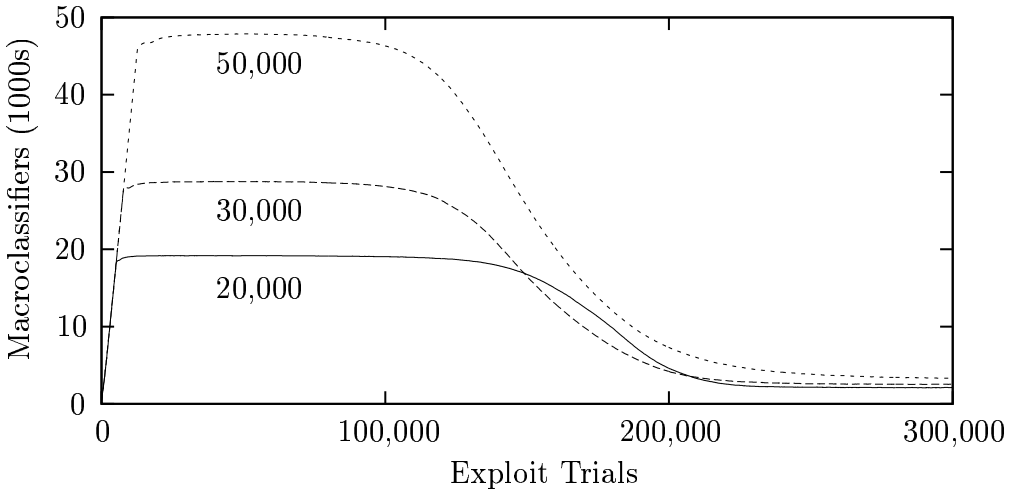


Figure 3.8: Population size in macroclassifiers for XCS on the 37-bit multiplexer, using three different population size limits (20,000, 30,000 and 50,000 rules).

First, fewer rules require less memory and computing power, since fewer rules need to be stored and matched against inputs. Second, since fewer rules are involved, it may be possible to learn partial maps given less experience with the environment (although in our comparison of XCS and SB–XCS in §3.6.4, SB–XCS actually took longer to learn its partial map than XCS did to learn its complete map).

How Parsimonious are Partial Maps?

Any benefit due to parsimony depends on the degree of that parsimony. So how much more parsimonious are partial maps than complete maps? We could try to compare the number of rules required in each case, but we note that the test function used is relevant as it determines what useful generalisations, and thus what accurate rules, are possible.[4] This makes a comparison of the number of rules needed difficult because we may be able to generalise more over one action than another, depending on the test function. In order to compare complete and partial maps we'd need to take this into account. Note, however, that if our space of possible test functions is the set of Boolean functions of a given length, an equal number of generalisations

[4]More carefully, it is the shape of the value function which determines what useful generalisations are possible.

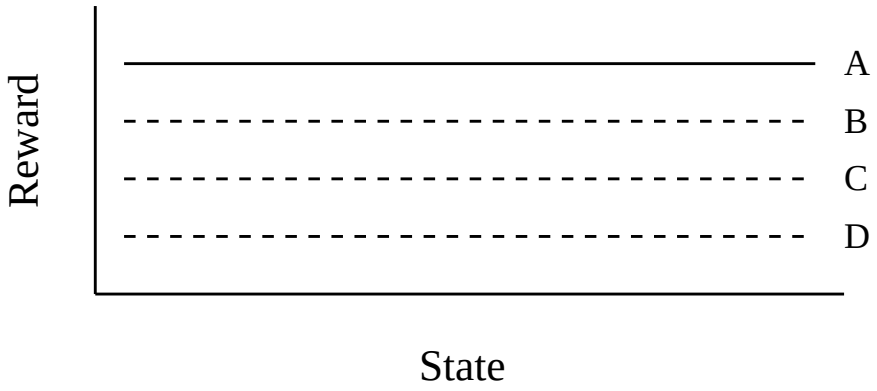


Figure 3.9: A reward function for a non-sequential task with 4 actions.

is possible over all actions. That is, over all such test functions, we cannot generalise more over any given action than any other.

For the moment let's ignore the issue of generalisation and, instead of considering the number of rules needed, let's consider the number of state-actions which must be represented. Using the standard ternary language this depends heavily on the number of actions available. If a actions are possible in each state, a best action map requires the value of $1/a$ state-actions be estimated per state, while a complete map requires all a actions be estimated. That is, a complete map requires the representation of a times more state-actions.

To illustrate this point, consider figure 3.9 which shows the reward function for a non-sequential task. Four actions are possible in each state, and each action always receives the same reward, regardless of which state it is taken in. (Consequently, the lines showing reward are flat.) Because in all states action A obtains the most reward, a best action map must represent only state-actions with action A. To emphasise this the line for action A has been drawn solid, while those for the other actions are dashed.

While a best action map represents only the most rewarding action in each state, a complete map must represent all four actions. Consequently, a complete map represents $a = 4$ times as many state-actions for this task.

The Degree of Parsimony In the case of the space of Boolean functions, where $a = 2$, the savings of using a best action map are not terribly impressive. In many cases a will not be much larger. However, as a rises the savings of a partial map become more significant. When the action space is continuous and $a = \infty$ a partial map (or some alternative representation) is

a necessity.

How Many Rules are Needed? Note that if we consider all Boolean functions of a given length, the ratio of state-actions to be represented is also the ratio of rules needed, if we assume we can generalise equally over all actions, on average. In this case, a complete map will require on average a times more rules.

This is the case for the function in figure 3.9, in which fully general rules can accurately predict the value of the state-actions they match. For example, the following rule constitutes a (minimal) best action map:

###→A

and the following four rules constitute a (minimal) complete map:[5]

###→A
###→B
###→C
###→D

XCS and SB–XCS The rules in the best action and complete maps above will be fit according to either XCS's or SB–XCS's fitness calculation, and for this particular task, XCS's minimal complete map is indeed four times the size of SB–XCS's minimal best action map.

However, XCS and SB–XCS differ in their ability to express generalisations, as we will see in §3.8. Consequently, for them the assumption made above that we can, on average, generalise equally over all actions is not in general valid.

This means that the conclusion that XCS's complete map will require on average a times more rules than SB–XCS's best action map does not hold in general (although it happens to be true in the example above). The conclusion that a times as many state-actions must be represented is, however, still valid since it is not affected by the ability to generalise.

3.7.2 Disadvantages of Partial Maps

The disadvantages of partial maps are that they may interfere with action selection (and exploration control more generally), and credit assignment. We discuss each of these in turn.

[5] This map is not strictly in the standard ternary language, since its actions are not from $\{0,1\}^l$. However, $\{A, B, C, D\}$ is effectively equivalent to $\{00, 01, 10, 11\}$.

Partial Maps and Exploration

This section suggests that managing the explore/exploit dilemma (§C.2) will be more difficult using a partial map since it does not allow us to record as much information on exploration as a complete map. If partial maps make exploration control more difficult, then the more difficult the exploration problem, the more of a disadvantage a partial map will be.

In the following we consider exploration control using a partial map in non-sequential and sequential tasks, following which we briefly discuss non-stationary environments.

Non-sequential Tasks In many LCS an action cannot be taken if it is not advocated by some rule, which means a partial map limits the range of actions it can take and hence its ability to explore its environment. Even if we allow an LCS to take an unadvocated action, we cannot record the result (the resulting immediate reward) since we have no matching rule.

Let's return to the non-sequential task of figure 3.9, but now suppose that the reward function is stochastic so that the true mean reward for taking a given action in a given state must be estimated by sampling it many times. How many times should we sample a state-action?

Some simple exploration strategies, for example taking a random action 10% of the time and the highest-valued one otherwise, do not require us to keep track of the number of visits made to a given state-action. Other more sophisticated strategies, however, require us to record information such as the number of visits, the mean reward, and the variance in it. Such information is easily added to rules.

However, a problem obviously occurs in a best action map representation of this reward function, in that we have no rule which can record information concerning actions B, C, or D. Consequently we cannot use such exploration control strategies. An alternative to storing exploration control information in the rules is to use a look-up table, but if we make use of such tables there seems no point in using a classifier system rather than a tabular Q-learner. In summary, partial maps would seem to restrict us to the use of less sophisticated, and less efficient, exploration strategies.

Sequential Tasks In sequential tasks we face the problem that we must explore not only the actions available in the current state, but the states and actions we may visit as a result of selecting an action in the current state, which makes exploration control much more difficult.

If we wish to implement distal exploration control schemes [319], in which we propagate information about exploration from state-action to state-

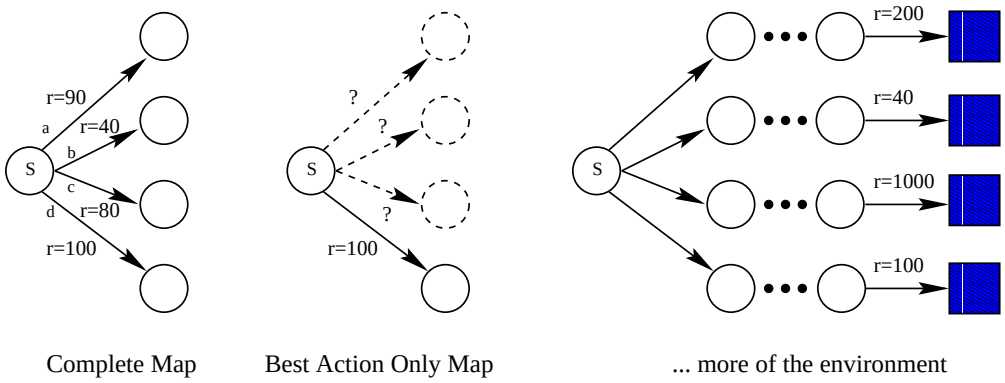


Figure 3.10: Complete and partial map representations of a sequential task.

action, we need a complete map in order to propagate exploration information properly (just as with propagating Q-values) and to record any statistics (e.g., on the number of times a state-action has been taken).

Partial maps tend to allocate rules to higher-valued state-actions, but an LCS must estimate the value of state-actions and can easily get their relative values wrong. Suppose an LCS is learning the task in figure 3.10, and initially has a complete map of the transitions from state S, as shown on the left. This corresponds to the left of figure 3.4, except that this time the rewards ($r = \ldots$) associated with each transition are shown, and each transition is labelled with the action (a,b,c,d) which produces it. Now suppose the LCS tries each of the actions in state S and determines that d generates the highest immediate reward. If it deletes some of the rules which cover the other transitions to obtain a partial map, it may not represent the transition due to action c. As long as this action is unadvocated the LCS will never further explore the consequences of taking it. Now consider the right of figure 3.10, which shows more of the task, including the four terminal states (shown as squares). The dots ($\cdots$) are meant to suggest that part of the environment is not shown. Suppose the optimal policy involves taking action c in order to reach the terminal state with a reward of 1000. It may be difficult for the learner to find this terminal state (and the optimal policy) if it maintains a partial map, since action c may not be represented.

Globally optimal behaviour often requires that the learner take locally suboptimal actions, i.e., actions which do not return the highest possible immediate reward, like action c. Even if the system does find the highest rewarding terminal state once, it will have difficulty propagating value back from it if state-actions along the path to it are missing.

Non-stationary Environments Hartley [110, 111] trained two classifier systems, XCS (which maintains a complete map) and NEWBOOLE (which maintains a partial map) on a binary categorisation task, then abruptly switched the category each stimulus belonged to. XCS quickly recovered from these changes by simply adjusting the strengths of the rules involved: consistently correct rules suddenly became consistently incorrect and vice versa. NEWBOOLE, in contrast, found that its rules suddenly all had low strength, and had to engage the genetic algorithm to generate new ones as it does not maintain low strength rules.

It is important to note, however, that the changes made in this experiment were very regular, as all the inputs belonging to one category were changed to the same new category. It is unknown whether a complete map offers any advantages in adapting to less systematic changes in the environment. In fact, given the suggestion on page 90 that it may be possible to learn partial maps more quickly, we might expect partial maps to adapt more quickly to non-stationary environments.

In Hartley's experiment the rewards were non-stationary. It is also possible for the variance in the rewards to be non-stationary, a situation which has yet to be examined.

Partial Maps and Sequential Credit Assignment

With the Temporal Difference algorithms (§E.4.5) commonly used with sequential tasks we update the estimate of a state-action's value based partly on the estimated value of a state-action which is visited later. That is, we propagate value from one estimator to another. However, if the estimator for the successor state is missing (e.g., if an action is not advocated by any rule) we cannot apply the basic 1-step Temporal Difference updates (e.g., the strength/prediction update used with XCS and SB–XCS – equation (2.4)). (Of course, if we do not allow the system to take unadvocated actions this problem does not occur.) We *could* apply n-step versions of our updates [265] and propagate value from state-actions visited farther in the future (i.e., n time steps in the future), but as far as the author is aware such updates have never been used with an LCS, and they are certainly not part of standard descriptions of the bucket brigade (e.g., [123, 126, 39, 94]).

Gridworld Examples To further illustrate how partial maps can make sequential credit assignment difficult, let us consider a number of simple gridworlds, and partial maps of them maintained by some hypothetical learning agent. In each of the following examples, the start state is marked **S** and the two terminal states are marked **T**. Upon entering a terminal state the agent

receives an immediate reward and is returned to the start state to commence the next episode. The immediate reward received upon entering a state is shown in the bottom right corner of that state, or is 0 if not shown. It is possible to take the actions {North, South, East, West} in any state, although actions which would move the agent out of the maze have no effect. Let us set the discount rate to $\gamma = 0.9$.

The state-actions of the agent's partial map are shown as arrows, e.g., in example 1, in the start state the agent's map represents the values of moving North and of moving East only.

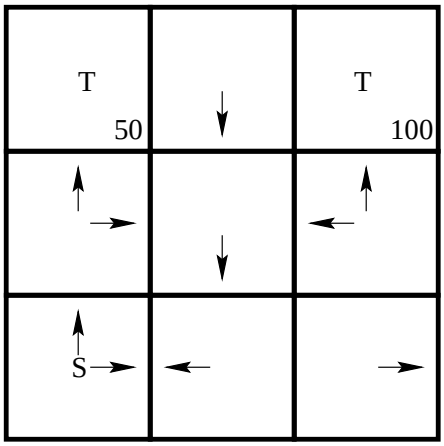


Figure 3.11: A partial map forming incomplete paths.

Example 1: Incomplete Paths The gridworld in figure 3.11 has two terminal states, with rewards of 50 and 100 respectively. The optimal policy for the learning agent (given $\gamma = 0.9$) is to follow one of several minimal paths to the terminal state in the upper right hand corner. We'll denote this state $\mathbf{T}_{100}$.

Note that the partial map for this gridworld does not form a complete path from $\mathbf{T}_{100}$ to **S**. The first problem this causes is that some systems, XCS and SB–XCS included, cannot reach $\mathbf{T}_{100}$ if no path leads to it, since they cannot select unadvocated actions (§2.3.4).

A second problem is that even if the agent was able to reach $\mathbf{T}_{100}$, an incomplete path means it is not possible to propagate value from $\mathbf{T}_{100}$ to **S** using the standard 1-step updates. Consequently, the credit assignment process will not be able to update the value estimates for moving North and East from **S** to take into account the existence of $\mathbf{T}_{100}$, and the agent will

not be aware of it. (All 1-step algorithms, including 1-step Q-learning and Bucket Brigades, will have the same problem.)

Furthermore, even if a path from $\mathbf{T}_{100}$ to **S** existed, the use of partial maps means an unfortunate deletion of a rule anywhere along the path would break it and disrupt the flow of value.

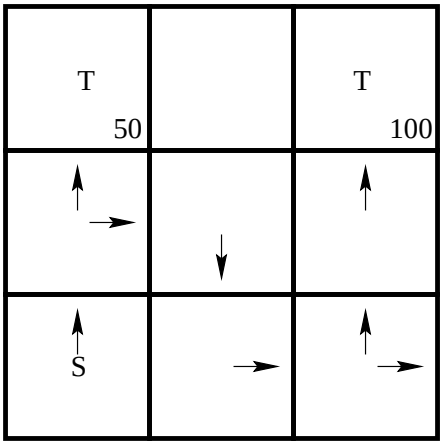


Figure 3.12: A partial map with interfering local optima.

Example 2: Interfering Local Optima The example in figure 3.12 is the same as the last, except that the agent's partial map differs. Now it has complete paths from **S** to both terminal states. Nonetheless, there are a number of problems with this scenario. For one, it is more difficult to reach $\mathbf{T}_{100}$ than $\mathbf{T}_{50}$ from **S** because the former is farther from it. This means both that more exploration is required to reach $\mathbf{T}_{100}$, and that propagating value from $\mathbf{T}_{100}$ to **S** will take longer than propagating value from $\mathbf{T}_{50}$ to **S**, using 1-step updates. Both factors makes it harder for the agent to learn to use a path to $\mathbf{T}_{100}$ and may cause it to converge instead on a suboptimal policy leading it to $\mathbf{T}_{50}$.

Both these problems occur using complete maps, but an extra problem with the use of partial maps compounds them: since rules leading to $\mathbf{T}_{50}$ are likely to gain value more quickly than those leading instead to $\mathbf{T}_{100}$, the latter may die out resulting in incomplete paths to $\mathbf{T}_{100}$.

Example 3: Discounting and Convergence to Suboptimal Policies A further problem in the same scenario as the last example is that the path to $\mathbf{T}_{100}$ is suboptimal (there exist shorter paths between **S** and $\mathbf{T}_{100}$). With

$\gamma = 0.9$, the value of $\mathbf{T}_{100}$ using this circuitous route is actually less than that of $\mathbf{T}_{50}$ (using a direct path), which may lead the system to converge on a policy which leads it to $\mathbf{T}_{50}$. This is more likely to occur when the agent is limited to a subset of paths by a partial map.

Example 4: Interfering Negative Rewards As a final example of how partial maps can interact harmfully with sequential credit assignment consider the maze in figure 3.13, in which a column of negatively rewarding states isolates $\mathbf{T}_{50}$ from $\mathbf{S}$. The agent's partial map is not shown in this example.

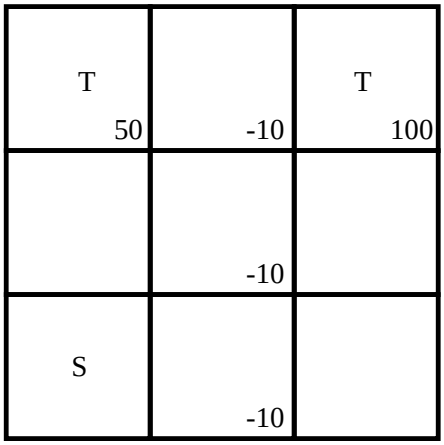


Figure 3.13: A gridworld with interfering negative rewards.

The undesirable states may tend to hide $\mathbf{T}_{100}$ from the learning agent, since it must pass through them to reach it. This makes the task of exploring the gridworld more difficult, since the agent may prematurely learn to avoid the unpleasant states.

Although this may happen with complete maps too, an extra difficulty in using partial maps is that they may tend to become sparser around the unpleasant states, exacerbating the difficulty of reaching $\mathbf{T}_{100}$ and propagating value back from it.

Summary of Partial Maps and Sequential Credit Assignment It is unclear just what the effects of partial maps are on the propagation of value, but it seems likely that any effects will be deleterious. The simplest and safest course would appear to be the maintenance of a complete map.

3.7.3 Complete Maps and Strength

If complete maps are useful, can we get a strength-based system like SB–XCS to maintain them? We could give rewards of 90 for incorrect actions and 100 for correct actions. If selective pressure is not too strong SB–XCS should be able to maintain classifiers for both correct and incorrect actions (i.e., a complete map). However, overgeneral rules would always have more strength (i.e., fitness) than accurate-but-incorrect rules, so the latter will be more likely to die out than the overgenerals. Such a system seems unlikely to adapt well as the overgenerals will interfere with action selection (and, of course, reproduction).

Distinguishing between Strength and Fitness

Alternatively, we could modify SB–XCS to distinguish between strength and fitness, and define a rule's fitness as:

$$\left| S_j - \frac{max - min}{2} \right|$$

where S_j is the strength of rule j, and max and min are the maximum and minimum possible rule strengths for the task. This system would give high fitness to both consistently correct and consistently incorrect rules. This system should be able to maintain complete maps for tasks which have only 2 rewards, but it will not work with arbitrary reward functions because of the strong overgenerals to be introduced in chapter 5.

Fitness Sharing

Another option is to introduce fitness sharing into SB–XCS. Because the strength of all rules tends towards equilibrium under fitness sharing [293, 47], the map produced with fitness sharing should be more complete. It is not clear, however, how complete it will be.

Steps Towards Accuracy

The maintenance of complete maps is one of XCS's distinctive features, and attempts to get SB–XCS to maintain complete maps are steps towards making SB–XCS more XCS-like.

3.7.4 Contrasting Complete and Partial Maps in RL Terminology

In Reinforcement Learning terminology, while XCS maintains a complete action-value function (i.e., Q-function), SB–XCS maintains an incomplete action-value function. Best action maps map states to actions – thus, in RL terminology they represent a policy, although since they are implemented with classifiers they contain information (e.g., strength values) which policies do not. SB–XCS does not search directly in the space of complete policies, but rather in the space of policy fragments – the fragments are (generalisations over) state-action pairs (i.e., classifiers).

3.7.5 Summary of Comparison

Partial maps have the advantage that, in principle, they require fewer rules, and so less processing. The degree of advantage depends on the particular task being learnt, and on the number of actions available; as the number of available actions increases, so does the advantage of a partial map.

Although they require more rules, complete maps may offer advantages with sequential credit assignment and exploration control, particularly in sequential tasks. Note that tabular Q-learners employ complete maps, and that convergence proofs for them involve infinite revisits to *each* state-action.

From this we might conclude that partial maps are more suitable for non-sequential tasks with large action sets, while complete maps may be more suitable for sequential tasks.

We can summarise the difference between the two by noting that partial maps provide a solution to a problem whereas complete maps, in contrast, provide a representation upon which Q-learning can operate in order to find a solution. Looked at this way, partial maps seem more suitable for systems which rely more on the GA to find solutions, i.e., those which fit the GA-view of classifier systems. Consequently partial maps should suit Pittsburgh classifier systems very well. Partial maps should also suit supervised learning LCS.

Further study is needed to confirm and quantify the merits of each type of map.

3.8 Ability to Express Generalisations

One of the subtle effects of the difference in fitness calculation between XCS and SB–XCS is the difference in the useful generalisations they can express.

3.8.1 Mapping Policies and Mapping Value Functions

In SB–XCS, a rule is a mapping from states to an action, and of course each such rule has an associated fitness. A mapping from each state to an action is a policy, so SB–XCS's rules are policy fragments, and SB–XCS generalises over its policy.

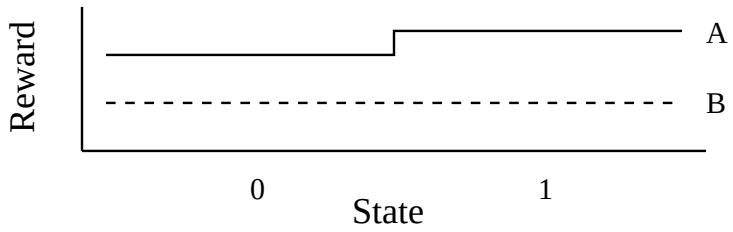
In XCS, in contrast, a rule is a mapping from states to an action to a prediction, and of course has an associated fitness. A mapping from states to actions to predictions is a value function, so XCS's rules are value function fragments, and XCS generalises over its value function.

A Non-sequential Example

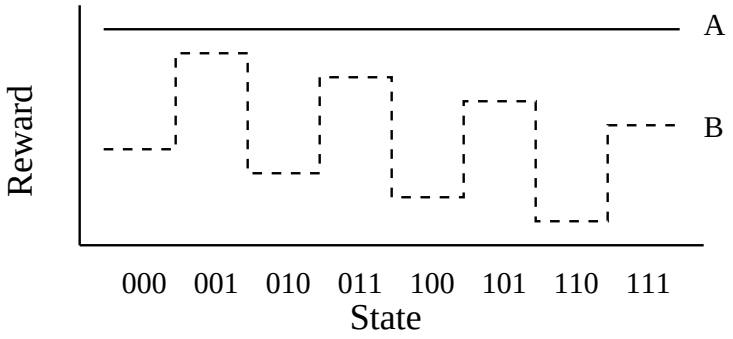
The difference means that XCS's rules are more constrained; consider the following reward function, which is shown in both tabular and graphic form.

State	Action	Reward
0	A	100
1	A	150

State	Action	Reward
0	B	0
1	B	0



To maximise reward, the learner should always take action A. SB–XCS can represent this policy with a single rule: $\#\rightarrow$A. XCS, however, has a problem with this rule, because it is alternately updated towards rewards of 100 and 150 and so has a considerable prediction error. If XCS's accuracy criterion ε_o is sufficiently strict this rule will be unfit and XCS will need to use two rules, $0\rightarrow$A and $1\rightarrow$A, in its place. Obviously we can produce larger problems in which SB–XCS can generalise better than XCS, for example the following one:



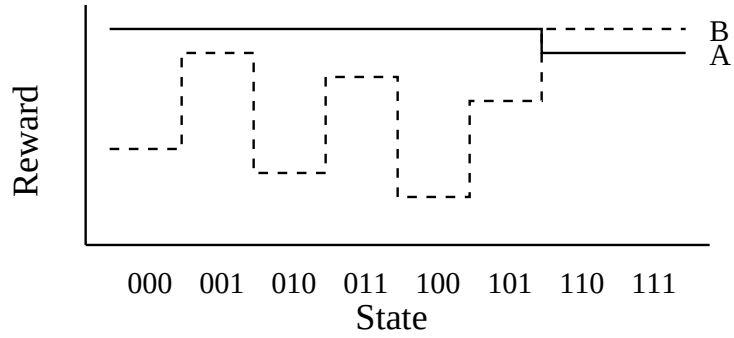
SB–XCS can represent the optimal policy for this function with a single rule, ###→A, while XCS requires 9 rules: ###→A and 8 fully specific rules advocating action B.

3.8.2 Adapting the Accuracy Criterion

In §3.8.1 we saw that XCS was unable to generalise as much as SB–XCS if its accuracy criterion was too strict. Certainly if we increase XCS's tolerance for prediction errors by increasing the accuracy criterion XCS will be able to generalise as well as SB–XCS on the examples we've just seen. But if we set the accuracy criterion too high, XCS may generalise excessively, and behave suboptimally. How can we know the optimal setting of the accuracy criterion? Ideally, XCS would find this value itself, and how it might do so is an interesting open question.

Limitations on the Adaptability of a Global Accuracy Criterion

XCS's accuracy criterion is global; it is used for all rules. This turns out, in some cases, such as the following reward function, to limit the extent to which we can adapt it to improve generalisation.



The range of rewards for action A is less than that for action B, meaning that as we increase the accuracy threshold ###→A will become accurate

before ###→B. However, ###→A is too general; it takes the wrong action in states 110 and 111. Consequently, XCS cannot raise the accuracy threshold enough to make ###→B accurate and yet still behave optimally.

SB–XCS can represent the optimal policy for this task with just three rules: 0##→A, #0#→A, and 1##→B (other minimal sets exist). XCS, however, requires these 3 rules and 7 others: 1##→A and 6 fully specific rules to round out the map for action B.

In order to generalise more fully, XCS may need a mechanism which adapts the accuracy criterion locally, for example for each rule, or for each action set. Such a mechanism is left for future work. In hindsight, however, it is not surprising that XCS should need such a mechanism; we can think of a classifier system as a function approximator [148], and a function approximator benefits from the ability to focus its resources more on some parts of its domain than others.

3.8.3 XCS-hard and SB–XCS-easy Functions

[154] shows that the minimal number of rules needed to represent a Boolean function correlates highly with the difficulty XCS has in learning to do so. One might hypothesise that the same holds for SB–XCS, given which the function in §3.8.2 should be a difficult one for XCS yet an easy one for SB–XCS, compared to other functions with the same number of states and actions. If so, this is significant as it demonstrates how to design XCS-hard yet SB–XCS-easy functions. We'll see another way of constructing such functions in §5.8.4.

3.8.4 Summary of Generalisation and Efficiency

Because SB–XCS's partial maps require the representation of fewer state-actions than XCS's complete maps, and SB–XCS can express at least some fit generalisations which XCS cannot it seems SB–XCS should often require many fewer rules than XCS to represent a task. Because this correlates with problem difficulty, SB–XCS should, at least in principle, find many tasks easier than XCS.

However, when compared on the 6 multiplexer, SB–XCS required more experience and more rules than XCS to find the solution (see page 61). Although this suggests that SB–XCS is not as efficient as XCS, the advantages of a partial map and more expressive generalisations may more than compensate for this reduced efficiency on tasks where they have the most effect.

At the same time, however, many non-sequential tasks can be formulated as supervised learning tasks (§C.1.1), which makes them less difficult for the

learner than an RL formulation (§C.2). Consequently, supervised learning classifier systems should outperform both XCS and SB–XCS (which are RL systems) on such tasks (§5.13). Indeed, unpublished work with a technique called weeding, which produced a supervised-learning-like XCS, resulted in performance improvement over the standard XCS on the 6 multiplexer test [145].

Further experimental work on these issues is called for.

3.9 Chapter Summary

This chapter has covered a lot of material, starting with simple rationales for adaptation in Holland's LCS, XCS and SB–XCS. We then inspected the populations evolved by XCS and SB–XCS for the 6 multiplexer and found differences in them (§3.5), most significantly that XCS represents accurate, general rules, regardless of whether or not they advocate the correct action. In §3.6 we formalised these differences, and argued that while SB–XCS tends towards partial maps of the state-action space, XCS tends towards complete maps. In §3.7 we compared the two, noting that SB–XCS's partial maps require fewer rules than XCS's complete maps, but that partial maps are likely to interfere with exploration and credit assignment.

§3.8 addressed the relative abilities of XCS and SB–XCS to express fit generalisations, a subject which deserves much more attention than it can be given here. It was shown that XCS's generalisation ability on a given task depends on the setting of the accuracy criterion, and that relaxing it can increase the fit generalisations which XCS can express. It was also shown, however, that a global accuracy criterion cannot always be set so as to both generalise optimally and represent an optimal policy. Consequently, it was suggested that mechanisms by which XCS could locally adapt the accuracy criterion be investigated.

Conclusions

Further comparisons of XCS and SB–XCS are needed, but the tentative conclusions of this chapter are that SB–XCS may have an advantage over XCS on non-sequential tasks with large action spaces, particularly those on which XCS has difficulty generalising. At the same time, XCS would seem to have advantages on sequential tasks thanks to its complete map. In addition, XCS's complete maps, with their more complete representation, may be of use in data mining by helping us understand the data better. This subject needs further investigation.

Chapter 4

What Should a Classifier System Learn?
(And How Should we Measure it?)

Chapter Outline

In this chapter we consider the issues of how a classifier system should learn to represent a Boolean function, and how we should measure its progress in doing so. We identify four properties which may be desirable of a representation; that it be complete, accurate, minimal and non-overlapping, and distinguish variations on two of these properties for the XCS system. We distinguish two categories of learning metric, introduce new metrics and evaluate them. We demonstrate the superiority of population state metrics over performance metrics in two situations, and in the process find evidence of XCS's strong bias against overlapping rules.

The material in this chapter began as a study of metrics for classifier systems, and so it remains, but as work proceeded the need to examine the objectives of learning in LCS soon became apparent. In order to address questions like "How well is it learning?" we must first address questions like "What is it trying to learn?". The latter turns out to be less straightforward than we might think, and, although the issue has been addressed in earlier work (e.g., [144]), it has been somewhat neglected in the LCS literature. Consequently, the scope of the work grew to address both questions.

The question "What is it trying to learn?" could be rephrased in the following ways. How should it represent a solution? What properties might a representation have, and which properties are desirable? What is an ideal solution for a given problem? What properties should an ideal solution have?

Some would suggest default hierarchies as an ideal representation of a solution, but, as explained in §3.6.1, XCS and SB–XCS do not support them, and so, for these systems at least, we are forced to consider alternatives.

In considering what an LCS should learn we relate learning in LCS, particularly XCS, to logic minimisation in order to borrow notation and terminology.

The Need for Comparisons and Metrics

The field of learning classifier systems encompasses a broad range of systems, and LCS are potentially applicable to a similarly broad range of problems, from data mining to on-line control. One problem with this diversity is the difficulty of comparing the effectiveness of these systems. Naturally there is little use in comparing the performance of LCS which are intended for different types of tasks. For example, Holmes's EpiCS [130] is simply not intended for use in sequential tasks and it makes little sense to evaluate it on such tasks. But many LCS are designed with the same tasks in mind, and even within a given system many alternative mechanisms are available. What's more, LCS mechanisms are often tunable. These alternatives must be compared.

Unfortunately there has been relatively little work comparing alternative LCS approaches, and similarly little work studying the parameterisation of LCS mechanisms. Such comparative studies are needed, and they will require better performance metrics and better test suites than are typically used.[1] This work introduces a distinction between what will be called performance metrics and population state metrics, and demonstrates advantages of the latter over the former. Although they have their limitations, it will be suggested that population state metrics can be a valuable tool in comparing alternative classifier systems, classifier system mechanisms, and parameterisations.

The Scope of this Study

We will consider the application of classifier systems to the learning of non-sequential tasks, or, more specifically, arbitrary Boolean functions. Boolean functions are a well-defined, and, for strings of finite length, finite class of

[1] The issue of test suites is not addressed here, but a start has been made in [154].

problems. Boolean functions are also easily represented and manipulated in the standard ternary LCS language.

We will not consider any issues specific to sequential tasks in this chapter, and we consider only a few extensions to the framework for learning Boolean functions introduced in the next section, but much of this work should be relevant to other cases. XCS [298] is taken as the focus of this work, but much of it should apply to other LCS.

Experimental Framework

To make matters more concrete, we will investigate what properties are desirable of a solution, and consider various on-line performance metrics, within the testing framework Wilson used with XCS in [298] (see also §2.3.3). As mentioned in §2.3.3, this scheme is less than fully satisfactory, but this chapter is concerned with solution properties and performance metrics used within the framework rather than the framework itself. In the process, however, we see some differences between Wilson's scheme and the more common train/test scheme, which should suggest deficiencies of Wilson's scheme.

Outline of the Chapter

This chapter is broadly structured as follows. In §4.1 we review a number of subjects to do with the representation of Boolean functions and condition-action rules, in preparation for later sections. In §4.2 we consider what properties the solutions found by a classifier system should possess. Then, in §4.3, we consider how to measure the performance of an LCS in finding solutions with the desired properties. Finally, §4.4 concludes.

4.1 Representing Boolean Functions

Here we review certain subjects to do with representation, which will enable us to deal more easily with Boolean functions and special sets of rules in later sections.

4.1.1 Truth Tables

We consider functions which are mappings of each possible binary string of a given length to either 0 or 1. Perhaps the most straightforward way to represent such a function is by its *truth table*, which is an exhaustive listing of all input/output pairs of the function. As an example, figure 4.1 shows the truth table for the 3-bit multiplexer function, so called because the first

bit is used as an index into the remaining two bits, and the value of an input string is the value of the indexed bit. For example, the value of 010 is 1 as the first bit dictates that the value of the string is that of the first of the remaining two bits. In contrast, the value of 110 is 0 as the first bit dictates that the second bit of the remaining two determines the value of the string.

4.1.2 On-sets and Off-sets

One way to represent a Boolean function more parsimoniously than with its truth table is to represent only the inputs which map to 1, called the *on-set* of the function, since we can assume the unrepresented inputs map to 0. Figure 4.2 represents the 3 multiplexer using its on-set.[2]

4.1.3 Sigma Notation

We can represent functions even more compactly by assuming the inputs in the truth table are in a regular order, and by identifying table rows, rather than explicitly listing the inputs they contain. This is the approach taken with sigma notation, in which the 3 multiplexer is represented as: $\Sigma(2, 3, 5, 7)$. (The first row in the table is considered row 0, and we list only the rows in the on-set.) This notation actually specifies a class of functions unless we know how many rows the table has. We can either assume a priori that we are working with a table of a given length, or we can assume the length of the table is the lowest power of 2 which will contain the specified rows. We will take the latter approach here.

4.1.4 Disjunctive Normal Form

The strings shown in the input column of figure 4.1 are implicit conjunctions of their components. We could have written the conjunctions explicitly, in which case the string in the first row would be $0 \wedge 0 \wedge 0$, where $\wedge$ is the logical AND operator. The two forms are logically equivalent so explicit use of $\wedge$ is redundant. An expression which is a conjunction of the characters in the input to a Boolean function is called a *minterm*. Each row in the input column of figure 4.1 is a minterm. A representation which expresses disjunctions of minterms is in *Disjunctive Normal Form* (DNF) (see, e.g., [140]).

[2]Equivalently, we can represent a function just by its *off-set*, the set of inputs which map to 0, although to avoid unnecessary complications we will not make further reference to this approach.

Input	Output
000	0
001	0
010	1
011	1
100	0
101	1
110	0
111	1

Figure 4.1: Truth table for the 3 multiplexer.

Input	Output
010	1
011	1
101	1
111	1

Figure 4.2: The 3 multiplexer, with only the on-set (rows mapping to 1) visible.

$0\,0\,\#\rightarrow 0$
$0\,1\,\#\rightarrow 1$
$1\,\#\,0\rightarrow 0$
$1\,\#\,1\rightarrow 1$

Figure 4.3: The 3 multiplexer represented using a set of classifiers.

4.1.5 Representing Functions with Sets of Rules

A classifier maps some subset of the possible input strings to an action, which can be taken as a partial specification of a function (or a full specification if the classifier happens to match all inputs). We can think of the classifier as equivalent to the rows in a truth table whose input column it matches. Any Boolean function can by fully specified by an appropriate set of classifiers; as an example, figure 4.3 shows one way of representing the 3 multiplexer using a set of classifiers.

Sets of Conditions are in DNF

When we denote a classifier as, e.g., $\#\#\#\rightarrow 0$, we are implicitly conjuncting the characters in its condition, as we did with the minterms of §4.1.4. Since the conditions of classifiers are satisfied (or not) independently, a set of conditions is implicitly disjuncted. Thus, a set of conditions in the standard ternary language is in DNF.

Note that a set of classifiers is *not* in DNF. Classifiers consist not only of

conditions, but also actions (and many other parameters, for that matter). We can, however, easily convert any set of classifiers into *two* DNF representations by dividing them into two sets, those with action 0 and those with action 1, and extracting their conditions. One way to think of a classifier is as a minterm (the condition) with an associated truth value (the action).

4.2 How Should a Classifier System Represent a Solution?

In simple, widely-used test problems like the multiplexers it is straightforward to determine by inspection whether a particular rule acts correctly and whether it is too general, or not general enough. But what should a set of rules look like? Even for simple tasks this is not so obvious.

Michalski provides another perspective on this issue:

> "For any given set of facts, a potentially infinite number of hypotheses can be generated that imply these facts. Background knowledge is therefore necessary to provide the constraints and a preference criterion for reducing the infinite choice to one hypothesis or a few most preferable ones.
>
> A typical way of defining such a criterion is to specify the preferable properties of the hypothesis - for example, to require that the hypothesis is the shortest or the most economical description consistent with all the facts..." [181] p. 89.

Michalski suggests that defining a preference for some hypotheses is not an academic issue but a necessity, given the number of possible hypotheses. This chapter suggests and discusses four properties which we may wish to require of hypotheses learnt by LCS. These four apply to other machine learning systems, and other properties are of course possible.

We begin this section by briefly contrasting the value of individual rules and the value of a set of rules. §4.2.1 demonstrates how we can determine the value of a single rule in isolation. Then, following sections discuss four properties which are inherent to *sets* of rules and which may be desirable. In §4.2.3 two important properties of any set of rules an LCS uses to represent a Boolean function are identified, that the set be complete and correct. Then, in §4.2.4 and §4.2.5, two further potentially desirable properties of a set, that it be minimal and that it lack overlaps between rules, are identified.

Following investigation of these four properties some additional subjects are discussed. In §4.2.6 so-called "optimal" sets of rules are discussed, in

§4.2.7 the interpretation of conflicting sets of rules is covered, and in §4.2.8 some differences between representation in XCS and other LCS are discussed.

Before we begin, note that we can distinguish between requiring that a rule set as a whole possess some property, or, less strictly, that some subset of it does. For example, we can insist that the classifier population *is* a minimal representation of a function. Alternatively, we may only insist that it *contains* a minimal representation. The latter may be sufficient depending on our purposes.

4.2.1 The Value of a Single Rule

Part of the appeal of classifier systems is that they work with condition-action rules which are easily interpreted. Using a simple, well-defined problem like the 3 multiplexer from §4.1.1, it is straightforward to evaluate a single rule in isolation. Let's assume the use of some kind of accuracy-based classifier system like XCS, in which the value of a rule is some function of its classification accuracy and its generality.

We can easily determine the classification accuracy of a rule using the 3 multiplexer. For example, it should be clear that $00\#\rightarrow 0$ is a fully accurate rule for this function, as it advocates the correct action (0) for both inputs it matches (000 and 001). The (formal) generality of a rule is even easier to assess, being equal to the number of # symbols in the rule's condition. (It is more difficult to assess the effective generality of a rule, that is, the number of inputs it matches in practice.)

Not only can we easily evaluate rules, we can also easily generate new rules, which allows us to search for superior rules. Returning to the example of $00\#\rightarrow 0$, we can exhaustively check all more general conditions ($0\#\#$, $\#0\#$, and $\#\#\#$) to see whether any of them are fully accurate. As none are, we know that no rule is superior to $00\#\rightarrow 0$ in this problem. (Note that a number of rules are equally valuable, e.g., $1\#1\rightarrow 1$.)

4.2.2 The Value of a Set of Rules

Although evaluating individual rules for simple problems is easy, evaluating *sets* of rules is not so straightforward. What makes one representation of a Boolean function more valuable than another? This question naturally applies not only to LCS but to other learning systems, and has been addressed in the machine learning literature. [181], for example, lists a number of criteria one might use to evaluate sets of rules.

This section identifies four properties of rule sets relevant to learning in classifier systems.

Input	Output
000	0
001	0
010	0
011	0
100	0
101	0
110	0
111	0

Figure 4.4: Truth table for the 3-bit constant 0 function.

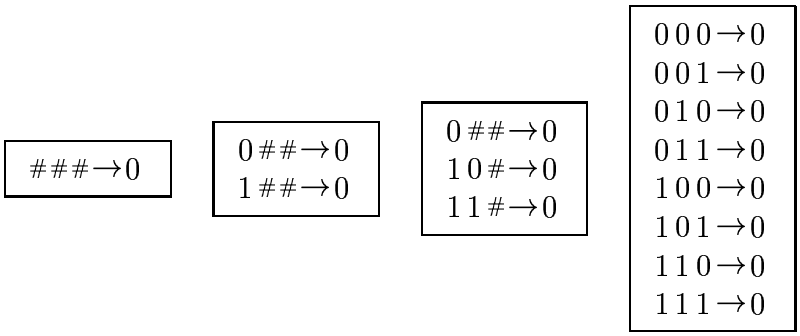


Figure 4.5: Four ways of representing the 3-bit constant 0 function with sets of classifiers.

4.2.3 Complete and Correct Representations

Let's consider a particularly simple function, the 3-bit constant 0 function, whose truth table is shown in figure 4.4. Figure 4.5 shows four representations of this function using the ternary LCS language. Notice that the rightmost set of rules corresponds directly to the truth table – it contains a rule for each row in the table. Valenzuela-Rendón called such a set a *one-to-one* rule set [281].

The four representations in figure 4.5 do indeed represent the 3-bit constant 0 function, whereas the two sets in figure 4.6 do not; the left represents only a subset of the input/output cases, while the right simply does not represent the desired input/output mapping.

The four rule sets in figure 4.5 suffer from neither of these problems. To emphasise this, I define two properties which they share:

Figure 4.6: Two sets of rules which misrepresent the constant 0 function.

Property 1. Completeness. *The rule set maps each possible input to an action.*

Property 2. Correctness. *The rule set correctly represents the intended input/output function by mapping each input to the correct action.*

A More Formal Statement

Michalski formalises these two concepts in [181], and we follow suit, treating rules as partial functions as noted on page 28.

- A set of classifiers C is complete for some function f (denoted $complete(C, f)$) iff:
$$\forall s \in \mathcal{S}(f)\ \exists c \in C\ s \in \mathcal{S}(c)$$
- A classifier c is correct with respect to f (denoted $correct(c, f)$) iff:
$$\forall s \in \mathcal{S}(c)\ f(s) = c(s)$$
- A set of classifiers C is correct with respect to f (denoted $correct(C, f)$) iff:
$$\forall c \in C\ correct(c, f)$$

Note that correctness implies completeness.

Although these properties may seem essential requirements of any ideal solution an LCS finds, the next section questions to what degree we should enforce correctness. Later, in §4.2.8, we'll see that these two properties are somewhat different when we apply them to XCS.

How Much Emphasis Should we Place on Correctness?

It seems uncontroversial to assert the desirability of a set of rules representing a function completely, rather than representing only a subset of it. At first, that the rule set also be correct would appear similarly uncontroversial. However, this section briefly considers problems more complex than the

basic framework of learning Boolean functions from noiseless input/output exemplars and suggest that in such cases it may be counterproductive to emphasise correctness excessively.

Section §2.3.3 outlined a testing framework in which the objective of the learning agent is to approximate the data upon which it is trained as closely as possible. In this framework the training data is perfect; all input/output cases are available, and the system is tested on the same data on which it is trained. In this case, it is indeed desirable that a rule set be a correct representation of what it has learnt.

Now consider dealing with more complex frameworks, e.g., mining data from a real world database, in which the training data is incomplete, may contain uncertainty (conflicting exemplars), or for some other reason only approximates the test data (which may contain its own uncertainty). If we emphasise obtaining a rule set which is correct – that is, fully consistent with the *training* data – we risk overfitting the training data (see, e.g., [184]).

4.2.4 Minimal Representations

While the four representations of the constant function in figure 4.5 are all complete and correct, the leftmost seems preferable in that it represents the function most compactly. In fact, it is a *minimal* representation of this function, using this language. In this section we consider the value of minimising the number of rules in a representation.

Property 3. Minimality. *The rule set contains the minimum number of rules needed to represent the function completely and correctly.*

A set of rules C is a minimal representation of a function f iff:

$$\begin{gathered} \mathit{complete}(C, f) \text{ and} \\ \mathit{correct}(C, f) \text{ and} \\ \forall r \in 2^R \text{ if } \mathit{complete}(r, f),\ \mathit{correct}(r, f) \text{ then } |r| \geq |C| \end{gathered}$$

where R is the set of all syntactically possible rules, and 2^R is the power set of rules.

Minimal DNF Representations

A set of rules with the three properties we have seen so far (a set which is correct, complete, and minimal) is a minimal DNF representation of a Boolean function. Thus, if the goal of our LCS is to evolve a complete, correct, and

minimal set of rules, its goal is to evolve a minimal DNF representation of a function (which, incidentally, is the goal of traditional minimisation algorithms such as the Quine-McCluskey method (see, e.g., [177, 6, 69])).

The Desirability of Minimality

The search for general, accurate rules – and consequently smaller populations of rules – has been a continuing theme in XCS research. Indeed, the minimisation of population size has been an explicit goal of some work (e.g., [298, 144, 304, 147]). Why has work with XCS assumed smaller, but nonetheless complete and accurate, populations are more desirable than larger ones?

First, in the absence of a drop in performance, a reduction in the size of the population can be taken as an indirect indication that an LCS is finding accurate and general rules. To decrease population size without affecting performance an LCS must remove redundant rules from its population. One way to make rules redundant, without increasing population size, is to increase the proportion of accurate, general rules in the population, since such rules subsume other rules.

As an example of subsumption, suppose a population includes two accurate rules $000 \rightarrow 0$ and $100 \rightarrow 0$. Now suppose the system introduces $\#00 \rightarrow 0$, which subsumes the first two rules. If the last rule is accurate, the first two become redundant and can be removed, resulting in a net reduction of the population size.

Another reason why a reduced population size is taken as a good sign is that a very low population size and good performance are only possible when existing rules are quite general. Otherwise the population cannot cover the entire function, which would result in poor performance.

General (but accurate) rules are valuable because, without generalisation, learning systems cannot hope to adapt to complex environments. (See [304] for a brief discussion.) One consequence is that we cannot hope to understand learning in animals without understanding generalisation.

Of course, it is possible for an LCS to find accurate, general rules but not remove the rules they subsume. A reduction in population size confirms that redundant rules are indeed being removed, and suggests the method by which the LCS selects rules for deletion is effective.

Smaller populations have also been sought because parsimony has at least two benefits, the first being efficiency. The main computational cost of running a classifier system lies in processing its population of rules. Minimising the rule population by removing less valuable rules reduces the cost of running an LCS. One way in which it does so is by reducing run-times on serial machines, and the number of processors used on parallel machines. Another

way in which removing redundant rules improves efficiency is that it can focus the search engine (typically a genetic algorithm) on the more valuable rules which remain in the population.

A second reason to value parsimony is the argument that more parsimonious representations are more comprehensible. In general, the fewer rules the human mind needs to interpret, the more comprehensible the set of rules should be. By removing subsumed rules, such as $000 \rightarrow 0$ above, the solutions found by a classifier system may be easier for us to understand.[3] Having said this, it is debatable whether direct inspection of rule populations is the best means of extracting information from them. Perhaps tools will be developed to assist us in this.

Finally, a bias towards parsimony is commonly used in choosing between inductive hypotheses, in both machine learning systems and scientific enquiry in general. However, the issues concerning the appropriateness of this bias are somewhat involved and require far more lengthy discussion than is possible here. See [184] for discussion of Occam's razor and the minimal description length principle.

Disadvantages of Minimality

In addition to the advantages suggested in the last section, the pursuit of minimality may have some disadvantages. For one, the more rules the system contains, the more hypotheses it evaluates, on average, for each input. This might allow the system to find good hypotheses more quickly. Additional disadvantages of minimality will be discussed in §4.2.5.

DNF Complexity and Problem Difficulty

As an aside, we can use the minimum number of rules needed to correctly and completely represent a function as a measure of the function's complexity in a given language [154]. Since the standard ternary LCS language is in Disjunctive Normal Form, this measure of complexity will be referred to as the *DNF complexity* of a function. As an example, the minimal DNF representation for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$ is shown in figure 4.7. Since 4 rules are needed to represent this function, its DNF complexity is 4.

[3] Wilson [304] also speculates that future introspective learning systems may benefit from more comprehensible representations just as humans do.

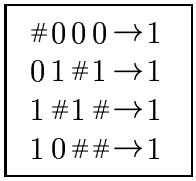


Figure 4.7: Minimal DNF rule set for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$.

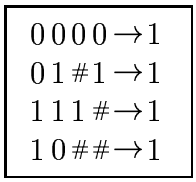


Figure 4.8: Minimal non-overlapping DNF set for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$.

4.2.5 Non-overlapping Representations

One feature of the rules in figure 4.7 is that they overlap; that is, there are inputs which are matched by more than one rule.[4] For example, rule 10## matches the inputs $\{1000, 1001, 1010, 1011\}$ and rule #000 matches the inputs $\{0000, 1000\}$. These rules overlap as they both match 1000.

Some sets of rules do not have such overlaps, which is the fourth and final property we will consider:

Property 4. A Lack of Overlaps. *No two rules advocating the same action match a common input.*

A set of classifiers C is non-overlapping iff:

$$\forall c_1, c_2 \in C \text{ where } c_1 \neq c_2,$$

$$\text{if } \mathcal{A}(c_1) = \mathcal{A}(c_2) \text{ then } \mathcal{S}(c_1) \cap \mathcal{S}(c_2) = \emptyset$$

We can produce a non-overlapping version of the rules in figure 4.7 by reducing the generality of some of the rules, as shown in figure 4.8. Like the overlapping set, this rule set also has a DNF complexity of 4.

Note that we can classify a pair of overlapping rules into two categories: 1) one rule is subsumed by the other, and 2) neither rule subsumes the other.[5]

[4]Note that these rules all advocate the same action. When rules advocating different actions match the same input we have a conflict – see §4.2.7.

[5]We could distinguish a third case in which the two rules are equivalent.

Input	Action	Reward
000	0	900
100	0	?
101	0	800

Figure 4.9: Part of a reward function.

Why Non-overlapping Solutions?

Two LCS which penalise overlapping rules are XCS, which has an *implicit* fitness bias against them, and Booker's endogenous-fitness LCS [36], which has an *explicit* fitness bias against them. In both cases, the bias against overlapping rules is designed to promote diversity in the rule population. That is, if we want to encourage a broad covering of the input (or input/action) space, we can do so by penalising redundancy in the existing covering. In other words, overlaps are penalised to encourage the completeness property. Wilson suggests another reason to penalise overlaps is to penalise subsumed rules, on the assumption that the more general of two valid hypotheses is preferable ([293] p. 227).

By penalising overlapping rules, however, we increase the odds of producing a non-overlapping solution, which is *not* a goal entailed by the desire to maintain diversity (completeness). That is, non-overlapping solutions are a byproduct of pressure to maintain diversity in a population of rules. Non-overlapping solutions are not entailed by the goal of removing subsumed rules either, but again may be a byproduct of penalties intended for such rules.

It is not clear that solutions with fewer (or no) overlaps, are advantageous. Consider what happens during training when the learner is presented with an input. If only one rule matches this input, it only has the opportunity to evaluate a single rule. But if many rules match, it can evaluate many rules using the same amount of experience.[6]

Now consider the problem of training on a subset of the input space and generating inductive hypotheses for the missing subset. In this case too it would seem advantageous to have overlaps in the missing inputs. In this case, the overlaps constitute additional hypotheses about the unknown subset which can be confirmed or refuted if experience with it later becomes available. (The same argument may hold when the training set consists of the entire input space, but when some inputs may occur infrequently, as when

Condition	Action	Prediction
#00	0	900
10#	0	800

Figure 4.10: Two overlapping rules which are consistent with the reward function in figure 4.9.

the training set is very large.)

As an example, consider the partially specified reward function for a 3-bit problem shown in figure 4.9. The inputs 000 and 101 occur, but the input 100 never occurs during the training phase. Figure 4.10 shows two rules which are consistent with the rewards in figure 4.9, and the rewards they predict they will receive. Note that these rules overlap, and that they predict different rewards: $10\#\rightarrow 0$ predicts a reward of 800 while $\#00\rightarrow 0$ predicts a reward of 900. In other words, the two rules represent two hypotheses about how much reward will be received for taking action 0 in response to input 100, if it ever occurs. Which rule is correct? We cannot know, since figure 4.9 doesn't specify a value for this case. Either, or neither, rule may turn out to be correct in light of further information.

Suppose our learning system is attempting to extract regularities from the data which are then to be inspected by a human expert. Given the absence of input 100 in the training set the best our learner can do is output a range of hypotheses regarding it.

With a powerful enough syntax the system could generalise *any* rule to cover the unseen input. So the hypotheses output by the system regarding 100 may just be a result of the system's bias, and there may be no reason to think they are particularly valuable. However, if it generates hypotheses regarding 100 by generalising from nearby regions of the syntactic space, such hypotheses will tend to have above average value if the data has appropriate local continuity. (See [288].)

A final problem with a lack of overlaps is that redundancy can help make a system robust. If a valuable rule is deleted by chance, rules which are otherwise redundant can help the system cope without it.

These difficulties suggest that rather than stressing minimality, or a lack of overlaps in general, perhaps we should specifically penalise subsumed rules. This would minimise the size of the population to some extent, without being prejudicial to diversity.[7]

[6] This is reminiscent of Holland's idea of implicit parallelism in genetic algorithms [119].

[7] This is precisely the effect of Wilson's subsumption deletion operators [304].

0#→0			#0→0
10→0	=		01→0
11→1			11→1

Figure 4.11: Two [O]s for the same function.

4.2.6 Optimal Rule Sets: [O]s

In earlier work the author called a rule set with the four properties just described (completeness, correctness, minimality and non-overlappingness) an *optimal rule set* (or an *optimal population*), denoted [O] [144].[8]

Just as there may be multiple minimal DNF representations for a function, there may be multiple [O]s, as demonstrated by the two [O]s for the same function in figure 4.11.

The four properties of an [O] were identified because XCS showed a tendency to evolve solutions with these properties. However, the suggestion in §4.2.5 that a lack of overlaps is a side-effect of pressure to promote diversity, rather than a desirable feature of a rule set, in turn suggests that its inclusion in the definition of an optimal population is unwarranted. Rather than focus on learning [O], perhaps we should focus on learning minimal DNF representations. We will evaluate XCS's ability to do so in §4.3.6.

As an aside, Wilson has long made use of [O] for the specific case of the 6 multiplexer (see §4.3.2), and it is this work which prompted me to use it and eventually consider its properties explicitly.

4.2.7 Conflicting Rules

When an LCS evolves rules there is typically a great deal of disagreement between them as to which action to take in a given state. To choose an appropriate action we rely on the *strength* of the rules to give us an indication of the utility of taking their actions. Consequently, for an LCS to act correctly it must assign rules appropriate strengths.

4.2.8 Representation in XCS

XCS differs from other LCS in that it maintains a complete map of the input/action space (§3.6.3). That is, it tends to evolve sets of rules in which each input/action pair is covered by at least one rule. Consequently, both

[8] In [154] the DNF complexity of an [O], denoted |[O]|, is used as a complexity metric.

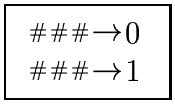


Figure 4.12: XCS represents both the correct and incorrect actions for any function, in this case a 3-bit constant function.

the correct and incorrect actions for each input will be advocated, as demonstrated by XCS's representation of a 3-bit constant function in figure 4.12. (More generally, XCS will represent all actions, so for a constant function with a actions XCS will tend towards a representation with a rules.)

This tendency towards a complete map means that, when dealing with XCS, we must modify the completeness property we saw earlier.

- For XCS, a complete rule set must map each input/action pair (not just each input).

In XCS, a set of classifiers C is complete with respect to a function f iff:

$$\forall s \in \mathcal{S}(f),\ \forall a \in \mathcal{A}(f),\ \exists c \in C \text{ such that } s \in \mathcal{S}(c),\ a \in \mathcal{A}(c)$$

Like other LCS, XCS relies on rule strength in order to choose between rules when selecting an action. (It also factors rule fitness into its calculation ([298], §2.3.4).) Consequently, the conflicting actions represented by the rules in figure 4.12 are not a problem for XCS as long as it has assigned them appropriate strengths. However, a complete map means the rule set cannot simply advocate the correct action for each input.

- For XCS, a rule set must be accurate, not just correct.

On page 43 a rule r in XCS was defined to be accurate iff:

$$\varepsilon_r < \varepsilon_o$$

That is, a rule is accurate if its prediction error falls under the accuracy criterion threshold.

4.3 How Should We Measure Performance?

In this section we turn to the question of how to measure the properties discussed in §4.2. There are many ways in which we might measure an LCS's performance in solving a problem, and many issues to be addressed.

Much of this broad subject is relevant to any learning system, and entire books are devoted to subjects such as statistical methods.

One study devoted specifically to metrics for use with LCS is that by Holmes [129], which introduced a number of metrics for use in two choice decision tasks and demonstrated their superiority over classification accuracy. Note that Holmes's study and the present work are complementary, and that Holmes's concerned itself solely with what we refer to herein as performance metrics.

We will restrict our considerations to on-line performance metrics for the learning of Boolean functions, such as those we would use with the experimental framework described in §2.3.3. Such metrics are divided into two categories. To give concrete examples, a number of metrics which have been used with XCS are described and classified as either performance metrics (§4.3.1) or population state metrics (§4.3.2). The two types are then contrasted, an advantage of population state metrics is demonstrated, and difficulties with using these metrics are discussed (§4.3.3). In §4.3.4 two new population state metrics are introduced, and they are evaluated in §4.3.5 and §4.3.6. Finally, in §4.3.7 the properties to which various metrics are sensitive are summarised.

4.3.1 Measures of Performance

Wilson's work with XCS makes use of two measures which depend on different aspects of performance.

Wilson's Performance Metric: $\mathcal{P}$

Wilson defines a measure of performance which he refers to simply as "performance" [298], but which we refer to as $\mathcal{P}$ in order to distinguish it from the more general notion of performance.

$\mathcal{P}$ was used in evaluating XCS and SB–XCS on the 6 multiplexer (§2.5.1), and is defined as a moving average of the proportion of the last n trials in which the system has responded with the correct action, where n is customarily 50. $\mathcal{P}$ is a form of classification accuracy metric, since it depends on the proportion of correct and incorrect classifications.

System Error

The second measure of performance in [298] is system error, which is a moving average over the last n trials of the difference between the system prediction (page 36) and the reward, where again n is customarily 50. (Wilson also

normalises system error to between 0 and 1.) Where $\mathcal{P}$ measures the proportion of XCS's actions which are correct, system error measures how well XCS predicts the rewards it receives.

4.3.2 Measures of Population State

Measures of performance, such as $\mathcal{P}$ and system error, are distinguished from measures of the population state, such as the following two.

Population Size

Wilson used a third metric with XCS, the population size, which is a count of the number of macroclassifiers in the population. (Macroclassifiers are rules with a *numerosity* parameters which specifies how many identical *virtual* classifiers they represent – §2.3.2.)

Proportion of an Optimal Rule Set: %[O]

%[O] is defined as the proportion of an optimal rule set which is present within the complete rule set on a given time step. This metric, for the specific case of the 6 multiplexer function, has long been used by Wilson [293, 294] who denoted it [S6].[9]

A difficulty in using this metric is that, as mentioned in §4.2.5, there may be more than one [O] for a given function. To date %[O] has never been used with a function with multiple [O]s, so these issues have never before been addressed. To deal with this problem we can select one of the [O]s at random for generating the %[O] statistic, although this is not entirely satisfactory since we are arbitrarily focusing on one [O] to the exclusion of the others. However, [O]s for a given function tend to be similar, which means that any population which contains one [O] is likely to contain, or be close to containing, the others. An alternative approach to this problem would be to define %[O] as the union of the [O]s for the function, although this will be a rather large set for some functions with longer input strings. A third alternative would be to measure %[O] for each [O] and use the maximum %[O].

A problem may arise in systems in which there is a great deal of pressure against overlapping rules. Such systems are more inclined to converge to a

[9] In fact, without explicitly identifying these properties, Wilson noted that [S6] forms a correct, complete and non-overlapping representation, and suggested that it is also minimal ([293] p. 212) (which it is, assuming this representation), thus foreshadowing the present work.

single [O], which could cause problems using either of the first two approaches suggested above. However, the use of larger population sizes, with their greater capacity to support diversity, should moderate this difficulty and make systems less likely to converge to a single [O]. This hypothesis is tested in §4.3.6.

4.3.3 Measuring Performance and Measuring State

$\mathcal{P}$ and system error *directly* measure a system's performance on some task, that is, they are generated by observing the interaction of the system with the problem environment. Such measures only *indirectly* measure the state of the population. Although the state of the population affects performance and hence performance metrics, such metrics are not generated from the population state.

Population state measures, in contrast, are generated based on the current population, and so are a *direct* measure of its state. However, they do not depend on how the system performs on the problem task, and so can give only an *indirect* indication of how the system is likely to perform.

Performance metrics are appropriate when we are interested in how well the system is interacting with its task environment. However, when we wish to learn about the rules which produce the system's behaviour – how it represents knowledge – we must use population state metrics. We must use population state metrics to study the four properties of §4.2.

Sensitivity of Metrics

Measures of population state can detect effects on the population which measures of performance do not. This can make population state measures particularly valuable in comparisons of alternative LCS mechanisms and their parameterisation, as they may affect the population state rather than (or more so than) the performance of the system.

To demonstrate this, XCS's performance using the t2 and t3 deletion schemes from [147] was evaluated on the 6 multiplexer (see §2.3.6). Using t2, a rule's probability of being selected for deletion is based on the extent to which it overlaps with other rules. In addition, a penalty against low fitness rules makes them more likely to be deleted. Using t3, deletion probability is calculated the same way, except that the low fitness penalty is applied only to rules which have been evaluated a minimum number of times. This protects newly generated rules until they have been well evaluated.

XCS was configured as in the original XCS paper, except that the niche GA ran in the action set, and GA subsumption deletion was employed [304].

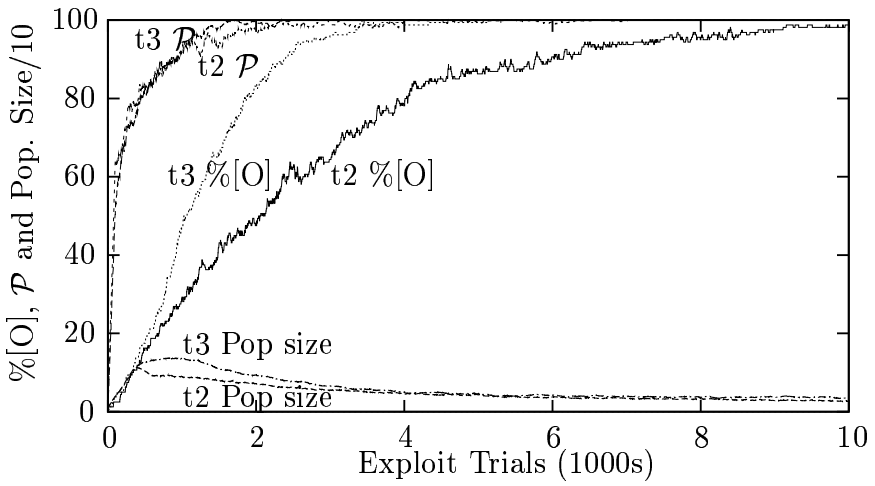


Figure 4.13: Performance and population state measures compared on the 6 multiplexer at distinguishing the effects of two deletion schemes t2 and t3. Curves are averages of 10 runs.

For this and subsequent tests the standard XCS settings were used (p. 59), and a delay of 25 was used for the t3 deletion scheme.

Figure 4.13 shows XCS's %[O], $\mathcal{P}$ and population size on this test, with curves averaged over 10 runs. t3 outperforms t2 slightly on the $\mathcal{P}$ metric, while t2 outperforms t3 on the population size metric. From consideration of only these two metrics we might conclude that t2 trades smaller population sizes for slightly reduced performance compared to t3. However, when we consider %[O], we can see that the use of t2 is highly detrimental to the evolution of the optimal rule set. Thus, %[O], by directly measuring the state of the population, reveals a problem with t2 which does not show up on $\mathcal{P}$, an indirect measure of the population state. Other, similar, population state metrics (such as %[m-DNF], to be introduced in §4.3.4) should show the effect %[O] does in this comparison, since the effect does not depend on a lack of overlaps.

This is not to suggest that population state metrics are simply superior to performance metrics. If, for example, we wished to observe the effect of an exploration control policy on performance, we would expect a performance metric to be more sensitive to it.

Obtaining Rule Sets for Population State Metrics

There are methods for computing the various rule sets (e.g., [O]) we might want to use with population state metrics given a complete specification

of the function to be learnt (see, e.g., [6, 69]). Sometimes, however, such information is unavailable. For example, we might reasonably apply an LCS to the problem of predicting a subset of attributes in a database from other attributes. In addition to the problem of the dataset providing an incomplete specification of the input/action space, there may well be inconsistencies in the input/output mapping specified by the database.

Obtaining ideal rule sets such as [O], minimal DNF rule sets, and [PI] (this last to be introduced in §4.3.4) means finding an ideal solution to the problem in advance of applying the LCS to it, which is clearly not always practical.

Particular difficulties with this approach can arise in using representations other than the standard ternary language. For one, it may be difficult to determine ideal rule sets using representations featuring variable-length rules. Additionally, in this case, and that of representations featuring real-valued conditions, it may well be unreasonable to expect a classifier system to find exactly the ideal solution simply due to the size of the search space and the minor differences between very similar rules. In such cases the approach of population state metrics such as %[O] may be infeasible. Note, however, that if our objective is to evaluate some component of a classifier system which is not specific to such a representation, for example a deletion scheme, we can evaluate it using the standard representation and avoid these difficulties. Presumably the merits and deficiencies a deletion scheme shows with the standard representation will tend to be found when it is used with more advanced representations. So a first step in evaluating a new mechanism could be to try it using simple, well-understood problems and the standard representation, where we can use more powerful metrics like %[O]. If it performs well there, we can then test it with weaker metrics using interesting problems and more advanced representations where we cannot use metrics like %[O].

As a consequence of the difficulties outlined here, metrics like %[O] are not suitable in all situations. However, their use can be advantageous under appropriate conditions, as demonstrated in the preceding section. We will see another case where they will prove advantageous in §4.3.5.

Computational Expense

The computational complexity of performance metrics is typically very low, since they are based only on the feedback provided by the learning environment to the learning agent, i.e., the correct action in the case of supervised learning and a reward in the case of reinforcement learning. The complexity of population state metrics, however, can depend on the size of the rule

population, and can be non-trivial. For example, in monitoring %[O] for the 37-bit multiplexer, we need to check each newly generated rule against a rule set of size 64. Worse, for a 10-bit parity problem we must check each rule against a set of size 1024 (for parity problems, the size of such sets is exponential in the number of input bits).

Another difficulty is that the longer the input string, the more minimal DNF and optimal solutions a function may have, increasing the computational expense of using %[O], and exacerbating the problems with %[O] discussed in §4.3.2.

4.3.4 New Population State Metrics

Section 4.2 questioned the utility of two properties: minimality and a lack of overlaps. This section introduces new metrics which are suitable when we are not interested in one or both of these properties.

%[m-DNF]

If we leave aside the non-overlapping property and deal with the remaining three we are looking for rule sets whose conditions are minimal DNF representations (see §4.1.4 and §4.2.4). We face the complication that multiple minimal DNF rule sets are possible for a given function, just as we faced the complication of multiple [O]s in using %[O] (see §4.3.2), and can again adopt the solutions of selecting one at random, using their union, or taking the maximum %[O].

The proportion of a minimal DNF representation present in the current population will be used as a population state metric. It will be denoted %[m-DNF], and it will be used with XCS in §4.3.6.

%[PI]

What should the representation of a function look like if we leave aside the requirements that it lack overlaps and be minimal? In this case we would like the rule set evolved by an LCS simply to contain as many of the *optimally general* rules as possible. (Recall that optimally general rules are rules which cannot be made any more general without violating the correctness property.) The conditions of optimally general rules are termed *Prime Implicants* in the minimisation literature, and we will denote their set [PI]. We can use the proportion of this set which is found in the rule population at the current time step as a population state metric, which we will denote %[PI], and which we will use with XCS in §4.3.5.

Unlike [O] and minimal DNF representations of a function, there is a unique [PI] for each function, and consequently we do not face the issue of selecting between alternative [PI]s or using their union. However, the utility of %[PI] may be limited by the large number of prime implicants a function may have: there can be on the order of 3^n prime implicants for a function of length n. For example, there are 10-bit functions which have around 59,000 prime implicants, a number which is likely to prohibit the use of %[PI]. This may tend to limit the use of %[PI] to relatively small functions.

Certainty-equivalent Minimisation

One case in which we might set aside the requirements of minimality and a lack of overlaps is when we intend to use condensation [298, 144] at the end of a run to obtain a more compact representation of the solution. Another case is an approach called *certainty-equivalent minimisation*, which is suggested here, but which cannot be elaborated much in the present work. As noted in §4.2, there is a parallel between learning in LCS, particularly XCS, and traditional function minimisation techniques such as the Quine–McCluskey method (see, e.g., [177, 6, 69]). An interesting approach would be to combine the two, namely, to employ an LCS to learn a representation of a function, and then apply some traditional minimisation technique to this representation, rather than requiring the LCS to do the minimisation. In this case %[PI] might be a suitable metric with which to evaluate the performance of the classifier system.

4.3.5 Testing XCS with %[PI]

To investigate the new metric %[PI] XCS was run, configured as in §4.3.3, on the standard 6 multiplexer benchmark (§2.5.1). (The t3 deletion scheme was used in this and the subsequent test.)

[PI] for the 6 Multiplexer

Figure 4.15 shows the set of conditions which are prime implicants for the 6 multiplexer. On the left are those in the off-set §4.1.2, and on the right are those in the on-set. Because XCS tries to learn complete maps §3.6.3, it will try to find a pair of rules for each of these conditions, one of which advocates action 1 and the other action 0. The union of the pairs of rules produced from these two sets of conditions was used as the 36-element set [PI].

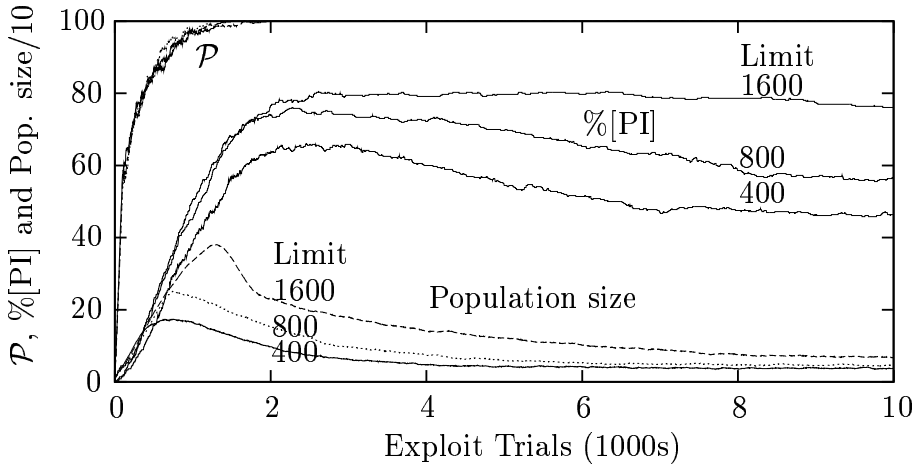


Figure 4.14: The effect of the population size limit on the proportion of the set of prime implicants %[PI] found by XCS for the 6 multiplexer. Population size limits of 400, 800 and 1600 were used. Curves are averages of 10 runs.

1.	000###	**10.**	001###
2.	01#0##	**11.**	01#1##
3.	0#00##	**12.**	0#11##
4.	10##0#	**13.**	10##1#
5.	11###0	**14.**	11###1
6.	1###00	**15.**	1###11
7.	#1#0#0	**16.**	#01#1#
8.	#00#0#	**17.**	#1#1#1
9.	##0000	**18.**	##1111

Figure 4.15: Prime implicants for the 6 multiplexer. On the left is the off-set and on the right the on-set.

Results and Discussion

It was found that XCS had great difficulty maximising %[PI], which was attributed to its strong pressure against overlapping rules. To investigate the effect of the population size limit on %[PI] the test was repeated with population size limits of 400, 800 and 1600, with results shown in figure 4.14. Increasing the population size limit improved %[PI], but even with a population of 1600 rules it was unable to find and maintain the complete set, even though this consists of only 36 rules.

That XCS fails to maintain all 36 rules is striking because its genetic algorithm generates approximately 12,000 rules during each run, from a space of only $3^6 \cdot 2 = 1458$ possible unique rules. (There are $3^6 = 729$ possible 6-bit conditions, and 2 possible actions for each.) Note that with the largest population size used (1600 rules), we could simply enumerate all 1458 possible rules and obtain a complete [PI]. Clearly XCS is strongly biased against maintaining the rules in [PI].

Though it does not maintain a complete [PI], XCS is known to reliably find and maintain the complete [O] for the 6 multiplexer test using a population of only 400 rules (e.g., [144], §2.5.1). That [O] is a non-overlapping 16-rule subset of [PI] should give a further indication of the magnitude of the bias against overlapping rules in XCS. The author has been aware of this bias for some time, but these results are the first indication of how very strong it is.

Note that the $\mathcal{P}$ metric – which produced the three largely overlapping curves in the upper left of the figure – gives no indication that XCS has difficulty maintaining the rules in [PI], and makes little distinction between the population size limits used. This constitutes another example of a population state metric yielding insights into the system which a performance metric overlooks.

Results using %[O] on the 6 multiplexer were shown in §2.5.1, and, for this particular function, [O] and [m-DNF] are identical. Consequently, results using %[m-DNF] are not shown.

4.3.6 Testing XCS with %[m-DNF]

To evaluate XCS (configured as before) and the %[m-DNF] metric $\Sigma(3, 5, 6, 8, 15)$ was chosen. This is a 4-bit function which has 32 minimal DNF representations, the maximum for a 4-bit function. This maximum was determined by the computationally expensive method of enumerating all possible 4-bit functions and finding all minimal DNF representations of each using the Quine–McCluskey method. The chosen function was then selected at random from among those with a maximal number of DNF representations. To evaluate XCS %[m-DNF] was defined as the proportion of the union of all 32 minimal representations, which forms a set of 36 rules.

%[m-DNF] for $\Sigma(3, 5, 6, 8, 15)$

Figure 4.16 shows the union of the conditions in the 32 minimal DNF representations of the 4-bit Boolean function $\Sigma(3, 5, 6, 8, 15)$, divided into those in the off-set (on the left) and those in the on-set (on the right). The union

1.	#001
2.	#100
3.	00#0
4.	0111
5.	1#01
6.	1#10
7.	10#1
8.	101#
9.	#010
10.	0#00
11.	110#
12.	11#0
13.	000#

14.	0011
15.	0101
16.	0110
17.	1000
18.	1111

Figure 4.16: Union of the minimal DNF representations of $\Sigma(3, 5, 6, 8, 15)$.

of the pairs of rules obtained from these sets is used as the 36-element set %[m-DNF].

Results and Discussion

Figure 4.17 shows XCS's performance on this function using population size limits of 400, 800 and 1600. Even with 1600 rules XCS was unable to find the complete set of 36 rules, even though there are only $3^4 = 81$ possible conditions of length 4, and (since there are 2 actions) $3^4 \cdot 2 = 162$ different rules in XCS's search space. This despite the fact that XCS generates approximately 12,000 rules during each test.

These results confirm the hypothesis of §4.3.2 that systems with strong pressure against overlapping rules would have difficulty finding the union of the minimal DNF representations of a function, and that metrics based on this approach, or the approach of selecting a minimal DNF solution at random, would be problematic for them. These results also emphasise the degree of bias against overlaps in XCS.

4.3.7 Summary of Metrics and Properties

We have reviewed a number of metrics from both the performance and population state categories. Figure 4.18 shows the properties to which various metrics are sensitive. The choice of metric depends on a number of factors including which properties are desired of the solution, and the difficulty of

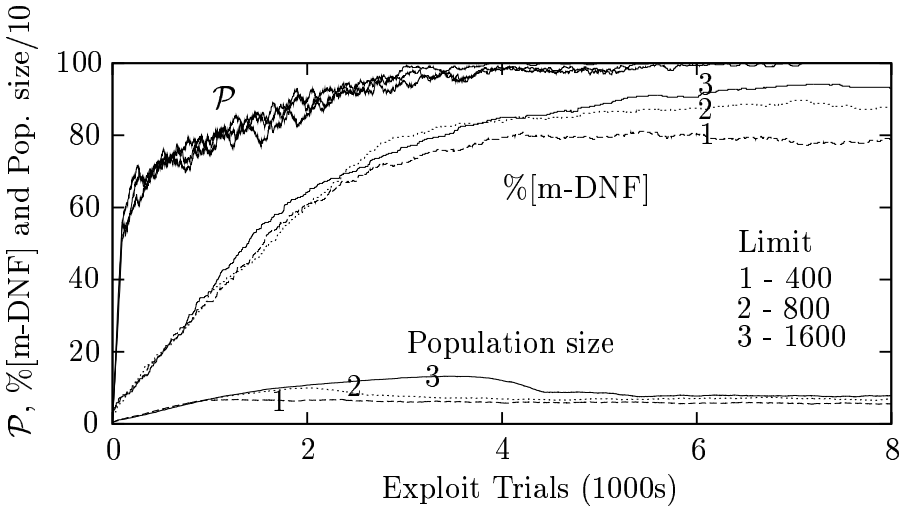


Figure 4.17: The effect of the population size limit on the proportion of the union of the minimal DNF solutions %[m-DNF] found by XCS for $\Sigma(3, 5, 6, 8, 15)$. Population size limits of 400, 800 and 1600 were used. Curves are averages of 10 runs.

	Correct	**Complete**	**Min.**	**Overlaps**	**Accurate**
$\mathcal{P}$	√	√			
System error					√
%[O]	√	√	√	√	
%[m-DNF]	√	√	√		
%[PI]	√	√	✓	✓	
Population size			√		
Mean generality			✓		

Figure 4.18: Properties to which various metrics are sensitive. A small ✓ indicates the metric is indirectly sensitive to the property.

using given metrics with the task at hand. Another dimension, not shown in the figure, is that performance metrics can be made sensitive to exploration or not depending on whether Wilson's pure explore/exploit scheme (§2.3.3) is used.

4.4 Chapter Summary

In response to the question "What should a classifier system learn?", we have distinguished four properties which may be desirable of a classifier system's

representation, and distinguished between the required properties for XCS and SB–XCS (§4.2.8). However, this work only begins to pose and answer questions of what an LCS should learn, and there is undoubtedly much more to be said on the subject. One direction in which this work should be pursued is to borrow from other areas of machine learning, some of which enjoy a greater level of sophistication than classifier systems currently do.

In response to the question "How should we measure what a classifier system has learnt?", we have distinguished two categories of metric: those based on the population state, and those based on performance. We saw the superiority of population state metrics in distinguishing the effects of alternative deletion schemes on the population (§4.3.3), and distinguishing pressure against non-overlapping rules (§4.3.5), and it was suggested that performance metrics will in turn be superior when we are interested in performance (§4.3.3).

In §4.2.5 the utility of finding non-overlapping solutions was questioned, and hence the utility of searching for [O]s (which are overlap free), and of the %[O] metric. The evaluation of %[O] and %[m-DNF] in §4.3.5 and §4.3.6 also showed that they are problematic when used with systems with a strong bias against overlapping rules, as a function may posses multiple [O]s or minimal DNF representations, and such systems may tend to converge to a single [O] or minimal DNF representation. In the process we saw the first evidence of how strong XCS's bias against overlapping rules is. §4.3.4 introduced new population state metrics which may be more suitable than the existing %[O]. §4.3.4 noted that there is a unique set of prime implicants [PI] for each Boolean function, so that the %[PI] metric does not suffer from this problem. Consequently, it may make a good choice of population state metric, although it was noted that the large number of prime implicants a function may have may limit the use of [PI].

Finally, §4.3.3 noted the difficulty of obtaining and using population state metrics such as %[O], %[m-DNF] and %[PI] in many situations. It was suggested, however, that selective use of such metrics can offer additional insight into the behaviour of classifier systems and that they may prove useful in comparing alternative classifier systems and classifier system mechanisms.

Chapter 5

Prospects for Adaptation

Chapter Outline

This chapter considers the conditions under which XCS and SB–XCS can be expected to adapt successfully, and in so doing develops some of the main results of the thesis. In particular, it develops a taxonomy of different rule types and considers the conditions under which they may occur. In order to do so an extreme simplification of the classifier system is made, which forces us towards qualitative rather than quantitative analysis. We begin with the basics, considering definitions for correct and incorrect actions, and then correct, incorrect, and overgeneral rules for both strength and accuracy-based fitness. The concept of strong overgeneral rules, which we claim are the Achilles' heel of strength-based classifier systems, are then analysed. It is shown that they depend on the structure of the reward function (or in sequential tasks, the value function). We distinguish between strong and fit overgeneral rules, and show that although strong overgenerals are fit in SB–XCS, they are not in XCS. We show how to design fit overgeneral rules for XCS, and consider why we might want to use reward functions which may produce strong overgenerals. Finally, we return to the Woods2 task of chapter 2 and explain SB–XCS's performance on it using theoretical arguments and empirical evidence.

XCS and SB–XCS are reinforcement learning classifier systems which evolve sets of rules in order to maximise the reward they receive from their task environment. They employ a genetic algorithm to generate rules, and to do so must evaluate the fitness of existing rules. In order for the GA to produce rules which are better adapted to the task, rule fitness needs somehow to be connected to the rewards received by the system – a credit assignment problem. Precisely how to relate LCS performance to rule fitness has been the subject of much research, and is of great significance because adaptation of rules and LCS alike depends on it.

This chapter undertakes an analysis of the causes and effects of certain rule pathologies in strength LCS and traces them ultimately to the relation between LCS performance and rule fitness – i.e., to the credit assignment system. We examine situations in which less desirable rules can achieve higher fitness than more desirable rules, which constitutes a mismatch between the goal of the LCS as a whole (adaptation to a task) and the goal of the GA (evolution of high-fitness rules).

To study rule pathology we undertake an analysis of what types of rules, and relationships between rules, are possible. Developing earlier work by Cliff and Ross [55], the notion of *strong overgeneral rules* (strong overgenerals, for short) is studied and it is shown exactly what requirements must be met for them to arise in both strength and accuracy-based LCS. In order to compare the two approaches we use the twins XCS and SB–XCS, since they were designed to differ as little as possible, allowing us to isolate the effects of the fitness calculation on performance. The analysis is undertaken using a number of simplifying assumptions outlined in §5.2, and deals first with non-sequential tasks.

It is argued that different definitions of overgenerality and strong overgenerality are appropriate for the two types of LCS (§5.3.2). Minimal conditions and tasks which will support strong overgeneral rules are presented (sections 5.4, 5.5 and 5.6), their dependence on the reward function is demonstrated (§5.4.1), and certain theorems regarding their prevalence are proved under simplifying assumptions (§5.5.2 and §5.6). It is shown that XCS and SB–XCS have different kinds of tolerance for biases (see §5.4.1) in reward functions, and (within the context of various simplifying assumptions) to what extent we can bias them without producing strong overgenerals (§5.6.2). It is also shown what kinds of tasks will not produce strong overgenerals even without our simplifying assumptions (§5.6.1 and §5.12.6).

Next fit overgeneral rules are distinguished (§5.3.4) and it is shown how XCS and SB–XCS differ in their response to them (§5.7). Although rules which are fit overgenerals for SB–XCS are not necessarily fit for XCS, we show that XCS can suffer from fit overgenerals, and we show how to produce

them (§5.8). Following this, the concept of a strong *under*general rule is introduced and it is noted that a generalisation bias in fitness is needed to avoid them (§5.9).

Having established that biased reward functions are needed to produce strong and fit overgeneral rules, we consider in §5.10 why biased reward functions might be used at all, and consider the idea that they might be a useful way of biasing rule allocation. However, in §5.11 we conclude that biasing the reward function is not in general an appropriate way of biasing rule allocation, and consider some alternatives.

Up to this point we will have considered the formation of strong and fit overgenerals in non-sequential tasks, and concluded that they can be avoided simply by using unbiased reward functions. In §5.12 we proceed to consider the more complex case of sequential tasks, and show that sequential tasks amplify difficulties with strong overgenerals. We conclude (§5.13) that SB–XCS is unsuitable for non-trivial sequential tasks and that it appears to have no niche (i.e., no useful domain of application), unless fitness sharing can resolve its problems (§5.14.1). The chapter concludes with consideration of the value of the approach taken and possible extensions (§5.14).

5.1 Known Problems with Strength LCS

In this section we review a number of known problems with strength LCS, which will serve as the starting point for the analysis presented later in this chapter.

Overgeneral Rules

Dealing with *overgeneral rules* – rules which are simply too general – is a fundamental problem for LCS. Such rules may specify the desired action in a subset of the states they match, but, by definition, not in all states, so relying on them harms performance. E.g., an overgeneral which matches 10 states may be correct in as many as 9 or as few as 1. Even overgenerals which are most often correct are (by definition) sometimes incorrect, so using them can harm the performance of the system.

Greedy Classifier Creation

Another problem faced by some LCS is what Cliff and Ross referred to as *greedy classifier creation* [55, 297]. To obtain better rules, a classifier system's GA allocates reproductive events preferentially to rules with higher fitness. This is simply the application of selective pressure in reproduction, one of

the components of the evolutionary process (§G.2), and of itself is not a problem. However, in many classifier systems a rule's fitness depends on the magnitude of the reward it receives. In such systems rules which match in higher-rewarding parts of the task will reproduce more than others. If the bias in reproduction of rules is strong enough there may be too few rules, or even no rules, matching low-rewarding states. (In the latter case, we say there's a gap in the rules' covering map of the input/action space.)

Strong Overgeneral Rules

Cliff and Ross [55] showed that the strength-based ZCS can have serious difficulty even with simple sequential tasks. They attributed ZCS's difficulties to the two problems above, and, in particular, to their interaction, an effect the author refers to [148, 150] as the problem of *strong overgeneral rules*. The interaction occurs when an overgeneral rule acts correctly in a high reward state and incorrectly in a low reward state. The rule is overgeneral because it acts incorrectly in one of the states, but at the same time it prospers because of greedy classifier creation and the high reward it receives in the other state.

Lettau and Uhlig [169, 170] independently discovered strong overgeneral rules using a very different approach from Cliff and Ross's. As they put it:

> " ... a suboptimal rule might dominate the optimal if it is applicable only in "good" states of the world: bad decisions in good times can "feel better" than good decisions in bad times." [170] p. 153.

Sequential and Non-sequential Tasks Although both Cliff and Ross and Lettau and Uhlig dealt exclusively with sequential tasks, the problems discussed above clearly also apply in non-sequential tasks, as this chapter will demonstrate. In fact, examples of trivial non-sequential tasks which produce strong overgenerals will be shown, and it is in these cases that analysis is simplest.

Significance The proliferation of strong overgenerals can be disastrous for the performance of a classifier system: such rules are unreliable, but outweigh more reliable rules when it comes to action selection. Worse, they may prosper under the influence of the GA, and may even reproduce more than reliable but low-rewarding rules, possibly driving them out of the population. For these reasons, and for their prevalence, strong overgenerals (and the related fit overgenerals of §5.3.4) are a major difficulty – perhaps the major difficulty – for strength-based classifier systems.

5.2 Methodology for Rule Type Analysis

Classifier systems are complex systems and analysis of their behaviour can be quite difficult. To make our analysis more tractable we'll make a number of simplifications, perhaps the greatest of which will be to study very small tasks. Although very small, these tasks illustrate different types of rules and the effects of different fitness definitions on them – indeed, they illustrate them better for their simplicity.

Another great simplification will be to deal initially with the much simpler case of non-sequential tasks rather than sequential ones. Sequential tasks present their own difficulties, but those present in the non-sequential case persist in the more complex sequential case; after all, non-sequential tasks are just the special case of sequential task in which $\gamma = 0$ (compare equations (2.2) and (2.3) on p. 41). Study of non-sequential tasks can uncover fundamental features of the systems under consideration while limiting the complexity which needs to be dealt with. Consequently, the analysis of rule types considers non-sequential tasks, and sequential tasks are dealt with only later in §5.12.

To further simplify matters we'll remove rule discovery from the picture and enumerate all possible classifiers for each task, which is trivial given the small tasks we'll consider. This effectively leaves us with something like a tabular Q-learner, where each entry in the table corresponds to a rule in the ternary language (see §6.1.3). In other words, some table entries aggregate elementary states, unlike in a standard tabular Q-learner. This approach simplifies matters since rule discovery is no longer an issue, and the behaviour of all possible rules is considered simultaneously. We'll restrict our considerations to the standard ternary LCS language of §2.2.1 because it is the most commonly used and because we are interested in fitness calculations and the ontology of rules, not in their representation.

At present we are concerned with rule type analysis; we would like to know, for example, the conditions under which it is possible for strong overgeneral rules to be created. We are not, however, interested at present in analysing the dynamic behaviour of rules over time. For example, we will not consider how the strength of a rule changes from a default initial value to a value which reflects its utility (as determined by the credit assignment system). Instead, we'll consider the steady state values of already adapted (i.e., evaluated) rules (that is, until §5.7, where we will see how the fitness of rules changes over time). In particular, we will not consider the effect of the learning rate β on rule updates, by assuming that it is declined appropriately in order that in the limit rule strengths approach the expected values shown.

Similarly, we are not interested in fluctuations in a rule's strength due to stochastic effects, and so we will consider only the expected values of rules in our calculations, and not deviations from expectations. In fact, we will consider deterministic reward functions, although by considering expected values we could compensate for stochasticity in the reward function. For our analysis of rules types, however, deterministic reward functions suffice.

As a final simplification we'll assume that, in all tasks, states and actions occur equiprobably. That is, on any time step the LCS has the same chance of sensing any of the possible environmental states, and it chooses an action at random. This makes the calculation of steady state strengths particularly simple. For example, figure 5.1 defines a simple task with two states and two actions, and a reward associated with each state-action pair. Figure 5.2 lists all possible rules for this task, along with their expected strengths. Since all states and all actions occur with equal probabilities, a rule's expected strength is simply the average of the rewards for the state-actions it matches.

What can this Sort of Analysis Tell us?

These simplifications reduce a complex dynamic system – the interaction of an LCS with a task – to a very simple static model, in which each rule has a single fixed expected strength. Because we have simplified matters so much, there is much that such a model cannot tell us, e.g., about the dynamic behaviour of rules. In fact, removing rule discovery and choosing actions at random does not leave us with much of a classifier system and our simplifications mean that any quantitative results we obtain do not apply to any realistic applications of an LCS. However, because the model is so simple it is amenable to the analysis we will perform. In particular, this approach seems well suited to the qualitative study of rule ontology, and it will give us a qualitative sense of the behaviour of two types of LCS. §5.3 contains examples of this approach.

Later, in §5.12.5, we will return to the Woods2 task of §2.5.2 and search for empirical evidence to support the analysis of this chapter.

Default Hierarchies

Default hierarchies (§3.6.1) have not been included in the analysis presented here because XCS and SB–XCS do not support them. Default hierarchies are potentially significant in that they may allow strength LCS to overcome some of the difficulties with strong overgeneral rules we will show them to have. If so, this would increase both the significance of default hierarchies and the significance of the well-known difficulty of finding and maintaining

them.

Fitness Sharing

Like default hierarchies, fitness sharing is a potential means for strength-based systems to escape problems with strong overgenerals. Its analysis is, unfortunately, beyond the scope of this work, and it is left as an important direction for future work. The incorporation of fitness sharing in SB–XCS would alter many, if not most, of the results in this chapter. It would also, however, greatly complicate the analysis presented here, and it is unlikely that the results obtained here would have been possible had fitness sharing been included in the analysis. This work is appropriately seen as a first step which identifies fundamental rule types; the effect of fitness sharing on them must await future work.

5.3 Analysis of Rule Types

5.3.1 Correct and Incorrect Actions

Since the goal of a reinforcement learning agent is to maximise the rewards it receives, it's useful to have terminology which distinguishes actions which do so from those which do not:

Correct action: In any given state the agent must choose from a set of available actions. A correct action is one which results in the maximum reward possible for the given state and set of available actions.

That is, an action c is correct, $\text{correct}(s, c)$, for a state s with respect to a reward function R iff:

$$\forall a \; R(s, a) \leq R(s, c)$$

Incorrect action: One which does not maximise reward.

Figure 5.1 defines a simple non-sequential task, in which for state 0 the correct action is 0, while in state 1 both actions 0 and 1 are correct. Note that an action is correct or incorrect only in the context of a given state and the rewards available in it.

State	Action	Reward
0	0	1000
1	0	500

State	Action	Reward
0	1	500
1	1	500

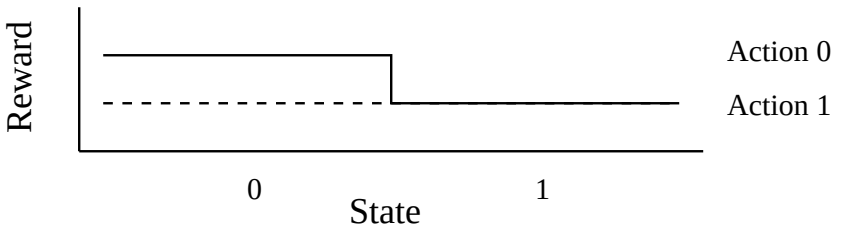


Figure 5.1: Reward function for a simple task.

Rule	Cond.	Action	E[Strength]	Strength Classification	Accuracy Classification
A	0	0	1000	Cons. Correct	Accurate
B	0	1	500	Cons. Incorrect	Accurate
C	1	0	500	Cons. Correct	Accurate
D	1	1	500	Cons. Correct	Accurate
E	#	0	750	Cons. Correct	Overgeneral
F	#	1	500	Overgeneral	Accurate

Figure 5.2: All possible classifiers for the simple task in figure 5.1 and their classifications using strength-based and accuracy-based fitness. ("Cons." stands for "consistently".)

5.3.2 Overgeneral Rules

Figure 5.2 shows all possible rules for the task in figure 5.1 using the standard ternary language of §2.2.1. Each rule's expected strength is also shown, using the simplifying assumption of equiprobable states and actions from §5.2. The classification shown for each rule will eventually be explained on pages 144 and 146.

We're interested in distinguishing overgeneral from non-overgeneral rules. Rules A, B, C and D are clearly not overgeneral, since they each match only one input. What about E and F? So far we haven't explicitly defined overgenerality, so let's make our implicit notion of overgenerality clear:

Overgeneral rule: A rule O from which a superior rule can be derived by reducing the generality of O's condition.

This definition seems clear, but relies on our ability to evaluate the superiority of rules. That is, to know whether a rule X is overgeneral, we need to know whether there is any possible Y, some more specific version of X, which is superior to X. How should we define superiority?

Are Stronger Rules Superior Rules?

Can we simply use fitness itself to determine the superiority of rules? After all, this is the role of fitness in the GA. In other words, let's say X is overgeneral if some more specific version Y is fitter than X.

In SB–XCS, our strength-based system, fitter rules are those which receive higher rewards, and so have higher strength. Let's see if E and F are overgeneral using strength to define the superiority of rules.

Rule E. The condition of E can be specialised to produce A and C. C is inferior to E (it has lower strength) while A is superior (it has greater strength). Because A is superior, E is overgeneral.

This doesn't seem right – intuitively E should *not* be overgeneral, since it is correct in both states it matches. In fact all three rules (A, C and E) advocate only correct actions, and yet A is supposedly superior to the other two. This seems wrong since E subsumes A and C, which suggests that, if any of the three is more valuable, it should be E.

Rule F. The condition of F can be specialised to produce B and D. Using strength as our value metric all three rules are equally valuable, since they have the same expected strength, so F is not overgeneral.

This doesn't seem right either – surely F *is* overgeneral since it is incorrect in state 0. Surely D should be superior to F since it is always correct.

Clearly using strength as our value metric doesn't capture our intuitions about what the system should be doing.

To define the value of rules let's return to the goal of the LCS, which, as mentioned earlier, is to maximise the reward it receives. Maximising reward means taking the correct action in each state. It is the correctness of its actions which determines a rule's value, rather than how much reward it receives (its strength).

Recall from sections 1.5.1 and 2.3.5 that strength is derived from reward. Strength is a measure of how good – *on average* – a rule is at obtaining reward. Using strength as fitness in the GA, we will evolve rules which are – *on average* – good at obtaining reward. However, many of these rules will

actually perform poorly in some states, and only achieve good average performance by doing particularly well in other states. Such rules are overgeneral; superior rules can be obtained by restricting their conditions to match only the states in which they do well.

To maximise rewards, we do not want to evolve rules which obtain the highest rewards possible *in any state*, but to evolve rules which obtain the highest rewards possible *in the states in which they act*. That is, rather than rules which are *globally* good at obtaining reward, we want rules which are *locally* good at obtaining reward. In other words, we want rules whose actions are correct in all states they match. What's more, each state must be covered by a correct rule because an LCS must know how to act in each state; that is, it must have a *policy* (§C.5.2).

To encourage the evolution of consistently correct rules, rather than rules which are good on average, we can use techniques like fitness sharing. But, while such techniques may help, there remains a fundamental mismatch between using strength as fitness and the goal of evolving rules with consistently correct actions. The effect of fitness sharing, and in particular its ability to combat overgeneral rules, deserves further study.

Strength and Best Action Maps

To maximise rewards, a strength-based LCS needs a population of rules which advocates the correct action in each state. If, in each state, only the best action is advocated, the population constitutes a best action map (§3.6.2). While a best action map is an ideal representation (in that it is minimal), it is still possible to maximise rewards when incorrect actions are also advocated, as long as they are not selected. This is what we hope for in practice.

Now let's return to the question of how to define overgenerality in a strength-based system. Instead of saying X is overgeneral if some Y is fitter (stronger), let's say it is overgeneral if some Y is more consistent with the goal of forming a best action map; that is, if Y is correct in more cases than X.[1] Notice that we're now speaking of the correctness of rules (not just the correctness of actions), and of their relative correctness at that. Let's emphasise these ideas:

Consistently Correct Rule: One which advocates a correct action in every state it matches. More formally, a classifier c is consistently correct w.r.t. a function f iff:[2]

[1] In RL terms, we could say X is overgeneral if some Y is more consistent with the optimal policy.

[2] The reader is referred to page 28 for notation.

$$\forall s \in \mathcal{S}(c)\ f(s) = c(s)$$

Consistently Incorrect Rule: One which advocates an incorrect action in every state it matches. That is, a classifier c is consistently incorrect w.r.t. a function f iff:

$$\forall s \in \mathcal{S}(c)\ f(s) \neq c(s)$$

Correctness of a Rule: The correctness of a rule is the proportion of states in which it advocates the correct action.[3]

The degree of correctness of a classifier c, correctness(c), w.r.t. a function f is the ratio between the states which it classifies correctly and the total number of states:

$$|\mathcal{C}| \ / \ |\mathcal{S}(c)|$$

where

$$\mathcal{C} = \{s \in \mathcal{S}(c) \mid c(s) = f(s)\}$$

Overgeneral Rule: One which advocates a correct action in some states and an incorrect action in others (i.e., a rule which is neither consistently correct nor consistently incorrect).

A classifier c is inconsistent w.r.t. a function f iff:

$$0 < \text{correctness}(c) < 1$$

That is, a classifier is inconsistent if it is neither consistently correct nor consistently incorrect.

The notion of the relative correctness of a rule allows us to say a rule Y is *more* correct (and hence *less* overgeneral) than a rule X, even if neither is consistently correct.

Now let's reevaluate E and F from figure 5.2 to see how consistent they are with the goal of forming a best action map. Rule E matches both states and advocates a correct action in both. This is compatible with forming a best action map, so E is not overgeneral. Rule F also matches both states, but advocates an incorrect action in state 0, making F incompatible with the goal of forming a best action map. Because a superior rule (D) can be obtained by specialising F, F is overgeneral.

[3]The correctness of a rule corresponds to classification accuracy in pattern classification.

Notice that we've now defined overgeneral rules twice: once on p. 142 and again above. For the tasks we're considering here the two definitions coincide, although they do not always. For example, in the presence of perceptual aliasing (where an input to the LCS does not always describe a unique task state) a rule may be overgeneral by one definition but not by the other. That is, it may be neither consistently correct nor consistently incorrect, and yet it may be impossible to generate a more correct rule because a finer distinction of states cannot be expressed.

The above assumes the states referred to in the definition of overgenerality are task states. If we consider perceptual states rather than task states the rule is sometimes correct and sometimes incorrect *in the same state* (which is not possible in the basic tasks studied here). We could take this to mean the rule is not consistently correct, and thus overgeneral, or we might choose to do otherwise.

Accuracy and Complete Maps

While all reinforcement learners seek to maximise rewards, the approach of XCS differs from that of strength-based LCS. Where strength LCS seek to form best action maps, XCS seeks to form a *complete map*: a set of rules such that each action in each state is advocated by at least one rule [298, 148]. This set of rules allows XCS to approximate the entire reward function and (hopefully) accurately predict the reward for any action in any state. XCS's fitness metric *is* consistent with this goal, and we'll use it to define the superiority of rules for XCS.

The different approaches to fitness mean that in strength-based systems we contrast consistently correct, consistently incorrect and overgeneral rules, but with accuracy-based fitness we contrast accurate and inaccurate rules.

In XCS, fitter rules are those with lower prediction errors – at least up to a point: small errors in prediction are ignored, and rules with small enough errors are considered fully accurate (see the accuracy update in §2.3.5). In other words, XCS has some tolerance for prediction error, or, put another way, some tolerance for changes in a rule's strength, since changes in strength are what produce prediction error. This tolerance for prediction error is used to define overgenerality in XCS; we say that a rule is overgeneral if its prediction error exceeds the tolerance threshold, i.e., if $\varepsilon_j \geq \varepsilon_o$. In XCS 'overgeneral' is synonymous with 'not-fully-accurate'.

Although this work uses XCS as a model, we hope it will apply to other future accuracy-based LCS. To keep the discussion more general, instead of focusing on XCS and its error threshold, we'll refer to a somewhat abstract notion of tolerance called τ. Let $\tau \geq 0$ be an accuracy-based LCS's tolerance

for oscillations in strength, above which a rule is judged overgeneral.

Like XCS's error threshold, τ is an adjustable parameter of the system. This means that in an accuracy-based system, whether a rule is overgeneral or not depends on how we set τ. If τ is set very high, then both E and F from figure 5.2 will fall within the tolerance for error and neither will be overgeneral. If we gradually decrease τ, however, we will reach a point where E is overgeneral while F is not. Notice that this last case is the reverse of the situation we had on p. 144 when using strength-based fitness. So which rule is overgeneral depends on our fitness metric.

Defining Overgenerality

To match the different goals of the two systems we need different definitions of overgenerality:

Strength-based overgeneral: For strength-based fitness, an overgeneral rule is one which matches multiple states and acts incorrectly in some, but not all.[4] That is, a rule c is a strength-based overgeneral w.r.t. a function f iff:

$$|\mathcal{S}(c)| > 1$$

$$\text{and } 0 < \text{correctness}(c) < 1$$

Accuracy-based overgeneral: For accuracy-based fitness, an overgeneral rule is one which matches multiple states, some of which return (sufficiently) different rewards, and hence has (sufficiently) oscillating strength. Here a rule is overgeneral if its oscillations exceed τ. That is, a rule c is an accuracy-based overgeneral w.r.t. a function f iff:

$$|\mathcal{S}(c)| > 1$$

$$\text{and } \varepsilon_c \geq \varepsilon_o$$

Note that the strength definition requires action on the part of the classifiers while the accuracy definition does not. Thus we can have overgenerals in a task which allows 0 actions (or, equivalently, 1 action) using accuracy (see, e.g., figure 5.5), but not using strength.

[4] This restatement of strength-based overgenerality is consistent with the two earlier definitions given on pages 142 and 145.

5.3.3 Strong Overgeneral Rules

Now that we've finally defined overgenerality satisfactorily let's turn to the subject of strong overgenerality. Strength is used to determine a rule's influence in action selection, and action selection is a competition between alternatives. Consequently it makes no sense to speak of the strength of a rule in isolation. Put another way, strength is a way of ordering rules. With a single rule there are no alternative orderings, and hence no need for strength.

In other words, strength is a relation between rules; a rule can only be stronger or weaker than other rules – there is no such thing as a rule which is strong in isolation. Therefore, for a rule to be a strong overgeneral, it must be stronger than another rule. In particular, a rule's strength is relevant when compared to another rule with which it competes for action selection.

Now we can define strong overgeneral rules, although to do so we need two definitions to match our two definitions of overgenerality:

Strength-based strong overgeneral: A rule which sometimes advocates an incorrect action, and yet whose *expected strength* is greater than that of some correct (i.e., not-overgeneral) competitor *for action selection.* That is, a rule c is a strength-based strong overgeneral w.r.t. a function f iff:

$$|\mathcal{S}(c)| > 1$$

$$\text{and } 0 < \text{correctness}(c) < 1$$

and there is a consistently correct rule r with strength(r) < strength(c)

with which c competes for action selection.

Accuracy-based strong overgeneral: A rule whose strength oscillates unacceptably, and yet whose *expected strength* is greater than that of some accurate (i.e., not-overgeneral) competitor *for action selection.* That is, a rule c is an accuracy-based strong overgeneral w.r.t. a function f iff:

$$|\mathcal{S}(c)| > 1$$

$$\text{and } \varepsilon_c \geq \varepsilon_o$$

and there is a consistently correct rule r with strength(r) < strength(c)

with which c competes for action selection.

The intention is that competitors be possible, not that they need actually exist in a given population.

The strength-based definition refers to competition with *correct* rules because strength-based systems are not interested in maintaining incorrect rules (see §5.3.2). This definition suits the analysis in this work. However, situations in which more overgeneral rules have higher fitness than less overgeneral – but still overgeneral – competitors are also pathological. Parallel scenarios exist for accuracy-based fitness. Such cases resemble the well-known idea of *deception* in genetic algorithms, in which search is led away from desired solutions (see, e.g., [94]).

5.3.4 Fit Overgeneral Rules

In our definitions of strong overgenerals we refer to competition for action selection, but rules also compete for reproduction. To deal with the latter case we introduce the concept of *fit overgenerals* as a parallel to that of strong overgenerals. A rule can be both, or either. The definitions for strength and accuracy-based fit overgenerals are identical to those for strong overgenerals, except that we refer to fitness (not expected strength) and competition for reproduction (not action selection):

Strength-based fit overgeneral: A rule which sometimes advocates an incorrect action, and yet whose *expected fitness* is greater than that of some correct (i.e., not-overgeneral) competitor *for reproduction.*[5]

Accuracy-based fit overgeneral: A rule whose strength oscillates unacceptably, and yet whose *expected fitness* is greater than that of some accurate (i.e., not-overgeneral) competitor *for reproduction.*

We won't consider fit overgenerals as a separate case in our initial analysis since in SB–XCS fitness and strength are the same, and so strong and fit overgenerals are similar.[6] Later, in §5.7, we'll see how XCS handles both fit and strong overgenerals.

5.3.5 Parallel Definitions of Strength and Fitness

At this point we must note two terminological issues. In strength-based systems, a rule's strength is used in action selection and reproduction, and so

[5]The more formal definitions of fit overgenerals are omitted as they differ from those for strong overgenerals only in that they refer to fitness and reproduction instead of strength and action selection.

[6]Nonetheless, there is still a difference between strong and fit overgenerals in strength-based systems, since the two forms of competition may take place between different sets of rules. See §5.4.

it is with SB–XCS.[7] Although XCS uses the same update as SB–XCS ((2.4)) (p. 42), Wilson refers to this value as *prediction*, rather than strength. For simplicity this value has generally been referred to simply as strength in this work.

The term "strength", however, really has two interpretations. One is the value updated by (2.4), and the other is the weight a rule has in action selection. When aggregated over the relevant rules, the latter value is referred to as system strength (p. 54) in SB–XCS and system prediction (p. 36) in XCS.

In SB–XCS, the two notions of strength coincide; in SB–XCS a rule's contribution to the system strength for an action is just its strength (weighted by numerosity – equation (2.11), p. 54). In XCS, however, a rule's contribution to the system prediction for an action is a function of both prediction and its fitness (equation (2.1), p. 36).

Consequently, in XCS we have two notions of strength; prediction, and contribution to system prediction. In discussing strong overgenerals, it should be understood that references to strength are to the latter value.

Fitness

A similar problem occurs in referring to the fitness of a rule. In XCS, fitness is a value updated by (2.9) (p. 45). Fitness, however, can also be interpreted (in fact, is normally interpreted in evolutionary computation) as the weight of a rule in reproduction. If we consider a single invocation of the niche GA, a rule's fitness parameter coincides with its weight in reproduction. However, if we consider all invocations of the GA, a rule's weight in reproduction is partly determined by its generality, thanks to the niche GA (in both XCS and SB–XCS).

In discussing fit overgenerals, it should be understood that references to fitness are to the weight of a rule in reproduction, not to the fitness parameter of a rule.

5.4 When are Strong and Fit Overgenerals Possible?

We've seen definitions for strong and fit overgeneral rules, but what are the exact conditions under which a task can be expected to produce them? If such rules are a serious problem for classifier systems, knowing when to expect

[7] Strength-based systems may, however, distinguish between shared and unshared strength and use them differently. See p. 199.

them should be a major concern: if we know what kinds of tasks are likely to produce them (and how many) we'll know something about what kinds of tasks should be difficult for classifier systems (and how difficult).

Not surprisingly, the requirements for the production of strong and fit overgenerals depend on which definition we adopt. Looking at the accuracy-based definition of strong overgenerality we can see that we need two rules (a strong overgeneral and a not-overgeneral rule), that the two rules must compete for action selection, and that the overgeneral rule must be stronger than the not-overgeneral rule. The task conditions which make this situation possible are as follows:

1. The task must contain at least two states, in order that we can have a rule which generalises (incorrectly).[8]

2. The task may allow any number of actions in the two states, including 0 actions, or, equivalently, 1 action. (We'll see later that strength-based systems differ in this respect.)

3. In order to be a strong overgeneral, the overgeneral must have higher expected strength than the not-overgeneral rule. For this to be the case the reward function must return different values for the two rules. More specifically, it must return more reward to the overgeneral rule.

4. The overgeneral and not-overgeneral rules must compete for action selection. This constrains which tasks will support strong overgenerals.

The conditions which will support fit overgenerals are clearly very similar: 1) and 2) are the same, while for 3) the overgeneral must have greater fitness (rather than strength) than the not-overgeneral, and for 4) they must compete for reproduction rather than action selection.

5.4.1 The Reward Function is Relevant

Let's look at the last two requirements for strong overgenerals in more detail. First, in order to have differences in the expectations of the strengths of rules there must be differences in the rewards returned from the task. So the values in the reward function are relevant to the formation of strong overgenerals. More specifically, it must be the rewards returned to competing classifiers

[8] We assume the use of the standard LCS language in which generalisation over actions does not occur. Otherwise, it would be possible to produce an overgeneral in a task with only a single state (and multiple actions) by generalising over actions instead of states.

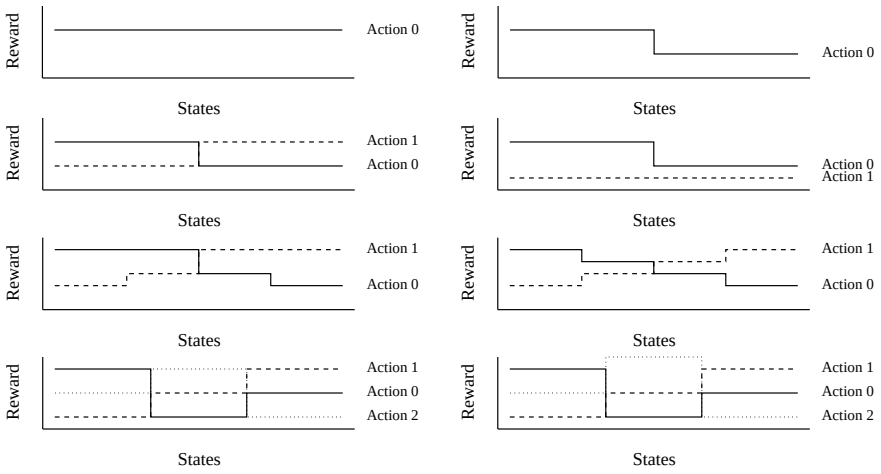


Figure 5.3: Reward functions with 1, 2 or 3 actions. Those on the left are unbiased while those on the right are biased.

which differ. So subsets of the reward function are relevant to the formation of individual strong or fit overgenerals.

Let us refer to the situation where different correct actions receive different rewards as a *bias* in the reward function. Reward functions which have no such biases are *unbiased*. More formally:

Unbiased reward function: A reward function R is unbiased iff there exists a constant r such that:

$$\forall s, a \text{ if correct}(s, a) \text{ then } R(s, a) = r$$

Figure 5.3 shows examples of unbiased reward functions on the left, and biased reward functions on the right. Note that the defining feature of the unbiased functions is that the highest reward in all states is the same constant value. If we plot the maximum reward for each state for any unbiased reward function we obtain a flat line.

For strong or fit overgenerals to occur, there must be a bias in the reward function at state-action pairs which map to competing classifiers. In the following section, we look at how classifiers can compete.

State \ Action			
	$R_{0,0}$	$R_{0,1}$	$R_{0,2}$
	$R_{1,0}$	$R_{1,1}$	$R_{1,2}$
	$R_{2,0}$	$R_{2,1}$	$R_{2,2}$

Figure 5.4: A reward function as a matrix indexed by state and action.

5.4.2 Competition for Action Selection

In XCS and SB–XCS, two classifiers c_1 and c_2 compete for action selection iff:

$$\mathcal{S}(c_1) \cap \mathcal{S}(c_2) \neq \emptyset \text{ and } \mathcal{A}(c_1) \neq \mathcal{A}(c_2)$$

Note that this relies on the property of the ternary language that classifiers are constant partial functions, i.e., that they advocate the same action in all states they match.

Figure 5.4 shows a reward function as a matrix with task state indexing the rows and LCS action indexing the columns. Using the standard ternary language rules can generalise over states but not actions, so a given rule will match some subset of a single column.

Rules compete for action selection when they occur in the same match set [M]. In XCS (and some other LCS) rules advocating the same action cooperate to have their action chosen, while rules advocating different actions compete. (We could say competition is between action sets [A]s, rather than individual rules.) In other LCS, rules which advocate the same action also compete against each other, and only the winner receives reward.

In either case, two rules which occur in different action sets within a match set will compete. The rules in each action set are updated towards the rewards returned for different actions in a state (rows in the matrix). Hence differences within a row of the matrix influence the strengths of competing classifiers and may result in strong overgenerals.

The rewards returned for taking the same action in different states (columns in the matrix) also affect the strengths of competing rules simply because rules can generalise over multiple states, and their strengths depend on the values in all of them. Hence differences within columns can result in strong overgenerals.

The effect, of course, of strong overgenerals on competition for action selection is that the system will tend to select an incorrect action.

5.4.3 Competition for Reproduction

The locus of competition for reproduction depends on the GA scheme used. With a panmictic GA[9] all rules in the population compete for reproduction, whereas with a niche GA only a subset of the population is eligible for reproduction. One way of running the niche GA is to restrict selection of rules to those in [M], in which case competition is between members of [M]. Similarly, with a niche GA in [A], rules in [A] compete.

More formally, using a niche GA in [M] two classifiers c_1 and c_2 compete for reproduction iff:

$$\mathcal{S}(c_1) \cap \mathcal{S}(c_2) \neq \emptyset$$

while using a niche GA in [A] c_1 and c_2 compete for reproduction iff:

$$\mathcal{S}(c_1) \cap \mathcal{S}(c_2) \neq \emptyset \text{ and } \mathcal{A}(c_1) = \mathcal{A}(c_2)$$

A niche GA limits the areas of competition within a population, and so limits how the rewards defined in the reward function interact. For example, with a GA in [A], rules belonging to different [A]s within an [M] do not compete for reproduction, so differences in the reward function between one [A] and another (i.e., differences within a row in the matrix) will not affect the reproduction of fit overgenerals in either.

Note, however, that different [A]s will overlap, as will different [M]s, and that a fully generalised rule is a member of all [M]s and all [A]s for its action. Even with a niche GA in [A] a fully general rule competes with all rules in the population which advocate the action it does. So even though a niche GA limits competition to subsets of the population, differences anywhere in a column can contribute to the fitness of two competing rules.

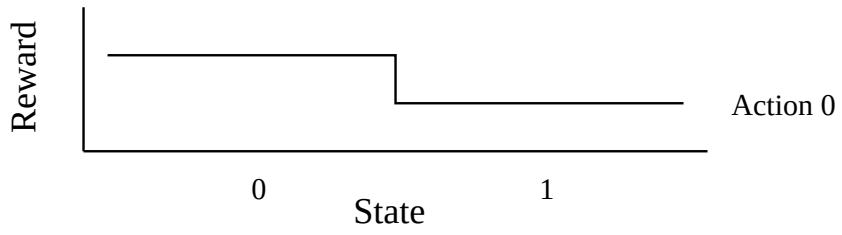
This means that with a GA in [A] differences in the rewards given for the same action may contribute to the fitness of rules competing for reproduction, but differences in rewards for different actions do not.

With a panmictic GA all rules compete, in which case a difference between any two parts of the reward function can contribute to the fitness of rules competing for reproduction.

The effect, of course, of fit overgenerals on competition for reproduction is that the fit overgenerals will tend to propagate in the population.

[9] That is, when all rules are eligible for selection (§2.3.6).

State	Action	Reward
0	0	$a = 1000$
1	0	$c = 0$



Rule	Condition	Action	E[Strength]
A	0	0	$a = 1000$
C	1	0	$c = 0$
E	#	0	$(a + c)/2 = 500$

Figure 5.5: A minimal (2x1) strong overgeneral task for XCS and all its classifiers.

5.5 Strong Overgenerals in XCS

In §5.4 we saw that, using the accuracy definition, strong overgenerals require a task with at least two states, and that each state can have any number of actions. We also saw that the reward function was relevant but did not see exactly how. Now let's look at a minimal strong overgeneral supporting task for accuracy and see exactly what is required of the reward function to produce strong overgenerals. Figure 5.5 shows a reward function for a task with two states and one action and all possible classifiers for it. As always, the expected strengths shown are due to the simplifying assumption that states and actions occur equiprobably (§5.2).

In §5.2 we made a number of simplifying assumptions, and for now let's make a further one: that there is no tolerance for oscillating strengths ($\tau = 0$), so that any rule whose strength oscillates at all is overgeneral. This means rule E in figure 5.5 is an overgeneral because it is updated towards different rewards. It is also a strong overgeneral because it is stronger than some not-overgeneral rule with which it competes, namely rule C. In §5.4 we saw that strong overgenerals depend on the reward function returning more strength to the strong overgeneral than its not-overgeneral competitor.

State	Action	Reward
0	0	w
1	0	x

State	Action	Reward
0	1	y
1	1	z

Rule	Condition	Action	E[Strength]	Overgeneral unless
A	0	0	w	never
B	0	1	y	never
C	1	0	x	never
D	1	1	z	never
E	#	0	$(w+x)/2$	$\lvert w-x\rvert \leq \tau$
F	#	1	$(y+z)/2$	$\lvert y-z\rvert \leq \tau$

Figure 5.6: A task which demonstrates that biases between actions do not result in strong overgenerals.

Are there reward functions under which E will not be a strong overgeneral? Since the strength of E is an average of the two rewards returned (labelled a and c in figure 5.5 to correspond to the names of the fully specific rules which obtain them), and the strength of C is c, then as long as $a > c$, rule E will be a strong overgeneral. Symmetrically, if $c > a$ then E will still be a strong overgeneral, in this case stronger than not-overgeneral rule A. The only reward functions which do not cause strong overgenerals are those in which $a = c$. So in this case *any* bias in the reward function makes the formation of a strong overgeneral possible.

If we allow some tolerance for oscillations in strength without judging a rule overgeneral, then rule E is not overgeneral only if $a - c \leq \tau$ where τ is the tolerance for oscillations. In this case only reward functions in which $a - c > \tau$ will produce strong overgeneral rules.

5.5.1 Biases between Actions do not Produce Strong Overgenerals

We've just seen an example where any bias in the reward function will produce strong overgenerals. However, this is not the case when we have more than one action available, as in figure 5.6. The strengths of the two fully generalised rules, E & F, are dependent only on the values associated with the actions they advocate. Differences in the rewards returned for different actions do not result in strong overgenerals – as long as we don't generalise over actions, which was one of our assumptions from §5.2.

5.5.2 Some Properties of Accuracy-Based Fitness

At this point it seems natural to ask how common strong overgenerals are and, given that the structure of the reward function is related to their occurrence, to ask what reward functions make them impossible. In this section we'll prove some simple, but perhaps surprising, theorems concerning overgeneral rules and accuracy-based fitness. However, we won't go too deeply into the subject for reasons which will become clear later in §5.7.

Let's begin by looking at a special kind of reward function in which strong overgenerals are impossible, and proving that this is so. Immediately after the proof we'll see the general conditions which make strong overgenerals impossible, but, first, the special case:

Theorem 1 *In XCS, overgeneral rules are impossible when the reward function is constant over each action.*

Proof. The strength of a rule is a function of the values towards which it is updated, and these values are a subset of the rewards for the action it advocates. If all such rewards are equivalent there can be no oscillations in a rule's strength, so it cannot be overgeneral. □

We can also prove this by reference to the XCS updates, as follows.

Proof. By (2.4), the strength of a rule j is:

$$p_j \leftarrow p_j + \beta(P - p_j)$$

By (2.5) the prediction error of j is:

$$\varepsilon_j \leftarrow \varepsilon_j + \beta\Big(|P - p_j| - \varepsilon_j\Big)$$

j is overgeneral iff $\varepsilon_j \geq \varepsilon_o$ where ε_o is a constant ≥ 0. If the reward function is constant over each action, then by the updates in chapter 2 P is constant, in which case for any rule j, $p_j = P$ and, assuming an initial ε_j of 0, $\varepsilon_j = 0$. (Recall that using MAM updates (§2.3.5), $\beta = 1$ on the first update, so that if we update prediction before prediction error, then $\varepsilon_j = 0$.) Since $\varepsilon_o > 0$ and $\varepsilon_j = 0$, j is accurate. □

More generally, strong overgenerals are impossible when the reward function is sufficiently close to constancy over each action that oscillations in any rule's strength are less than τ. Now we can see when strong overgenerals are possible:

Theorem 2 *In XCS, if the task structure meets requirements 1 and 4 of §5.4 at least one overgeneral rule will be possible for each action for which the reward function is not within τ of being constant.*

Proof. A fully generalised rule matches all inputs and its strength is updated towards all possible rewards for the action it advocates. Unless all such rewards are within τ of equivalence it will be overgeneral. □

In other words, if the rewards for the same action differ by more than τ the fully generalised rule for that action will be overgeneral. To avoid overgeneral rules completely, we'd have to constrain the reward function to be within τ of constancy for each action. That overgeneral rules are widely possible should not be surprising. But it turns out that with accuracy-based fitness there is no distinction between overgeneral and strong overgeneral rules:

Theorem 3 *In XCS, all overgeneral rules are strong overgenerals.*

Proof. Let's consider the reward function as a vector $R = [r_1\ r_2\ r_3\ \ldots\ r_n]$, and, for simplicity, assume $\tau = 0$. An overgeneral matches at least two states, and so is updated towards two or more distinct values from the vector, whereas accurate rules are updated towards only one value (by definition, since $\tau = 0$) no matter how many states they match. For each r_i in the vector there is some fully specific (and so not overgeneral) rule which is only updated towards it. Consequently, any overgeneral rule (which must match at least two states) competes with at least two accurate rules.

Now consider the vector $X = [x_1\ x_2\ x_3\ \ldots\ x_y]$ which is composed of the subset of vector R towards which the overgeneral in question is updated. Because we've assumed states and actions occur equiprobably, the strength of a rule is just the mean of the values it is updated towards. So the strength of the overgeneral is $\bar{X}$, the mean of X.

The overgeneral will be a strong overgeneral if it is stronger than some accurate rule with which it competes. The weakest such rule's strength is $\min x_i$. The inequality $\min x_i < \bar{X}$ is true for all reward vectors except those which are constant functions, so all overgenerals are strong overgenerals. □

Taking theorems 2 and 3 together yields:

Theorem 4 *In XCS, if the task structure meets requirements 1 and 4 of §5.4 at least one* **strong** *overgeneral rule will be possible for each action for which the reward function is not within τ of being constant.*

In short, using accuracy-based fitness and reasonably small τ only a highly restricted class of reward functions and tasks do not support strong overgeneral rules.

These 4 theorems are independent of the number of actions available in a task. Note that the 'for each action' part of the theorems depends

on the inability of rules to generalise over actions, a syntactic limitation of the standard LCS language. If we remove this arbitrary limitation then we further restrict the class of reward functions which will not support strong overgenerals.

5.6 Strong Overgenerals in SB–XCS

We've seen how the reward function determines when strong overgeneral classifiers are possible in accuracy-based systems. Now let's look at the effect of the reward function using SB–XCS, our strength-based system. Recall from the strength-based definition of strong overgenerals that we need two rules (a strong overgeneral and a not-overgeneral correct rule), that the two rules must compete for action selection, and that the overgeneral rule must be stronger than the correct rule. The conditions which make this situation possible are the same as those for accuracy-based systems, except for a change to condition 2: there needs to be at least one state in which at least two actions are possible, so that the overgeneral rule can act incorrectly. (It doesn't make sense to speak of overgeneral rules in a strength-based system unless there is more than one action available.)

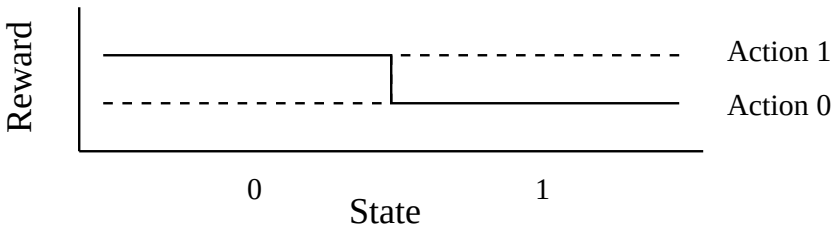
A second difference is that in strength-based systems there is no tolerance for oscillations in a rule's strength built into the update rules. This tolerance is simply not needed in SB–XCS where all that matters is that a rule advocate the correct action, not that its strength be consistent.

A complication to the analysis done earlier for accuracy-based systems is that strength-based systems tend towards best action maps (§3.6.2). Simply put, SB–XCS is not interested in maintaining incorrect rules, so we are interested in overgenerals only when they are stronger than some correct rule. For example, consider the binary state binary action task of figure 5.7. Using this unbiased reward function, rules E & F are overgenerals (since they are sometimes incorrect), but not strong overgenerals because the rules they are stronger than (B & C) are incorrect. (Recall from the definition of a strong overgeneral in a strength LCS in §5.3.3 that the strong overgeneral must be stronger than a *correct* rule.) This demonstrates that in strength-based systems (unlike accuracy-based systems) not all overgeneral rules are strong overgenerals.

What consequence does this disinterest in incorrect rules have on the dependence of strong overgenerals on the reward function? The reward function in this example is not constant over either action, and the accuracy-based concept of tolerance does not apply. In an accuracy-based system there must be strong overgenerals under such conditions, and yet there are none here.

State	Action	Reward
0	0	1000
1	0	0

State	Action	Reward
0	1	0
1	1	1000



Rule	Condition	Action	E[Strength]	Status
A	0	0	1000	Correct
B	0	1	0	Incorrect
C	1	0	0	Incorrect
D	1	1	1000	Correct
E	#	0	500	Overgeneral
F	#	1	500	Overgeneral

Figure 5.7: An unbiased reward function and all its classifiers. Unbiased functions will not cause strong overgenerals.

5.6.1 When are Strong Overgenerals Impossible in SB–XCS?

Let's begin with a first approximation to when strong overgenerals are impossible. Later, in §5.6.2, we'll ask when strong overgenerals *are* possible, and we'll get a more precise answer to the question of when they are impossible.

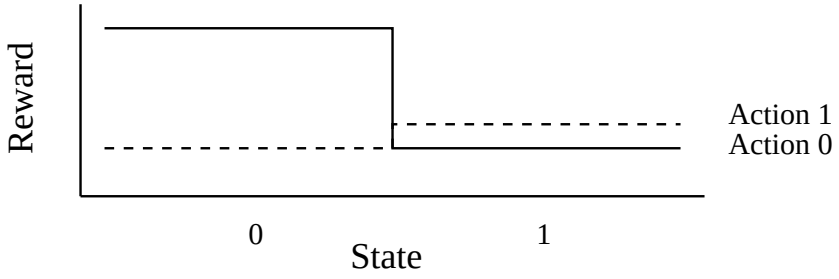
Theorem 5 *In SB–XCS, strong overgenerals are impossible when the reward function is unbiased (i.e., constant over correct actions).*

Proof. A correct action is one which receives the highest reward possible in its state. If all correct actions receive the same reward, this reward is higher than that for acting incorrectly in *any* state. Consequently no overgeneral rule can have higher strength than a correct rule, so no overgeneral can be a strong overgeneral. □

To make theorem 5 more concrete, reconsider the reward values in figure 5.6. By definition, a correct action in a state is one which returns the highest reward for that state, so if we want the actions associated with w and z to

State	Action	Reward
0	0	$w = 1000$
1	0	$x = 0$

State	Action	Reward
0	1	$y = 0$
1	1	$z = 200$



Rule	Condition	Action	E[Strength]	Strong overgeneral if
A	0	0	$w = 1000$	never
B	0	1	$y = 0$	never
C	1	0	$x = 0$	never
D	1	1	$z = 200$	never
E	#	0	$(w + x)/2 = 500$	$(w + x)/2 > z$
F	#	1	$(y + z)/2 = 100$	$(y + z)/2 > z$

Figure 5.8: A 2x2 biased reward function which is a minimal strong overgeneral task for strength-based LCS, and all its classifiers.

be the only correct actions then $w > y, z > x$. If the reward function returns the same value for all correct actions then $w = z$. Then the strengths of the overgeneral rules are less than those of the correct accurate rules: E's expected strength is $(w + x)/2$ which is less than A's expected strength of w and F's expected strength is $(y + z)/2$ which is less than D's z, so the overgenerals cannot be strong overgenerals. (If $w < y$ and $z < x$ then we have a symmetrical situation in which the correct action is different, but strong overgenerals are still impossible.)

5.6.2 What Makes Strong Overgenerals Possible in SB–XCS?

It is possible to obtain strong overgenerals in SB–XCS by defining a reward function which returns different values for correct actions. An example of

a minimal strong overgeneral supporting task for SB–XCS is given in figure 5.8. Using this reward function, E *is* a strong overgeneral, as it is stronger than the correct rule D with which it competes for action selection (and for reproduction if the GA runs in the match set or panmictically – see p. 47).

However, not all differences in rewards are sufficient to produce strong overgenerals. How much tolerance does SB–XCS have before biases in the reward function produce strong overgenerals? Suppose the rewards are such that the actions associated with w and z are correct (i.e., $w > y, z > x$) and the reward function is biased such that $w > z$. How much of a bias is needed to produce a strong overgeneral? I.e., how much greater than z must w be? Rule E competes with D for action selection, and will be a strong overgeneral if its expected strength exceeds D's, i.e., if $(w+x)/2 > z$, which is equivalent to $w > 2z - x$. So a bias of $w > 2z - x$ means E will be a strong overgeneral with respect to D, while a lesser bias means it will not.

E also competes with A for reproduction, and will be fitter than A if $(w + x)/2 > w$, which is equivalent to $x > w$. So a bias of $x > w$ means E will be a fit overgeneral with respect to A, while a lesser bias means it will not. (Symmetrical competitions occur between F & A (for action selection) and F & D (for reproduction).)

We'll take the last two examples as proof of the following theorem:

Theorem 6 *In SB–XCS, if the task structure meets requirements 1 and 4 of §5.4 and the modified requirement 2 from §5.6, a strong overgeneral is possible whenever the reward function is biased such that* $(w + x)/2 > z$ *for some* w, x *&* z.

SB–XCS's Tolerance for Reward Biases

The examples in this section show there is a certain tolerance for biases (differences) in rewards within which overgenerals are not strong enough to outcompete correct rules. Knowing what tolerance there is is important as it allows us to design reward functions which will not produce strong overgenerals. Unfortunately, because of the simplifying assumptions we've made (see §5.2) these results do not apply to more realistic tasks. However, they do tell us how biases in the reward function affect the formation of strong overgenerals, and give us a sense of the magnitudes involved. An extension of this work would be to find limits to tolerable reward function bias empirically. Two results which do transfer to more realistic cases are theorems 1 and 5, which tell us under what conditions strong overgenerals are impossible for the two types of LCS. These results hold even when our simplifying assumptions do not.

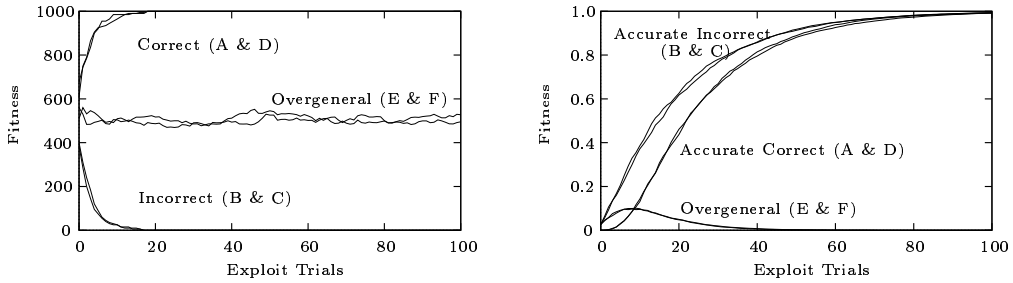


Figure 5.9: Rule fitness using strength-based SB–XCS (left) and accuracy-based XCS (right) on the unbiased function from figure 5.7.

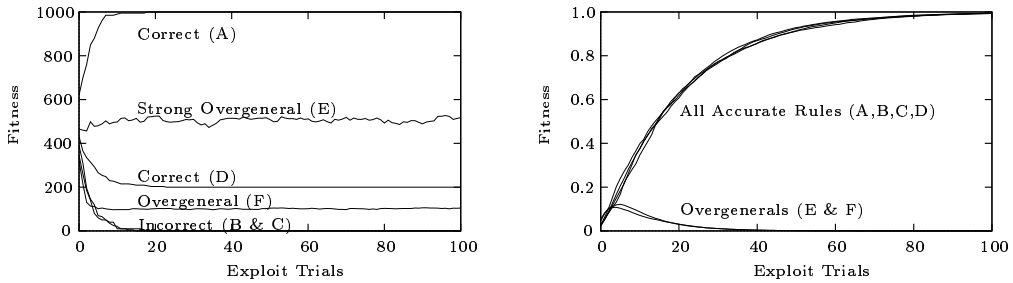


Figure 5.10: SB–XCS (left) and XCS (right) on the biased function from figure 5.8.

5.7 Fit Overgenerals and the Survival of Rules under the GA

We've examined the conditions under which strong overgenerals are possible under both types of fitness. The whole notion of a strong overgeneral is that of an overgeneral rule which can outcompete other, preferable, rules. But, as noted earlier, there are two forms of competition between rules: action selection and reproduction. Our two systems handle the first in the same way, but handle reproduction differently. In this section we examine the effect of the fitness metric on the survival of strong overgenerals.

XCS and SB–XCS were compared empirically on the tasks in figures 5.7 and 5.8. For these tests the GA was disabled and all possible rules inserted in the LCS at the outset. The following settings were used: $\beta = 0.2$, $\varepsilon_o = 0.01$ (see §2.3.5). Wilson's pure explore/exploit scheme (§2.3.3) was used.

5.7.1 Comparison on an Unbiased Reward Function

First we compared XCS and SB–XCS on the reward function from figure 5.7. Figure 5.9 shows the fitness of each rule using strength (left) and accuracy (right), with results averaged over 100 runs. The first thing to note is that we are now considering the development of a rule's strength and fitness over time (admittedly with the GA turned off), whereas until this section we had only considered steady state strengths (as pointed out in §5.2). We can see that the actual strengths indeed converge towards the expected strengths shown in figure 5.7. We can also see that the strengths of the overgeneral rules (E & F) oscillate as they are updated towards different values.

Using strength (figure 5.9, left), the correct rules A & D have highest fitness, so if the GA was operating we'd expect SB–XCS to reproduce them preferentially and learn to act correctly in this task.

Using accuracy (figure 5.9, right), all accurate rules (A, B, C & D) have high fitness, while the overgenerals (E & F) have low fitness. Note that even though the incorrect rules (B & C) have high fitness and will survive with the GA operational, they have low strength, so they will not have much influence in action selection. Consequently we can expect XCS to learn to act correctly in this task.

5.7.2 Comparison on a Biased Reward Function

While both systems seem to be able to handle the unbiased reward function, compare them on the same task when the reward function is biased as in figure 5.8. Consider the results shown in figure 5.10 (again, averaged over 100 runs). Although XCS (right) treats the rules in the same way now that the reward function is biased, SB–XCS (left) treats them differently. In particular, rule E, which is overgeneral, has higher expected strength than rule D, which is correct, and with which it competes for action selection. Consequently E is a strong overgeneral (and a fit overgeneral if E and D also compete for reproduction).

5.7.3 Discussion

In these trivial tasks XCS's accuracy-based fitness is effective at penalising overgeneral, strong overgeneral, and fit overgeneral rules. This shouldn't be surprising: for accuracy, we've defined overgeneral rules precisely as those which are less than fully accurate. With fitness based on accuracy these are precisely the rules which fare poorly.

With SB–XCS's use of strength as fitness, strong overgenerals are fit overgenerals. But with XCS's accuracy-based fitness, strong overgenerals – at least those encountered so far – have low fitness and can be expected to fare poorly.

Notes on Figures 5.9 and 5.10

The reader may have noticed that on the right of figure 5.9, the accurate incorrect rules (B & C) gain fitness more quickly than the accurate correct rules (A & D), and that this does not occur on the right side of figure 5.10.

The effect in figure 5.9 is due to the use of an initial prediction of 0 for all rules in that experiment. Because Accurate Incorrect rules update towards a prediction of 0, they have no prediction error, and hence their fitness is immediately updated towards 1.

In contrast, the predictions of Accurate Correct rules are updated towards 1000. Since their initial predictions are 0, this gives them a considerable prediction error. This error, however, is transient. Once these rules reach a prediction of 1000, their prediction error reaches 0, and they too are updated towards a fitness of 1. This transient prediction error accounts for the slower rise in fitness of the accurate correct rules. The opposite effect (slower rise in fitness of Accurate Incorrect rules) could be achieved by using an initial prediction of 1000.

This effect does not occur in figure 5.10 because in that experiment the initial prediction error was reset to 0 after the first update. Because the MAM technique (p. 39) is used on the prediction updates, the first prediction update to a rule makes its prediction equal to the value towards which it is updated (i.e., the learning rate is 1 on the first update). This, combined with resetting the initial prediction error, means there is no bias as a result of the initial prediction value. (An alternative to resetting the prediction error to 0 after its first update is to update prediction error *before* prediction.)

That the predictions of correct and incorrect rules require almost 20 updates to converge is due to the fact that the curves are averaged over 100 runs, and that states and actions were selected equiprobably so sometimes a rule's first update did not occur for many steps. In other words, the curves are an artifact of the averaging done; without averaging, each rule would jump straight to its expected prediction. Similar results should be obtained, however, if one were to use a constant learning rate of, e.g., 0.2, rather than the MAM technique.

5.8 Designing Strong and Fit Overgenerals for XCS

We have already examined the effect of bias in the reward function, and seen that it may produce strong overgenerals in both XCS and SB–XCS, and fit overgenerals in SB–XCS. However, we have also seen that biases in the reward function do not produce fit overgenerals in XCS, since overgenerals (including strong overgenerals) have low fitness under accuracy-based fitness.

That is, accuracy-based fitness gives XCS immunity to fit overgenerals caused by biases in the reward function. Is there a way in which we can induce strong and fit overgenerals in XCS? Perhaps there is another form of bias to which XCS is susceptible.

5.8.1 Biased Variance Functions

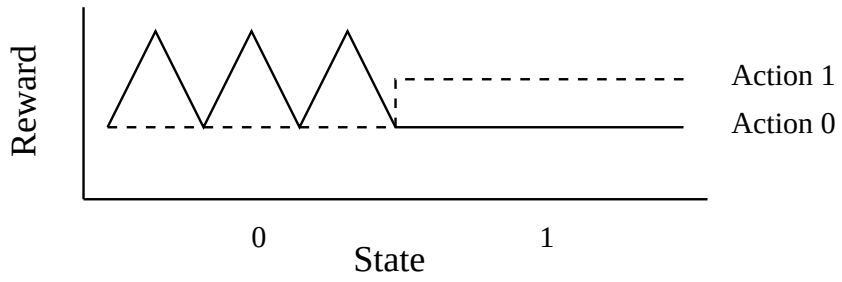
Just as we can bias the rewards themselves, so we can bias the variance in the rewards. That is, we can construct a reward function which has more variance in the reward for one state-action than another. This gives us what we might call a biased *variance function*. (We might also say it gives us a biased *accuracy function*. Suppose we applied tabular Q-learning to a task with a biased variance function and calculated the accuracy of each state-action using XCS's accuracy-based fitness updates. The accuracies of the state-action pairs give us a form of accuracy function, and it will be biased according to the bias in the variance function.)

It was hypothesised that XCS would suffer from strong and fit overgenerals if the variance in the reward function was suitably biased. For example, figure 5.11 shows a task with unbiased rewards, but vastly different variance (σ^2) in the rewards for different state-actions; while the other state-actions all have 0 variance, state-action 0:0 has variance 100. A sawtooth waveform is used in the appropriate part of the state/reward diagram in figure 5.11 in order to suggest variance in the reward for this state-action. Note that the rewards for all other state-actions are shown with flat lines to indicate they have 0 variance. An alternative way to show the variance function for this reward function is to collapse the state-actions onto the x-axis and show the variance on the y-axis, as in figure 5.12.

Because the variance in the reward for state 0 action 0 is so high, rule A will have low fitness in XCS (unless the accuracy criterion ε_o is quite high). In fact, rules E and F, which are overgeneral, may have higher fitness than A, and be fit overgenerals with respect to A. They may also be strong

State	Action	Reward	σ^2
0	0	100	100
1	0	90	0

State	Action	Reward	σ^2
0	1	90	0
1	1	100	0



Rule	Condition	Action	E[Strength]	Fitness
A	0	0	100	low
B	0	1	90	high
C	1	0	90	high
D	1	1	100	high
E	#	0	95	medium
F	#	1	95	medium

Figure 5.11: A task in which XCS may suffer from fit overgenerals. The sawtooth wave indicates a state-action in which rewards are stochastic.

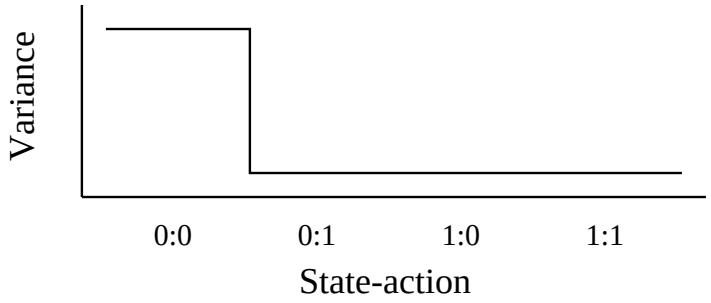


Figure 5.12: The biased variance function for the task in figure 5.11.

overgenerals with respect to A; although they have less strength than A, the inclusion of fitness in the prediction array calculation (p. 36) might discount the prediction of A enough to give E and F more influence in action selection.

5.8.2 Empirical Results

In order to determine whether XCS would develop fit overgenerals in such conditions, XCS was evaluated in the environment in figure 5.11 with different levels of variance for state-action 0:0. For each level of variance, XCS was initialised with a fixed population consisting of the 6 possible rules in figure 5.11. At the start of each run the parameters of these rules were initialised to default values unrelated to the strength and fitness values shown there. Rule discovery was disabled so that XCS operated like a tabular Q-learner. Parameter settings were as on page 59.

XCS was first tested using the rewards in figure 5.11 with no variance; that is, the rewards were deterministic. Next, the reward for state-action 0:0 was made stochastic, and its variance increased incrementally. For each test, the fitness of each of the 6 rules was recorded after 1000 time steps, which was enough time for them to converge to stable values. Figure 5.13 shows the fitness of rules A and E as the variance in the reward for 0:0 increased from test to test. This figure shows the variance in terms of the range of rewards for 0:0. In each test, rewards for 0:0 were obtained by generating a uniform random number r between 0 and R inclusive. The R used for various tests is shown on the x-axis of figure 5.13. The reward given to XCS for state-action 0:0 was:

$$100 + r - 0.5R$$

so that, for example, with $R = 20$, rewards were between 90 and 110, and with $R = 40$, rewards were between 80 and 120. As the range in rewards R increases from 30 to 40, the two curves cross over, and E becomes a fit overgeneral with respect to A.

5.8.3 Avoiding Fit Overgenerals

To avoid fit overgenerals we could modify XCS so that it takes the variance in state-actions into account in calculating fitness. One approach might be to adapt the accuracy criterion ε_o (p. 43) for different parts of the task, which §3.8 has already suggested is necessary to allow XCS to generalise better.

Related Work

Lanzi and Colombetti [165] took a related approach, and estimated the contribution to a rule's prediction error due to two sources: overgeneralisation and error inherent in stochastic environments. The prediction error update was then modified in order to subtract the estimated error inherent in the

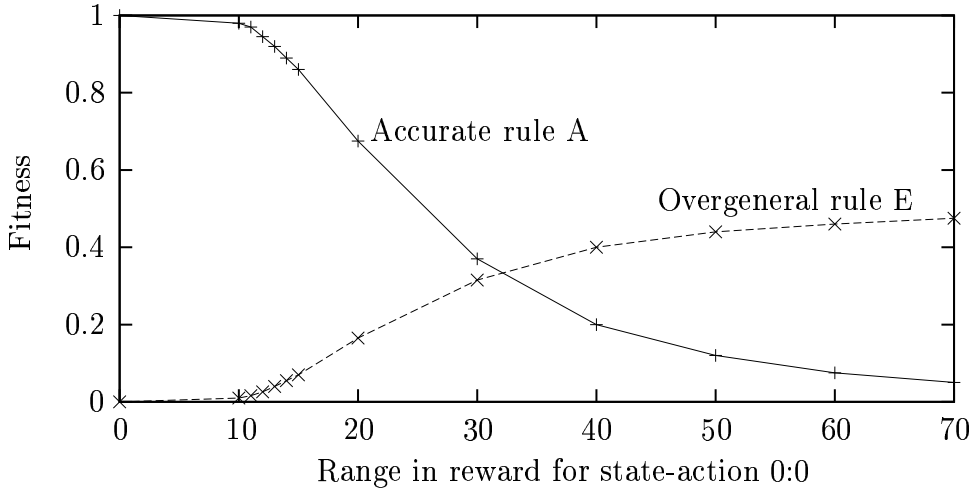


Figure 5.13: Fitness of two rules from figure 5.11 at different levels of variance bias. Rule E becomes a fit overgeneral in XCS when the variance in reward for state-action 0:0 is sufficiently high.

environment. This is the only work in the literature (apart from the present work) in which XCS is applied to a task with a biased variance function. XCS was applied to grid world tasks in which the probability of the intended state transition occurring varied from one state to another, which would have introduced some bias in the variance of rewards. They showed that XCS was able to reach optimal performance when the degree of uncertainty in state transitions (and hence variance in rewards) was significant but limited, but when the uncertainty was too high XCS was unable to reach optimal performance. We can hypothesise that Lanzi and Colombetti's results were due to the presence of fit overgenerals, although no such analysis was performed. Their results agree with the results presented in figure 5.13, where a small bias in the variance function did not make rule E a fit overgeneral, but more substantial bias did. In other words, both studies indicate XCS has some tolerance for bias in the variance function.

By subtracting the estimated error inherent in the environment from a rule's prediction error, Lanzi and Colombetti improved performance in cases of greater uncertainty (higher variance). We can hypothesise that this modification reduced the incidence of fit overgenerals.

5.8.4 SB–XCS and Biased Variance Functions

How does SB–XCS fare with biased variance functions? Will it suffer from fit overgenerals (as XCS does) in the task in figure 5.11? Expected rule strength is not affected by the increase in variance, so SB–XCS should not be affected, other than by stochastic effects, e.g., long sequences of atypical rewards drawn from the uniform distribution, which will occasionally happen.

Thus, SB–XCS suffers from biases in the reward function while XCS suffers from biases in the variance function. Neither suffers from the other's problem. In other words, each is immune to the conditions which cause fit overgenerals in the other!

The obvious question is: which problem is it better to suffer from? It is difficult to say without further study, but, given the results we will see on the prevalence of value function bias in sequential tasks in §5.12, susceptibility to variance function bias may be the lesser of two evils. Study of the amount of bias in each function on different types of tasks, and the effect on each system (given that each has some tolerance for bias) is needed.

Note that SB–XCS's immunity to biases in the variance function gives us another way of constructing SB–XCS-easy yet XCS-hard functions (§3.8.3).

5.9 Strong and Fit Undergeneral Rules

We have seen strong and fit overgeneral rules, which are overgenerals that are stronger or fitter than some correct competitor, and that they result from biases in the reward or variance function. Now we will see that it is also possible for biases in the reward function to produce undergeneral rules which are stronger and fitter than a more-general-yet-correct competitor. The problem with this is that it interferes with the evolution of (accurate) general rules, and so accurate, general representations of the task at hand.

The task in figure 5.14 demonstrates such a case. At the bottom of the figure are three rules of particular interest, their expected strengths and an evaluation of their generality. Although rule C is the most general-yet-accurate, rules B and A have higher strength. (The principle can be demonstrated with only 2 states, but the example seemed clearer with 4.) This is not a problem in action selection since they all advocate the same action, but it is in reproduction. That is, strong undergeneral rules are not a problem, but fit undergeneral rules are. Let's consider SB–XCS and XCS in turn.

SB–XCS

With SB–XCS's strength/fitness, the less general rules are fitter, which is clearly undesirable. This effect of fit undergenerals depends on the reward function being biased; with an unbiased reward function, fit undergenerals cannot occur. However, even with unbiased reward functions there must still be some bias towards generality, otherwise SB–XCS will not prefer the more general of two consistently correct rules. This is a real problem, since without a generality preference the population will swell up with an enormous number of correct but overly specific rules.

These problems demonstrate that SB–XCS must factor generality into rule fitness if it is to evolve accurate, general rules. Happily, the niche GA provides an effective fitness bonus which does just this.

XCS

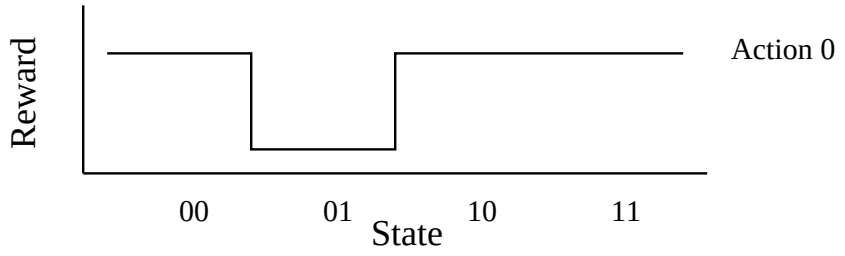
In XCS, the fitness of A depends on the setting of the accuracy criterion. If it is strict enough, A is, by definition, an overgeneral, and the low fitness it receives is appropriate.

If, however, A is not overgeneral according to the accuracy criterion, it is only as fit as B and C. In other words, XCS will have equal preference for the three, on the basis of their fitness. Thus, although XCS does not suffer from fit overgenerals (unless the variance function is sufficiently biased §5.8), it still needs some bias towards generality, which, happily, it has thanks to the niche GA.

5.10 Why Bias the Reward Function?

The reward function is the means by which we define the goals of our learning systems when we design reinforcement learning experiments (see [265]). The reward function is part of the definition of the task; choosing different reward functions results in different learning tasks. The trick is to choose a reward function which results in the desired behaviour. Given that the experimenter defines the reward function, SB–XCS's problems with biased reward functions can potentially be avoided by using unbiased ones (or only slightly biased ones). Why use biased reward functions if they can cause problems for strength-based systems?

State	Action	Reward
00	0	1000
01	0	200
10	0	1000
11	0	1000



Rule	Condition	Action	E[Strength]	Generality
A	00	0	1000	undergeneral
B	1#	0	1000	undergeneral
C	##	0	800	optimally general

Figure 5.14: A task which supports strong undergeneral rules.

5.10.1 Some State-actions are more Important than Others

One plausible reason to bias the reward function is to get the system to allocate more resources (classifiers) to more important parts of the task. For example, we might consider the detection of an impending meltdown to be more important than optimising the output of a reactor. Similarly, in scanning medical images for signs of cancer, we might want to avoid false negatives more than false positives. In a sequential task, we might want to give more reward for finding water than food if a simulated frog needs to drink more often than it eats.

5.10.2 A Rule Allocation Bias can Focus Resources

If some state-actions are more important than others, we should allocate more rules to them to help ensure the correct action is learnt there. This is analogous to giving more weight to minimising errors over parts of a training set in a neural network. Seen in this light, the problem of greedy classifier

creation from §5.1 is really a useful feature rather than a harmful bug: all we need to do is assign higher rewards to state-actions which we consider more important, and greedy classifier creation will ensure more rules are devoted to it. (This trick will not work with XCS, since in XCS a rule's fitness is not proportional to the rewards it receives.)

A rule allocation bias only becomes a problem when it is inappropriate, for example, by emphasising the importance of the wrong states, or by being too extreme and causing gaps in the covering map. In any case, if we're really concerned with gaps in the covering map we should be able to modify our rule allocation algorithms to ensure they don't occur. We could, for example, only delete a rule if all the states it matches are also matched by some other rule (or rules).

5.11 Rule Allocation Reconsidered

In §5.10 we saw that biasing reward functions might be useful, as it allows us to bias rule allocation. However, there are two problems to consider. First, biased reward functions make strong overgenerals possible, which is problematic for SB–XCS if not XCS. Second, it is not clear that this is an appropriate way to bias rule allocation in the first place.

In non-sequential tasks biasing the reward function in one state does not affect other states (apart from influencing rule allocation). The situation is, as usual, more complicated in sequential tasks. First, rule allocation is based on the value function, not the reward function, although since the value function is derived from the reward function the reward function still influences rule allocation. Second, biasing the reward function in one state affects the value function in other states. Due to discounting, the value of states declines the further they are from reward, and, as Wilson pointed out in [297] (section 5.3), it is not clear that more rules should be allocated closer to rewarding states. Is it really more important that we act correctly in these states? Let's express the importance of acting optimally in a state as a function $\mathcal{I}(s)$. We can express the hypothesis that it is more important to act correctly closer to rewards as:

$$\mathcal{I}(s) = \max_a Q(s, a) \tag{5.1}$$

In the next section we'll see a shortcoming of this formulation of the importance of acting correctly.

5.11.1 Knowing What Not to Do

Is it really more important to minimise errors in higher-valued states? We might reason that the goal of any reinforcement learner is to maximise the reward it receives, so a state's (or rule's) importance is determined by its contribution to obtaining reward. This would suggest that rules which lead to higher rewards are more important. But rules can contribute indirectly to maximising reward, which suggests the importance of a rule is not the same as the desirability of taking the action it advocates. Surely rules which can save us from disasters are also important; as well as knowing what to do, we may need to know what *not* to do.

For example, a robot might have a rule whose use would lead it to fall down a flight of stairs and damage itself. We definitely do not want the robot to take the action advocated by this rule, so we associate a very low reward (or, equivalently, a large negative reward) with this outcome. At the same time, however, we definitely do not want the robot to forget this rule, because it is very important not to forget the consequences of its use. Not all tasks have disasters like this, but some do.

Let's rephrase the importance of acting correctly in a state s in terms of the cost $\mathcal{C}$ of acting suboptimally there. One way to define the cost is to look at the worst case, that is, the cost of taking the worst possible action in s rather than the best:

$$\mathcal{C}(s) = \max_a Q(s,a) - \min_a Q(s,a) \tag{5.2}$$

This worst-cast cost function would give high value to states in which the robot risks falling down stairs by acting inappropriately.

5.11.2 Managing Exploration

Avoiding disasters seems like an extreme case of the more general problem of managing exploration (§C.2). After all, once we've learned to behave satisfactorily in our task we no longer need to remember what not to do – all we need is to remember what to do. It's only when exploring – when there is some chance of taking what appears to be a suboptimal action – that we need to explicitly guard against taking very suboptimal actions.

But it is not only when there are potential disasters to avoid that we can benefit from keeping track of suboptimal actions. Rules may well appear suboptimal and yet if we haven't tried them (and the alternatives) sufficiently we can't be sure they really are suboptimal. A single trial is sufficient in deterministic non-sequential tasks, but in other cases many trials may be

needed to find a good estimate of the true value of the rule. Exploration control is extremely important to efficient reinforcement learning, particularly in sequential tasks, because inefficient exploration can be very wasteful of trials.

5.11.3 Complete and Partial Maps Revisited

Recall that a complete map contains both consistently correct and consistently incorrect classifiers. However, maintaining consistently incorrect classifiers in the population does not mean we have to take an incorrect action each time one is applicable. In fact, quite the opposite is true. If a classifier is consistently incorrect, this suggests we should *not* take the action it advocates. If we did not keep this consistently incorrect classifier in the population, how would we know it was a bad idea to take its action? If we delete it, we have no record of the utility, or lack thereof, of taking that action in that state, and we might be tempted to try the suboptimal action again and again in the future. We need to keep incorrect classifiers in order to keep track of guesses which have not paid off; in other words to manage our exploration.

If we do maintain rules which tell us how to avoid particularly bad outcomes we are maintaining a more complete map than SB–XCS does. To keep track of all actions requires a complete map.

5.11.4 Alternatives to Biasing the Reward Function

Although biasing the reward function may be a useful way of biasing rule allocation in SB–XCS in some cases, there are three problems with it. First, it can cause strong and fit overgenerals in SB–XCS. Second, it doesn't consider the problem of learning what not to do. Third, this approach cannot be used to bias rule allocation in XCS, since it does not allocate rules to states in proportion to reward. Given these problems, can we bias rule allocation without biasing the reward function?

One way may be to vary the frequency with which individual states are encountered, so that more important states are presented more frequently. In non-sequential tasks we may have complete control over which states are visited. In sequential tasks, however, the agent and task determine which states are encountered, not the experimenter. When we cannot control the task we may be able to record experiences and play them back in simulation with a frequency bias towards more important ones.

Alternatively, we could split the classifiers into multiple subpopulations, each with a different size limit, and assign each to a subset of task states.

More important parts of the task would get larger subpopulations. This makes the system more complicated, but has the advantage of allowing the use of domain knowledge to generalise a priori by limiting the input to each subpopulation (see [135, 142]).

A third alternative is simply to define a function (e.g., the worst-case cost (5.2)) to weight each state, and to modify the system to take it into account when allocating rules.

5.11.5 Can SB–XCS Avoid Strong and Fit Overgenerals?

We saw in §5.6.1 that strong and fit overgenerals in SB–XCS are impossible when the reward function is unbiased, so one way to try to ensure SB–XCS will adapt to a task is to use an unbiased reward function. Unbiased reward functions may suffice, particularly when used with the alternative means of biasing rule allocation just outlined.

This suggests a way for SB–XCS to avoid strong and fit overgenerals and adapt to more non-sequential tasks than it would otherwise. However, in §5.12 we will see that SB–XCS cannot escape serious difficulties with sequential tasks.

5.12 Sequential Tasks

In non-sequential tasks a reinforcement learner approximates the reward function defined by the experimenter, which is in principle enough to maximise the reward it receives. In sequential tasks, however, consideration of the reward function alone is not sufficient, as it defines immediate reward (the reward on a given time step) only (§C.3.1). In sequential tasks the learner's actions influence the state of the task and hence which rewards it may receive in the future. Consequently the learner must take into account future consequences of current actions if it is to maximise the total amount of reward it receives over multiple time steps. Many reinforcement learning systems do so by learning a *value function*, which maps each state to an estimate of its long-term value. The value function for a task is implicitly defined by the task definition.

Q-learning

We'll examine the learning of sequential tasks by Q-learning agents, since XCS and SB–XCS both use the Q-update to estimate rule strength. Q-

learners take long-term consequences into account by learning a *Q-function* (also called an *action-value* function) which maps state-action pairs to an estimate of their long term value called their *Q-value*. (Q stands for quality of the state-action pair.) A Q-function is really just a special kind of value function, in which we estimate the long-term value of state-action pairs rather than states. In fact, we can define a value function $V(s)$ as:

$$V(s) = \max_a Q(s, a) \tag{5.3}$$

That is, the value of a state s is the value of its highest Q-value.

In non-sequential tasks XCS and SB–XCS both use the non-sequential definition of P (p. 41). The strengths of rules are estimates of immediate reward, and the population of rules estimates the reward function. In sequential tasks, when using the sequential definition of P (p. 41), the strength of a rule is an estimate of long-term value, and the population of rules and their strengths approximate the value function.

Just as in non-sequential tasks, strong and fit overgenerals depend on the reward function, so in sequential tasks they depend on the value function (or Q-function). (In fact, we can refer to non-sequential value functions by using a non-sequential definition of return (C.1), in which case the value function is the reward function (§E.2). However, it seems clearer to refer to reward functions in the non-sequential case and value functions in the sequential case.)

Consequently, it is of interest to define a unbiased value functions, under which strong and fit overgenerals are impossible following theorems 1 and 5.

Unbiased value function: A value function V is unbiased iff there exists a constant c such that:

$$V(s) = c \tag{5.4}$$

for all $s \in \mathcal{S}$. That is, a value function is unbiased if it is a constant function.

Recall from §5.4.1 that a reward function is unbiased if it is constant *over correct actions*, not simply constant. The difference occurs because the value function is not parameterised by actions; by (5.3) the value of a state is the value of its highest-valued state-action.

5.12.1 The Need to Pass Values Back

Q-values are estimates of the long-term value of taking a given action in a given state, not just the immediate reward for doing so. To effect this, each Q-value is updated towards a fraction of the value of its successor. In this way, the estimate of the value of acting now comes to reflect some of the value of what happens later.

To see how this works, let's look at the Q-update, which for convenience we repeat from page 265 in Sutton and Barto's notation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\left[\overbrace{r_{t+1} + \gamma \max_a Q(s_{t+1}, a)}^{Target} - \overbrace{Q(s_t, a_t)}^{OldEstimate} \right]}_{Error} \quad (5.5)$$

where t is the time step, and α controls the learning rate (equivalent to β in Wilson's notation on page 42).

The Q-value of a state-action pair $Q(s, a)$ is updated towards the target component of (5.5), that is, towards the immediate reward r plus some fraction γ of the value of the state which follows it ($\max_a Q(s_{t+1}, a)$). (We say the value of the following state s_{t+1} is discounted and passed back to its predecessor s_t.) In this way a state-action which receives an immediate reward of 0 will have a Q-value greater than 0 if the state it leads to has a Q-value greater than 0 (assuming $\gamma > 0$). In other words, by passing value back a state takes on some of the value of its successor. In this way, Q-updates drive Q-values towards the long-term value of a state-action.

A simple task is shown in figure 5.15 to illustrate Q-learning. The immediate reward r and true Q-value Q for each state-action, assuming a discount rate of $\gamma = 0.9$, are shown. We define the value of the terminal state to be 0 so that the value of the transition labelled c is due entirely to the immediate reward there.

We can initialise estimates of each Q-value arbitrarily, and the Q-updates will move them towards the true Q-values shown. (In fact, given infinite revisits to each state-action, and appropriately declined α, the estimated Q-values are guaranteed to converge to their true values.)

Knowledge of the true (as opposed to estimated) Q-function for a task is sufficient to act optimally by simply taking the action in the current state with the highest associated Q-value. In figure 5.15 this means taking a' rather than i' and a rather than i.

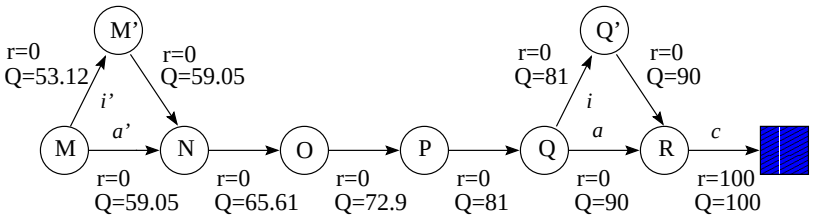


Figure 5.15: A simple sequential task showing immediate rewards r and Q-values Q for state transitions using $\gamma = 0.9$. M is the start state and the square state is terminal.

5.12.2 The Need for Discounting

We've just seen that we must pass value back in order to take long-term consequences into account. But how much value should we pass back? In the Q-update, this is parameterised by the discount rate γ. We can think of γ controlling how much consideration the system gives to future rewards in making decisions. At $\gamma = 1.0$ no discounting occurs, and the system will learn the path through the task's states which results in the most reward, regardless of how long the path is. This is often not what we want. For example, being paid £5 a year from now is not as desirable as being paid £5 today. If we used $\gamma = 1.0$ in figure 5.15 then both the i and a transitions would have Q-values of 100 and the system would be unable to choose between them. At the other extreme, if we set γ to 0.0 the system will be shortsighted and take no interest in the future consequences of its actions. This too is often undesirable, as it would lead the system to choose £5 today rather than £1000 tomorrow. In figure 5.15, $\gamma = 0.0$ would give i and a Q-values of 0, and again the system would be unable to choose between them. Typically we will want to set γ to some value between 0 and 1 in order to give possible future rewards a suitable weighting.

5.12.3 How Q-functions become Biased

Passing values back from one state to another is necessary for the system to take future consequences of its actions into account. Discounting of these values is necessary for the system to find shorter paths through the state space. However, passing values back and discounting tend to produce one of the two criteria for the production of strong overgenerals: a biased Q-function. Figure 5.15 demonstrates this effect. Notice that even though there are only two different immediate rewards (r is either 0 or 100), there are many Q-values. Even though the reward function has only two different

rewards for correct actions (and so is a little biased) the Q function has many different rewards for correct actions (and so has more biases than the reward function).

The other criterion for the formation of strong overgenerals, that it be possible to act incorrectly, is met in all non-trivial tasks, specifically, in all tasks in which we have a choice of alternative actions in at least one state.

5.12.4 Examples

Let's look at some examples of strong overgenerals based on figure 5.15.

Short Sequences can Produce Strong Overgenerals

Imagine the situation where an overgeneral matches the state-actions labeled c and i. Now imagine a second rule which is correct and which advocates only the state-action labeled a. This rule competes with the overgeneral for action selection in state Q, and its strength is 90.

Using the strength definition of strong overgenerality (p. 148), the overgeneral is a strong overgeneral if it is stronger than a correct competitor. If we assume the overgeneral experiences both the transitions it advocates with equal frequency, we can use the inequality from theorem 6 to tell us whether it is a strong overgeneral:

$$(c + i)/2 > a \tag{5.6}$$

which evaluates to:

$$\begin{aligned}(100 + 81)/2 &> 90\\ 90.5 &> 90\end{aligned}$$

This example demonstrates that strong overgenerals can be obtained even with very short sequences of states, even the minimal sequence shown here.

Any $0 < \gamma < 1$ can Produce Strong Overgenerals

If the value of a state s is $V(s)$, discounting results in $\gamma V(s)$ being passed to the state preceding s. More generally, a state n steps ahead of s receives $\gamma^n V(s)$ from s. Let n be the number of steps between c and both i and a. Then we can rewrite the strong overgeneral inequality (5.6) as:

$$(c + \gamma^n c)/2 > \gamma^n c \tag{5.7}$$

which is true for any $c > 0$, $n \geq 1$, and, significantly, any $0 < \gamma < 1$. In other words, according to our approximate expression, passing back *any* fraction of the value of a state (other than all of it, i.e., $\gamma = 1$) will produce a Q-function capable of supporting strong overgenerals in this task. This does not occur if $\gamma = 1$, but we've already seen in §5.12.2 that passing back the entire value of a state often does not produce the results we want. Of course our calculations have been greatly oversimplified, but it should be clear that all but the simplest sequential tasks can support at least some strong overgenerals.

Longer Sequences Increase the Bias

Now let's look at another example, in which the overgeneral matches in states R and M, and the correct rule matches only in state M. We now use the Q-values labelled a' and i' for a and i in $(c+i)/2 > a$ and obtain $76.56 > 59.05$. Notice that in this example the overgeneral acts incorrectly farther from the goal than in the first example, but its strength exceeds the threshold required of a strong overgeneral by a greater amount. The farther i and a are from the goal, the stronger the strong overgeneral will be, compared to the correct classifier. Notice also that the farther i and a are from the goal, the easier it is to produce strong overgenerals because there are more state transitions in which c can occur and gain enough strength to produce a strong overgeneral. (We can show the same thing by increasing n in (5.7).)

5.12.5 Woods2 Revisited

When in §2.5.2 tabular Q-learning, XCS and SB–XCS were evaluated on the sequential Woods2 task, it was found that although the first two achieved optimal performance of an average of 1.7 steps to food, SB–XCS only reached approximately 3 steps to food. Why is this? This section has argued that sequential tasks have biased value functions, and that this tends strongly to cause strong and fit overgenerals in SB–XCS. Let's look at Woods2 in more detail.

Although the diagram of Woods2 on page 62 shows a large gridworld, Woods2 is very regular and consists of a basic 5x5 pattern (shown in figure 5.16) repeated many times. In some copies of the pattern, G is substituted for F, and Os may be substituted for Qs. Since both kinds of rock and both kinds of food behave identically, the only effect of these substitutions is to increase the number of inputs which the animat may experience, and to create equivalence classes among the inputs, over which the animat can generalise.

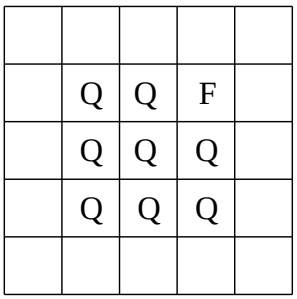


Figure 5.16: The basic Woods2 pattern.

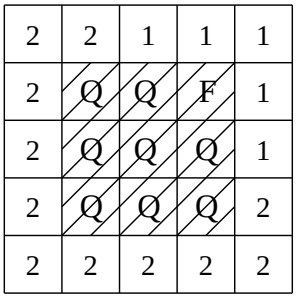


Figure 5.17: The basic Woods2 pattern with steps to food from each blank cell.

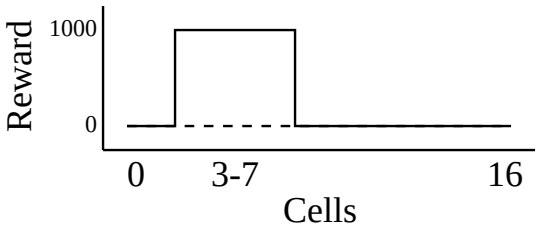
Figure 5.17 shows the basic Woods2 pattern, but with the number of steps needed to reach a goal state shown in each blank cell, and the food and rock states crossed out for legibility. Of the 25 cells in the basic pattern, 16 are blank. Of these, 5 are 1 step from a goal state and 11 are 2 steps. (That the majority of the cells are 2 steps from the goal is only obvious when the basic pattern is repeated, as on page 62.)

Given random initial placement of the animat in a blank cell, the average number of steps to food is $(5 \cdot 1) + (11 \cdot 2)/16 = 1.6875 \approx 1.7$. That at most 2 actions – i.e., a *minimal* sequence – are needed to reach a goal state suggests Woods2 is not a particularly difficult sequential decision task, even if random behaviour averages 27 steps to food.

The Reward Function

Although Woods2 contains 5 types of cell (2 types of food, 2 types of rock, and blanks), the animat can only occupy blank cells. Given the sensory coding specified in §2.5.2 there are 70 distinct states (input strings) the animat can

encounter. These 70 states occur in the 16 blank cells in the basic pattern which the animat can occupy, and we can group states according to the cell in the basic pattern in which they occur. That is, cells can be thought of as macro-states, or equivalence classes among states. Transitions to food states result in a reward of 1000, while all other actions result in a reward of 0. This produces a biased reward function, as in some states the correct action results in 1000 reward and in others 0 reward. The following figure shows the reward for state-actions grouped according to which of the 16 empty cells they occur in, numbering the cells from the top left of the basic pattern and working around its edges clockwise. For each state $\max_a R(s,a)$ is shown as a solid line, while the reward for all others state-actions is shown with a dashed line.



Given the analysis earlier in this chapter, a biased reward function immediately suggests SB–XCS may suffer from strong and fit overgenerals. Let us look next at the Q-function function.

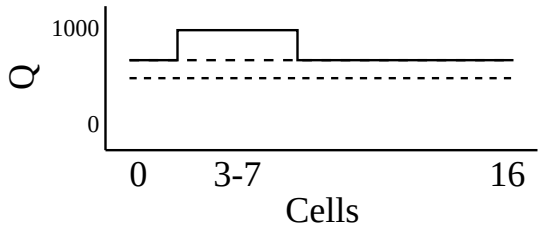
The Q-function

Given that the animat can be at most 2 steps from a food state, we can classify all actions as follows: those which transition to food, those which transition to a state 1 step from food, and those which transition to a state 2 steps from food. Consequently, the Q-function for Woods2 is particularly simple, having a range of only 3 values, one for each case above.

Actions which transition to a food state receive a reward of 1000, and, since food states are terminal, no value is backed up from successor states. Assuming $\gamma = 0.71$, actions which transition to a state 1 step from food have a Q-value of $\gamma 1000 = 710$, while those which transition to a state 2 steps from food have a Q-value of $\gamma^2 1000 \approx 504$. (Since the furthest the animat can start from the food is 2 steps, transitions *to* a state 2 steps from food only occur either when it moves from a state 1 step from food (i.e., moves the wrong way), or when it attempts to move into a wall from a state 2 steps from food.)

The following figure shows the Q-function for state-actions belonging to the 16 empty cells, numbering them clockwise from the top left as before.

The solid line indicates the Q-value of the optimal action in each state, while the dashed lines show the values of suboptimal actions.



Notice that the Q-function is less biased than the reward function, in that the Q-values are closer together than the rewards. It is more biased, however, in that it may take on 3 values, whereas the reward function has a range of only 2 values. This means there are more points at which the Q-function is biased than the reward function. That is, there are more pairs of state-action pairs whose Q-values differ than there are pairs whose rewards differ.

How Hard is Woods2?

Given that the Q-function for Woods2 is biased, we can expect SB–XCS to suffer from strong and fit overgenerals. Does this account for SB–XCS's performance of approximately 3 steps to food? Let's consider how well SB–XCS is really doing.

If random behaviour averages 27 steps to food, SB–XCS's 3 steps seems quite good, and is nearly identical to ZCS's performance on the related Woods1 task [297]. (See, however, [47] and comments on page 186.) On the other hand, in the experiment on page 66, SB–XCS's performance fell to 28 steps to food when exploratory actions were disabled, indicating it was relying on occasional exploratory actions to get it out of loops, and so had not learnt an effective policy. Considering that the animat is only ever at most 2 steps from food, we have to ask how well SB–XCS (and, incidentally, ZCS) are really adapting to this task when they achieve 3 steps to food.

One way to approach this question is to ask how well an agent which cannot learn about sequences of actions would do. That is, how well would an agent do if it learnt how to act in cells adjacent to food, but could not propagate information from these cells to cells further from the food? To find out, tabular Q-learning was run in Woods2 with $\gamma = 0$, so that it would not adapt except in those cells immediately adjacent to food. Other parameters were as in §2.5.2.[10]

[10]The tabular Q-learner was implemented as discussed in §6.1.3.

Averaged over 10 runs, this system converged to approximately 4.3 steps to food, despite the fact that it adapted in only 5 of 16 blank cells and behaved randomly in the others. (In all runs it adapted in those 5 cells which were adjacent to food.) In other words, the performance of SB–XCS and ZCS at 3 steps to food is not much better than a system which is incapable of learning sequences of actions!

How is the relatively good performance from the non-sequential learner possible? To begin, in 5/16 of the trials the animat starts in a cell adjacent to food. In these states it is able to learn the optimal policy, thanks to the immediate reward it receives when it takes the correct action. In the other 11/16 of the trials, it behaves randomly, and usually stumbles by chance across a state adjacent to food – in which it knows how to act – within a few steps. (The system did, however, occasionally time out after 100 time steps.) Inspection of the learnt Q-values confirms that this is how the system operates; state-actions which lead to food have Q-values of 1000, and all others have Q-values of 0. Essentially, Woods2 is not a very difficult sequential decision task. A system which can learn the optimal policy in states adjacent to food (which requires no sequential learning), and which acts randomly elsewhere performs reasonably well.

Inspection of rules evolved by SB–XCS suggests that it adopts the same technique of learning to act optimally in states adjacent to food. (Note that if we consider only these states, the reward function is unbiased.) Given this, its performance in other states must be particularly bad. Rules with numerosity greater than 3 are shown in figure 5.18, sorted first by numerosity and then by strength. The most numerous rules are unsurprisingly those with strength 1000, i.e., those which map the states adjacent to food. There are, however, a number of high-numerosity overgeneral rules present. Such rules are easily detected in the list as their strengths differ considerably from the true Q-values towards which a tabular Q-learner will converge in this task (1000, 710 and 504).[11] And yet, these rules have considerable experience, suggesting that they would by now have converged to stable values were they not overgeneral. Additionally, the prediction error ε is high for these rules. (SB–XCS calculates prediction error to provide additional statistics on rules, but does not use it itself.)

These rules are in fact strong and fit overgenerals, as they compete with less strong and fit accurate rules. The less strong competitors do not appear in figure 5.18 simply because they have numerosity less than 4, suggesting that the strong and fit overgenerals are effectively depriving them of nu-

[11]The true Q-value of a state-action is defined mathematically by the task definition. The Q-values learnt by an LCS or tabular Q-learner are estimates.

merosity, and, in many cases, actually driving them from the population.

The only correct rule with strength less than 1000 is the last shown. Notice that despite having numerosity 4, it only has experience 6, indicating that it was created only recently. There are very few other correct rules with strength less than 1000 in the population, all with numerosity less than 4, and all with low experience. This suggests that although SB–XCS is able to find some correct rules, they do not tend to survive long, nor do they accumulate large numerosities. (The numerosity of 4 for the last rule shown is probably unusually high for such a rule.) This makes sense; although desirable, such rules have lower fitness than their fit overgeneral competitors. SB–XCS's fitness calculation does not value many of the rules it needs to adapt to this task.

That SB–XCS and ZCS perform slightly better than the $\gamma = 0$ Q-learner indicates that they *are* adapting somewhat in the states 2 steps from food, which is likely to result from the higher strengths of rules which move towards food. It is clear, however, that even in this very simple task, with a simple, not very biased Q-function, that SB–XCS has not learnt the optimal policy in states 2 steps from the goal. That is, its ability to learn sequences of actions in this task appears to be extremely limited. This suggests that strong and fit overgeneral rules are a considerable problem even when the Q-function is not heavily biased. This is emphasised by the fact that strong overgenerals actually make up a small proportion of the total numerosity in figure 5.18, and yet appear to prevent SB–XCS from adapting.

ZCS Redux Recently, Bull and Hurst [47] have shown that different parameter settings, and turning the GA off after the system has adapted, allow ZCS to reach near-optimal performance on Woods1. Clearly, in this case it is able to adapt in states which are not adjacent to food. It is possible that SB–XCS could reach similar performance if it were better parameterised. However, it may be that ZCS's fitness sharing is responsible for its near-optimal performance in [47], in which case SB–XCS (lacking fitness sharing) should be unable to match ZCS. This matter deserves further investigation.

The Mountain Car Task

The Q-function for Woods2 is rather simple. As a more interesting example, consider the value function for the mountain car task, in which an underpowered car is trapped between two mountains. To escape, the car must learn to rock back and forth in the valley between the two in order to gain sufficient momentum to reach the peak (see figure 5.19).

Condition	Act.	Str.	ε	Num.	Exp.
0##00#00 ##0##000 ##0##1##	NW	1000	0.0	26	1517
##0#0#00 000#00## 1#00##0#	SW	1000	0.0	17	2253
0###0#00 #11##### ##000#0#	SE	1000	0.0	16	387
#000#00# #0#0##0# 1##1##0#	W	1000	0.0	9	376
0######0 0#00#0#0 #00000##	SE	656	0.08	9	318
00000#00 #11##### ##0#0#0#	SE	1000	0.0	8	295
#000#00# ##0#0### ###1##0#	W	1000	0.0	7	292
##00##00 000##0## 1#00##0#	SW	1000	0.0	7	1026
0##00#00 ##0##000 1####1##	NW	1000	0.0	6	153
00000000 000#1### ########	S	1000	0.0	6	197
#00##0## #0#00### ###1##0#	W	1000	0.0	6	324
0#000#00 ##0##000 ##0##1##	NW	1000	0.0	6	464
0##00#00 000#00## 1#00##0#	SW	1000	0.0	6	1809
#000#00# #0#00### ###1##0#	W	1000	0.0	6	1577
##000### #0##000# ##00##0#	SW	772	0.13	6	217
#00#0#00 ######## #####0##	SW	701	0.14	6	499
0###0#00 #11##### ##0#0#0#	SE	1000	0.0	5	314
##0#0#00 000#00## 1000##0#	SW	1000	0.0	5	461
000###0# ##0##1## ##0####0	S	1000	0.0	5	317
#0###### ##0#0### ###1##0#	W	1000	0.0	5	367
0###0000 011##### ##0###0#	SE	1000	0.0	5	477
##0#0#00 000#00## 1#000#0#	SW	1000	0.0	5	1335
00##00#0 0##0##00 1##1##0#	W	1000	0.0	5	499
##00##0# 000##0## 1#00##0#	SW	1000	0.0	5	1467
#0000000 000#1### ##0#####	S	1000	0.0	5	963
00##000# 0####0## #####00#	W	914	0.15	5	263
#00#0#00 ######## ####00#0	SW	743	0.15	5	446
#0000000 000#1### ##0####0	S	1000	0.0	4	154
000#0#00 ##0#1### ##0####0	S	1000	0.0	4	184
0###0000 011##### ##0#0#00	SE	1000	0.0	4	286
#0#0000# ##0#1### ###000#0	S	1000	0.0	4	330
##0#0#00 000#0##1 ######0#	SW	1000	0.0	4	1303
#0####0# ###00### ###1##0#	W	1000	0.0	4	530
00000#00 #11##### ##00##0#	SE	1000	0.0	4	1367
##0###00 000#00## 1#00##0#	SW	1000	0.0	4	2326
#000#00# ##0#0### ###1#0#0	W	941	0.16	4	229
00###00# ##0#0### ###1####	W	893	0.22	4	260
0##01#0# 0#0##### ##00###1	SE	709	0.0	4	6

Figure 5.18: The most numerous rules evolved by SB–XCS for Woods2.

The reward function returns -1 on each time step of the task, and so is unbiased. Figure 5.20 shows the value function for the mountain car task with $\gamma = 1$, using the discretisation found by Reynolds's adaptive resolution reinforcement learning system [206, 207, 208]. That is, for each state $V(s) = \max_a Q(s, a)$ is shown. (Figures 5.19 and 5.20 appear courtesy of Stuart I.

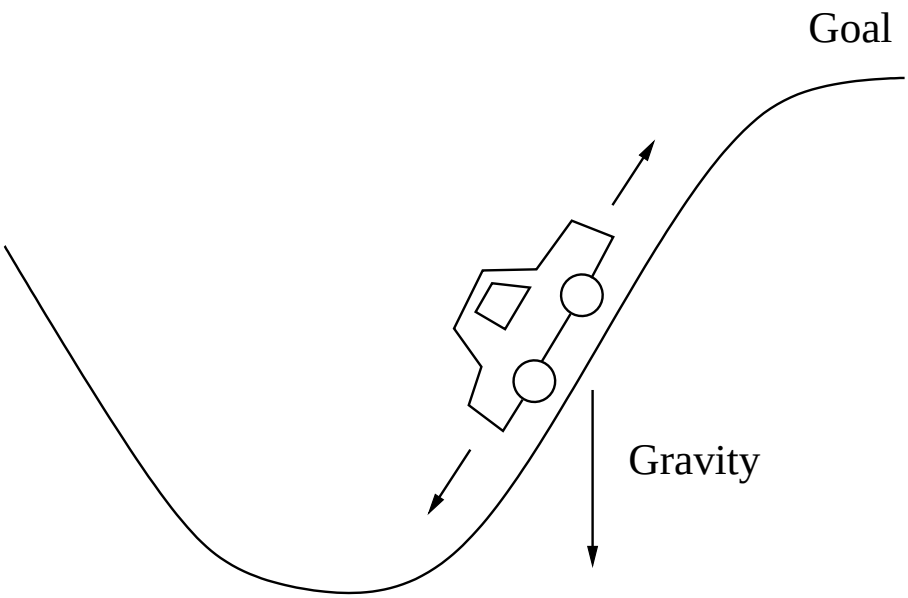


Figure 5.19: The mountain car task.

Reynolds.)

Although the value function contains many plateaus (thanks to the state aggregation performed by this learning system), it is nonetheless decidedly very highly biased! Given the great biases in this function, we can expect SB–XCS to produce an overwhelming number of strong and fit overgeneral rules and so be unable to adapt. This task and its complex, highly irregular value function are more typical of tasks studied in the mainstream reinforcement learning literature than Woods2. This task has has been included here simply to illustrate the complexity of a typical value function, and to suggest the difficulty which SB–XCS should have on such tasks.

5.12.6 When Will the Value Function be Unbiased?

SB–XCS should be able to adapt to tasks with unbiased value functions, since this makes strong and fit overgenerals impossible (p. 160). Under what conditions will a value function be unbiased?

To address this, let's pose the question more carefully. In the task in figure 5.15, under what reward functions and values of γ will we obtain a value function which is unbiased over non-terminal states? (We do not ask that the terminal state have the same value as the other states since we defined its value to be 0.)

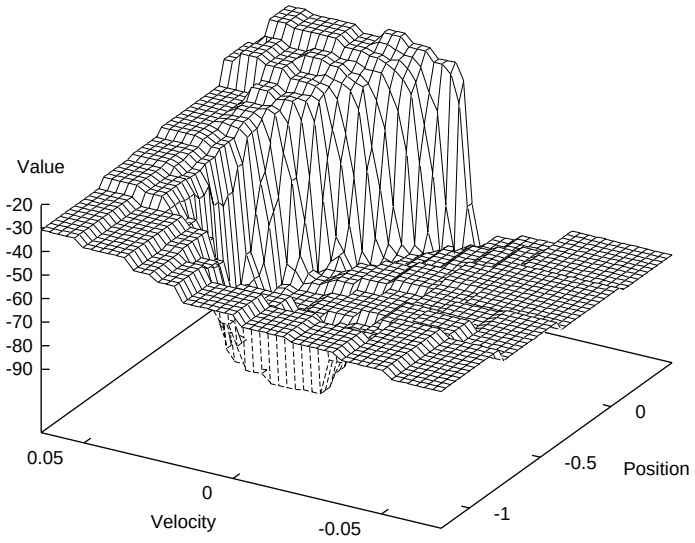


Figure 5.20: Value function $V(s) = \max_a Q(s,a)$ for the mountain car task.

In terms of the reward function, the value function will be unbiased over non-terminal states only when:

$$\max_a R(s,a) = (1-\gamma)V(s') \tag{5.8}$$

for all $s, s' \in \mathcal{S}$, that is, when the reward function exactly makes up for the value lost from the successor state due to discounting. Two cases where this occurs are:

1. It occurs in non-sequential tasks when the reward function is unbiased.

2. It occurs when the reward function is constant 0 over correct actions (γ can take any value).

In other words, the value function is only unbiased either in non-sequential tasks with unbiased reward functions (case 1), or in uninteresting sequential tasks with a degenerate reward function (case 2).

Unbiased Value Function = Non-sequential Task

Note that to assert that the value function is unbiased is to assert that all states have equal value. That is:

$$V(s) = V(t) \tag{5.9}$$

for all $s, t \in \mathcal{S}$. If this is the case, the task is effectively non-sequential, since there are no sequential decisions to make; being in any state is as good as being in any other. The only issue is what action to take, and, for a classifier system, how to generalise over states and actions.

5.13 What Tasks can we Solve with SB–XCS?

The circumstances under which a value function will be unbiased, and so under which SB–XCS can be expected to adapt, are extremely limited, consisting of non-sequential tasks with unbiased reward functions, and uninteresting sequential tasks which are effectively non-sequential.

Furthermore, the unbiased non-sequential tasks for which SB–XCS is suitable are probably often better modelled as supervised learning tasks (§C.1.1). If we can specify the correct action in each state, we have enough information to do supervised learning. Since the supervised learning paradigm provides the learner with more information (allowing it to avoid the explore/exploit dilemma), agents should be able to adapt more quickly when a task is formulated as supervised learning. As noted in §3.8.4, unpublished work has shown a supervised-learning-like XCS to outperform the standard (RL) XCS on the 6 multiplexer [145]. Given this, one must ask what the value is in modelling non-sequential tasks as reinforcement learning tasks (other than to study the workings of a reinforcement learning system, as we did with the 6 multiplexer (§2.5.1)).

This analysis is slightly pessimistic, given that SB–XCS has some tolerance for biases in the reward function (§5.6.2), and so may adapt in some cases where the value function is biased. However, such cases are likely to be very few and far between. (Recall that in the task in figure 5.15 any discounting, and any length of action sequence was sufficient to produce strong overgenerals, under our simplifying assumptions (§5.12.4).)

This suggests SB–XCS will suffer from strong and fit overgenerals in essentially all interesting tasks. How much of a problem are strong and fit overgenerals? Experiments with Woods2 show that SB–XCS adapts reasonably well, but largely because much of the task is non-sequential (§5.12.5). In

the sequential aspects of the task, the relatively few strong overgenerals prevent it from learning an effectively policy, meaning it must rely on occasional random actions to break it out of loops. In the unbiased 6 multiplexer, SB–XCS is able to adapt well but is outperformed by XCS (§2.5.1), and should be outperformed by supervised learners.

This analysis would seem to leave SB–XCS with little or no niche. (The addition of fitness sharing, however, might change this.)

5.14 Extensions

This section briefly considers some extensions to the work presented in this chapter.

5.14.1 Fitness Sharing

This work has considered XCS, which we claim avoids strong and fit overgenerals because its accuracy-based fitness penalises overgeneral rules (§5.7). This work has also considered SB–XCS, which we claim cannot adapt to tasks with biased reward functions, because it suffers from strong and fit overgenerals. This work has not, however, considered the addition of fitness sharing to SB–XCS. Fitness sharing is known to counter the propagation of overgeneral rules [249, 134, 47], at least in some cases. The addition of fitness sharing to SB–XCS, and its use in other systems, may allow successful adaptation to tasks with biased reward functions, although this has yet to be demonstrated conclusively. Clearly, this is an important direction for future work, and the analysis of rule types in this chapter and of representations in chapter 3 are two possible starting points for such work.

Although such extensions are beyond the scope of the current work, the following line of reasoning suggests some hypotheses:

- It has been shown that SB–XCS can adapt to a task with an unbiased reward function (the 6 multiplexer – §2.5.1) despite its lack of fitness sharing.
- This suggest fitness sharing is not needed for such tasks.
- That SB–XCS found an optimal representation for the 6 multiplexer task (§3.6, §3.6.4) suggests that fitness sharing will not assist SB–XCS in finding good representations.

- Although fitness sharing has been shown to help strength-based ZCS avoid strong overgeneral rules [47], the rate of adaptation is slow, compared to the rate of adaptation of SB–XCS and XCS to the 6 multiplexer (§2.5.1).
- Consequently, fitness sharing seems unlikely to speed the adaptation of strength-based systems on tasks with unbiased reward functions.
- Because fitness sharing takes time to counter overgeneral rules [47], the more biased the reward function, the longer it should take a strength-based system to adapt.
- Because XCS penalises overgeneral rules more directly than fitness sharing, its difficulty in adapting to biased reward functions should not be as great.
- Although the comparison was limited, XCS was able to adapt more quickly to the 6 multiplexer than SB–XCS (§2.5.1, §3.6.4).
- Consequently, there is reason to believe that strength-based systems employing fitness sharing may adapt more slowly than XCS, particularly in tasks with biased reward functions.

These hypotheses are highly speculative and certainly need evaluation.

5.14.2 Other Factors Contributing to Strong Overgenerals

This chapter has emphasised the role of the reward and value functions, and of fitness calculation, in the formation of strong and fit overgenerals. Clearly these are major factors, but there are others. Unfortunately, the analysis in this chapter is a gross oversimplification of more realistic learning tasks, in which it can be very difficult to determine how much of a problem strong and fit overgenerals are likely to be. Some additional factors are:

the classifiers - they often apply in many states, not only two which in isolation make strong or fit overgenerals possible;

the explore/exploit policy - the strategy adopted affects how often classifiers are updated towards their different rewards;

the frequency with which given states are seen - in the non-sequential case this depends on the training scheme, and on the learner and the task itself in the sequential case;

the selection mechanisms - how high selective pressure is in reproduction and deletion;

the fitness landscape - to what extent strong and fit overgenerals compete with stronger and fitter correct rules.

As a simple example of these factors, an overgeneral might act correctly in 10 states with reward c and incorrectly in only 1 with reward i. Using the strength-based strong overgeneral inequality from page 162, its expected strength would be $(10c + i)/11$, and it would be a strong overgeneral if this value exceeded the strength of some accurate competitor. Similarly, the overgeneral might match in 10 states with reward i and only 1 with reward c.

Although the complexity of the issue makes a more complete analysis difficult it should be clear that the nature of the reward and value functions is an important factor in determining the prevalence of strong and fit overgenerals, and that they are not uncommon.

In the mainstream RL literature strength-like values are often stored using look-up tables with an entry for each state-action pair. Such tabular systems are relatively insensitive to the form of the reward and value functions, which may account for the lack of attention this subject has received in the mainstream RL literature. SB–XCS, however, is clearly sensitive to the form of the reward and value functions. Other strength-based LCS, even with fitness sharing, must still be influenced by the form of the reward and value functions. That is, even if fitness sharing is able to completely overcome strong and fit overgenerals, and allow strength-based LCS to adapt regardless of the form of the value function, complex value functions are still likely to be more difficult for strength-based LCS and require greater effort to learn. Fitness sharing may overcome strong overgenerals, but with some effort. This constitutes an important difference between strength-based LCS and tabular reinforcement learners. It is curious that the form of these functions has not received more attention in the LCS literature given their sensitivity to them.

5.14.3 Qualitative and Quantitative Approaches

We could extend the approach taken in this work by removing some of the simplifying assumptions made in §5.2 and dealing with the resultant additional complexity, and by including the factors in §5.14.2. For example, we could put aside the assumption of equiprobable states and actions, and extend the inequalities showing the requirements of the reward function for the

emergence of strong overgenerals to include the frequencies with which states and actions occur. Taken far enough such extensions might allow quantitative analysis of non-trivial tasks. Unfortunately, while some extensions would be fairly simple, others would be rather more difficult.

At the same time, the most significant results from this approach may be qualitative, and some such results have already been obtained: we have refined the concept of overgenerality and argued that strength and accuracy-based LCS have different goals (§5.3.2), and introduced the concepts of fit overgeneral rules (§5.3.4), and strong and fit undergeneral rules (§5.9).

We've seen that, qualitatively, strong and fit overgenerals depend essentially on biases in the reward or value function, and that they are very common. We've also seen that the newer XCS has, so far, dealt with them much better than SB–XCS (although we have not considered fitness sharing or default hierarchies). This is in keeping with the analysis in §5.3.2 which suggests that using strength as fitness results in a mismatch between the goals of the LCS and its GA.

Applying our qualitative analysis to Woods2, we predicted that its biased reward and value functions would produce strong and fit overgenerals. Empirical results showed that this is indeed the case and that they are responsible for SB–XCS's poor performance there (§5.12.5).

In addition to these qualitative and empirical results, some interesting quantitative results have been obtained, despite our simplifications. We've seen that unbiased reward and value functions will not support strong overgenerals (sections §1 and §5.6.1), and we've seen the conditions under which a value function will be unbiased (§5.12.6).

Rather than pursue further quantitative results it would be preferable to extend the qualitative approach used here to consider the effects of default hierarchies and mechanisms to promote them, and fitness sharing. Further study of persistent strong and fit overgenerals in XCS is of interest, as are are hybrid strength/accuracy-based fitness schemes, as opposed to the purely strength-based fitness of SB–XCS and purely accuracy-based fitness of XCS.

5.15 Chapter Summary

We've analysed and extended the concept of overgeneral rules under different fitness schemes. Dealing with such rules is a major issue for Michigan-style evolutionary rule-based systems in general, not just for the two classifier systems we have considered here. For example, the use of alternative representations (e.g., fuzzy classifiers), rule discovery systems (e.g., evolution strategies) or the addition of internal memory should not alter the funda-

mental types of rules which are possible. In all these cases, the system would still be confronted with the problems of greedy classifier creation, overgeneral, strong overgeneral, and fit overgeneral rules. Only by modifying the way in which rule fitness is calculated, or by restricting ourselves to benign reward functions, can we influence which types of rules are possible.

Although we haven't described it as such, this work has examined the fitness landscapes defined by the reward function, γ, the structure of the task, the representation of rules and the fitness scheme used. We have examined the idea that we can try to avoid pathological fitness landscapes by using benign reward functions, but found this limiting, particularly in sequential tasks. Consequently, to avoid pathological fitness landscapes our best bet is to select the most suitable fitness scheme. That is, we should alter the classifier system to suit the task, rather than vice versa.

Chapter 6

Classifier Systems and Q-Learning

The LCS is usually described as a method*: a set of algorithmic details that define a way to solve a class of problems. However, in many ways the LCS is more of an* approach*: a set of conceptual details that define a certain direction for developing methods. Therefore, the defining issues for the LCS are not necessarily algorithmic, but conceptual.*

Robert E. Smith paraphrasing Lashon B. Booker [242] p. 2.

Chapter Outline

The previous chapter investigated the ontology of rules in great detail. This chapter, the last to present original results, takes a step back to look broadly at how XCS and SB–XCS combine Genetic Algorithms and Q-learning, and returns to the question of the GA and RL-views of classifier systems posed in the introduction.

In the introductory chapter two alternative views of classifier systems were suggested: the GA-view and the RL-view. Throughout the thesis, we've compared two algorithmically similar, but, it has been argued, conceptually very different LCS. In this chapter we return to the issue of the two views, which we address by considering how the two systems address their twin goals of policy learning and generalisation.

To begin with, however, §6.1 considers the relationship between Q-learning and classifier systems, and between the GA and Q-learning in classifier systems. It is argued that XCS is a proper generalisation of tabular Q-learning and that SB–XCS is not.

6.1 Classifier Systems and Q-Learning

SB–XCS was designed as the strength-based twin of XCS, and from its specification in §2.4 we can see that they are very similar algorithmically. Of particular interest in this section is the fact that both make use of two learning systems, one evolutionary (the genetic algorithm) and the other based on dynamic programming (Q-learning). Virtually all LCS are hybrid evolutionary/reinforcement learning systems of some kind.

Despite this kinship, this section makes the case that XCS and SB–XCS are fundamentally very different systems, and that they learn policies in very different ways. It is claimed that XCS is in fact a type of Q-learning system, and that the means by which it learns policies has more in common with such systems than with that of its twin SB–XCS.

6.1.1 Q-Learning in Classifier Systems

In contrast to most LCS, XCS and SB–XCS use a version of the Q-learning update, and their strengths are thus Q-values: predictions of the discounted return to be received for taking the advocated action and following the optimal policy thereafter. (This is why Wilson refers to strength as prediction in XCS.)

Predicting Return with Multiple Classifiers

While tabular Q-learning updates a single state-action's estimate, XCS and SB–XCS update a number of estimates (each classifier is an estimate) and each estimate may apply to multiple state-action pairs.

XCS and SB–XCS differ slightly from tabular Q-learning when applied to sequential tasks. In tabular Q-learning the value towards which the estimate is updated is obtained from a single estimate for a single state-action, while in XCS and SB–XCS the estimate is updated towards P, which, in the sequential case (equation (2.3)), is an average of the estimates of many classifiers. (In the non-sequential case (equation (2.2)), there is no difference in P for the tabular and LCS versions.)

Fitness Sharing and Q-learning

Wilson points out that XCS's approach is closer to tabular Q-learning than that found in other Q-learning-like LCS which share strength between rules:

> "Several articles (e.g., [217, 279, 74, 297]) have drawn attention to the relationship between the Q-learning update procedure and various versions of the bucket-brigade algorithm, especially a version in which the payoff value is, as in Q-learning, a discounted sum of immediate reward and the sum of strengths of the maximum strength action in the next match set [297]. The major difference is precisely that it is this sum of strengths that represents the [Q-value], not a single number as in Q-learning. That is, assuming sharing of strength as discussed in Section 2, the system's [Q-value] information is distributed over sets of classifiers, sets that are subject to abrupt membership changes due to the GA. In XCS, however, the relation to Q-learning is closer and more stable because each classifier uses Q-learning to predict the payoff directly, independent of the other classifiers, and the system prediction is an average instead of a sum." [298] p. 21–22.

Wilson also explains that fitness sharing makes the meaning of a rule's strength unclear:

> "Sharing eliminates takeover effects but then a classifier's strength no longer directly predicts payoff [i.e., reward]; instead, the total of the shared strength (among matching classifiers advocating the same action) predicts the payoff. This division of the prediction becomes problematic since a given classifier, with its single strength value, is often involved in numerous distinct matching sets, so that the meaning of the strength value becomes unclear." [298] p. 3–4.

However, there is a straightforward solution to this dilemma, which is to maintain two strength values for each rule; a shared strength and an unshared strength. The first can be used for reproduction and the second for predicting return, just as strength and fitness are used separately in XCS.

6.1.2 Is it Really Q-Learning?

While the update used by XCS and SB–XCS more closely resembles tabular Q-learning than do those of other LCS, XCS's update is not identical to that of tabular Q-learning because the LCS version involves multiple estimators.

Nonetheless, two points support the idea that we should consider XCS a type of Q-learner. First, XCS converges to essentially the same value function that a tabular Q-learner would, as shown in Woods2 in [298], in which high fitness classifiers converge to the same Q-values a tabular Q-learner would. Second, tabular Q-learning is not the only form of Q-learning: other Q-learning systems use some generalisation mechanism to generalise over the value function: e.g., Watkins [286] and Sutton [264] used CMACs, and Lin a neural network [173]. The author's position is that XCS is another form of Q-learner, one which uses a GA as generalisation mechanism.

Why SB–XCS is not so close to Tabular Q-learning

One feature of tabular Q-learning is that it uses a complete map. XCS evolves such maps, but, as discussed in §3.6.2, SB–XCS does not in general do so. This is an important difference, since the lack of a complete map is likely to interfere with the Q-learning process (§3.7.2). Consequently, SB–XCS differs significantly from tabular Q-learning in this respect.

6.1.3 XCS is a Proper Generalisation of Tabular Q-Learning

The nature of XCS as a Q-learner is well illustrated by considering what little modification is required to turn it into a tabular Q-learner. All that is necessary is to disable the rule discovery component and instead insert a list of all syntactically possible classifiers which do not contain hashes (#). This list is a Q-table and each classifier is an entry in the table. In this case XCS *is* a tabular Q-learner. Without these modifications XCS is a Q-learner which uses a function approximation GA [148] to generalise over equivalent states in the Q-table. In contrast, SB–XCS is *not* a proper generalisation of tabular Q-learning, since its partial maps do not represent complete value functions.[1]

Implementation Notes

The simplest way to obtain a tabular Q-learner from an implementation of XCS is to set θ_{GA} to a very large number, so that the GA is never invoked, and set $P_{\#} = 0$, so that covering generates fully specific rules. In order for good performance to be achieved, it is necessary to use action set covering, rather

[1] That a classifier system could be a generalisation of tabular Q-learning was first shown for a highly simplified LCS by Dorigo and Bersini [74].

than Wilson's original method (§2.3.6). This is how the tabular Q-learner used in §2.5.2 and §5.12.5 was implemented.

The approach of using covering to generate rules as needed has the advantage that the Q-table contains only as many state-actions as needed, since covering only occurs in states actually encountered; in some tasks there are far more syntactically possible states than are ever encountered. For example, Woods2 has only 556 states, and yet the coding of the environment can express 2^{24} syntactically possible states. A disadvantage is that inserting a complete Q-table at the outset would allow random access to its entries and save run-time.

6.1.4 Summary

In summary, while most reinforcement learning LCS are hybrid evolutionary/reinforcement learning systems, and a number of LCS use the Q-learning update, XCS is closest to Q-learning as it, like tabular Q-learning, uses complete maps. In fact, XCS is a proper generalisation of tabular Q-learning, since it reduces to it when its generalisation mechanism is disabled. This conceptual link is significant as it more firmly unites two somewhat separate research traditions (the LCS community and the RL community) and clarifies the comparison between different types of learning systems.

6.2 The GA-View and RL-View Revisited

Section §1.6.2 outlined two views of classifier systems: the GA-view – that the LCS is essentially the application of a GA to a problem – and the RL-view – that the LCS is some kind of RL system, that is, something like Q-learning, perhaps with some kind of evolutionary component.

Which view is correct? Or, more likely, which view is most appropriate? Perhaps it depends on the LCS in question. Are the two view compatible? In the following sections some answers are suggested.

The Policy's the Thing

Classifier systems for reinforcement learning have the same goal as any RL system: to maximise return. To do so, an RL agent must find an optimal policy (§F.2). However, an important feature of classifier systems – many would say the whole point of using them, rather than, say, tabular Q-learning – is their in-built capacity to exploit environmental regularities. We could say the primary tasks an LCS faces are finding good policies and finding useful generalisations while doing so. Consequently, we can address the issue of the

GA- and RL-views by considering how an LCS finds (selects, determines) policies, and how it finds generalisations.

In fact, we will overlook the issue of finding generalisations. They are expressed by #s in rule conditions, and rule conditions are the concern of the rule discovery system. In both SB–XCS and XCS the main component of the rule discovery system is the GA; in other words, in both systems it is up to the GA to find useful generalisations. Of course, for the GA to do so the credit assignment system must provide it with useful information; it must give useful generalisations higher fitness than less useful or less general conditions. But let's ignore generalisation and focus on the learning of policies. Two broad approaches are presented in the appendices: evolution and generalised policy iteration. Which approach does a (Michigan) classifier system use? Let's consider SB–XCS and XCS in turn.

6.2.1 How SB–XCS Determines Policies

We know that, to a first approximation, SB–XCS's rule population determines its policy, and that SB–XCS's GA evolves its rules, so we could say that SB–XCS's GA determines its policy. But this is a superficial analysis, and in fact we could apply the same reasoning to XCS. Let's consider SB–XCS's operation in more detail.

A policy is a mapping of states to actions. A rule population may not specify a unique policy, since often conflicting rules occur, and some states may have no matching rules. Given this, how can we tell what an LCS's policy is?

Deriving a Policy from a Value Function: Tabular Q-Learning

Let's consider a tabular Q-learning system. It maintains a value function, and just as a set of rules is not a policy, neither is a value function. (A set of rules can represent a value function, but it can also represent conflicting actions for the same state; it is more general than a policy.) From its value function, a tabular Q-learner derives a policy *as needed*. That is, each time the system must select an action it consults its value function for the current state, and applies some action selection method (e.g., ϵ-greedy selection). Since the value function is constantly being updated, so is the policy. The point here is that there is no explicitly represented policy; it is generated as needed. (Some RL systems do, however, explicitly represent policies, e.g., actor-critic methods [265].)

Deriving a Policy from a Value Function: SB–XCS

An LCS selects an action based on the currently matching rules, taking the need to explore into account, in the same way a tabular Q-learning system does based on its value function. From this description an LCS operates just like a tabular Q-learner. But in some cases there's a difference. Recall that SB–XCS maintains partial map representations (§3.6.2), and that consequently sometimes (in fact, usually) some of the actions available in a state have no value estimate. SB–XCS (following XCS) cannot select actions whose values are not estimated (i.e., which are not advocated by a matching rule).

In contrast, in a tabular Q-learning system, all actions are represented and can be selected. Is this significant? Yes, because the presence or absence of rules is determined by the GA. By determining which actions are represented, the GA influences action selection.

6.2.2 How XCS Determines Policies

XCS and SB–XCS can be configured to select actions using the same method (e.g., ϵ-greedy selection), and neither can select actions which no matching rule advocates. However, XCS's strong tendency towards complete maps means that typically all actions are represented by some rule. The significance is that in XCS the GA *does not* influence action selection by preventing some actions from being selected. In XCS, all actions are eligible for selection, just as in tabular Q-learning.

This point is somewhat subtle, since the same action selection method can be used with both systems. The behaviour of the chosen method differs in the two systems because of the form of representation upon which it operates (partial versus complete maps). In complex systems like LCS, this sort of interaction between components can easily occur by chance, although in this case XCS's designer, Stewart Wilson, intended action selection to operate on complete maps ([298] p. 5). What Wilson may not have intended was the resulting decoupling of the GA and action selection, i.e., of the GA and policy determination.

That XCS's GA is detached from action selection seems highly significant to the issue of how it determines policies. It is also relevant to the issue of the GA and RL-views, as we'll soon see in §6.2.4.

6.2.3 Three Approaches to Determining a Policy

We can distinguish three ways to determine a policy. The first is to evolve and evaluate complete policies, in which case policies are completely determined by the evolutionary algorithm. This is the approach used in Pittsburgh LCS (§2.1.1). A second approach is to derive a policy from a (complete) value function, as we do in tabular Q-learning. This is the approach used with XCS, where the value function is stored by the rule population.

Finally, the third approach, that used by SB–XCS, lies somewhere between the first two. In SB–XCS, the policy is derived from the incomplete representation of the value function SB–XCS's partial map provides. Significantly, the policies which can be derived depend on which actions are advocated, which in turn depends on which rules the GA maintains in the population. Thus, the policy is determined by a combination of Q-learning and evolution (and any additional rule discovery mechanisms, such as covering).[2]

6.2.4 The GA-View and the RL-View

In this section we relate the two learning subsystems to the two primary tasks of the LCS, consider the emphasis to be placed on the role of the GA, and finally propose an answer to the question of whether LCS are GA-based or RL-based.

Which Subsystem Does What?

Although all classifier systems contain a rule discovery and credit assignment system, we can now see that different LCS operate on very different principles. Although XCS and SB–XCS are very similar algorithmically, XCS relies entirely on Q-learning to obtain policies, while SB–XCS combines Q-learning with evolution. Figure 6.1 summarises the differences between XCS, SB–XCS, and the Pittsburgh approach.

The Importance of the GA

The two approaches place different emphasis on the role of the GA; in SB–XCS the GA is more significant as it directly affects the policy. This emphasis on the role of the GA is part of the original rationale for the LCS.

In case there is any doubt concerning the emphasis traditionally put on the role of the GA, let's refer to Holland and Reitman's discussion of the two

[2] The fighter aircraft LCS [246] is an interesting case in which a default policy is applied in the absence of any matching rule.

	XCS	SB–XCS	Pitt LCS
Policy	QL	QL & GA	GA
Generalisation	GA	GA	GA

Figure 6.1: The method (Genetic Algorithm or Q-Learning) employed by each system for finding policies and generalisation.

learning subsystems in the first classifier system, CS-1 [127]. It is clear that they consider the GA the primary learning system and the credit assignment system decidedly secondary:

> "The second process [credit assignment] is a form of simple learning; after a series of actions, it stores in memory information about the consequences of these actions. The third process [the GA] is a more complex learning process ... the novelty of the model [the LCS] is not so much in the performance or simple learning processes [credit assignment], but rather in the process that changes memory [the GA]." [127] p. 470.

Although the quote above is from 1978, this view is not confined to early work on classifier system, as the following quote from 1998 suggests:

> "The learning classifier system (LCS) is an application of the genetic algorithm (GA) to machine learning." [62] p. 299.

In case there is any doubt concerning the emphasis to be placed on the role of Q-learning in XCS, we refer the reader to the exploration of the connection between XCS and tabular Q-learning in §6.1.

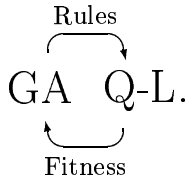
Which View for Which System?

The RL-view seems entirely appropriate for XCS, since it derives its policy solely from a value function, just as tabular Q-learning does. In contrast, the GA-view seems entirely appropriate for Pittsburgh LCS, which rely on the GA for both policy learning and generalisation. Somewhere between the two extremes lies SB–XCS, in which the GA and Q-learning both contribute to determining the policy.

6.2.5 Combining Evolution and Q-learning

XCS and SB–XCS represent different ways of combining evolutionary search and Q-learning in a Michigan classifier system. The following figure shows

the general approach both follow, in which the GA produces rules which Q-learning uses and whose fitness it evaluates. (Of course, XCS uses not only Q-learning, but the extra updates for accuracy-based rule fitness – §2.3.5.)



As the figure shows, the GA and Q-learning interact, each feeding into the other. Hence, policy learning and generalisation interact, at least indirectly, even in XCS, since one feeds into the other. Nonetheless, the extent of GA influence on policy learning is much reduced in XCS compared to SB–XCS.

We can refine this picture of XCS and SB–XCS's operation to take into account the role of the GA in each. First, however, let's consider different ways of looking at rules in the two systems.

What are Classifiers?

Classifiers are mappings of states to actions, with some associated parameters such as strength and fitness. Throughout this work we've assumed XCS and SB–XCS use the same standard ternary language to represent their rules. However, because the two systems function differently, we can look at their rules from different perspectives. We consider two sources of evidence: rule structure and the role of the GA in action selection.

Rule Structure In SB–XCS, a rule is a mapping from states to an action, and of course each such rule has an associated fitness. A mapping from each state to an action is a policy, so SB–XCS's rules are policy fragments, and SB–XCS generalises over its policy.

In XCS, in contrast, a rule is a mapping from states to an action to a prediction, and of course has an associated fitness. A mapping from states to actions to predictions is a value function, so XCS's rules are value function fragments, and XCS generalises over its value function.

The GA and Action Selection Because the GA in SB–XCS influences action selection, it again seems reasonable to think of rules as fragments of a policy. We can think of the GA generating plausible policy fragments which the production system can apply, and which the credit assignment

system evaluates. The GA searches for high-valued policy fragments, i.e., rules which, together, form a good policy and result in high return.

In XCS, however, the GA does not influence the policy as it does in SB–XCS, so again it seems reasonable to think of XCS's rules as value function fragments rather than policy fragments. The credit assignment system assigns a prediction to the value function fragments, as some average of the Q-values of their constituent state-actions, and action selection occurs based on these predictions. Credit assignment also assigns a fitness to the fragments, based inversely on the error in the predictions. The GA searches for useful ways of fragmenting the value function, rather than fragments of a high-valued policy.

In summary, rules can be thought of as aggregations of state-actions which are:

1. Policy fragments (in SB–XCS).

2. Value function fragments (in XCS).

Summary of Operation of XCS and SB–XCS

Now that we've seen two ways of looking at rules, we can summarise the operation of XCS and SB–XCS as follows:

How SB–XCS Operates:

- Generates policy fragments with the GA.
- Performs Q-learning on policy fragments.
- Policy fragment's fitness = its Q-value.

How XCS Operates:

- Generates value function fragments with the GA.
- Performs Q-learning on value function fragments.
- Fragment's fitness = value negatively correlated with error in its Q-update.

6.3 XCS is Closer to Tabular Q-Learning than to SB–XCS

The analysis presented in this chapter suggests XCS has more in common with tabular Q-learning than it does with SB–XCS and other strength-based LCS. That is, despite its origins and algorithmic similarity to SB–XCS, the issues faced by XCS, and its means of finding policies, are closer to those of tabular Q-learning than to SB–XCS or other strength-based (or Pittsburgh-style) classifier systems.

6.4 Summary

This chapter has investigated unexpected consequences of the different representations towards which XCS and SB–XCS tend as a result of their different fitness calculations.

Section §6.1.3 argued that XCS is a proper generalisation of tabular Q-learning, and showed that it reduces to tabular Q-learning when generalisation is disabled. That section also argued that SB–XCS is not a proper generalisation of tabular Q-learning because its partial maps do not represent complete value functions. Section §6.2.1 explained that partial maps were again significant because they allow SB–XCS's GA to influence action selection and thereby restrict the policies Q-learning can follow. XCS's complete maps make no such restrictions. This suggests the RL-view of classifier systems suits XCS, while the GA-view is more suitable for SB–XCS. All this suggests policy learning in XCS has more in common with tabular Q-learning than with SB–XCS.

Chapter 7

Conclusion

This concluding chapter first outlines the conclusions reached regarding the capacities of different types of classifier systems (§7.1). Next, it lists the contributions of the thesis in order to make them explicit (§7.2). Following this, it emphasises the main points of the thesis (§7.3), discusses extensions of this work (§7.4), and, finally presents some concluding remarks (§7.5).

7.1 The Capacities of Various Types of LCS

As mentioned in §2.1.1, Wilcox [291] suggests that although Michigan LCS are faster than Pittsburgh LCS, they are limited to simpler tasks. Wilcox, writing in 1995, did not include XCS in his assessment of Michigan LCS, as XCS had only just appeared. This section proposes an extension to Wilcox's analysis to include XCS and SB–XCS Michigan LCS:

Pittsburgh LCS

- Can solve complex problems [291].
- Are slow (since each chromosome is a set of rules) [291].

Strength-based SB–XCS

- Cannot solve complex problems (§2.5.2, and chapter 5).
- Is fast (since each chromosome is an individual rule) [291].

Other Strength-based Michigan LCS

- May have difficulties with complex problems (e.g., [291, 55], and chapter 5), which may either prevent them from adapting, or slow adaptation. More study is needed.
- Are fast (since each chromosome is an individual rule) [291].

Accuracy-based Michigan LCS (i.e., XCS)

- Can solve complex problems (since they do not appear to suffer from strong and fit overgenerals – chapter 5).
- Are fast (since each chromosome is an individual rule).

We can summarise these characteristics as follows, where N/A stands for 'Not Applicable', and '?' indicates further study is needed before a prediction can be made.

	XCS	SB–XCS	Strength	Pitt
Simple	Fast	Fast	Fast	Slow
Complex	Fast	N/A	?	Slow

7.2 Contributions

The contributions listed in this section are organised by chapter.

Introduction

- distinguished the GA and RL-views of classifier systems (§1.6.2).

Learning Classifier Systems

- provided the first analysis of the advantages and disadvantages of Wilson's explore/exploit framework for testing learning systems (§2.3.3).
- provided the most comprehensive description and discussion of XCS to date (§2.3), including previously unpublished material on covering (p. 46), mutation (p. 48), and fatigue (p. 47).
- clarified the details of XCS by providing, in some cases for the first time, updates in formal notation, and the algorithms in figures 2.1, 2.3 and 2.5.

- introduced a mutation operator (p. 48), a covering operator (p. 46), and a deletion scheme (p. 50) which appear to be superior to those used by Wilson's original implementation of XCS.

- showed that the delta rule subsumes the MAM technique (§2.3.5, p. 39).

- introduced the strength-based classifier system SB–XCS as a tool for comparing strength and accuracy-based fitness (§2.4).

- made an initial comparison of XCS and SB–XCS (§2.5), which showed that they perform similarly on the 6 multiplexer, but that XCS does much better than SB–XCS on Woods2. It was also shown that XCS's ability to evolve small rule populations depends on the use of subsumption deletion rather than on its accuracy-based fitness.

How Strength and Accuracy Differ

- extended Wilson's XCS generalization hypothesis (§3.3.2) to show that it applies more broadly than a strict interpretation would allow.

- introduced a schema theorem for the niche GA (§3.3.2).

- introduced a rationale for adaptation by SB–XCS (§3.4).

- provided empirical evidence that XCS and SB–XCS reliably evolve arguably optimal solutions to the 6 multiplexer (§3.6.4).

- compared partial and complete maps, noting the advantage (potentially smaller population size) and disadvantages (interference with credit assignment and exploration control) of partial maps compared to complete maps (§3.7).

- suggested that partial maps are more suitable for Pittsburgh LCS. With Michigan LCS, partial maps may be more suitable for non-sequential tasks with large action sets, and complete maps may be more suitable for sequential tasks (§3.7.5).

- showed that for XCS to generalise well the accuracy criterion must be set appropriately (§3.8).

- showed that XCS cannot always generalise as well as SB–XCS, no matter how the global accuracy criterion is set, and suggested the need for adaptive local accuracy criteria (§3.8.2).

What Should a Classifier System Learn?

- identified and formalised four properties which may be desirable of a set of rules found by an LCS; that they be complete, accurate, minimal and non-overlapping (§4.2).
- showed that pressure towards completeness produces pressure against overlaps as a perhaps undesirable byproduct (§4.2.5, p. 118).
- distinguished performance and population state metrics and demonstrated advantages of the latter (§4.3).
- introduced %[m-DNF] and %[PI], two new population state metrics (§4.3.4).
- uncovered the first evidence of XCS's very strong bias against overlapping rules (§4.3.5).

Prospects for Adaptation

- formalised the notions of correct and incorrect actions, correct and incorrect rules, overgeneral rules, fit overgeneral rules, and strong overgeneral rules (§5.3).
- defined the minimal conditions under which strong and fit overgenerals can occur (§5.4).
- explained the dependence of strong and fit overgeneral rules on the reward/value function and on competition for action selection (§5.4.2) or reproduction (§5.4.3).
- introduced the notion of bias in reward/value functions, and of biased and unbiased functions (§5.4.1), which are useful in understanding overgeneral, strong and fit overgeneral rules.
- showed that, using the standard ternary LCS language, biases in the reward/value function between actions do not result in strong overgenerals (§5.5.1).
- showed that although in XCS all overgenerals are strong overgenerals (§5.5.2), they have low accuracy and hence low fitness (§5.7).
- showed that strong overgenerals are impossible in SB–XCS when the reward function is unbiased (§5.6.1), and possible when it is sufficiently biased (§5.6.2).

- introduced the concept of fit overgeneral rules for XCS, showed that they depend on bias in the variance of the reward function, demonstrated a simple task which produces them, and related this work to earlier work by Lanzi and Colombetti on XCS in stochastic environments (§5.8). Also showed that SB–XCS does not suffer from biases in the variance function.

- introduced an inequality which determines when, under highly simplified conditions, a rule will be a strong overgeneral (equation (5.6), p. 180).

- introduced the concept of strong and fit *under*general rules and showed that both XCS and SB–XCS need a bias towards generality in order to evolve general rules (§5.9).

- suggested that we can bias the reward function in order to bias rule allocation in strength-based classifier systems (§5.10).

- suggested that it can be important to learn what *not* to do, and that the value of acting correctly in a state is not the same as its expected return (§5.11).

- suggested three alternative ways of biasing rule allocation in classifier systems (§5.11.4).

- showed that non-trivial sequential tasks have biased value functions, thanks to the passing back of value and discounting (§5.12), meaning that they are likely to produce strong overgenerals in SB–XCS.

- showed that very short sequences of actions, and any discount rate, can produce sufficient biases in the value function to produce strong overgenerals (§5.12.4).

- showed theoretically why SB–XCS should suffer from strong and fit overgeneral rules in Woods2 (the reward and value functions are biased) and showed empirically that SB–XCS does indeed suffer from them (§5.12.5).

- showed that a $\gamma = 0$ Q-learner, an agent incapable of learning sequences of actions, is able to perform reasonably well on the sequential Woods2 task, and that the relatively good performance of SB–XCS (and probably that of ZCS on Woods1 [297]) is largely due to non-sequential learning (§5.12.5).

- showed the limited conditions under which a task's value function will be unbiased, and hence in which SB–XCS can be expected to adapt (§5.12.6).

Classifier Systems and Q-Learning

- showed that XCS is a proper generalisation of tabular Q-learning while SB–XCS is not (§6.1.3).
- showed that the GA in XCS is almost entirely disconnected from policy determination, unlike in SB–XCS and other LCS (§6.2).
- related XCS, SB–XCS and Pittsburgh LCS to the GA and RL-views of classifier systems (§6.2.4).
- showed the role of the GA and of Q-learning in addressing the problems of finding a policy and generalisation in XCS, SB–XCS and Pittsburgh LCS (§6.2.4).
- suggested policy learning in XCS has more in common with tabular Q-learning than with SB–XCS and other strength-based LCS (§6.3).

Further Contributions

- Section §7.5 will highlight the need for a complex systems design methodology and suggest much progress with LCS has been made thanks to work on minimalist systems.
- Appendix A compares XCS with and without macroclassifiers and finds they have little effect on learning rate, but reduce run-time.
- Appendix B walks through an XCS cycle in unprecedented detail, in order to clarify its operation.
- Appendix H will show that Sarsa is due to Wilson as much as to Rummery and Niranjan.

7.3 The Take-home Message

The most significant points of this work are summarised as follows.

- XCS and SB–XCS operate on very different principles, despite their algorithmic similarity (§6.2):

  - XCS performs Q-learning on value function fragments.
  - SB–XCS performs Q-learning on policy fragments.

- XCS is a proper generalisation of tabular Q-learning, and as much a Q-learning system as neural network-based Q-learners (§6.1).
- Policy learning in XCS has more in common with tabular Q-learning than with SB–XCS and other strength-based classifier systems (§6.3).
- The definition of fitness in strength-based systems is fundamentally at odds with the behaviour we would like the system to show (page 143).
- SB–XCS suffers from strong and fit overgeneral rules in tasks with biased reward/value functions and this problem is enough to prevent it from adapting (chapter 5).

  - Consequently, SB–XCS can only be expected to adapt to a limited class of tasks, essentially those with unbiased value functions. Unbiased non-sequential tasks are probably often better modelled as supervised learning (§5.13), and unbiased sequential tasks are effectively non-sequential (§5.12.6). Consequently, SB–XCS appears to have no niche. However, the addition of fitness sharing to SB–XCS might allow it to overcome such problems.

- Fitness sharing should help combat strong and fit overgenerals (see [47]). However:

  - Combating strong overgenerals with fitness sharing, even if completely successful, is hypothesised to require some effort on the part of the LCS (§5.14.1). Thus, it is predicted that strength-based LCS will be slower to adapt to tasks which can produce strong and fit overgenerals than to tasks which cannot.
  - XCS, with its accuracy-based fitness, directly penalises strong and fit overgenerals (§5.7), and so should not need fitness sharing.
  - This, and preliminary comparisons on an unbiased task (§2.5.1), suggest XCS may adapt more quickly to tasks with biased reward functions than strength-based systems. However, this hypothesis remains to be tested.

- XCS suffers from its own form of fit overgenerals when the bias in the variance in the reward function is sufficient (§5.8).

- Neither system suffers from fit overgenerals in the conditions which cause them in the other system (§5.8.4). Further study is needed to determine which problem is more serious.
- Although the complete maps of XCS appear to offer many advantages over less complete maps (§3.7), they may also make certain tasks more difficult for XCS to learn (§3.8.3).
- XCS's accuracy-based fitness restricts the generalisations it can express, compared to SB–XCS (§3.8).

7.4 Open Problems and Future Work

The comparison of XCS and SB–XCS presented here is by no means exhaustive. Additionally, only two specific systems have been explicitly compared, while a great range of strength-based systems appear in the literature, and a range of XCS-like systems are conceivable. Analysis of specific systems other than XCS and SB–XCS must be left for future work.

As noted in §2.1, our limited understanding of the relationships between different LCS makes classification of them problematic. Extension of the analysis presented in this work to other systems should help clarify this issue. In particular, it would be of interest to extend the analysis of chapters 3 and 5 to less SB–XCS-like strength-based systems, hybrid strength-accuracy systems, Frey and Slate's accuracy-based LCS [91], Booker's endogenous-fitness LCS [36, 35, 37], and the fitness scheme used with CS-1 [127, 100].

7.4.1 Fitness Sharing and Strength-based Fitness

This work has attempted to establish that SB–XCS is unsuitable for most applications due to problems with strong and fit overgenerals. SB–XCS, however, does not incorporate fitness sharing, which counters problems with strong and fit overgenerals and should make SB–XCS more useful. An important open question is the degree to which fitness sharing can help with strong and fit overgenerals.

Although the analysis of SB–XCS in this work has not addressed this issue, it has, however, improved the foundations for such study. The lack of fitness sharing in SB–XCS makes it a more extreme (and limited) system, but also simplifies it. This simplification has allowed us to analyse SB–XCS extensively, to obtain rationales for its adaptation (§3.4), and failure to adapt (chapter 5), to characterise the partial map representations it prefers (§3.6.2), and to produce a taxonomy of strength-based rule types (§5.3). It

is doubtful that such progress would have been made if the complication of fitness sharing had been considered from the beginning. Now that we have obtained this foundation, however, we can further our understanding of the effect of fitness sharing by adding it to SB–XCS and analysing its effect on the representations the modified system produces, and how the various rule types are affected.

7.4.2 Further Study of Accuracy-based Fitness

Further study of accuracy is still greatly needed. We have some theory to explain the operation of XCS (chapter 3), but more is needed, and it must be integrated with existing work in reinforcement and machine learning. A particular area which must be addressed is the question of to what class of problems XCS is well suited. This work has shown serious limitations to the applicability of SB–XCS, but has shown few limitations for XCS, other than the preliminary work on variance bias (§5.8) and generalisation (§3.8). This work has certainly *not* shown that that XCS is suitable for all, or even a wide range of classification and reinforcement learning tasks. Nor has it addressed the question of to which tasks XCS is best suited – all inductive systems have a bias which makes them better suited to some tasks than others [183, 224] – although a start was made in [154].

In addition, although XCS is a breakthrough in classifier systems technology, it is unlikely to be the last word in classifier systems. Rather, as a "new direction" in classifier systems research [298], it is likely to contain many quirks. Rather than being *the* means of using accuracy-based fitness, it is surely only one of many possible approaches, a point in an almost completely unexplored design space of accuracy-based systems. XCS is the focus of active, indeed unprecedented, research; many modifications and extensions have been studied (e.g., [161, 304, 162, 163, 164, 165, 147, 309]), and many more will be. At this point, such work is just beginning, and the author still considers XCS to be poorly understood; its strengths and limitations have yet to be mapped, and surely many surprises remain.

7.5 Concluding Remarks

Classifier systems have a long history and a considerable literature; the first implemented LCS appeared in 1978, and over 600 LCS publications are catalogued in [152]. LCS have a broad and important range of potential applications, from pattern classification to machine discovery to reinforcement learning. In RL, they intrinsically address the two major problems of gener-

alisation and policy learning. LCS can be defined very broadly, and when so defined seem a fundamental approach to learning and representation.

7.5.1 The Moral of the Story: The Need for a Complex Systems Design Methodology

Given all this, what lessons can we learn from more than 20 years of classifier systems research that we might apply elsewhere? One problem encountered with LCS (and other evolutionary systems) is over-reliance on the adaptive power of the genetic algorithm. GAs *are* powerful, robust (widely-applicable), domain-independent algorithms. But it is not enough to invent a coding and fitness function for a problem and assume the GA will be able to find optimal (or even good) solutions. In order to adapt, the fitness function must give the GA the information it needs. We have seen that different fitness functions (i.e., the XCS and SB–XCS fitness schemes) make all the difference to an LCS's capabilities, and that SB–XCS's superficially plausible Q-value fitness does not in fact lead the GA towards the populations we would like (§5.3.2).

Another lesson – one which applies very broadly – is that we cannot simply construct complex systems on paper and expect them to work as intended; the interactions which characterise complex systems make this approach infeasible. Instead, we need a design methodology for complex systems which interleaves incremental design and evaluation. One approach is to start with simple learning agents and build complexity by adding one component at a time. In doing so, we must constantly confirm that our expectations of the system are being met; we need to check whether the components are interacting in the way in which we intend. This is consistent with Wilson's animat path to Artificial Intelligence [296], in which we iteratively design minimal agents to adapt to the current task and then increase the complexity of the task. Evidence of the utility of a minimalist approach comes from the fact that, although the classifier system is a complex framework in which many mechanisms have been investigated, much of the progress made with LCS has come through study of minimalist systems such as SCS, ZCS and XCS.

The animat path is part of the *design-based approach* (p. 15) which seeks to understand systems from the point of view of an engineer trying to design a system which meets certain requirements, and is inspired by software engineering and conceptual analysis in philosophy. Given that software engineering addresses the task of designing complex systems, perhaps its methodology could prove useful in designing systems such as LCS. As the sophistication of Artificial Intelligence systems increases, so will the difficulties of engineering them and consequently so does the need for good design methodology.

7.5.2 Classifier Systems and Reinforcement Learning

Despite considerable work on LCS relatively little progress was made for many years, and, during the 1980's, "mainstream" RL appeared and overtook LCS research. There is no doubt that LCS research is difficult, as it addresses fundamental Artificial Intelligence issues of learning and representation. However, so does RL. We can attribute the difference in success to the separation of the two problems of finding a policy and generalisation in mainstream RL. LCS research, in contrast, has tackled both problems simultaneously, since LCS intrinsically combine both. This results in a much harder problem. It also results in much more complex systems, as anyone who has implemented both tabular Q-learning and a classifier system can attest. One indication is that, of the 12 parameters listed on page 59, all are used by XCS and only one by tabular Q-learning. It is difficult to exaggerate the burden this additional complexity imposes on the LCS researcher in implementing, debugging, parameterising, and analysing these systems. Is it any wonder that complex LCS remain poorly understood while simple tabular RL systems are much better understood?

By linking classifier systems and mainstream RL, LCS should benefit from the great strides made in RL in recent years. This and a shift to accuracy-based fitness promise to finally from classifier systems from the quagmire in which they have languished for so long.

7.5.3 The Future

Despite the difficulty of working with classifier systems, much progress has been made in recent years, and the next five years should see the field blossom. I hope that in this time they will become less of an abstract, potential approach, and a more practical, widely-used, competitive machine learning technique. This will require more theory and better integration with RL and ML. This thesis has attempted to provide some needed theory by starting to explain the differences between strength and accuracy-based systems, Pittsburgh and Michigan systems, and classifier systems and mainstream RL.

Appendix A

Evaluation of Macroclassifiers[1]

In order to evaluate the effect of the use of macroclassifiers on a typical problem, two series of tests were run using the 6 multiplexer (§2.5.1). In the first series of tests, macroclassifiers were used as normal, but in the second they were disabled by omitting the test for matching classifiers upon insertion of the new classifier into [P]. Thus in the second case all classifiers had a numerosity of 1, and classifiers with identical conditions and actions were permitted to coexist. In neither case was subsumption deletion used, as this only makes sense when macroclassifiers are available, and to use it in one and not the other would have biased the results.

Figure A.1 shows the results of these runs. System parameters were set as in §2.5.1. The reward function returned either a high reward for the correct action or a low reward for the incorrect action. Although performance on the $\mathcal{P}$ metric (§2.5.1) is highly similar, there is a difference in the system error (§4.3.1) for the two cases: the system with macroclassifiers generally had a slightly lower system error. This experiment was repeated using reward functions with 16 and 32 different possible rewards in different states and in neither case was there any observed difference between the use of macroclassifiers and their lack.

When macroclassifiers are used, new duplicate classifiers are discarded and instead the numerosity of an existing macroclassifier is increased. As a result, an existing better-evaluated classifier becomes stronger, and the new one, which would have some form of initial value for prediction and prediction error, is not introduced into the population. This may account for the observed difference in system error. In the reward functions with more diverse rewards the actual reward may tend to be closer to the new-classifier-influenced system prediction because the rewards are closer. This

[1]This appendix originally appeared in [144].

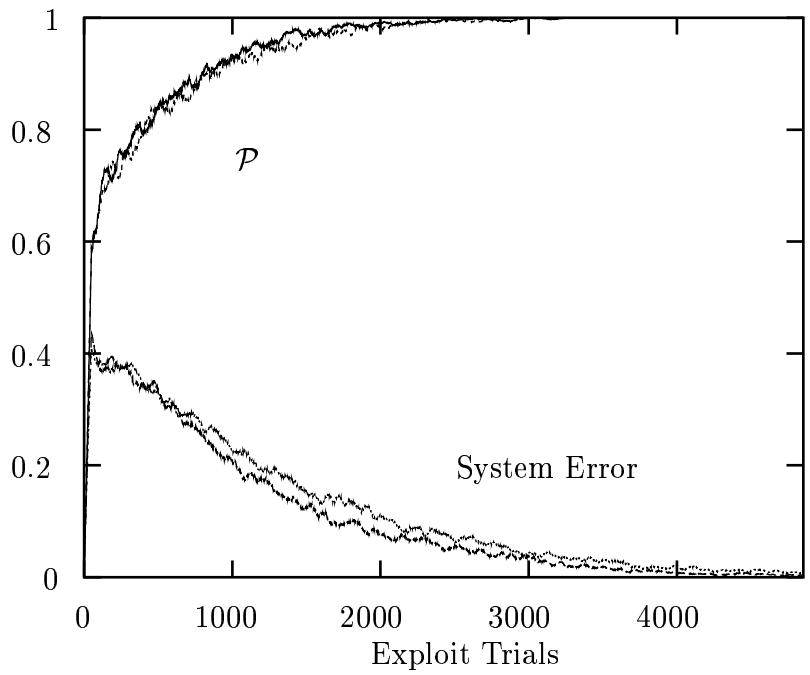


Figure A.1: Comparison of XCS with and without macroclassifiers on the 6 multiplexer with 2 reward levels. Curves are the average of 30 runs. System error is slightly lower when macroclassifiers are used on problems with 2 rewards, but unchanged when more rewards are used. Performance is very similar for the two cases.

could account for the lack of effect outside the 2 reward case. In any case reducing the level of system error is a beneficial if slight side-effect of the use of macroclassifiers.

With macroclassifiers, the population on the 6 multiplexer and similar tests approaches a minimal size (and can reach it using condensation [298, 144]). Without macroclassifiers, the population tends to fill up with duplicate microclassifiers.

In summary, experience with XCS strongly supports the use of macroclassifiers; all comparisons indicate that, if they have any effect, they improve the system. They appear to perform essentially as equivalent microclassifier populations, increase run time speed and offer interesting statistics. The only drawback to the use of macroclassifiers is the additional complexity of implementing the system.

Appendix B

Example XCS Cycle

In this appendix we walk through XCS's behaviour during a single time step, as specified in chapter 2 and in particular the algorithms on pages 36, 41 and 48, in order to clarify the process. Suppose XCS is learning the 3 multiplexer problem (see §2.5.1), that rewards of 1000 for correct responses and 0 for incorrect responses are used, and the population of rules [P] consists of the following:

Rule	**Cond.**	**Action**	**p**	ε	**F**	**Num.**
a	##1	0	531	611	0.2	2
b	0#0	0	512	553	0.3	1
c	01#	1	1000	0	1	1
d	11#	1	491	448	0.2	1
e	#00	0	1000	0	1	1
f	111	1	1000	0	1	2
g	111	0	0	0	1	1
h	110	1	0	501	0.2	1

where **p** indicates the prediction of a rule, ε its prediction error, and **F** its fitness (see page 31 and §2.3.5). **Num.** indicates the numerosity (number of virtual copies) of the rule (§2.3.2). This population has been contrived to show interesting features of the XCS algorithm. A real population would likely be larger, and differ subtly in terms of **p**, ε and **F**.

B.1 The Performance System Algorithm

We repeat the XCS performance system algorithm from page 36 for convenience:

1. Obtain the single input string from the environment.
2. Form the match set [M] of classifiers in [P] whose conditions match the input string (p. 36).
3. Calculate the system prediction for each action using the rules in [M] in order to fill the prediction array (p. 36).
4. Select an action based on the system predictions (p. 37).
5. Send the selected action to the environment and receive a reward.
6. Form the action set [A] of classifiers in [M] which advocated the action selected in 4 above.
7. Call the credit assignment algorithm (figure 2.3).

Step 1. Obtain the environmental input.

The first step is to receive an input string, which we will assume is 1 1 1 , from the environment.

Step 2. Form the match set.

The second step is to form the match set [M], that is, the subset of [P] which matches the current input. [M] is:

Rule	Cond.	Action	p	ε	F	Num.
a	##1	0	531	611	0.2	2
d	1 1 #	1	491	448	0.2	1
f	1 1 1	1	1000	0	1	2
g	1 1 1	0	0	0	1	1

Step 3. Calculate the system prediction (equation (2.1)) for each action.

Action 0 The only rules in [M] which advocate action 0 are a and g, so according to equation (2.1) the system prediction for action 0 is:

$$
\begin{aligned}
P(a_0) &= \frac{\sum_{x \in [\mathrm{M}]_{a_0}} F_x p_x}{\sum_{x \in [\mathrm{M}]_{a_0}} F_x} \\
&= (F_a p_a + F_g p_g)/(F_a + F_g) \\
&= (0.2 \cdot 531 + 1 \cdot 0)/(0.2 + 1) \\
&= 106.2/1.2 \\
&= 88.5
\end{aligned}
$$

Action 1 The two rules in [M] which advocate action 1 (rules d and f) result in a system prediction of:

$$
\begin{aligned}
P(a_1) &= (F_d p_d + F_f p_f)/(F_d + F_f) \\
&= (0.2 \cdot 491 + 1 \cdot 1000)/(0.2 + 1) \\
&= (98.2 + 1000)/1.2 \\
&= 915.167
\end{aligned}
$$

Step 4. Select an action.

Let's assume XCS alternates between pure explore and exploit cycles (§2.3.3), and is currently in an explore cycle, meaning it selects an action at random from among the two advocated actions. Let's assume action 1 is chosen.

Step 5. Send the selected action to the environment and receive a reward.

As action 1 is, according to the 3 multiplexer function, the correct response to input 1 1 1, XCS receives $r_t = 1000$.

Step 6. Form the action set.

Form the action set [A], the subset of [M] which advocates the selected action. [A] is:

Rule	**Cond.**	**Action**	**p**	ε	**F**	**Num.**	**Exp.**	**GA**
d	1 1 #	1	491	448	0.2	1	43	4030
f	1 1 1	1	1000	0	1	2	18	4010

This time we have also shown the experience of each rule (**Exp.**), and the time step the rule was last in an action set in which the GA was invoked (**GA**). We will use these parameters in later calculations.

Step 7. Invoke the credit assignment and rule discovery algorithms to update classifiers in [A] and check for GA invocation.

We will cover these algorithms in the following sections.

B.2 The Credit Assignment Algorithm

The previous section covered the algorithm with which XCS generates a response to an input. In this section, we cover the credit assignment algorithm XCS uses to update statistics regarding the rules used in generating its response. The credit assignment algorithm from page 41 is repeated here for convenience:

```
1.  If the previous time step's action set [A]_-1 is not empty then
2.      Update [A]_-1 (p. 42).
3.      Do action set subsumption in [A]_-1 (p. 52).
4.      If conditions for GA invocation in [A]_-1 are met (p. 49) then
5.          Call the rule discovery algorithm in [A]_-1 (p. 47).
6.  If the episode has just ended then
7.      Update [A].
8.      Do action set subsumption in [A].
9.      If conditions for GA invocation in [A] are met then
10.         Call the rule discovery algorithm in [A] (p. 47).
11. else
12.     [A]_-1 ← [A] (in sequential mode only).
```

The 3 multiplexer is a non-sequential task so, as noted on page 40, only steps 7 through 10 apply.

Step 7. Update [A].

This involves the five updates of the XCS fitness calculation from §2.3.5, which we label as steps 7.1 through 7.5.

7.1 Update Prediction We begin by updating each rule's prediction according to (2.4). Since the 3 multiplexer is a non-sequential task we use $P = r_t$ (2.2). Let's use a learning rate of $\beta = 0.2$ and begin by updating rule d:

$$\begin{aligned} p_d &\leftarrow p_d + \beta(P - p_d) \\ &\leftarrow 491 + 0.2(1000 - 491) \\ &\leftarrow 592.8 \end{aligned}$$

after which we update rule f:

$$\begin{aligned} p_f &\leftarrow 1000 + 0.2(1000 - 1000) \\ &\leftarrow 1000 \end{aligned}$$

Note that the order in which rules are updated is unimportant since the updates are independent.

7.2 Update Prediction Error Next we update each rule's prediction error according to (2.5). First rule d:

$$\begin{aligned} \varepsilon_d &\leftarrow \varepsilon_d + \beta\Big(|P - p_d| - \varepsilon_d\Big) \\ &\leftarrow 448 + 0.2\Big(|1000 - 491| - 448\Big) \\ &\leftarrow 448 + 0.2(509 - 448) \\ &\leftarrow 448 + 0.2(61) \\ &\leftarrow 460.2 \end{aligned}$$

and then rule f:

$$\begin{aligned} \varepsilon_f &\leftarrow 0.0 + 0.2\Big(|1000 - 1000| - 0.0\Big) \\ &\leftarrow 0.0 + 0.2(0.0 - 0.0) \\ &\leftarrow 0.0 \end{aligned}$$

Notice that the prediction error of rule d has increased even though it advocated the correct action on this time step. This is because there was a large error between its reward prediction ($p_d = 491$) and the actual reward ($P = r_t = 1000$). d's large error is due to it being overgeneral; it advocates the correct action in response to input $1\,1\,1$, but the incorrect action in response to input $1\,1\,0$.

7.3 Calculate Accuracy Let's use the typical values of $\alpha = 0.1, v = 5$ and $\varepsilon_o = 10$ to parameterise the accuracy equation (2.6). The accuracy of rule d is:

$$\kappa_d = \begin{cases} 1 & \text{if } \varepsilon_d < \varepsilon_o \\ \alpha(\varepsilon_d/\varepsilon_o)^{-v} & \text{otherwise} \end{cases}$$

Since ε_d exceeds the threshold ε_o (that is, $\varepsilon_d = 460.2 > \varepsilon_o = 10$), its accuracy is:

$$\begin{aligned}\kappa_d &= \alpha(\varepsilon_d/\varepsilon_o)^{-v} \\ &= 0.1(460.2/10)^{-5} \\ &\approx 0\end{aligned}$$

Since $\varepsilon_f < \varepsilon_o$, the accuracy of f is 1.

7.4 Calculate Relative Accuracy The relative accuracy of d is:

$$\begin{aligned}\kappa'_d &\leftarrow \frac{\kappa_d \cdot numerosity(d)}{\sum_{x \in [A]} \kappa_x \cdot numerosity(x)} \\ &\leftarrow \frac{0 \cdot 1}{(0 \cdot 1) + (1 \cdot 2)} \\ &\leftarrow 0\end{aligned}$$

while the relative accuracy of f is:

$$\begin{aligned}\kappa'_f &\leftarrow \frac{1 \cdot 2}{(0 \cdot 1) + (1 \cdot 2)} \\ &\leftarrow 1\end{aligned}$$

7.5 Update Fitness The fitness of rule d is updated towards its relative accuracy:

$$\begin{aligned}F_d &\leftarrow F_d + \beta(\kappa'_d - F_d) \\ &\leftarrow 0.2 + 0.2(0 - 0.2) \\ &\leftarrow 0.2 - 0.04 \\ &\leftarrow 0.16\end{aligned}$$

and F_f is updated towards κ'_f:

$$\begin{aligned}F_f &\leftarrow F_f + \beta(\kappa'_f - F_f) \\ &\leftarrow 1 + 0.2(1 - 1) \\ &\leftarrow 1 + 0 \\ &\leftarrow 1\end{aligned}$$

We can see the overgeneral rule d's fitness is approaching 0, while accurate rule f's fitness remains at 1.

Step 8. Action Set Subsumption.

Having completed the updates to the rule parameters, it is time to test for action set subsumption (§2.3.6). No subsumption occurs in this [A] as the most general of the accurate and experienced rules (rule d) does not logically subsume any other rule in [A].

Step 9. Check for GA invocation.

Now we check to see if the GA should be invoked in the current [A]. Let's assume the current time step $t = 4050$, and that θ_{GA}, the threshold for GA invocation, is 25. According to (2.10), the GA is invoked if:

$$\frac{\sum_{x \in [\mathrm{A}]} \left(t - x_{\mathrm{GA}}\right)}{|[\mathrm{A}]|} > \theta_{GA}$$

where t is the current time step, x_{GA} is the time step on which rule x was last in an action set in which the GA was invoked (or the time step of its creation if it has never been in such an action set), and $|[\mathrm{A}]|$ is the size of the action set. This evaluates to:

$$\begin{aligned} \frac{(t - d_{\mathrm{GA}}) + (t - f_{\mathrm{GA}})}{2} &> 25 \\ \frac{(4050 - 4030) + (4050 - 4010)}{2} &> 25 \\ 30 &> 25 \end{aligned}$$

so we will indeed call the GA. The operation of the GA is specified by the rule discovery algorithm, which we cover in the next section.

B.3 The Rule Discovery Algorithm

The rule discovery algorithm specifies how XCS generates new rules using a Genetic Algorithm, and how they are incorporated into the rule population. Rule discovery operates exclusively in [A] in non-sequential tasks, and in sequential tasks either in [A] or $[\mathrm{A}]_{-1}$ as specified by the credit assignment algorithm.

The rule discovery algorithm from page 48 is repeated here for convenience:

```
1.  Reset GA counters of rules in [A] (or [A]_{-1}).
2.  Select two parents P1 and P2 from [A] (or [A]_{-1}).
3.  Cross P1 & P2 with probability χ otherwise clone to obtain C1 & C2.
4.  Mutate each bit in C1 and C2 with probability μ.
5.  Initialise parameters of C1 and C2.
6.  Delete rules as needed.
7.  If C1 subsumed by P1 or P2 then
8.      Increment numerosity of subsuming parent.
9.  else
10.     Insert C1 into [P].
11. If C2 subsumed by P1 or P2 then
12.     Increment numerosity of subsuming parent.
13. else
14.     Insert C2 into [P].
```

Step 1. Reset GA counters.

To begin, we reset the GA counters of the two rules in [A] (d and f) to the current time step so that they will tend to delay the next application of the GA in their action sets through (2.10).

Step 2. Select two parents.

To select two parents from [A] we construct a roulette wheel with a slot for each rule in [A], the size of which is determined by the fitness of that rule. Consequently, our wheel has two slots, of size $F_d = 0.16$ and $F_f = 1$ respectively.

Next we spin the wheel twice to select two parents. Let's assume that rule f is selected both times, which is not unlikely given its fitness is much greater than d's.

Step 3. Obtain two children.

With probability χ we obtain offspring by applying 1 point crossover to the parent's conditions, and with probability $1 - \chi$ we simply clone each parent once. In this particular case, even if crossover occurs, the resulting offspring will be identical to their parents, since the same rule f is used as parent for both offspring.

Step 4. Mutate C1 and C2.

We apply point mutation with probability μ to each bit in the conditions and actions of the children. Let's assume one of the offspring is mutated so that its condition is $1\,1\,0$.

Step 5. Initialise C1 and C2.

The newly generated rules are initialised as described in §2.3.6. We now have two new rules, which we label f' and f'':

Rule	Cond.	Action	p	Num.	Exp.	GA
f'	$1\,1\,0$	1	1000	1	0	4050
f''	$1\,1\,1$	1	1000	1	0	4050

Step 6. Delete rules as needed.

We have just generated two new rules and, before inserting them in the population, must ensure that doing so does not cause the population size to exceed its limit N. If the current population size in *micro*classifiers $+2 > N$ we delete sufficient rules from [P] so that the current size $+2 = N$. Let's assume the population size is already at the limit and that two rules must be deleted to make room for the new ones.

To select a rule for deletion, we construct a roulette wheel with a slot for each rule in [P] (note: not just each rule in [A]), the size of which is determined by the precise deletion scheme in use. Let's assume rule a is selected for deletion. Because it is has numerosity 2, instead of removing the rule from the population, we simply decrement its numerosity by 1; there is now one less virtual copy of rule a, and the population size in *micro*classifiers (though not *macro*classifiers) is one less.

Now the roulette wheel must be adjusted to reflect the loss of numerosity from rule a, after which we spin it a second time to select another rule for deletion. Let's assume that b is the unlucky rule this time. Because there is only one copy of b it is removed from the system (conceptually, its numerosity becomes 0).

Steps 7–14 GA Subsumption and Rule Insertion.

We have generated and initialised two new rules and made space for them, but we are still not quite ready to insert them into the rule population [P]. First we must check to see if the new rules are subsumed by their parents through GA subsumption (§2.3.6). Let's assume θ_{sub}, the minimum experience required for one rule to subsume another, is 25. Since rule f, the parent of f' and f'', only has experience 18 it cannot subsume other rules.

Rule Insertion Although the two new rules have escaped subsumption by their parent, there is still a final check to make before inserting them into [P]. There should never be two distinct macroclassifiers with the same condition and action, and to prevent this we check each new rule against each other in the population before inserting it. If a rule with the same condition and action is found, the new rule is discarded and the numerosity of the existing rule is incremented. This check reveals that f'' is actually a copy of its parent f, so f'' is discarded and the numerosity of f is incremented by 1. Rule f', however, survives this final hurdle and is inserted into [P], which is now:

Rule	Cond.	Action	p	ε	F	Num.
a	##1	0	531	611	0.2	1
c	01#	1	1000	0	1	1
d	11#	1	592.8	460.2	0.16	1
e	#00	0	1000	0	1	1
f	111	1	1000	0	1	3
f'	110	1	1000	53.13	0.06	1
g	111	0	0	0	1	1
h	110	1	0	501	0.2	1

Conclusion

This concludes the current time step, so the system returns to step 1 of the performance algorithm (figure 2.1) to commence the next time step, unless a termination condition, e.g., the prescribed number of completed time steps, is met.

Appendix C

Learning from Reinforcement

And thus do we of wisdom and of reach,
With windlasses and with assays of bias,
By indirections find directions out

William Shakespeare. Hamlet. Act II Scene i.

Appendix Outline

This appendix presents a somewhat idiosyncratic introduction to reinforcement learning tasks, tailored to the work presented in the thesis. Unusual attention is paid to non-sequential decision tasks, which are the subject of most of the results presented in this work. For a more complete introduction the text by Sutton and Barto [265] is highly recommended.
This appendix is structured to initially provide a gentle, intuitive introduction to RL tasks and some of their features, with the introduction of a more formal framework reserved until the end. Algorithms which can solve RL tasks are introduced in later appendices.

Throughout this thesis learning problems are described in terms of interaction between a learning agent and its environment (or task, or problem). The nature of this interaction defines different learning paradigms, or classes of problems, of which the broad paradigm called Reinforcement

Learning (RL) is one. A wide range of tasks can be modelled using the RL paradigm, although most work in RL is concerned with sequential decision making (§C.3). Many methods for solving sequential decision tasks exist, and RL's niche is in tasks in which there is uncertainty about the effect of actions. More specific problems to which RL has been applied include robot navigation [203, 76, 67, 108], adversarial games such as chequers [221, 222] and backgammon [267, 268, 269], control of dynamic systems [17, 264, 189], elevator dispatching [63, 64], dynamic channel allocation in cellular telephone networks [233, 274], and job shop scheduling [323, 324, 322]. See [265] for case studies.

For introductory material on RL tasks and algorithms which solve them see [265, 184, 220, 235, 3, 262]. The survey by Kaelbling, Littman and Moore [141] provides a good introduction for those already familiar with Machine Learning. Wyatt [320] provides a more recent survey.

C.1 Three Learning Paradigms

Let's assume a model of interaction between learner and environment based on a sequence of discrete time steps (or trials). On each time step, the agent receives some sensory information from the environment and responds with an action. In addition, the environment may then respond with further information concerning the action just chosen by the agent. Three learning paradigms are defined by the nature of the feedback from the environment, and in explaining RL it helps to contrast the three.

C.1.1 Supervised Learning

In *supervised learning*, after the learner has chosen an action, the environment indicates what the correct action was in that situation. In this case the environment acts as a teacher. Supervised learning can be used when we have a training set of exemplars, for each of which we know the correct action. In supervised learning we teach the learner a particular way of acting, and the goal of the learner is always to choose the action the teacher considers correct. An example would be training an agent to classify images as being pictures of either tanks or flowers. The following dialogue (based on one in [141]) illustrates such a supervised learning problem:

```
Time Step: 1
Environment: Does picture 1 show a tank or a flower?
Learner: A flower.
```

```
Environment: It's a tank.

Time Step: 2
Environment: Does picture 2 show a tank or a flower?
Learner: A flower.
Environment: It's a flower.

Time Step: 3 ...
```

The following more abstract dialogue is equivalent:

```
Time Step: 1
Environment: You are in state 1. Choose action A or B.
Learner: I'll take action A.
Environment: The correct action was B.

Time Step: 2
Environment: You are in state 2. Choose action A or B.
Learner: I'll take action A.
Environment: The correct action was A.

Time Step: 3 ...
```

C.1.2 Reinforcement Learning

In *reinforcement learning*, the environment returns a *reward* (typically a real valued number) indicating the value of the action taken, as illustrated by the following dialogue:

```
Time Step: 1
Environment: You are in state 1. Choose action A or B.
Learner: I'll take action A.
Environment: Your reward is 100.

Time Step: 2
Environment: You are in state 9. Choose action A, B, or C.
Learner: I'll take action A.
Environment: Your reward is 200.

Time Step: 3 ...
```

Notice the key difference between RL and SL – that in RL the environment returns a measure of how good each action was, rather than indicating

the correct action. We can think of RL as training by reward and punishment, which, as one might suspect, makes it useful when we know what we would like to have done, but we don't know just how we would like it done.[1] This ignorance of how to achieve the desired results can have two aspects. First, we may not know the solution to the problem we pose. For example, we may have perfect knowledge of the rules of a game like chess (the task), but not know the optimal solution (i.e., how to play perfectly). The second form of ignorance concerns the mechanisms by which solutions are found. We ignore such mechanisms when we treat the RL agent as a black box. An example would be training a dog to fetch a ball – we can observe its behaviour and reward or punish it, but have little idea of what is going on inside its head.

In SL the fundamental goal of the learner is to act correctly as often as possible. In RL the agent's overall goal is similar: roughly speaking it is to obtain as much reward as possible. (We'll return to the goals of RL agents in §C.4.) Because the learner is not told if its action was correct, it can only deduce this by trying alternative actions in response to the same input and comparing the reward received in each case. Consequently, in RL the learner faces the *explore/exploit dilemma*: the choice of whether to use its best guess at choosing the correct action (exploit its current knowledge), or whether to try actions it has not yet tried (explore new actions) and so improve its guesses in the future. This means that, unlike in supervised learning, the learner will not always try to choose the correct action since it must explore to improve its knowledge.[2]

From the preceding we can see that RL requires less than SL of the person who specifies the problem and yet requires more from the learner, since it receives less feedback on its actions. Because RL requires less of the problem specification it is a more broadly applicable paradigm (though more difficult for the learner, as we will see in §C.2).

RL can be used to model aspects of learning in animals (including humans) – in fact there is a related field within psychology also called reinforcement learning. We consider the animal's environment to return a reward when appropriate, e.g., when it consumes food it receives a positive reward, while when it injures itself it receives a negative reward. (The source of the reward is actually part of the animal's brain which classifies events as good or bad to some degree, but we can consider this part of the brain as external to the reinforcement learning part, and thus part of its environment.)

[1] For our purposes punishments need only be the relative lack of reward. We will refer to punishments, negative and positive rewards simply as rewards.

[2] Other definitions of supervised learning might allow a role for exploration.

C.1.3 Unsupervised Learning

In the third paradigm, *unsupervised learning*, the environment gives no feedback as to the value of the action chosen by the learner. In this case the learner seeks to classify inputs based on regularities inherent in the data it observes, rather than learn to optimise its evaluation by an external source as in the other two paradigms.

C.2 The Explore/Exploit Dilemma: a Feature of RL

RL and SL differ in that in SL, when the agent visits a state and chooses an action, it is told what the correct action for that state was. This is all it needs to know to act optimally in this state if it ever revisits it (assuming the world does not change, and that the agent can distinguish the state when it encounters it). In RL, in contrast, the agent is only given a reward, and does not know if some other action would have generated more reward. Consequently, if it is to learn which action generates the most reward in that state it must *explore* the other possible actions to find how much reward they generate. In general the rewards associated with actions may be stochastic, in which case an agent must sample each action many times to obtain good estimates of its value. This need to explore sets RL apart from SL, and makes RL that much harder for the agent. See [270, 141, 299, 319, 265, 179] for introductions to and surveys of exploration in RL.

The need to explore options arises from the overall goal of an RL agent to maximise the reward it receives – it cannot do so unless it knows which action in a state maximises reward. The more an agent knows about the rewards associated with states and actions the better it will be able to maximise rewards. Thus there is value in learning about the environment – exploring – because this knowledge can be *exploited* to maximise payoff.

While there is value in exploration, ultimately the agent is interested in maximising rewards. Clearly it cannot spend all its time exploring alternative actions as this involves taking those with suboptimal rewards. Equally clearly it cannot from the outset take its current best guess at the best action, because at the outset it knows nothing of what the best actions are. (When an agent takes the best-looking action, we say it is acting *greedily*.)

To optimise the amount of reward received the agent clearly needs to balance exploration with exploitation, hence the explore/exploit dilemma. Roughly, the agent needs to explore at the outset in order to gain knowledge of the problem, and to exploit this knowledge (explore less) later on.

The question of just how to balance exploration and exploitation optimally is a rather difficult one, particularly since optimal behaviour depends on the how many times the agent will return to the same state. If never, it makes sense to take the best guess at maximising reward. But as the number of expected revisits increases, so does the value of exploring alternative choices.

The explore/exploit dilemma can be cast in perhaps more familiar terms by considering the problem Stuart encountered during his travels, when he visited a restaurant in which he could understand neither menu nor waiters. Lacking any knowledge of the available dishes, Stuart could only choose one at random by pointing to the menu. However, upon returning to the restaurant the following day, Stuart had a choice: whether to choose the same dish (exploit his knowledge) or to choose another one (explore his options). In this example the length of the interaction is relevant: if Stuart expected to return to the restaurant many times it would make sense to sample many dishes to find favourites. If, on the other hand, he was on a short holiday and would not return after the second visit, exploration may not have been worth the risk of choosing something inedible.

Other examples would be the problem of a fisherman choosing between many locations on a lake to optimise the number of fish caught, or the classic example of a gambler choosing between two slot machines to optimise his winnings.

To make matters more difficult, in the case of sequential decision problems exploration is complicated by the need to consider the long-term effects of actions.

C.3 Sequential and Non-sequential Tasks

We can classify all our learning tasks as either sequential or non-sequential decision tasks. This is an important distinction as non-sequential decision tasks are in an important sense much simpler, and some learning techniques we can apply to them cannot be used with sequential tasks.

A *non-sequential decision task* (sometimes called a *single-step task*) is one in which the response of the learner has no effect on which states the environment will visit in the future. In contrast a *sequential decision task* (or *multi-step task*) is one in which the actions of the learner *do* influence state transitions.[3]

[3]It is common to refer to sequential tasks as *delayed reward tasks*, but this is technically incorrect as sequentiality and delay of reward are independently variable problem dimensions.

Non-sequential decision tasks are simpler in that the agent only needs to consider the current state when making decisions, whereas in sequential tasks it must consider what states its actions may lead to. Some examples may help to illustrate the distinction. Classifying a set of pictures as of either dogs or cats is a non-sequential decision task, since the category each picture belongs to does not depend on any other picture.[4] On the other hand, choices of destination in the travelling salesman problem (in which a salesman attempts to find the shortest route between a number of cities) are sequential, as each destination will affect how easy it is to get to any other.

As another example, consider a *non-sequential* decision task in which the learning agent has perfect knowledge of the reward for each action in each state, so that exploration is not an issue. In this case the best action to take to maximise the reward received from the environment is simply the one which produces the highest reward *right now*. (That is, the best policy is to act greedily.)

Now consider a *sequential* decision task in which the agent again has perfect knowledge of rewards. Following the greedy policy will maximise the reward on each time step, but because actions now affect which states will be seen in the future, acting greedily may force the agent to enter states with low rewards sometime in the future. This means the greedy policy may result in the agent receiving less total reward than it might otherwise. An example would be eating too much ice cream – rewarding now, but costly later if it makes you feel ill. In general, sequential decision tasks require short-term pain for long-term gain.

C.3.1 Immediate Reward and Long-term Value

In a non-sequential decision task, the value of taking an action in a state on a given time step is just its *immediate reward* – that is, the reward the agent receives just for taking that action in that state. (Normally immediate reward will be referred to simply as reward.) If rewards are stochastic the true value of a given state-action pair is its expected reward. In non-sequential tasks optimal behaviour requires only that we maximise the reward received on each time step, independently of what we do on any other time step.

But in the preceding section we saw that in a sequential task considering only immediate rewards can easily lead to long-term suboptimality. This is because *reward is an indication only of the immediate value of an action.* Clearly, if we want to speak of the real (long-term) value of an action in a

[4] Such a pattern classification task could well be learnt using supervised learning methods, but, equally, we can make it an RL task by only providing feedback in the form of rewards.

sequential problem we need a separate notion of value, one which considers the long-term performance of the learner. (In appendix F we'll see a way in which an agent may obtain long-term estimates of value by learning a value function.) In sequential tasks, optimal behaviour requires that we maximise the long-term value of our actions, not just the immediate reward.

C.3.2 Sequential Decisions Imply RL

Supervised learning requires a correct training set so that the learner can be told the correct action on each time step. This means that for sequential problems the designer (the one who sets up the learning experiment) needs to work out the long term consequences of actions in advance. Consequently a supervised learner never faces a sequential problem; the designer has already solved the sequencing aspect and translated the problem into a non-sequential one.

In RL, reward is a measure of only the immediate value of an action. The designer leaves it to the learner to learn the long term consequences of actions. Consequently, for sequential problems it is often easier to invent a suitable reward function than to find a correct training set, i.e., it is easier to do RL than SL.

This is not to trivialise the problem of SL. In appendix D we'll see that SL tasks typically involve not only the accurate classification of states, but also accurate generalisation over these states. Further, even though in SL a teacher provides the learner with a set of correctly classified examples, we typically wish to train the learner on a subset of possible examples and for it to generalise usefully to examples on which it has not been trained.

C.3.3 Episodic and Continuing Tasks

Some tasks have termination conditions, that is, they reach a point where nothing further happens. For example, if a robot's goal is to deliver a letter, the task finishes once the letter has been delivered. Such tasks are called *episodic tasks*. If, on many occasions, we ask the robot to deliver a letter, it encounters the same task many times. Each instance of the task is called an *episode* (or *trial*). From the environment's point of view, each episode is independent: what happens during one episode has no influence on what happens during another. Consequently it is appropriate to think of the environment resetting itself at the end of the episode. Of course if the agent learns, then from its perspective the episodes are not independent, since it can apply what it learned during a previous episode to a later one.

Not all tasks are episodic: *continuing tasks* simply keep going. An example would be the task faced by a house control system, which, day after day, must regulate the temperature and lighting and watch for signs of fire.

C.4 The Agent's Goal: Maximising Return

Earlier it was stated that, roughly speaking, the goal of any RL agent is to maximise the reward it receives in the long run. In this section we'll see that although this is the basic idea, it is an oversimplification.

A first problem with this formulation of the agent's goal is that an RL agent may be interested in learning about (exploring) its environment rather than maximising reward. For example, it may wish to learn about the environment's state transition dynamics or the rewards associated with states and actions. Although an agent may sometimes be set the task of maximising such information rather than maximising reward, such considerations are beyond the scope of this work, and we will concentrate on the typical case where the goal of the agent is maximising rewards, rather than information.

A second problem with our initial formulation of the goal of an RL agent is due to the distinction between the immediate value of an action (as indicated by reward) and its long-term value. In non-sequential tasks the agent's goal is indeed to maximise rewards individually, but in sequential tasks it is concerned with the sequence of rewards, not just individual ones.

C.4.1 Return and Reward

To deal with these problems, and avoid further imprecision, we'll use the idea of a *return* which can be defined as needed for different tasks. This allows us to finally define the goal of an RL agent. For our purposes:

Definition 1 *The goal of an RL agent is to maximise the return it receives.*

The return at time t, denoted R_t, is some function of the sequence of rewards the agent receives from that time step.

In §C.1.2 we presented the RL paradigm as a dialogue between a learner and its environment, in which the agent is rewarded on the time step during which it acts. Sutton and Barto [265], in contrast, reward the agent on the time step following its action, that is, the agent acts at time t and is rewarded at time $t+1$ ([265] p. 52). From here on we adopt Sutton and Barto's model in order to synchronise our notation with theirs. Consequently, the return

at time R_t is some function of the rewards which occur on subsequent time steps: $r_{t+1}, r_{t+2}, r_{t+3} \ldots$.

For non-sequential tasks, the events during each time step are independent of those which occur during any other, so it suffices for the agent to maximise return on each time step independently:

$$R_t = r_{t+1} \tag{C.1}$$

Because each time step is independent, maximising r_t on each time step maximises the return the agent receives in the long run. That is, for non-sequential tasks, maximising return *is* equivalent to maximising rewards.

C.4.2 Sequential Formulations of Return

For sequential tasks the agent needs to consider the rewards it will receive in the future. One way to do so is to define R_t as the sum of the rewards it will receive in the future:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots r_T = \sum_{k=t+1}^{T} r_k \tag{C.2}$$

where T is the last time step of the environment. Using this definition the agent must consider what will happen in the future if it is to maximise R_t, since future rewards contribute to R_t. Now maximising return is not the same as maximising rewards – the agent may choose a smaller reward at time $t+1$ in order to get a bigger reward at time $t+2$, and hence a bigger R_t. Note that the contribution of a given reward r_i towards the total return is independent of when r_i occurs.

Discounted Return

Defining return with equation (C.2) may be suitable when the task is *episodic*, that is when T is finite. However, in *continuing* tasks, where $T = \infty$, (C.2) may easily produce an infinite sum, which is inconvenient. Partly for this reason it is very common to use a different definition of the return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \gamma^T r_T = \sum_{k=0}^{T} \gamma^k r_{t+1+k} \tag{C.3}$$

where a *discount rate* $0 \leq \gamma < 1$ bounds the infinite sum. Note that (C.3) is a proper generalisation of (C.2); if we set γ to 1.0 no discounting occurs and (C.3) reduces to (C.2).

With $\gamma < 1$ rewards contribute less towards R_t the further in the future they occur. Thus γ controls how much weight the agent should give to future rewards in making decisions. Consequently, γ is useful not only as a mathematical convenience, but can be seen as an interest rate, a probability of living another step, or a bias towards shorter paths to a goal. When $\gamma = 1.0$ the agent will value most highly the sequence of states which results in the most reward, regardless of how long that path is. This is often not what we want. For example, being paid £5 a year from now is not as desirable as being paid £5 today for various reasons, e.g., we may not live another year, we can earn interest on the £5 over the next year, and we may need the £5 right now. At the other extreme, if we set γ to 0.0 the agent will be shortsighted and take no interest in the future consequences of its actions. This is often undesirable, as it would lead the system to choose £5 today rather than £1000 tomorrow. Note, however, that if $\gamma = 0$ equation (C.3) reduces to (C.1); that is, the discounted sequential return reduces to the non-sequential return. This makes perfect sense, as in non-sequential tasks there are no future consequences to consider.

For sequential tasks we typically want to set γ to some value between 0 and 1 in order to give possible future rewards a suitable weighting. See [141] for a brief survey of alternative definitions of return.

C.5 Formalising RL Tasks

Now that we've seen many of the components of reinforcement learning tasks informally, we present them in a more formal way.

C.5.1 Environment

Non-sequential Tasks

We'll define non-sequential learning environments consisting of:

- A discrete time step t.[5]
- A set of states $\mathcal{S}$. The environment is in exactly one state $s_t \in \mathcal{S}$ on any given time step.

[5] Work has been done on continuous time RL (see [289, 23, 81]) but we will not consider it here.

- A function $\mathcal{A}$ which returns the set of possible actions which an agent may take in a given state $\mathcal{A}(s) \rightarrow a$.

- A reward function $\mathcal{R}^a_{ss'}$ which returns the expected immediate reward r_t given a state s, successor state s' and action a. That is,

$$\mathcal{R}^a_{ss'} = E\Big\{r_{t+1} \;\Big|\; s_t = s, s_{t+1} = s', a_t = a\Big\} \tag{C.4}$$

Note that the sequence in which states occur is not defined in the above, and in particular is not influenced by the actions taken by the learner.

Sequential Tasks and MDPs

For sequential tasks it is customary to adopt the formalism of Markov Decision Processes (MDPs) to model environments (although many applications of RL violate the Markov property). To define an MDP, to $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, t\}$ we add:

- A state transition function $\mathcal{P}^a_{ss'}$ which specifies the probability of transition from a state s to another state s' when taking action a. That is, $\mathcal{P}^a_{ss'} = \Pr\Big\{s_{t+1} = s' \;\Big|\; s_t = s, a_t = a\Big\}$.

An MDP allows some agent to influence the state transitions of the system by selecting an action on each time step; the transition probabilities may be sensitive to the agent's action. Note that while $\mathcal{P}$ may be sensitive to the agent's actions, we normally assume that the environment $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\}$ does not change over time (except, of course, for the time step t) – only the learner changes in response to its interaction with the environment.[6]

C.5.2 Learning Agent

Policies

$A(s)$ defines the possible actions for a state s. It is up to the learner to select which action to take at a given point.

- We define a *policy* π as a mapping from states to actions $\mathcal{S} \rightarrow \mathcal{A}$, i.e., a rule which the learner uses to decide which action to take in a particular state.

[6]If $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\}$ does not depend on t we can think of t as just a notational construct rather than an intrinsic part of the environment.

All agents have a policy, and we can view any mapping from states to actions as a policy – e.g., random action selection is a policy. A policy may be implemented in many ways, e.g., by a neural network or a look-up table. A policy can be deterministic or stochastic.

Learning is the process of adapting the policy in order to optimise performance with respect to the agent's goal (which we defined in §C.4). Agents differ in whether they learn or not, and in how their learning component works. Some learners adjust their policies directly, others derive them from a mapping from states (or states and actions) to some estimate of utility.

Exploration Policies

- RL agents also need an *exploration policy*, which determines how they address the explore/exploit dilemma.

Many approaches to exploration control are possible (see e.g., [265]). A simple exploration policy, the greedy policy, is to always select the best-looking action. Just as the agent can learn to optimise its rewards by adjusting its policy, it can also learn to optimise its exploration by adjusting its exploration policy. This subject is, however, beyond the scope of this thesis.

What is an RL Agent?

At this point we anticipate the question of what constitutes an RL agent. Following convention [141, 265] we define an RL agent thus:

Definition 2 *An RL agent is any agent which solves RL problems.*

C.5.3 Agent-environment Interaction

Interaction between the agent and environment proceeds as follows on each time step t:

- The environment returns a reward to the agent $\mathcal{R}^a_{ss'} \rightarrow r_t$ for acting on the previous time step $t-1$. (If this is the first time step we return some default value, e.g., 0.)
- The agent senses the current environmental state s_t.
- The agent selects exactly one of the available actions $a_t \in \mathcal{A}(s_t)$.

In episodic tasks the sequence of time steps is divided into episodes, each of which consists of a number of time steps. Since each episode is, from the environment's perspective, independent, we reset the time step t at the start of each episode (§C.3.3).

C.6 Summary

We've distinguished reinforcement, supervised and unsupervised learning on the basis of the interaction between learning agent and environment. We've encountered the ubiquitous RL problem of balancing exploration and exploitation, a problem which does not occur in supervised learning (as it has been formulated here). We've distinguished between the simpler non-sequential and more complex sequential decision tasks, and considered some precise definitions of the goal of an RL agent in both cases making use of the concept of a return. Finally, as is customary in the RL literature, we've introduced a formal model of RL problems based around Markov Decision Processes.

We'll consider methods for evaluating policies in appendix E, and methods for finding better policies (i.e., solving RL problems) in appendices F and G, but, before doing so, in appendix D we consider another class of learning problem, generalisation problems, and discuss their interaction with the RL problem.

Appendix D

Generalisation Problems

Categorization is the system's sine qua non *for combating the environment's perpetual novelty.*

John H. Holland [121] p. 92.

Appendix Outline

Appendix C presented reinforcement learning problems, one of the two major classes of problems with which this thesis is concerned. This appendix presents the second major class, generalisation problems. After a brief, general introduction to problems of generalisation we focus on the problem of state aggregation, the aspect of generalisation with which this work is most concerned. State aggregation and the utility of aggregations are formalised towards the end of the appendix. The relationship of generalisation and RL problems is briefly examined, as is the peril inherent in attempting both simultaneously. A later appendix will return to this in more detail, and will consider specific examples of generalising RL agents.

All learning systems generalise in the sense that they abstract general principles from observed data. For example, a tennis player learns to play tennis (a somewhat abstract skill covering potential tennis games)

thanks to experience of actual tennis games. Generalisation is a rather general concept, encompassing many problems.

Learning systems address various generalisation problems in different ways, e.g., modularisation (e.g., [142, 135]), lifelong learning (e.g., [209, 271, 107]) and shaping (e.g., [76]) are all ways of generalising. This thesis is concerned with generalisation in classifier systems, for which an important form of generalisation is state aggregation; generalising by lumping states together into classes.

D.1 Why Generalise?

Just as there are many forms of generalisation, there are many reasons why we might want to generalise, perhaps the most compelling of which is that many problems of interest simply cannot be tackled with our favourite basic methods without adding some form of generalisation. Why? Because many optimisation techniques suffer from what Bellman called the "curse of dimensionality" [21]; that computational requirements (time and/or storage) grow with the size of the state space. In particular, in combinatorial problems the size of the state space grows exponentially as we add state variables, and so too do the computational requirements of the algorithm.

D.1.1 The Curse of Dimensionality

A trivial example should help illustrate this point. A straightforward and consequently popular method of associating values with states is to use a look-up table to exhaustively list each state and its value. This approach does not generalise over states and so is only practical for problems with small state spaces. Suppose we wish to implement a robot controller using a look-up table and some algorithm which updates the values associated with states as they are visited. If the robot has n sensors, each of which can take on m values, there are n^m possible states it can perceive. With only 5 sensors, each capable of producing 10 values, there are 5^{10} or nearly 10 million states. To illustrate the curse of dimensionality and combinatorial explosion and that occurs when we increase the input space consider that adding a 6^{th} such sensor results in 6^{10} or over 60 million states.

If we estimate the value of state-action pairs (as in Q-learning), rather than just states, matters are even worse.

D.1.2 The Need for Generalisation

One difficulty with mapping each state to a table entry is that it may be difficult to allocate enough storage for a table with many millions of entries. But a much greater problem is that it will take a great deal of learning (experience with the task) to fill it with useful information because the sheer size of the state space means it may be a very long time before a given state is visited, or between revisits. Because the state space is so large, if we do not generalise, visiting a state tells us only about a very small part of the task (namely that state).

In short, we often find that although there exist methods capable of solving a given problem in principle, the time and space required to do so make application of these methods impossible. Were time and space complexity not an issue (e.g., if we had limitless computational power) we would not need efficient methods; random search or enumeration would suffice for any problem.

In some cases, even limitless computational power may not suffice in the absence of generalisation. In very large or continuous spaces we may never visit exactly the same state twice, and in such cases, without generalisation we simply *cannot* adapt to the problem.

D.2 Generalisation in RL

For the sake of exposition reinforcement learning and generalisation have been presented as distinct problems. Sutton and Barto [265] take the same approach, describing generalisation in RL agents as a matter of supervised learning, with any number of SL methods being applicable. Taking this approach, the problem of state aggregation described in the following section can be thought of purely as a supervised learning problem, for which the SL form of dialogue presented in §C.1.1 is appropriate.

Alternatively, generalisation can be thought of as part of the RL problem. Certainly generalisation is needed for practical RL: the main drawback of RL is poor scalability of basic approaches, and generalisation is an important means of addressing this drawback. But more than this, attempts to solve the two problems simultaneously interact heavily, a consequence of which is that in the generalisation problem the teacher cannot state definitively what the correct action is for each input. That is, the teacher's knowledge is based on the current solution to the RL problem, and as the solution to the RL problem improves, so too does the teacher's choice of correct and incorrect actions. In other words, the generalisation problems which arise while doing RL are non-stationary. The reader is advised to bear this interaction in mind,

although no further reference is made to it in this appendix. §6.2.5 refers to the interaction briefly.

D.2.1 Generalising Over Policies and Value Functions

There are two approaches to generalisation in RL which concern us in this work: generalising over policies and generalising over value functions. As explained in §3.8.1, classifier systems may do either. We can describe both as state aggregation, which we cover in the following section.

D.3 State Aggregation

Suppose the elements of set $\mathcal{S} = \{A, B, C, D, E, F\}$ fall into two equivalence classes $A = B = C$ and $D = E = F$, and that some agent attempts to represent these classes with rules of the form: {<set of states in class> $\rightarrow$ <class label>}, where <class label> is a unique integer for each class. Two rules suffice to represent the two equivalence classes in $\mathcal{S}$: $\{A, B, C\} \rightarrow 1$ and $\{D, E, F\} \rightarrow 2$. (In fact, since there are only two classes, one rule suffices if we assume any $s \in \mathcal{S}$ not covered by that rule is implicitly part of the other class.) We can generalise the first rule so that it includes D, yielding: $\{A, B, C, D\} \rightarrow 1$. This is an inaccurate aggregation, but the point is that we can modify the generality of these rules by adding and removing states from the set of states they contain. This kind of generalisation is called *state aggregation*. Not all learning systems explicitly aggregate states in this way, but classifier systems do. The goal of state aggregation is to assign all states to some generalisation, and for the generalisations to be as accurate and general as possible.

D.4 State Space and Generalisation Space

In §C.5 we defined $\mathcal{S}$ as the set of states the task can be in. To formalise the problem of aggregating states we introduce the notion of a space of generalisations in $\mathcal{S}$ called a *generalisation space*.

- A generalisation space $\mathcal{G}$ for a state space $\mathcal{S}$ is the power set of that state space i.e., $\mathcal{G} = 2^{\mathcal{S}}$.

The power set of a set $\mathcal{S}$ is the set of all subsets of $\mathcal{S}$. For example, if $\mathcal{S} = \{A, B, C\}$ then $\mathcal{G} = \{\{\}, \{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$. For a state space of size n, generalisation space is of size 2^n. (This includes the

null set – otherwise it is of size $2^n - 1$.) In practice, the choice of representation may limit the range of generalisations which can be expressed, but, typically, the size of generalisation space grows much more quickly than the size of the state space.

We can use elements of $\mathcal{G}$ to represent classes or regularities in $\mathcal{S}$. In the worst case no classes exist in $\mathcal{S}$, and the only useful generalisations are those trivial ones which include only a single state (e.g., $\{A\}$). But even when considerable regularities exist in $\mathcal{S}$ we are faced with the counterintuitive concept of searching a much larger space $\mathcal{G}$ in the hope of simultaneously improving learning (of an RL task) in a smaller space $\mathcal{S}$.

D.5 Summary

This work is primarily concerned with two classes of problems. The RL problem of appendix C is essentially the problem of finding a policy which optimises return. This appendix has introduced the second class of problems, generalisation. Within generalisation, of particular interest in this work is the problem of aggregating states in a way that is consistent either with the policy or with the value function. The RL and state aggregation problems are intertwined because, on the one hand, aggregations must be consistent with the policy or value function, and on the other, they influence what policies may be found.

Appendix E

Value Estimation Algorithms

Appendix Outline

Evaluating actions, and more generally policies, is an essential part of the process of searching for better policies; if we don't know how good two policies are, how can we choose between them? This appendix first introduces methods of evaluating actions and policies in non-sequential tasks, and then extends them to the more complex sequential tasks. For non-sequential tasks it suffices to estimate the expected reward for a state-action, so we are effectively approximating the environment's reward function. As we'll see, in sequential tasks we must instead estimate the long-term value of actions, which means approximating a value function which is only implicit in the task specification. For sequential tasks we distinguish Monte Carlo and Temporal Difference approaches to evaluation. At the end of this appendix we consider predicting the value of aggregations of state-actions, and how to store value estimates.

This appendix and appendix F together present what are normally referred to as "Reinforcement Learning algorithms", by first, in this appendix, reviewing algorithms for evaluating actions and policies, and later, in appendix F, reviewing one approach to improving policies. The subject of "RL algorithms" has been divided into these two parts in order to emphasise

the two component processes, and to emphasise that the very same methods for evaluating policies are available when using a very different approach to improving policies, namely the Evolutionary Algorithms of appendix G.

The RL framework Recall from §C.5 that our framework for describing RL tasks consists of a set of states $\mathcal{S}$, a function returning the actions available in a state $\mathcal{A}$, a reward function $\mathcal{R}$ and (in sequential problems) a state transition function $\mathcal{P}$. We assume that $\mathcal{R}$ and $\mathcal{P}$ are inaccessible to the learning agent, except through its interactions with the environment, consisting of sequences of states, actions and rewards. (Otherwise we could apply the efficient methods of Dynamic Programming (DP) [21, 22, 204], which require direct access to $\mathcal{R}$ and $\mathcal{P}$.)

The goal of any RL agent is to maximise the return it receives, and it has some policy $\mathcal{S} \rightarrow \mathcal{A}$ which tells it how to act in any given state. The problem addressed in this appendix is to evaluate how good an agent's policy is at achieving the goal of maximising return. More generally, we are interested in how good individual state-actions and aggregations of state-actions are at maximising return.

Tabular implementation For the time being we'll assume that the value of each state-action is estimated independently. That is, we assume a look-up table with an entry for each state-action. Later, in §E.5, we'll consider estimation of the value of aggregations of state-actions.

Prediction and Control We often use an estimate of the expected return as a prediction of the return to be obtained in the future. In this sense, this appendix is all about prediction of future returns based on experience with past returns.

We will only take up the matter of how to maximise return in later appendices. Maximising return means finding good policies, and policies are the means by which an agent controls its environment (to the extent that it can).

E.1 The Value of State-actions

Definition 1 (p. 241) stated that the goal of any RL agent is to maximise the return it receives. Consequently, the value of a state-action to an agent is a function of both its tendency to result in return and the magnitude of that return. More specifically:

Definition 3 *The true value of a state-action pair is the expected return which follows it.*

We can express definition 3 symbolically if we denote the true value of taking action a in state s as $Q^*(s,a)$, where the Q stands for Quality (or value) of the state-action pair, and the * indicates it is the true value and not an estimate. Then:

$$Q^*(s,a) = E\Big\{R_t \;\Big|\; s_t = s,\ a_t = a\Big\} \tag{E.1}$$

where $E\Big\{R_t\Big\}$ is the expected return for taking action a in state s. In the following section we'll see what happens when we plug the non-sequential definition of return into (E.1), while in §E.4.2 we'll see a more specific formulation of (E.1) which applies to sequential definitions of return.

E.2 Non-sequential RL: Estimating Reward Functions

E.2.1 The Value of State-actions in Non-sequential Tasks

Combining (E.1) and the non-sequential definition of return (C.1), i.e., $R_t = r_{t+1}$, means the true value of a state-action pair is simply its expected reward:

$$\begin{aligned} Q^*(s,a) &= E\Big\{R_t \;\Big|\; s_t = s,\ a_t = a\Big\} \\ &= E\Big\{\mathcal{R}_t \;\Big|\; s_t = s,\ a_t = a\Big\} \end{aligned} \tag{E.2}$$

where $E\Big\{R_t\Big\}$ is the expected return and $E\Big\{\mathcal{R}_t\Big\}$ is the expected reward for taking a in s at time t. That is, in approximating the value of a state-action (its expected return) we are approximating part of the reward function.[1]

In §E.3 we consider updates which we can use to approximate the reward function. In particular, in §E.3.4 we see a simple, but complete algorithm for doing so.

[1] When we address the issue of control in appendix F, we'll see that, in non-sequential tasks, to maximise expected return in a state s we need only select the action $a \in \mathcal{A}(s)$ with the highest expected reward.

E.3 Estimating Expectations with Sample Averages

An obvious way to estimate the expected reward of a state-action pair is to use the average of the observed rewards generated by that state-action.[2] If we write the *estimated value* of a state-action pair as $Q(s, a)$ then:

$$Q(s, a) = \frac{r_1 + r_2 + r_3 \ldots r_n}{n} \tag{E.3}$$

where, for notational convenience in equation (E.3) only, we'll assume there is only 1 state-action pair, so that all rewards $r_1 \ldots r_n$ are applied to the same $Q(s, a)$. The accuracy of this estimate can be expected to improve with the number of samples n by the law of large numbers. That is, the estimate $Q(s, a)$ approaches the true value $E\{R_t\}$ as n approaches ∞.

E.3.1 Incremental Updates

Let's think about how to implement the average update (E.3) incrementally. To simplify notation in this section, for the time being let's forget about estimating the value of a state-action pair and simply consider the more general problem of averaging a series of values $r_1, r_2, r_3 \ldots r_n$. That is, we want to find the average avg of the series:

$$avg = \frac{r_1 + r_2 + r_3 \ldots r_n}{n} \tag{E.4}$$

Clearly we don't need to explicitly store each r_i – we can instead just store the incremental sum of the r_i and a counter for n, the number of samples seen. We initialise sum and n to 0, and then each time we see a new value r_i the following updates yield the new average:

$$\begin{aligned} sum &\leftarrow sum + r_i, \\ n &\leftarrow n + 1, \\ avg &\leftarrow \frac{sum}{n} \end{aligned}$$

[2] We remind the reader that expectation is the population mean, while average is the sample mean.

With a little more computation we can even do away with *sum*: all we have to do is generate *sum* as the product of the average *avg* and the number of values n. We then add the new value r_i and reaverage:

$$avg \leftarrow \frac{\overbrace{\big[avg \cdot n\big]}^{sum} + r_i}{n+1} \tag{E.5}$$

remembering to update n *after* reaveraging:

$$n \leftarrow n+1$$

E.3.2 A General Form of Incremental Update

Note that we can rewrite (E.5) as:

$$\begin{aligned} avg &\leftarrow \frac{avg \cdot n + r_i + avg - avg}{n+1} \\ &\leftarrow avg + \frac{r_i - avg}{n+1} \\ &\leftarrow avg + \frac{1}{n+1}\big[r_i - avg\big] \end{aligned} \tag{E.6}$$

(E.5) and (E.6) are ways of computing the average of a series of values incrementally. Update (E.6) will look familiar to many readers as it is an instance of the following general form of incremental update:

$$NewEstimate \leftarrow OldEstimate + StepSize \overbrace{\Big[Target - OldEstimate\Big]}^{error} \tag{E.7}$$

specific instances of which include forms of the *Widrow-Hoff delta rule* (also called the *delta rule*), the Least Mean Squares (LMS) update, the Temporal Difference rules to be introduced shortly, the arithmetic mean (E.6), and standard deviation.[3]

Error in Incremental Updates

The expression $\big[Target - OldEstimate\big]$ is called the *error* in $OldEstimate$. The idea is to reduce the error in $NewEstimate$ by making it closer to $Target$, i.e., $NewEstimate$ is equal to $OldEstimate$ plus some proportion ($StepSize$) of the error.

[3]Update (E.7) was taken from [265] section 2.5, which provided inspiration for some of the material in this section. An alternative derivation of (E.6) also appears there.

E.3.3 Setting *StepSize* in Incremental Updates

Note that *StepSize* decreases in (E.6) on each time step, in particular, in a way that guarantees convergence in the limit to the true mean of the rewards for that state-action pair. In practice, however, $0 < StepSize \leq 1$ is often a constant, in which case (E.7) is an (exponential) recency-weighted average rather than an unweighted average of the samples (see [265]). That is, because *StepSize* is fixed, the further in the past an error occurred, the less influence it has on the current estimates. An advantage of recency-weighted averages over unweighted averages is that their devaluation of older samples allows them to follow changes in non-stationary functions. An advantage of using a constant *StepSize* is that we need not keep track of n. We'll adopt the convention of denoting *StepSize* as α, yielding the following from equation (E.6):

$$avg \leftarrow avg + \alpha\Big[r_i - avg\Big] \tag{E.8}$$

E.3.4 A Prediction Algorithm for Non-sequential RL

To return to the issue of estimating the value of a state-action pair, we can use the incremental updates of the previous sections to estimate the true value based on a series of sampled rewards. For example, from E.8 we obtain:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} - Q(s_t, a_t)\Big] \tag{E.9}$$

where we note that the reward at time $t+1$ is used to update the state-action pair of the previous time step s_t, a_t (see p. 241).

Given a policy π, to approximate the value of its component state-actions, it is in principle sufficient to use a look-up table with an entry for each $Q(s, a)$, and apply (E.9) to each ($Q(s, a)$ a sufficient number of times. The algorithm in figure E.1 does this, commencing by initialising the estimates $Q(s, a)$ to arbitrary values; uniform initialisation and random initialisation are two obvious methods. On each time step, an action is selected using π. Following this the environment returns the successor state and the immediate reward. Each estimate $Q(s, a)$ which is updated infinitely often will converge to its true value $Q^*(s, a)$, provided α is declined appropriately.

Approximating the Reward Function

If more than one action is possible in at least some states, and if π maps each state to a single action, then following π we only visit a subset of state-

For each $s \in \mathcal{S},\ a \in \mathcal{A}(s)$
    $Q(s,a) \leftarrow$ arbitrary number // *Initialisation*
For each episode:
    $s \leftarrow initialState()$
    For each time step of current episode:
        $\pi(s) \rightarrow a$
        $environment(a) \rightarrow (r, s')$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r - Q(s,a)\Big]$
        $s \leftarrow s'$

Figure E.1: Tabular algorithm for incrementally approximating expected rewards using sample averages.

actions. In such cases, the algorithm in figure E.1 approximates a subset of the reward function. To approximate the entire reward function it suffices to set π to random action selection, in which case all state-actions will be visited.

E.4 Sequential RL: Estimating Long-term Value Functions

We've seen that *for non-sequential tasks* the value of a state-action is its expected reward. However, in §C.3.1 we saw that *for sequential tasks* we somehow need to consider the long-term consequences of our actions if we want to maximise reward over a *sequence* of actions. How can we learn about long-term consequences when environmental feeback on the value of an action (i.e., reward) does not tell us anything about them?

E.4.1 Long-term Value Functions

One way to learn about long-term consequences from observation of immediate consequences is to learn an *action-value function* (also simply called a value function) which maps state-actions to an estimate of their *long-term* value. (We'll see in appendix F that if we know the true long-term value of actions, we can behave optimally simply by choosing the action $a \in \mathcal{A}(s)$ which has the highest long-term value.)

Note that this means that whereas in non-sequential problems the task is to approximate (a subset of) the reward function, in sequential tasks it is to approximate a long-term value function. This long-term value function

is not part of the MDP which specifies an RL problem (§C.5), although it is implicit in the problem specification in that it can be computed given the MDP and the agent's definition of return. It is in this way that we can specify an RL problem (as reward function etc.) without knowing its optimal solution(s) (and the true long-term value function *is* the solution to the prediction problem).

Backing up values The basic idea behind constructing a long-term value function is to propagate (back up) part of the value of a successor state-action to its predecessor; if state-action m leads to state-action n we propagate part of n's value back to m. This way m's value reflects n's – now m's value is an estimate of the long-term value of taking m, not just its value on the time step we take it.

An example The backing up of value is illustrated in figure E.2, the top third of which shows a number of states as circles labeled with captial letters, actions as arrows connecting states and labeled with lower case letters, and the immediate reward for each state-action as an integer value. The two squares on either side denote terminal states. The scenario is that Axel, upon arriving home (state H) craving ice cream, faces a decision about how much to eat. He can either eat a reasonable amount of ice cream (action b), which is moderately rewarding ($r = 5$), or consume all the ice cream (action c), which is very rewarding ($r = 10$). Without considering future consequences of his actions Axel will see eating all the ice cream as the better option (as illustrated by the bar chart in the middle of figure E.2). However, action b allows him to go out to a film (state F) later that night, which is itself moderately rewarding ($r = 5$), while eating all the ice cream means he will spend the evening feeling ill in bed (state B), which is decidedly unrewarding ($r = -20$). By backing up part of the value of a to b, and of d to c, we can create an action-value function which allows Axel to choose the course of action which is most rewarding in the long term, only by considering the action-values for the state H in which he must make a decision (i.e., he need only compare $Q(H, b)$ and $Q(H, c)$). This action-value function is illustrated with a bar chart in the lower third of figure E.2, in which all the value (discount rate $\gamma = 1$) from state-actions a and d has been backed-up (added) to their predecessors. (I.e., $Q(H, b) = 5 + 5$ and $Q(H, c) = 10 - 20$.)

Interdependencies Of course for the long-term value approach to work, we need to back up values from each state-action to each of its predecessors. And if we alter n's value based on its succesor o, this in turn affects m's value,

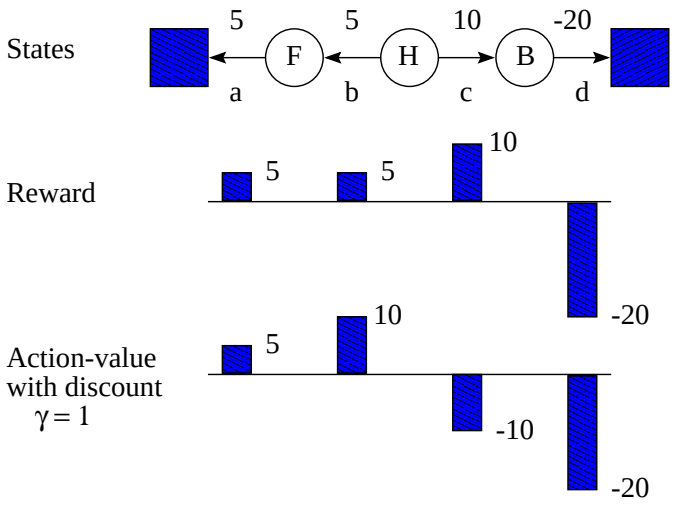


Figure E.2: Backing up values between state-action pairs.

since it is based partly on n's. It should be clear that there is generally an extensive set of interdependencies in the action-values of state-actions.

E.4.2 The Value of State-actions in Sequential Tasks

Value depends on the policy followed The example in figure E.2 demonstrates how we can back up value from one state-action to another. Notice, however, that the only state in which the agent needs to choose between alternative actions is the starting state H – in states F and B there is only one action, and so no choice to make. This means there is no choice of value to back up. But in general, successor states may have multiple state-actions. Which of these values should we back up to the predecessor? From definition 3, the value of a state-action is its expected return. To determine the sequence of rewards upon which the return is based we must know how the agent will act in subsequent states, which is given by its policy. So – in sequential tasks – a state-action's value depends on what policy will be followed in the future.

We can now extend definition 3 to a more precise form for the sequential case:

Definition 4 *In sequential tasks, the true value of a state-action pair is its expected return* **under a given policy**.

We can express definition 4 formally by combining the general expression

for the value of a state-action pair (E.1) and the general expression for the discounted return (C.3):

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\Big\{R_t \;\Big|\; s_t = s,\ a_t = a\Big\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \;\middle|\; s_t = s,\ a_t = a\right\} \end{aligned} \tag{E.10}$$

where $Q^\pi(s,a)$ is the true value of a state-action under a policy π, and E_π is the expectation under π. Note that a need not be the action recommended by π in s. It suffices to take a in s and follow π thereafter.

E.4.3 The Value of a Policy

Equation (E.10) defines the value of each state-action under a policy π. Given this, we can compare policies: π is better than or equal to a policy π' if, for each state, the expected return of its best state-action is better than or equal to that of π'. That is, $\pi \geq \pi'$ if and only if for all $s \in \mathcal{S}$:

$$\max_a Q^\pi(s,a) \geq \max_a Q^{\pi'}(s,a) \tag{E.11}$$

E.4.4 Estimating Values with Monte Carlo Methods

A simple method of estimating long-term value Since $Q^\pi(s,a)$ is the expected return for taking a in s and following π thereafter, we can estimate it by averaging actual observed returns. That is, after the agent (which is following π) takes a in s at time t we can record the rewards it receives until the end of the episode and calculate a return R_t for s, a for that episode. This single R_t is an estimate of the true value of $Q(s,a)$, and to obtain better estimates of $Q(s,a)$ we can average many returns over many episodes. As the number of episodes and sampled returns grows, the average return $Q(s,a)$ will approach the true return $Q^\pi(s,a)$.

This method is the direct extension to the sequential case of the averaging we used for non-sequential tasks in §E.3.4; now instead of averaging immediate rewards to approximate the reward function we average complete returns to approximate the value function.

Incremental implementation To perform the averaging described above, we could record all the returns and average them at the end of each episode:

$$Q^\pi(s,a) \leftarrow \frac{R_e}{E} \tag{E.12}$$

where R_e is the return on episode e and E is the number of episodes.

However, a better approach is to use an incremental update for this averaging (as we did in §E.3), which we obtain by substituting R_e for r_{t+1} in (E.9):

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha\Big[R_e - Q^\pi(s,a)\Big] \tag{E.13}$$

Note that this update is incremental in the sense that we do not reaverage all the values after each episode (unlike with (E.12)). But it is *not* incremental in another sense, since the $Q^\pi(s,a)$ estimates are only updated at the end of each episode, and not *within* an episode.

Monte Carlo Methods Following Sutton and Barto [265] we'll refer to any method which averages *complete* returns (i.e., from t till the end of the episode) as a *Monte Carlo* (MC) method, although outside RL the term is used more broadly. A problem with Monte Carlo methods is that the estimates $Q(s,a)$ are only updated at the end of each episode, meaning the agent does not adapt during an episode. Consequently Monte Carlo methods for evaluating policies converge very slowly. Additionally, application of Monte Carlo methods to continuing tasks is awkward as they do not break naturally into episodes, and without episodes there are no opportunities to perform updates.

First-visit and Every-visit MC Two ways of formulating the return are used with MC methods: *first-visit* MC averages returns following the first visit to s,a, while *every-visit* MC averages returns following every visit to s,a. The MC algorithm at the start of this section did not specify whether first or every visit averages were made, and can be used with both.

E.4.5 Estimating Values with Temporal Difference Methods

A novel contribution of RL is an alternative to Monte Carlo updating, called *Temporal-Difference* (TD) updates. The difference is that, where Monte Carlo updates must wait for the end of an episode to perform a backup, TD updates wait only for the next state, meaning they perform a backup on

each time step. Not surprisingly, TD updates usually converge more quickly than Monte Carlo updates ([265] p. 139). TD methods are able to update on each step by using the value of a later state-action as an estimate of part of the return. This is where the name "Temporal Difference" comes from: an estimate at time t is updated based on its difference with another estimate at some later time.

Sarsa Let's consider the value of a state-action:

$$Q^{\pi}(s,a) = E_{\pi}\Big\{R_t \;\Big|\; s_t = s,\ a_t = a\Big\}$$

defined as the expected discounted return:

$$= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+1+k} \;\middle|\; s_t = s,\ a_t = a\right\}$$

which we can split into the expected reward for the current state-action r_{t+1} and the expected return which follows on subsequent time steps following policy π:

$$= E_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+2+k} \;\middle|\; s_t = s,\ a_t = a\right\}$$

The expected return which follows is equivalent to the discounted true value of the state-action at $t+1$:

$$= E_{\pi}\Big\{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) \;\Big|\; s_t = s,\ a_t = a\Big\} \tag{E.14}$$

Equation (E.14) gives the value of a state-action $Q^{\pi}(s_t, a_t)$ as a function of the expected immediate reward and the discounted value of the successor state-action $Q^{\pi}(s_{t+1}, a_{t+1})$. That is, it gives us a way of relating the value of one state-action to another.

Equation (E.14) is one way of writing the *true* value of s, a, but how can we estimate this value from interaction with the task? We can use the immediate reward r_{t+1} to sample $E_{\pi}\Big\{r_{t+1}\Big\}$, but what about the remainder of (E.14)? This is where the temporal differencing comes in: we update the value of one state-action using the value of its successor, that is, we update $Q(s_t, a_t)$ based on the value of $Q(s_{t+1}, a_{t+1})$. In other words, we use $\gamma Q(s_{t+1}, a_{t+1})$ to sample $\gamma Q^{\pi}(s_{t+1}, a_{t+1})$.

Substituting these samples for the true values allows us to drop the expectation from (E.14) and define the sample estimate $Q(s,a)$ of the true value of s,a:

$$Q(s_t,a_t) = r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) \tag{E.15}$$

which we can plug into our general incremental update (E.7) to obtain:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \underbrace{\left[\overbrace{r_{t+1} + \gamma Q(s_{t+1},a_{t+1})}^{Target} - \overbrace{Q(s_t,a_t)}^{OldEstimate} \right]}_{Error} \tag{E.16}$$

which is known as Sarsa [265] for the sequence of events it makes use of: $s_t, a_t, r_t, s_{t+1}, a_{t+1}$. Strictly speaking we have presented the one-step tabular version of Sarsa.[4] For tabular implementations Sarsa has been shown to converge with probability 1 to the optimal policy, assuming infinite revisits to every state-action and appropriately declined α [234]. See appendix H for more on the origins of Sarsa.

Q-learning If we modify (E.16) so that instead of using the observed state-action $Q(s,a)$, we instead take the best state-action in the successor state as an estimate of the return to follow we have:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \underbrace{\left[\overbrace{r_{t+1} + \gamma \max_a Q(s_{t+1},a)}^{Target} - \overbrace{Q(s_t,a_t)}^{OldEstimate} \right]}_{Error} \tag{E.17}$$

which is the widely used 1-step Q-learning update [286, 265]. Tabular implementations of Q-learning have been shown to converge with probability 1 to the optimal policy, assuming infinite revisits to every state-action and appropriately declined α [287, 138, 276].

On-policy and Off-policy Evaluation

The idea of using the value of the best state-action of the successor state as part of the target is to update the estimate $Q(s,a)$ directly towards the

[4] "One-step" refers to the update only taking the next state's value into account. More generally, *n-step* algorithms use the value of the state n steps into the future.

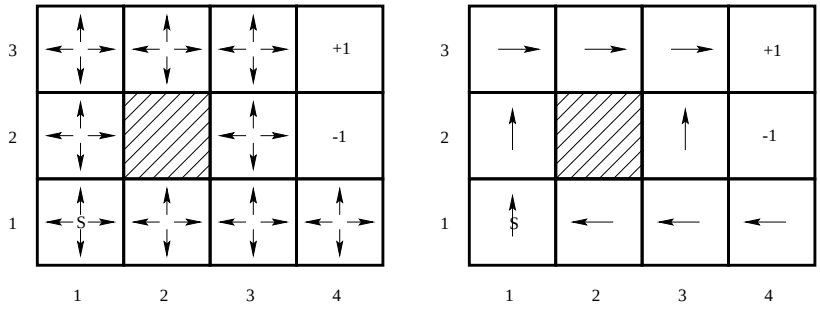


Figure E.3: A maze with available actions (left) and the optimal policy (right).

return for following the optimal policy, rather than whatever policy the agent happens to be following. Thus Q-learning is an *off-policy* algorithm: it learns about one policy while following another. Sarsa, by contrast, is an *on-policy* algorithm: it learns the value of the policy it is following. Q-learning and Sarsa are briefly compared in [265] (p. 149).

E.4.6 Russell and Norvig's Maze: A Sequential RL Task

As a simple example of a sequential decision task let's consider the maze in figure E.3, which was taken from [220] (p. 499). The left side of the figure shows the layout of the states and the actions available in each state. The right side of the figure shows the optimal policy. The maze has a start state marked S in the lower left corner (position 1,1), and two terminal states in the upper right area marked $+1$ and -1, which is the respective immediate reward for entering these states. At all other state transitions the agent receives a reward of -0.04. In each state the agent can take one of four actions: North, South, East and West. The action has the intended effect with probability 0.8, otherwise the agent moves equiprobably to a square either $+90°$ or $-90°$ from the intended one. If the action takes the agent into a wall no state transition occurs.

We can solve this problem by constructing a table of value estimates $Q(s, a)$ of size $|\mathcal{S}| \cdot |\mathcal{A}(s)| = 9 \cdot 4 = 36$, where s can be any state since in this case all have the same $|\mathcal{A}(s)|$. We can initialise these values randomly and update them based on trials in the maze using the Monte Carlo or TD updates we've just seen. Note that the optimal action in state (3,1) is to go West rather than North, despite the fact that this is a much longer route and

there is a reward of -0.04 on each step. This action is optimal because the penalty for taking the longer route is less than that for sometimes accidentally ending up in the -1 terminal state as a result of stochastic state transitions.

E.4.7 Summary of Sequential RL

Backing up values means that, unlike the non-sequential case where we simply approximate the reward function, we are approximating a long-term value function, which takes into account not just the immediate value of an action, but the value of states and actions which follow. What states and actions follow is determined by the policy the agent follows and the environment's dynamics. Consequently, in the sequential case, the value of a state-action depends on the policy the agent will use in the future. We can think of the value function as compiling information about the value of future actions, under a given policy, into the value estimate of a state-action.

E.5 State Aggregation

In appendix D we saw that generalisation is an important, even essential, part of learning, and that one way to generalise was to aggregate state-actions. In this section we'll examine the interaction of aggregation and the prediction problem, and consider a non-adaptive form of aggregation. Classifier systems are related to this approach in that classifiers are aggregations.

E.5.1 Fixed and Adaptive Aggregation Schemes

So far we've assumed the prediction methods in this appendix have been implemented using look-up tables, with one value estimate stored for each state-action. A simple way to add (non-adaptive) state aggregation to this approach is to expand the table to include entries for aggregations of state-actions. That is, the value of an entire aggregation is estimated, and its estimate occupies a cell in the table. We assume the mapping from the table to the aggregations is fixed – otherwise we would not consider this a tabular method.

An issue which obviously needs to be dealt with is the selection of aggregations to use. A poor choice will result in poor performance, and, because the aggregations are fixed, there is no hope of improvement through learning. Domain knowledge and trial and error experimentation may be required to find suitable aggregations for the task at hand, difficulties which make adaptive state-aggregation methods more attractive.

AE-tables

One alternative to adaptive aggregation and to hand-crafted aggregation is to choose aggregations at random. Another alternative is to include all possible aggregations in the table. We'll call this approach *Aggregation Enumeration*, and refer to tables of exhaustively enumerated aggregations as *AE-tables*. We can use AE-tables with any prediction method, e.g., Sarsa or other TD updates, Monte Carlo updates, techniques from Dynamic Programming, and, for non-sequential tasks, averaging rewards.

In normal tabular implementations, the table maps $\mathcal{S} \times \mathcal{A} \rightarrow \Re$. AE-tables, in contrast, map $\mathcal{G} \times \mathcal{A} \rightarrow \Re$, where $\mathcal{G}$ is the generalisation space defined as the power set of the state space in §D.4. Thus AE-tables enumerate state-aggregation-action space rather than state-action space.

Although this way of aggregating states is non-adaptive, it essentially avoids the issue of which aggregations to use by using all possible aggregations in $\mathcal{G}$ simultaneously. AE-tables make optimal use of interactions with the environment in the sense that each input updates all possible relevant aggregations simultaneously.

Scalability of AE-tables

The limiting factor of AE-table approach is that it scales particularly badly because generalisation space grows much more quickly than state space (see §D.4). This growth means that for most problems unreasonable amounts of memory are required to store the AE-table, and, on serial machines, unreasonable amounts of time are required to perform updates on the table.

AE-tables scale so badly that they cannot be used except for tasks with the smallest of state spaces. We can obtain some improvement in scaling by using some restricted subset of aggregations, e.g., only those aggregations possible using the ternary classifier systems language (see chapter 2), but even then AE-tables can still only be applied to very small state spaces.

However, even if AE-tables are impractical, we can use them as a benchmark for state-aggregation RL systems (on tasks with small state spaces) because of their very efficient use of interactions with the environment. Also, because AE-tables are conceptually simple, they may clarify the issues involved in prediction and control with aggregations in the way non-aggregating tabular representations clarify the issues involved in, e.g., Q-learning.

E.5.2 The Value of Aggregations I: Return

§E.1 defined the value of an individual state-action as its expected return, which, in the sequential case, turns out to depend on the policy being followed (§E.4.2, equation (E.10)). This definition holds for aggregations of state-actions, and we can write the true value of an aggregation as:

$$Q^*(g) = E_\pi\Big\{R_t \;\Big|\; (s_t, a_t) \in g\Big\} \tag{E.18}$$

where $g \in \mathcal{G}$ is an aggregation over state-actions.

§E.4.3 ordered entire policies based on the values of their state-actions. Since policies are simply aggregations of states and actions, when g is an entire policy equation (E.18) defines the value of an entire policy.

E.5.3 The Value of Aggregations II: Predictive Utility

Clearly if all possible aggregations are maintained, many will normally be inaccurate. Although we do not remove inaccurate aggregations from an AE-table, we can simply ignore their predictions, that is, if we can identify them as inaccurate. In other words, even though we do not adapt the aggregations, we still need to evaluate them.

Recall from §D.3 that we illustrated the problem of state aggregation by considering the supervised learning problem of assigning elements of a set to classes. In this appendix we're concerned with the problem of estimating the value of state-actions, so we put two state-actions in a class if they have exactly the same value. If the values of two state-actions are identical, aggregating them has no detrimental effect on value prediction.

Accuracy

We can aggregate states which have different values, in which case the accuracy of an aggregation is determined by how close the true values of the aggregated state-actions are. That is, aggregating state-actions with less similar values will make for worse value predictions. In some cases the benefits of aggregation will outweigh the loss in predictive accuracy.

Another reason why aggregating state-actions with non-identical values may be useful is that because we don't have access to the true values of state-actions and are using estimates, even if two state-actions have the same true value their estimates are likely to differ. The fact that we are using estimates may disguise the similarity of the true values.

Estimating Accuracy

Normally we do not have access to the true values of state-actions, and, in general, we do not even have direct estimates of the individual state-actions which form an aggregation. This means we must evaluate the accuracy of an aggregation by looking only at the value estimate for the entire aggregation.

We can do so by recognising that the value of an aggregation g of state-actions x and y is a function of x's value and y's value. That is, g's value will be updated towards the values of x and y. We can estimate the difference in the values of x and y (and so the accuracy of g) by observing the variance in the value of g: the higher g's variance, the more dissimilar the values of x and y are likely to be. XCS's accuracy calculation (§2.3.5) is actually an example of this approach.

Unfortunately this approach is complicated by the fact that variance in g's value may be due to other sources, e.g., the bias due to an initial estimate, changes in the values of successor estimates in TD learning, differences in sequences of rewards due to taking different paths through the state space and/or simply to stochastic rewards. A particular complication is that the effect of all these factors may vary from aggregation to aggregation, meaning we cannot simply ignore them by setting a fixed threshold on the variance, below which we ignore it. (See §5.8 for an example.)

E.6 Storing Value Estimates

Until §E.5 we assumed that a table enumerating all state-actions was maintained, so that each state-action had an explicit estimate of its value. While this approach is conceptually simple it is certainly not the only one. In this section we see some alternatives.

E.6.1 Storing Estimates of Aggregations

A Single Estimator for a Policy

In §E.5 we saw that we could directly estimate the value of an aggregation of state-actions. Aggregations can be of different sizes, and in the extreme case can form an entire policy, in which case an estimate of the aggregate is an estimate of the value of the entire policy. This can be stored as a single numeric value – effectively a table of size 1. This is perfectly feasible, and, in fact, we'll see an example of this approach in §G.6.

Covering of the State-action Space by Aggregations

In general we may have a number of aggregations and maintain a table with an estimate of the value of each. The aggregations may form a partition of the state-action space or they may overlap. They may cover the entire state-action space, or may cover it only sparsely.

Estimator Density and Choice of Update

We can think of the density of value estimates as one dimension along which a learning system can vary, and the TD-MC balance of the updates as another. Many other dimensions exist as well.

These dimensions are not entirely independent: the sparseness of estimators affects which methods we can use. With a single value estimate for an entire policy we are restricted to pure MC estimation, since there are no other estimators available to use in temporal differencing. More generally, the fewer the estimates we store the more MC our updates must be, since TD updates can be employed only to the extent that successor estimates can be found.

E.6.2 Sparse Estimators, Models and Search

Given a model of the environment in the form of $\mathcal{R}$ and $\mathcal{P}$ (§C.5.1), or even in the form of a simulation, we can search through possible sequences of actions and state transitions. This allows us to store estimators sparsely in the state-action space, and to search from the current state-action to the nearest estimator(s), whose value(s) can then be used to estimate that of the current state. The ability to compute values from nearby ones as needed means we can use TD updates flexibly, despite the sparseness of our estimators.

There are many ways to distribute (and redistribute) estimators through the state-action space.

E.6.3 Function Approximators

We can train any of a wide range of supervised learning function approximators (e.g., neural networks) to approximate the value estimates we would otherwise store in a table by using each update as a training instance for the function approximator. This approach is frequently used in the RL literature (and has even been given a name: "Neuro-Dynamic Programming" [23]).

The use of a function approximator itself provides generalisation, meaning this approach avoids the need to work with aggregations of state-actions.

E.7 Summary

We've seen Monte Carlo and Temporal Difference approaches to evaluating state-actions, and that we can determine the value of aggregations of state-actions, and entire policies, by considering the values of their component state-actions. We've also seen that estimators need not be restricted to individual state-actions – that we can directly estimate the value of aggregations and entire policies. Finally, we looked briefly at some alternatives to the simple table of state-action estimators we'd earlier assumed.

In the next two appendices we'll see how to incorporate these methods into control algorithms which can solve RL tasks by evaluating an existing policy and then finding an improved one based on it.

Appendix F

Generalised Policy Iteration Algorithms

Appendix Outline

This appendix is the first to introduce methods which can solve the RL tasks introduced in appendix C. Here we review methods of Generalised Policy Iteration, a subset of which are better known simply as "Reinforcement Learning Algorithms". GPI methods combine the policy evaluation methods of appendix E with an idea called policy improvement. Together these methods allow iterative approximation of optimal control policies.

In appendix E we saw methods for evaluating actions and policies, that is, methods which allow us to *predict* the return for an action or policy. We turn now to the issue of *control*, that is, to the problem of finding a policy which maximises return.

An alternative, and more conventional, title for this appendix would be "Reinforcement Learning Algorithms", which is the name normally used for many of the Generalised Policy Iteration (GPI) algorithms introduced here. Instead, they are referred to herein as GPI algorithms in order to distinguish them from Evolutionary Algorithms, which are themselves a form of RL algorithm (see definition §2, page 245). The term "GPI algorithm" is intended to encompass both sequential and non-sequential methods, both action-value

and state-value functions (see §F.4), and methods which incorporate any of the prediction methods – Dynamic Programming, Monte Carlo and Temporal Difference methods – from appendix E. Referring to GPI methods emphasises their defining feature, one which sets them apart from evolutionary methods, and which thus seems an appropriate choice, even if its use runs counter to convention. This deviancy is necessitated by the unusual range of systems with which the thesis deals: most discussions are concerned only with either GPI methods, or EAs, or, most commonly, some subset of either. For introductory material on GPI methods see [265, 184, 220, 235, 3]. Kaelbling et al. [141] provide a good survey for those already familiar with machine learning.

F.1 Policy Improvement

In appendix E we saw various methods for obtaining an action-value function for a policy. Now we see how to go in the opposite direction: how to obtain a policy from an action-value function. The purpose of this exercise is to obtain a better policy than the one to which the action-value function belongs, and hence the process of evaluating a policy π and obtaining another π' from its value function is called *policy improvement*. We can illustrate policy improvement this way:

$$\pi \xrightarrow{\text{Evaluation}} Q(s,a) \xrightarrow{\text{Improvement}} \pi'$$

Comparing state-actions The goal of any RL agent is to maximise return, and a state-action value $Q(s,a)$ is an estimate of the return which will come from taking a in s and (in the sequential case) following a policy π thereafter. The agent's best guess at maximising return in s is simply to take the action $a \in \mathcal{A}(s)$ with the highest $Q(s,a)$. That is, the agent should act greedily with respect to its knowledge of the available actions $a \in \mathcal{A}(s)$.

An action-value function may suggest improvements Having evaluated the state-action value function Q^π for a policy π, we often find that the greedy action in a state s is not the action recommended by π. This suggests that we can obtain an improved policy by creating a policy π' which is identical to π except in s, where it takes the greedy action. That is,

$$\pi'(s) = \arg\max_a Q(s,a)$$

where $\arg\max_a$ denotes the argument $a \in \mathcal{A}(s)$ which maximises the term which follows it.

The greedy policy Just as we can improve a policy in a given state, so we can improve it in all states simultaneously. To do so yields the policy π' which is greedy with respect to Q^π. The new policy π' will be better than π except when both are *optimal policies*.

Exploration Of course the explore/exploit dilemma (§C.2) is everpresent, but we can address it by modifying our greedy policy using some form of exploration policy (§C.5.2).

F.2 Optimal Policies

Recall from appendix E that being able to evaluate policies allows us to rank them, and that we used equation (E.11) to define partial orderings over policies. For any problem, there is always at least one policy which is better than or equal to all other policies in all states. Any such policy (there may be more than one) is called an *optimal policy*, denoted π^*. All optimal policies share the same optimal action-value function Q^* defined as:

$$Q^* = \max_\pi Q^\pi(s,a) \tag{F.1}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. An optimal policy is one which acts greedily with respect to Q^*:

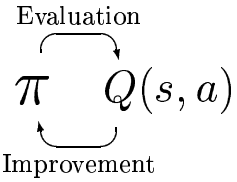
$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{F.2}$$

for all $s \in \mathcal{S}$. In other words, an optimal policy is one which takes the highest valued state-action in all states. (There are multiple optimal policies when multiple actions maximise a given $Q(s)$.)

F.3 Generalised Policy Iteration

Having obtained an improved policy from a value function, the obvious next step is to evaluate the new policy and reiterate policy improvement. This

leads to cycles of policy evaluation and policy improvement which Sutton and Barto [265] term *Generalised Policy Iteration* (GPI). We can illustrate GPI this way:



Stopping condition GPI proceeds until the greedy policy π' is no better than the policy π which generated the most recent action-value function, in which case an optimal policy has been found.

F.3.1 How Well must we Evaluate a Policy?

If we use a table to store the $Q(s,a)$, visit state-actions infinitely many times, and decline α appropriately, the $Q(s,a)$ are guaranteed to converge to their true values under a range of prediction methods from appendix E. While convergence proofs of this sort afford some confidence that a prediction method is working towards the right target, we cannot wait infinitely before switching to policy improvement. In practice, the policy is only evaluated until changes in the action-value function fall below some threshold.

How well must we evaluate a policy before performing policy improvement? Happily, policy improvement can take place after any change to the action-value function. In fact, in terms of computational effort it is often more efficient to only roughly evaluate a policy before improving it. In constructing a greedy policy we don't need to know exactly how good or bad a state-action is, only how it compares to the alternatives. In fact, many similar action-value functions have identical greedy policies. It is only the ranking of the $Q(s,a)$ which matters, not their exact values, since we simply take the action with the highest value. Also, an improvement in one state is better than no improvement at all, so it is useful to improve a policy of which only a subset has been evaluated. Having said this, convergence rates may be optimal when somewhat more than this minimum evaluation is done.

Creating Control Algorithms from Prediction Algorithms

To obtain a control algorithm we can use a prediction algorithm such as Q-learning or Sarsa and, on each time step, choose an action which is greedy

For each $s \in \mathcal{S},\ a \in \mathcal{A}(s)$
    $Q(s,a) \leftarrow$ arbitrary number // *Initialisation*
For each episode:
    $s \leftarrow initialState()$
    For each time step of current episode:
        Choose an action a greedily from $Q(s,a)$ // *Policy improvement*
        $environment(a) \rightarrow (r, s')$
        Perform evaluation update, e.g., for Q-learning:
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$

Figure F.1: A generic form of GPI control algorithm.

with respect to the current action-value function. That is, on each time step we perform policy improvement for the state we are currently in. Of course we will normally moderate this greediness with our exploration policy, which will sometimes result in a non-greedy action being selected. A generic algorithm of this form is shown in figure F.1.

F.3.2 Convergence Properties of GPI Control Algorithms

GPI control algorithms will converge to optimal policies and their true action-value functions given infinite revisits to every state-action, the policy converges to the greedy policy, and appropriately declined α.

F.3.3 Initialising Value Functions

Since policy improvement normally produces an improved policy π' which differs only slightly from the old policy π, it is possible to use the value function for π as the initial value function for π' at the start of policy evaluation. This is an optimisation since these initial values should be a better approximation of the true values than random or uniform values.

F.3.4 What Characterises GPI Algorithms?

Note the following aspect of GPI:

- Policy improvement *explicitly* compares *all* the actions for a state, and uses the action with the highest $Q(s,a)$ in the new policy.

This point is important because in appendix G we'll see evolutionary methods which do not explicitly compare value estimates. This difference is the basis upon which the two approaches to RL are distinguished in this work, and the reason for which the term "GPI algorithm" is used for the first approach.

F.4 State-value Functions

So far we have dealt with action-value functions exclusively, but an alternative approach is possible. *State-value functions* are just like action-value functions, except that they map states, rather than state-actions, to an estimate of their value, i.e., $\mathcal{S} \to \Re$ rather than $\mathcal{S} \times \mathcal{A} \to \Re$. A state's value $V^\pi(s)$ is the value of its most valuable state-action:

$$V^\pi(s) = \max_{a \in \mathcal{A}(s)} Q^\pi(s,a) \tag{F.3}$$

We can use any of the general prediction methods of appendix E to evaluate state-value functions, and use policy improvement to obtain better policies from them.

Disadvantages A difficulty arises in using state-value functions for control in that the agent must know something about the effect of its actions on state transitions in order to select the action which is most likely to bring it to the highest valued state. In other words, using state-value functions requires knowledge of the environment's state transition function $\mathcal{P}$ (§C.5.1), or an estimate of it, whereas using action-values does not.

State-value functions are of no use in non-sequential RL tasks, as the agent is unable to direct itself towards high-valued states, and state-value functions do not tell it which action to take in the current state.

Advantages An advantage of state-value functions is that only $|\mathcal{S}|$ value estimates need be maintained, rather than the $\Sigma_{s \in S} |\mathcal{A}(s)|$ estimates needed for action-value functions. As discussed in appendix D, this not only reduces memory requirements, but reduces the number of updates required in learning, although the savings may be small.

F.5 Summary

Given an action-value function it is easy to obtain a policy: all we need to do is act greedily with respect to the action-value function. That is, our

best guess at maximising return is simply, in each state, to take the action for which we estimate return is highest. We call this policy improvement because the policy obtained will improve on the policy which defined the action-value function (except when both policies are optimal).

The simplicity of obtaining a policy from an action-value function means the *true* action-value function for a task is effectively an optimal solution to it, in that the agent can behave optimally simply by acting greedily. In other words, it can choose the next action in the sequence of optimal actions without searching through sequences of possible actions. We can think of an action-value function as compiling information about the value of an entire sequence of actions into individual action-values. Once the true action-values are known the sequential aspect of the task is no longer a problem.

Appendix G

Evolutionary Algorithms

Appendix Outline

This appendix, following the GPI methods of appendix F, introduces a second form of control method which can solve RL tasks, called Evolutionary Algorithms. We begin with a general introduction to Evolutionary Algorithms, see how they can be applied to function optimisation problems and the generalisation problem of appendix D and review some adaptations which can make them more suitable for the latter. Next we see how EAs can be applied to RL tasks, first non-sequential and then sequential tasks. Evolutionary learning of RL tasks is incorporated into the RL framework of appendix C, and the use of predictions methods with EAs is discussed. Finally, there is a brief comparison of GPI and evolutionary methods for RL.

Evolutionary Algorithms (EAs) are search methods inspired by evolution in nature. A number of general forms of EA exist, including Genetic Algorithms (GAs), Evolution Strategies, Evolutionary Programming, and Genetic Programming, each of which has been instantiated in many ways. Because this thesis is concerned with learning classifier systems, we'll focus in this appendix on GAs, a popular form of EA which is found as the evolutionary component of most classifier systems. For introductory material on GAs the reader is referred to [97, 19, 20, 180, 182, 316, 184, 112]. An impressive survey of EAs and their applications can be found in [8].

G.1 Evolution

Just as in nature populations of organisms evolve over many generations to adapt to their environment, in an EA populations of candidate solutions evolve to adapt to the requirements of some problem we wish to solve. EAs have innumerable applications falling into diverse areas including planning, design, simulation and identification, control and classification ([8]). To choose just a few examples from among thousands, EAs have been used for structural design of aircraft [41], to find the topology and set the weights in neural networks [321], to find control rules for simulated organisms [30] and a six-legged robotic insect [253], to find the three dimensional structure of a protein given its amino acid sequence [174], to find novel fighter aircraft manoeuvres from combat simulation [246], to segment cardiac images [202] and, finally, to set the timetable for a set of examinations [59].

Diversity and Fitness

To return to the analogies with nature, EAs are population-based algorithms, in which, just as in nature, the diversity in the population means that, for whatever reason, some of its members are better suited to the environment than others. Some may have greater strength, a greater wingspan, an immunity to a virus, or any combination of myriad factors which affect their ability to survive and reproduce. This ability to reproduce is termed the *fitness* of an individual. In nature it is "calculated" through a creature's lifetime of interaction with the world around it: to the extent it reproduces it is fit. In simulating evolution, we must define how the EA calculates an individual's fitness ourselves, in such a way as to encourage the evolution of individuals with the properties we seek.

Fitness and Reproduction

The evolution of fitness-enhancing traits occurs simply because fitter individuals are more likely to reproduce, and thus contribute their genetic material (and hence their genetically influenced traits) to the next generation. Any trait which helps an individual reproduce is more likely to appear in the next generation than it would be if it did not help. This tendency for fitter individuals to reproduce more than less fit individuals, whether in nature or in simulation, is known as *selection*, and gives rise to Darwin's famous phrase "survival of the fittest". That is, fitter individuals are selected, by nature or by the EA, to contribute more to the next generation.

The Origins of Diversity

Diversity results from transcription errors during the reproductive process (*mutation*). In species which reproduce sexually, additional variation is due to the mixing of genes from multiple parents (*recombination*). Selective pressure imposes a bias towards parents having advantageous traits, although not necessarily the same ones. If offspring inherit from multiple parents, some will combine the best features of their parents and be fitter still.

Evolution is Stochastic

Of course there are no guarantees in life: an individual with above-average fitness may, simply through bad luck, fail to reproduce. However, we would expect, by definition, its chances to be better than average. Consequently, although the selection of advantageous traits for propagation to successive generations is not guaranteed, it is probable. Evolution is a stochastic process.

Artificial Evolution

Evolutionary algorithms are not far removed from the ancient practice of breeding plants and animals to develop desired traits. Suppose a breeder wants to breed bigger dogs with thicker coats. From the dogs available he selects the biggest and those with the thickest coats and mates them. Hopefully some of their offspring will be bigger still and have yet thicker coats than their parents. From this new generation he again selects those with the desired traits and mates them. After many generations he is very likely to obtain dogs with the traits he is after. Note that in breeding, the breeder's artificial selective pressure has largely replaced nature's. In EAs exactly the same principles are at work, except with vastly less complicated individuals and on a far smaller time scale. Since the selection of parents for breeding (and the rest of the EA) almost always runs without human intervention many generations can often be created each second. The speed of this process is only limited by the computing power available, and, happily, EAs are amenable to various models of parallel implementation. It should be emphasised that EAs are far simpler than evolution in nature, both in terms of the individuals evolved, and also in terms of the genetic operators used. Nonetheless EAs and natural evolution operate using many of the same fundamental principles.

Evolution is Blind

Notice that our dog breeder needs absolutely no idea of the mechanisms of inheritance (other than mating), how genes are expressed in fully developed animals, the biochemistry involved or any other of these very complicated subjects. Similarly, EAs are most useful when knowledge of the domain is limited and when analytical solutions are unavailable. Put simply, we don't need to know how or why something works in order to encourage evolution to produce it. Evolution encourages us to treat individuals as black boxes.

G.2 Elements of EAs

EAs incorporate the following abstractions of Darwinian evolution:

- A population of competing individuals.
- Evaluation of the fitness of individuals.
- Fitness-biased probabilistic selection of individuals for reproduction.
- Probabilistic variation of offspring based on parents.

The individuals in question represent candidate solutions to the problem of interest, or to parts of it. By analogy with nature they are often referred to as *chromosomes*. The second step is performed by a *fitness function* which is passed a chromosome and returns the chromosome's fitness (a number representing the value of the individual). Parents are then selected and *genetic operators* applied to transform them into offspring.

G.2.1 A Generic EA

Perhaps the best way to introduce evolutionary algorithms is by example. Figure G.1 shows a very abstract generic evolutionary algorithm. The algorithm is not detailed enough to allow implementation without filling in some missing details, in the process of which it which could be extended into any of the EA paradigms previously mentioned (GAs, GP etc.).

We'll focus on the canonical fixed-length binary bitstring GA as it is the most relevant to this thesis. A GA is still general enough to be independent of the problem to which it is applied, and the representation used. Applying a GA to a particular problem is largely a matter of providing suitable encodings for individuals, a suitable fitness function and suitable genetic operators, examples of which we'll see shortly.

```
randomly initialise population with n individuals

repeat until stopping conditions met
    evaluate fitness of each unevaluated individual
    probabilistically select parents based on fitness
    generate n offspring by probabilistically transforming parents
    remove n individuals to make room for the new offspring
endrepeat
```

Figure G.1: A generic evolutionary algorithm.

G.2.2 Population-based Search

That EAs rely on populations of individuals is an important point, one which sets them apart from otherwise similar algorithms like simulated annealing. Members of the population are not independent; they interact by exchanging information when they mate.

A population of n interacting individuals is quite different from a series of n independent trials with a single individual. The hope is that, by helping each other, a population of individuals will more efficiently find a solution.

G.2.3 Fitness Functions

The role of the fitness function is to evaluate each individual in a population, in preparation for the selection of parents. In nature, the fitness function's role is played by an organism's environment, which tests its ability to survive and reproduce. The fitness function is one aspect of an EA that must be tailored to the problem at hand, since it determines which qualities are valued and will tend to emerge.

One interesting form of fitness function is when the chromosomes are used to construct simulated animals (called *animats* [293]), which must survive in a simulated environment. In this case the fitness function is implicit in the environmental dynamics, just as in nature.

There is a great diversity in the ways in which chromosomes code solutions to problems. In principle any combination of data and data structure can be operated on by EA. Most commonly, however, individuals consist of floating point numbers, or strings of binary bits, or higher level symbols. In many EAs individuals consist of fixed-length vectors, but in others variable length vectors, lists, trees or graphs are used.

In the canonical GA chromosomes are strings of binary bits. These strings may be decoded and interpreted in any way, or the fitness function may act

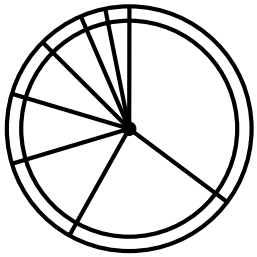


Figure G.2: A roulette wheel with fitness-proportional slot sizes.

directly on them. For example, in the famous *learning tasks!1-max* problem, the fitness of a string is simply the number of 1s it contains. Strings with more 1s will be fitter and tend to reproduce more; eventually a string consisting entirely of 1s will be found.

The fitness function is where we define the goal of the evolutionary process. If we wanted to find a string consisting entirely of 0s, we could define fitness as the number of 0s. Of course these examples have rather trivial optimal solutions. But consider the case of an animat in a simulated world. It may be far from trivial to find the optimal solution there, even if it is relatively easy to build the simulation. (See [265] p. 114 for an example.)

G.2.4 Probabilistic Selection of Parents

Once the fitness of each individual is known, selective pressure can be exerted on the population by selecting individuals for reproduction based on their fitness. This can be achieved in many ways, perhaps the most conceptually straightforward of which is *roulette wheel* selection. Following this method, we construct a conceptual roulette wheel, the circumference of which is equal to the summed fitness of the population members. Each individual is then allocated a slot on the wheel, the size of which is proportional to its fitness, i.e., fitter individuals get larger slots, as illustrated in figure G.2. To select a parent, we spin the wheel, drop a ball in and take the individual whose slot the ball comes to rest in. This process is modelled simply by generating a random number between 1 and the circumference of the wheel. Fitter individuals, with their larger slots, are more likely to be selected, although using this particular scheme none is guaranteed to be selected.

To select n parents we simply spin the wheel n times, with or without allowing already selected individuals to compete, as desired.

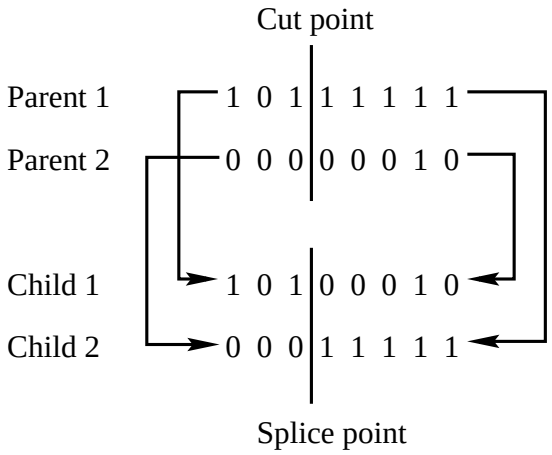


Figure G.3: Single point crossover.

G.2.5 Genetic Operators

Having selected a parent to reproduce, we can simply clone it, but this will not introduce any novel individuals. An essential principle of evolution is that the space of possible individuals be sampled by introducing novelty into the population. To do so we can mutate the clone. Simple *point mutation* for a binary bitstring works as follows: each bit in the chromosome is mutated to its complement (0 or 1) with some small probability.

This mutation introduces novelty into the population – it can generate new chromosomes never seen before. Additionally, it is guaranteed, given infinite time, to sample all points in the space of possible chromosomes, and thus to sample the optimal solution.

What mutation does not do, however, is combine information from different individuals, which is needed for population-based search. We can achieve this by spinning the roulette wheel twice to select two parents and applying *crossover*. A simple form is *single point crossover*, in which we choose a random point at which to split both chromosomes. Each parent then contributes one of the substrings thus formed to each of two offspring, as illustrated in figure G.3.

Crossover is generally applied to pairs of strings only with some probability. If crossover does not take place, and no mutation occurs, the offspring are clones of their parents.

G.2.6 Replacement

Normally an EA works with a fixed population size so generating new individuals means existing ones must be replaced. There are two replacement paradigms: *generational* and *steady-state* EAs. In the former, an entire generation is constructed by reproduction, and the entire previous generation is discarded. (Using *elitism* we allow some outstanding individuals from the previous generation to survive.)

With a steady-state EA, the offspring are inserted into the existing population, prior to which the same number of individuals are selected for deletion. (Typically we want to delete lower fitness individuals, and so may want to select them based on an *inverse* function of their fitness.) Thus, in a steady-state EA, selection occurs at two points: reproduction and deletion. A steady-state scheme has a less disruptive effect on the population, and allows more gradual evolution.

Generational and steady-state schemes are actually two extremes of a continuum; a steady-state EA can replace any number of individuals, and one which replaces 90% of the population during each reproductive event is not much different from a generational EA.

G.3 EAs as Search

If we think of EAs as search methods, then chromosomes are points in a *search space*, and EAs are parallel (population-based) search methods. Genetic operators are then search operators, which probabilistically take us from one point in space to another. The exact nature of the operators used is very important because they determine just how we can move in this space; with one operator two points may be far apart, while with another they may be adjacent. In fact, we can use the search operators to determine how points are connected in a representation of the space. That is, given a set of operators and a fitness function we can define a *fitness landscape* which the EA searches, an idea due to the biologist Sewell Wright [317]. Fitness landscapes are so named because their features resemble those of geographical landscapes, with hills, valleys, ridges, canyons and so on.

To illustrate a landscape, figure G.4 shows function $x^2+2y^2-0.3(\cos(3\pi x)\cdot \cos(4\pi y))+0.3$, from [24]. In this case adjacency is the landscape is simply adjacency along the real number line. (We could say the search operators increment and decrement x, y coordinates at some fine-grained resolution.) Although optimisation has thus far been framed as the problem of maximising fitness, the inverse problem of minimising fitness is in general equivalent, and the goal of this particular task is to find the lowest point in the landscape.

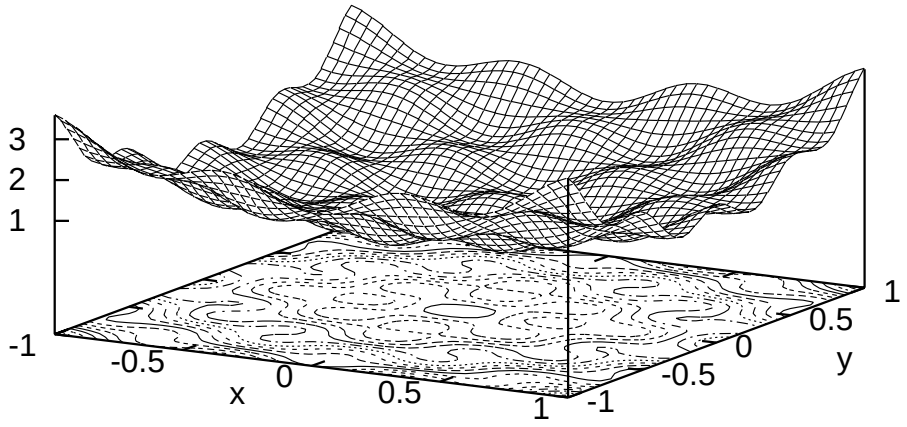


Figure G.4: A fitness landscape with a contour plot beneath it.

G.3.1 Local and Global Optima

Each individual occupies a particular point in the landscape and reproduction generates new points. Because adjacency in the landscape is defined by the search operators used, their transformations of parents tend to produce offspring which are nearby in the landscape. The idea is that over many generations the population will work its way by leaps and bounds from point to point down to the lowest point in the entire landscape. (Or, equivalently on other problems, the highest point.) This point is referred to as the *global optimum* (or *global maximum* or *global minimum*, as the case may be).

A pitfall in this search process is that it may get trapped in a *local optimum*, that is a valley (or peak) which is not the global optimum. That is, from a local optimum short movements in any direction produce worse solutions, but better solutions do exist somewhere. So-called *hill climbing* methods, which only consider moves to nearby points, are particularly susceptible to this problem. One of the advantages claimed for population-based search is that by sampling different parts of the landscape simultaneously they can better avoid local optima.

Although the landscape in figure G.4 contains many local optima it is comparatively simple, compared, for example, with a 10-dimensional equivalent.

G.4 The Generalisation Problem

We can easily apply EAs to the generalisation problem outlined in appendix D. All we need to do is to code chromosomes which represent aggregations over states, in which case the EA searches in generalisation space.

One problem with this approach is that the EA will tend to converge towards the global optimum. In many problems that is exactly what we want, but not here: recall from §D.3 that the goal of state aggregation involves assigning *all* states to some generalisation. The global optimum is the fittest generalisation $g \in \mathcal{G}$, but usually only covers a small subset of $\mathcal{S}$. We want a set of fit generalisations which collectively cover the *entire* space $\mathcal{S}$. To encourage this we can employ EA techniques like those introduced in the following section.

G.5 Niching and Mating Restriction

In some cases we wish to find not a single global optimum, but to find many optima in different parts of the fitness landscape. A fairly extreme case is the generalisation problem of appendix D in which we want to find and maintain enough optima to cover the entire state space. We can address such problems by encouraging the formation of specialised subpopulations. Even if we *are* seeking a single globally optimal point it may be helpful to encourage the formation of subpopulations in order to maintain genetic diversity and so continue searching in many regions in parallel.

To encourage the distribution of the population across the fitness landscape and the development of subpopulations we can use EA techniques such as *niching* and *mating restriction* [8]. We can divide niching techniques into *fitness sharing* and *crowding*.

G.5.1 Fitness Sharing

Fitness sharing techniques use the idea is that there are only fixed amounts of resources (fitness) available in any part of the environment, which must be shared between members in the same region. Consequently, the presence of other individuals in a region makes it less attractive, while less densely populated areas become more attractive. This has the effect of discouraging the population from converging to a common region.

Fitness sharing involve transforming an individual's "raw" fitness into a *shared fitness* based on some *similarity metric* which defines the distance

between any two individuals, and a *sharing function* which defines how to compute the shared fitness.

For the generalisation problem, we can define the similarity σ of two individuals as the number of states which are classified by both:

$$\sigma(i,j) = |i \cap j|$$

where i, j are any two individuals. A suitable generic sharing function would be:

$$sh(\sigma) = \begin{cases} 1 - (\frac{\sigma}{radius})^{\beta} & \text{if } \sigma < radius \\ 0 & \text{otherwise} \end{cases}$$

where $radius$ is the radius of the region around an individual within which it must share fitness with any others, and β is a scaling constant. This allows us to define:

$$f_{share}(i) = \frac{f_{raw}(i)}{\sum_{j=1}^{|\mathrm{P}|} sh(\sigma(i,j))}$$

where $f_{share}(i)$ is the shared fitness of i, $f_{raw}(i)$ is its raw fitness, and $|\mathrm{P}|$ is the size of the population.

If we introduce fitness sharing our EA becomes a *co-evolutionary* system, that is, one in which the fitness of an individual depends on the other individuals present in the population at the same time. In a co-evolutionary system the fitness landscape changes over time, which complicates both search and analysis.

G.5.2 Crowding

Where fitness sharing modifies fitness and hence the selection of parents, crowding techniques modify the selection of individuals to replace. They work by biasing this form of selection so that newly generated individuals are likely to replace similar existing ones. This reduces the chances of an individual in one region being replaced by one in another region, and thus the effects of one subpopulation (i.e., the individuals in one region) on another.

G.5.3 Mating Restriction

Mating restriction is a further class of techniques which can help distribute the population in the landscape. The idea is that only parents which are sufficiently similar, according to some similarity metric, are allowed to mate.

This restriction means the offspring produced are likely to be similar to both parents. Consequently, the offspring of one region will tend to compete with their parents and other similar individuals, rather than invade regions belonging to other individuals. Like crowding, mating restriction reduces the interactions between subpopulations.

G.6 RL with EAs

Unfortunately any account of the use of EAs for RL presented here is necessarily quite limited. A more complete treatment is that of Moriarty, Schultz and Grefenstette [187], which refers to Evolutionary Algorithms for Reinforcement Learning (EARL), which others have called simply Evolutionary Reinforcement Learning (ERL).

Terminology On page 273 use of the term "GPI algorithm" rather than the standard "RL algorithm" was justified on the basis that EAs are also RL algorithms, and that it is desirable to distinguish the two forms of RL algorithm. The assertion that EAs are RL algorithms is justified simply by definition: EAs are, by definition, reinforcement learners since they can solve reinforcement learning problems, including the full RL problem involving sequential decisions and balancing exploration and exploitation.

Equating fitness and reward In appendix C the RL paradigm was defined on the basis of the interaction between a learning agent and its environment, in which feedback to the learner consists only of a scalar value. This being the case, for EAs to be RL algorithms they must learn from scalar values, which they do: EAs learn from a fitness function. In other words, the fitness function is the reinforcement signal from which EAs learn.[1] We'll explore this equation of reward and fitness functions further in the following sections.

G.6.1 Non-associative RL with an EA

If EAs are RL algorithms, what kind are they? First let's consider the 1-max problem from §G.2.3, which we can fit into the RL framework of states,

[1] Russell and Norvig ([220] p. 619) consider EAs as RL methods on the basis that offspring are rewards. This is a similar view, since offspring are allocated on the basis of fitness. See also [188] p. 12.

actions and rewards by imagining a dialogue such as this:

```
Time Step: 1
Environment: You are in state 1.
Learner: I'll take action 00100111.
Environment: Your reward is 4.

Time Step: 2
Environment: You are in state 1.
Learner: I'll take action 11010011.
Environment: Your reward is 5.

Time Step: 3 ...
```

1-max is non-sequential, so we've assumed the state is always 1. Consequently we (and the EA) can ignore it. The actions taken by the agent are chromosomes in the population, evaluated one after another in any order since the problem is non-sequential. The reward is the fitness assigned to each chromosome. Once all chromosomes in the current population have been evaluated the EA can generate new ones and have them evaluated.

Definition of fitness The goal of the EA is to evolve the fittest chromosome it can, that is, to maximise the fitness it receives on a given time step. Since we've equated fitness with reward, the EA's goal is to maximise reward, and since 1-max is non-sequential we can use the equation for non-sequential returns (equation (C.1)) as our definition of return. More generally,

$$F(\pi) = E_\pi\Big\{R\Big\} \tag{G.1}$$

where $F(\pi)$ is the fitness of a policy (or more generally a chromosome) π.

What kind of evaluation method are we using? Appendix E is all about methods of estimating the value of state-actions and aggregations of state-actions. How does the evaluation of chromosomes for the 1-max problem fit into appendix E's framework? It is particularly easy to evaluate chromosomes in 1-max, partly because the fitness function is deterministic. Suppose, on the other hand, that the fitness function sampled some distribution of fitnesses around an expected value. Then a single invocation of the fitness function might not give us a good idea of the value of a chromosome, but we could obtain better estimates by averaging the results of multiple calls to the fitness function (for the same chromosome).

Since the non-sequential return is appropriate for 1-max, averaging returns means averaging complete returns. In other words we are performing Monte Carlo evaluation (albeit a degenerate form, since we're just averaging rewards).

How well must we evaluate a chromosome? Note that although improved evaluation of fitness is desirable, evolutionary processes, being stochastic themselves, are robust to noise and that precise evaluation of the chromosomes is not necessary. In fact, optimal use of computing resources may call for rather approximate evaluation of each policy.

Formalising Non-Associative RL for an EA Applying a simple EA to 1-max fits nicely into the RL framework developed in appendices C and F. In our RL notation, $\mathcal{S}$ consists of only 1 state, and the only issue is which $a \in \mathcal{A}(s)$ to choose to maximise immediate reward. That is, the EA does not associate actions with any given state; in RL terms it is said to be *non-associative* [265].

G.6.2 Associative RL with an EA

Solving the Maze Problem with an EA

Representing Solutions Now that we've seen an EA solve a simple non-associative, non-sequential problem let's look at the maze problem from §E.4.6, which is both sequential and requires association of actions with states. An obvious way to evolve paths through the maze, though by no means the only way, is to code a complete solution in each chromosome. Each position in the chromosome will code the action to take in a given state in the maze. To simplify matters, instead of binary strings let's use chromosomes which are strings of the values $\{N, S, E, W\}$, for North, South, East and West. Now each chromosome is a policy, mapping from $\mathcal{S} \to \mathcal{A}$. The genetic operators produce new policies, so we're searching directly in the space of complete policies. There are $|\mathcal{A}(s)|^{|\mathcal{S}|} = 4^9 = 262,144$ different policies for this maze, only one of which is optimal.

Evaluating fitness To evaluate the fitness of a particular chromosome we can simulate moving through the maze, taking note of the rewards obtained on each step of the trial. The chromosome's fitness can then be some function of these rewards. Because the environment is stochastic a chromosome may not obtain exactly the same rewards each time it runs through the maze, and to compensate for this we may wish to run it through many times and

average the results in some way. We'll return to the subject of precisely how to evaluate a chromosome in §G.6.3.

Interfacing chromosomes and environments Note that we need to add some extra machinery to interface the chromosomes with the maze environment: we need some code which detects the current environmental state and looks up the specified action in the appropriate part of the chromosome. This additional machinery is not complex but it is essential.

Making EAs Associative

Fixed Associations If GAs are non-associative, how is it that they can be applied to the maze problem? In this problem there are 9 states, and each needs an action associated with it. Our solution was to evolve complete policies (coding an action for each state), to store the actions in a certain order, and to add some extra machinery to decode the chromosomes as needed based on the order used to store them. It is this machinery which solves the problem of associating states and actions for the GA.

Adaptive Associations Another way to get an EA to perform associative RL is to code the states to which an action applies within the chromosome itself, in which case the EA can modify the mapping from states to actions. This is the approach taken by the Learning Classifier Systems introduced in chapter 2. However, this approach also requires some extra machinery to decode the chromosomes and apply the actions at the correct time. In both cases it is the additional machinery, not some fundamental part of the EA, which solves the problem of associating states with actions. This view is consonant with the statement in §G.6.1 that EAs are non-associative algorithms.

Formalising Associative RL for an EA Normally we would find it convenient to describe a problem such as the maze in terms of a set of states $\mathcal{S}$ and a set of actions available in a given state $a \in \mathcal{A}(s)$ (as outlined in §C.5). For example, a dialogue for an EA in the maze problem could include different states. This makes sense when considering the maze from the perspective of the entire learning system, that is the EA *plus* the machinery which associates states and actions.

If, on the other hand, we look at the maze strictly from the EA's point of view, we should describe it as having only a single state $|\mathcal{S}| = 1$, since the EA – on its own – does not distinguish states. To represent the environment

from this point of view we can collapse all states into a single one, that is, $|\mathcal{S}| = 1$. This single state has an action for each state-action in the alternative associative framework, i.e., $|\mathcal{A}(x)| = \sum_{s \in \mathcal{S}} |\mathcal{A}(s)|$, where x is the single state in the non-associative representation of the environment.

G.6.3 Sequential RL with an EA

The maze example requires not only association of states and actions, but consideration of the long-term consequences of actions. We evolved complete policies, which we evaluated using some function of the rewards observed when using them.

Formulating return What function of the rewards should we use? In general, in sequential decision tasks fitness is defined as some (long-term) return such as the sum of rewards (equation (C.2)) or sum of discounted rewards (equation (C.2)). In either case the general equality of fitness and return (equation (G.1)) holds, although in practice we may only crudely estimate the true return (fitness) of a chromosome.

Storing estimates of value The simplest approach is to associate an estimate of return with the chromosome, that is, with an entire policy. Although it was not discussed there, this is the approach we took with 1-max.

Evaluation Methods for EAs

The standard approach to evaluating fitness in a sequential task seems to be to sum the rewards during an episode, and to average these returns over many episodes. This is a matter of averaging complete returns, and so is Monte Carlo evaluation. That EAs use Monte Carlo evaluation whereas GPI methods such as Q-learning have almost exclusively used TD methods is one difference between them. However, it need not be. Although the literature does not appear to contain any work using TD evaluation for the evolution of complete policies with fixed state associations (as in our solution to the maze problem), there is no reason why it cannot be done.

G.7 Comparing GPI and EA Methods for RL

Despite the great overlap in the problems to which GPI and EA methods can be applied there has been little comparison of them, which is attributable to the rather different approaches they take and to the fact that they have been developed within different research communities.

G.7.1 Similarities between GPI and EA Methods

Not only can EAs be applied to the same problems as GPI methods, they share certain features:

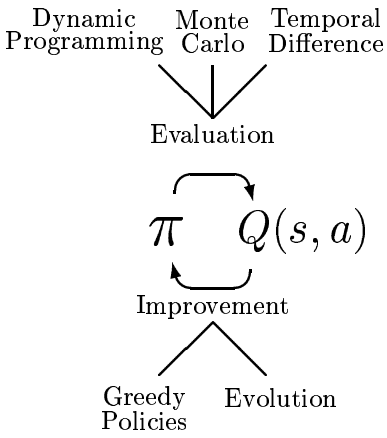
> "Both approaches assume limited knowledge of the underlying system and learn by experimenting with different policies and using reinforcement to alter those policies. Neither approach requires a precise mathematical model of the domain, and both may learn through direct interactions with the operational environment."
>
> Moriarty, Schultz & Grefenstette [187] p. 224.

> "The GA approach shares with RL the twin benefits of being based on training, not programming, and being responsive to changes in the environment through the possibility of online retraining."
>
> Singh, Norvig & Cohn [235] p. 8.

That they resemble each other is only natural, since it is their similarities which make them suitable for the same broad class of problems.

A Common Framework for GPI and Evolution

The Generalised Policy Iteration of §F.3 and evolution are broadly similar in that both consist of cycles of evaluation and improvement. In fact, both can use the same basic approaches to policy evaluation from appendix E, and both may only roughly evaluate policies before performing policy improvement. Despite the strong overall similarities of GPI and evolutionary methods, they differ fundamentally in how they perform policy improvement. To include both GPI and evolution we can refine the GPI diagram from page 275 to show the difference in policy improvement:



The addition of evolutionary computation to this figure should suggest potentially fruitful and largely unexplored combinations of EC and evaluation methods.

G.8 Summary

This appendix has introduced stochastic, population-based methods known as Evolutionary Algorithms, which we can use as a second elementary class of control method for RL problems. We've seen that EAs can be applied to the same problems as GPI methods, that they can use the same prediction methods, and that EAs and GPI methods have an overall resemblance in that they iterate evaluation and improvement phases.

Appendix H

The Origins of Sarsa

Sutton and Barto ([265] p. 159) attribute the Sarsa algorithm for value estimation (p. 265) to Rummery and Niranjan [219], although they mention that the bucket brigade evolved into a nearly identical algorithm [297].

In this appendix it is argued that neither invented the canonical tabular Sarsa, but that they independently developed very similar algorithms for non-tabular systems at roughly the same time. For reference we repeat the one-step tabular Sarsa update (E.16) from page 265:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\left[\overbrace{r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})}^{Target} - \overbrace{Q(s_t, a_t)}^{OldEstimate} \right]}_{Error} \quad \text{(H.1)}$$

In 1994 a connectionist version of this update was introduced by Rummery and Niranjan [219] who called it *Modified Connectionist Q-learning* (MCQ-L). Rummery acknowledges Wilson's work:

> "Wilson [297] noted the similarities between Q-learning and the bucket brigade classifier system [123]. Using this interpretation, the bucket brigade algorithm is equivalent [to] a TD(0) form of Modified Q-learning [Sarsa]." [218] p. 20, footnote 4.

About the same time Wilson [297] introduced a rule-based version for his ZCS classifier systems, which he referred to as an *implicit bucket brigade*, following Goldberg's introduction of the term [94] in reference to an earlier, related variant of the bucket brigade by Wilson in 1985 [292]. The name Sarsa first appeared when Rummery and Niranjan mentioned that Sutton suggested the name *SARSA* for their MCQ-L ([219] p. 6), and was next used by Sutton in [264].

H.1 Modified Connectionist Q-learning

Rummery and Niranjan ([219] p. 6) define MCQ-L as:

$$\Delta \mathbf{w}_t = \eta[r_t + \gamma Q_{t+1} - Q_t] \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla_w Q_k \tag{H.2}$$

where $\Delta \mathbf{w}_t$ is the change in the neural network weights $\mathbf{w}_t$, η is α (that is, the learning rate) in Sutton & Barto's notation, $\nabla_w Q_k$ is a vector of the output gradients calculated by backpropagation, and λ is the eligibility trace decay rate from the TD(λ) algorithm [265].

In short, MCQ-L is a connectionist version of tabular Sarsa(λ).

H.2 ZCS's Implicit Bucket Brigade

In Wilson's notation ([297][1] p. 6) ZCS's update is:

$$S_{[\mathrm{A}]} \leftarrow S_{[\mathrm{A}]} - \beta S_{[\mathrm{A}]} + \beta r_{imm} + \beta\gamma S_{[\mathrm{A}]'} \tag{H.3}$$

where $S_{[\mathrm{A}]}$ is the total strength of the members of the action set [A] at time t and $S_{[\mathrm{A}]'}$ that at time $t+1$. We can convert partly to Sutton and Barto's notation by replacing β with α and r_{imm} with r_t, yielding:

$$\begin{aligned} S_{[\mathrm{A}]} &\leftarrow S_{[\mathrm{A}]} - \alpha S_{[\mathrm{A}]} + \alpha r_t + \alpha\gamma S_{[\mathrm{A}]'} \\ &\leftarrow S_{[\mathrm{A}]} - \alpha\Big[r_t + \gamma S_{[\mathrm{A}]'} - S_{[\mathrm{A}]}\Big] \end{aligned} \tag{H.4}$$

Wilson performed an additional update ([297] p. 6), deducting a small tax τ from classifiers in [M] which are not in [A]. This measure was intended to make the system more likely to select the same action consistently in a given state, and is thus a form of exploration control and not part of the main update. Further, Wilson considered this technique provisional and suggested there are better approaches to exploration control ([297] p. 7).

ZCS's update (H.4) differs only from tabular one-step Sarsa (H.1) in that $S_{[\mathrm{A}]}$ is normally determined by many classifiers, whereas Sarsa's $Q(s,a)$ comes from exactly one table entry. Note that in the case where $|[\mathrm{A}]| = 1$ they are *identical*. In other words, ZCS's update is a generalisation of tabular one-step Sarsa.

[1] Page numbers for this reference refer to the electronic version.

H.3 Who Invented Sarsa?

Clearly Rummery and Niranjan on the one hand and Wilson on the other independently invented very similar versions of Sarsa at around the same time (1993/94), although neither of them gave the exact tabular version of the algorithm presented in [265] and p. 265.

Appendix I

Notation

The classifier systems and XCS notation presented below is a superset of that in [298, 53], while the reinforcement learning notation is from [265].

Mathematical Notation

$\arg\max_a F(a)$	The argument a which maximises $F(a)$.
$\max_a F(a)$	The maximum value returned by any argument a.
$\cup$	Set union operator.
$\cap$	Set intersection operator.
$\in$	Set membership operator: $x \in X$ means x is an element of X.
$\notin$	$x \notin X$ means x is not an element of X.
$\|\mathcal{S}\|$	The cardinality (i.e., size) of a set $\mathcal{S}$.
$2^{\mathcal{S}}$	The power set of a set $\mathcal{S}$.
$\forall$	Universal quantifier. $\forall o$ is read: "For all objects o ... ".
$\exists$	Existential quantifier. $\exists o$ is read: "There exists some object o ... ".
$\leftarrow$	Assignment operator.
$\rightarrow$	Assignment operator.
$a \approx b$	Indicates that a and b are approximately equal.
E	The expectation (i.e., population mean).
$\Re$	The set of real numbers.
∞	Infinity.
exp	The exponential function.
$\square$	Denotes the end of a proof.
σ^2	Variance.
$\sum$	Summation, e.g., the sum of a series of values X_1 to X_n is $\sum_{i=1}^{n} X_i$. We can also sum over the elements of a set, e.g., the sum of predictions in [A] is $\sum_{j \in [A]} p_j$.

General Classifier Systems Notation

[P]	The panmictic (i.e., global) set of classifiers in a classifier system (p. 31).
[M]	The match set; the set of classifiers matching the current environmental input (p. 36).
$[M]_{a_i}$	The subset of of [M] advocating action a_i (p. 36).
[A]	The action set; the subset of [M] advocating the currently selected action (p. 37).
$[A]_{-1}$	The previous time step's action set (p. 41).
$\|[A]\|$	The size of the action set.
[O]	The optimal set of classifiers for a given problem (p. 120).
%[O]	The proportion of the optimal set present in [P] (p. 123).
$\mathcal{P}$	Wilson's performance metric (p. 58).
[m-DNF]	The minimal DNF representation of a function (p. 127).
%[m-DNF]	The proportion of rules in an [m-DNF] representation present in [P] (p. 127).
$0\,0\,0 \rightarrow 1$	A classifier (p. 26). The symbol $\rightarrow$ separates the condition and action.
[PI]	The set of rules whose conditions are Prime Implicants for a function (p. 127).
%[PI]	The proportion of rules in [PI] present in [P] (p. 127).
τ	The tolerance for prediction error in an accuracy-based system (p. 146).

XCS Rule Update Notation

P	The target towards which the prediction of a rule is updated (p. 41).
$P(a_i)$	The system prediction for action a_i (p. 36).
r_t	The reward at time t.
p_j	The prediction of rule j (p. 42).
ε_j	The prediction error of rule j (p. 43).
κ_j	The accuracy of rule j (p. 43).
κ'_j	The relative accuracy of rule j (p. 44).
F_j	The fitness of rule j (p. 45).

XCS Parameter Notation

θ_{sub}	Threshold experience required for a rule to subsume another (p. 52).
θ_{GA}	Threshold used to control GA invocation (p. 50).
θ_{del}	Threshold experience above which a rule's deletion probability may increase using t3 deletion (p. 51).
θ_{mna}	Threshold number of rules in [M] below which covering occurs (p. 46).
δ	Threshold controlling how unfit a rule must be for its deletion probability to increase (p. 51).
N	The population size limit in *micro*classifiers (p. 31).
β	The learning rate (p. 42).
α	Constant controlling the rate of decline in accuracy when ε_o is exceeded (p. 43).
v	Another constant controlling the rate of decline in accuracy when ε_o is exceeded (p. 43).
ε_o	The accuracy criterion (a constant controlling the tolerance for prediction error p. 43).
γ	The discount rate (p. 41).
χ	The probability of applying crossover to two parents (p. 48).
μ	The probability of applying mutation to a bit in a condition or action (p. 48).
μ_g	Probability of mutating a 0 or 1 to a # in Wilson's biased mutation scheme (p. 49).
$P_\#$	The hash probability (p. 46).
p_I	Initial prediction of a new rule (p. 51).
ε_I	Initial prediction error of a new rule (p. 51).
F_I	Initial fitness of a new rule (p. 51).

Reinforcement Learning Notation

t	Discrete time step of the learning environment.
s_t	Environmental state at time t.
a_t	Agent's action at time t.
r_t	Immediate reward at time t.
$\mathcal{R}$	The immediate reward function.
R	The return which an RL agent attempts to maximise.
$\mathcal{S}$	The set of all environmental states for a given task.
$\mathcal{A}(s)$	The set of actions possible in state s.

$\mathcal{P}$	The state transition function.
π	A policy.
$\mathcal{S} \to \mathcal{A}$	A policy.
V^π	The value function for π.
$\mathcal{S} \to \Re$	A value function (more carefully, a state-value function).
$\mathcal{S} \times \mathcal{A} \to \Re$	An action-value function, also called a Q-function.
$Q(s, a)$	The estimated value of a state-action pair (an estimated Q-value).
$Q^\pi(s, a)$	The true value of a state-action pair (a true Q-value) under a policy π.
$Q^*(s, a)$	The true value of a state-action pair (a true Q-value) under an optimal policy.
E_π	The expectation under a policy π.
α	The step size term in an update. Also called the learning rate.

Miscellaneous Notation

§	Section in this thesis.
(1.1)	A numbered equation.
w.r.t.	"With Respect To ... ".

Bibliography

[1] *Collected Abstracts for the First International Workshop on Learning Classifier System (IWLCS92)*, 1992. October 6–8, NASA Johnson Space Center, Houston, Texas.

[2] Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2000), in the Joint Workshops of SAB 2000 and PPSN 2000, 2000. Pier Luca Lanzi, Wolfgang Stolzmann and Stewart W. Wilson (workshop organisers).

[3] Reinforcement Learning Repository at Michigan State University. http://web.cps.msu.edu/rlr/, 2002.

[4] Manu Ahluwalia and Larry Bull. A Genetic Programming-based Classifier System. In Banzhaf et al. [9], pages 11–18.

[5] Rudolf F. Albrecht, Nigel C. Steele, and Colin R. Reeves, editors. *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*. Spring-Verlag, 1993.

[6] A. E. A. Almaini. *Electronic Logic Systems*. Prentice–Hall, 2nd edition, 1989.

[7] C. W. Anderson. *Learning and Problem solving with multilayer connectionist systems*. PhD thesis, University of Massachussets, Amherst, MA, USA, 1986.

[8] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press, 1997.

[9] W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 1999.

[10] Alwyn Barry. Hierarchy Formulation Within Classifiers System – A Review. In E. G. Goodman, V. L. Uskov, and W. F. Punch, editors, *Proceedings of the First International Conference on Evolutionary Algorithms and their Application EVCA'96*, pages 195–211, Moscow, 1996. The Presidium of the Russian Academy of Sciences.

[11] Alwyn Barry. Aliasing in XCS and the Consecutive State Problem: 1 – Effects. In Banzhaf et al. [9], pages 19–26.

[12] Alwyn Barry. Aliasing in XCS and the Consecutive State Problem: 2 – Solutions. In Banzhaf et al. [9], pages 27–34.

[13] Alwyn Barry. Specifying Action Persistence within XCS. In Whitely et al. [290], pages 50–57.

[14] Alwyn Barry. *XCS Performance and Population Structure within Multiple-Step Environments*. PhD thesis, Queens University Belfast, 2000.

[15] Alwyn Barry. The Learning Classifier Systems Web. The University of Bath, 2002. http://www.cs.bath.ac.uk/~amb/LCSWEB/.

[16] A. G. Barto, P. Anandan, and C. W. Anderson. Cooperativity in networks of pattern recognizing stochastic learning automata. In *Proceedings of the Fourth Yale Workshop on Applications of Adaptive Systems Theory*, pages 85–90, 1985.

[17] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike Adaptive Elements That Can Solve Difficult Learning Problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13(5):834–846, 1983.

[18] Eric Baum. Towards a model of intelligence as an economy of agents. *Machine Learning*, 35(2):155–185, 1999.

[19] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[20] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

[21] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[22] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 1995.

[23] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[24] Ihor O. Bohachevsky, Mark E. Johnson, and Myron L. Stein. Generalized simulated annealing for function optimization. *Technometrics*, 28(3):209–217, 1986.

[25] Andrea Bonarini. ELF: Learning Incomplete Fuzzy Rule Sets for an Autonomous Robot. In Hans-Jürgen Zimmermann, editor, *First European Congress on Fuzzy and Intelligent Technologies – EUFIT'93*, volume 1, pages 69–75, Aachen, D, September 1993. Verlag der Augustinus Buchhandlung.

[26] Andrea Bonarini. An Introduction to Learning Fuzzy Classifier Systems. In Lanzi et al. [167], pages 83–104.

[27] Andrea Bonarini, Claudio Bonacina, and Matteo Matteucci. Fuzzy and Crisp Representations of Real-valued Input for Learning Classifier Systems. In Lanzi et al. [167], pages 107–124.

[28] Pierre Bonelli and Alexandre Parodi. An Efficient Classifier System and its Experimental Comparison with two Representative learning methods on three medical domains. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA-91)*, pages 288–295, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[29] Pierre Bonelli, Alexandre Parodi, Sandip Sen, and Stewart Wilson. NEWBOOLE: A Fast GBML System. In *International Conference on Machine Learning*, pages 153–159, San Mateo, California, 1990. Morgan Kaufmann.

[30] Lashon B. Booker. *Intelligent Behavior as an Adaptation to the Task Environment.* PhD thesis, The University of Michigan, 1982.

[31] Lashon B. Booker. Improving the performance of genetic algorithms in classifier systems. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA-85)*, pages 80–92, Pittsburgh, PA, July 1985. Lawrence Erlbaum Associates.

[32] Lashon B. Booker. Classifier Systems that Learn Internal World Models. *Machine Learning*, 3:161–192, 1988.

[33] Lashon B. Booker. Triggered rule discovery in classifier systems. In Schaffer [225], pages 265–274.

[34] Lashon B. Booker. Representing Attribute-Based Concepts in a Classifier System. In Gregory J. E. Rawlins, editor, *Proceedings of the First Workshop on Foundations of Genetic Algorithms (FOGA91)*, pages 115–127. Morgan Kaufmann: San Mateo, 1991.

[35] Lashon B. Booker. Classifier systems, endogenous fitness, and delayed reward: A preliminary investigation. In *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2000), in the Joint Workshops of SAB 2000 and PPSN 2000* [2]. Extended abstract.

[36] Lashon B. Booker. Do We Really Need to Estimate Rule Utilities in Classifier Systems? In Lanzi et al. [167], pages 125–142.

[37] Lashon B. Booker. Classifier systems, endogenous fitness, and delayed rewards: A preliminary investigation. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. Morgan Kaufmann, 2001.

[38] Lashon B. Booker and Richard K. Belew, editors. *Proceedings of the 4th International Conference on Genetic Algorithms (ICGA91)*. Morgan Kaufmann, July 1991.

[39] Lashon B. Booker, David E. Goldberg, and John H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.

[40] Lashon B. Booker, Rick L. Riolo, and John H. Holland. Learning and Representation in Classifier Systems. In Vassant Honavar and Leonard Uhr, editors, *Artificial Intelligence and Neural Networks*, pages 581–613. Academic Press, 1994.

[41] M. F. Bramlette and E. E. Bouchard. Genetic algorithms in parametric design of aircraft. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 10, pages 109–123. Van Nostrand Reinhold: New York, 1991.

[42] Larry Bull. *Artificial Symbiology: evolution in cooperative multi-agent environments.* PhD thesis, University of the West of England, 1995.

[43] Larry Bull. On ZCS in Multi-agent Environments. In A. E. Eiben, T. Baeck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings Parallel Problem Solving From Nature (PPSN-V)*, volume 1498 of *Lecture Notes in Computer Science*, pages 471–480. Springer–Verlag, 1998.

[44] Larry Bull. On using ZCS in a Simulated Continuous Double-Auction Market. In Banzhaf et al. [9], pages 83–90.

[45] Larry Bull, Terence C. Fogarty, S. Mikami, and J. G. Thomas. Adaptive Gait Acquisition using Multi-agent Learning for Wall Climbing Robots. In *Automation and Robotics in Construction XII*, pages 80–86, 1995.

[46] Larry Bull, Terence C. Fogarty, and M. Snaith. Evolution in Multi-agent Systems: Evolving Communicating Classifier Systems for Gait in a Quadrupedal Robot. In Larry J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA95)*, pages 382–388. Morgan Kaufmann Publishers, 1995.

[47] Larry Bull and Jacob Hurst. ZCS Redux. *To appear in Evolutionary Computation*, 2002.

[48] Martin Butz, David E. Goldberg, and Wolfgang Stolzmann. New challenges for an ACS: Hard problems and possible solutions. Technical Report 99019, University of Illinois at Urbana-Champaign, Urbana, IL, October 1999.

[49] Martin Butz and Wolfgang Stolzmann. Action-Planning in Anticipatory Classifier Systems. In Wu [318], pages 242–249.

[50] Martin V. Butz, David E. Goldberg, and Wolfgang Stolzmann. Investigating Generalization in the Anticipatory Classifier System. In *Proceedings of Parallel Problem Solving from Nature (PPSN VI)*, 2000. Also technical report 2000014 of the Illinois Genetic Algorithms Laboratory.

[51] Martin V. Butz, David E. Goldberg, and Wolfgang Stolzmann. Probability-enhanced predictions in the anticipatory classifier system. In *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2000), in the Joint Workshops of SAB 2000 and PPSN 2000* [2]. Extended abstract.

[52] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. How XCS Evolves Accurate Classifiers. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H Garzon, and Edmund Burke, editors, *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 927–934. Morgan Kaufmann, 2001.

[53] Martin V. Butz and Stewart W. Wilson. An Algorithmic Description of XCS. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, number 1996 in LNAI, pages 253–272. Springer–Verlag, 2001.

[54] Keith Chalk and George D. Smith. Multi-Agent Classifier Systems and the Iterated Prisoner's Dilemma. In George D. Smith, Nigel C. Steele, and Rudolf F. Albrecht, editors, *Artificial Neural Networks and Genetic Algorithms*, pages 615–618. Springer, 1997.

[55] Dave Cliff and Susi Ross. Adding Temporary Memory to ZCS. *Adaptive Behavior*, 3(2):101–150, 1995.

[56] Marco Colombetti and Marco Dorigo. Learning to Control an Autonomous Robot by Distributed Genetic Algorithms. In J. A. Meyer H. L. Roitblat and S. W. Wilson, editors, *From Animals to Animats 2. Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92)*, pages 305–312. A Bradford Book. MIT Press, 1992.

[57] Marco Colombetti and Marco Dorigo. Robot Shaping: Developing Situated Agents through Learning. Technical Report TR-92-040, International Computer Science Institute, Berkeley, CA, 1993.

[58] Marco Colombetti and Marco Dorigo. Training agents to perform sequential behavior. *Adaptive Behavior*, 2(3):247–275, 1994. ftp://iridia.ulb.ac.be/pub/dorigo/journals/IJ.06-ADAP94.ps.gz.

[59] D. Corne, P. Ross, and H.-L. Fang. Fast practical evolutionary timetabling. In *AISB Workshop on Evolutionary Computing*, volume 865 of *Lecture Notes in Computer Science*, pages 250–263. Springer: Berlin, 1994.

[60] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their*

Applications, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[61] Henry Brown Cribbs III and Robert E. Smith. Classifier system renaissance: New analogies, new directions. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 547–552, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[62] Henry Brown Cribbs III and Robert E. Smith. What Can I do with a Learning Classifier System? In C. Karr and L. M. Freeman, editors, *Industrial Applications of Genetic Algorithms*, pages 299–320. CRC Press, 1998.

[63] R. H. Crites. *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents.* PhD thesis, University of Massachusetts, Amherst, 1996.

[64] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In Touretzky et al. [275], pages 1017–1023.

[65] Bart de Boer. Classifier Systems: a useful approach to machine learning? Master's thesis, Leiden University, 1994. ftp://ftp.wi.leidenuniv.nl/pub/CS/MScTheses/deboer.94.ps.gz.

[66] Kenneth A. De Jong and Willliam M. Spears. Learning Concept Classification Rules Using Genetic Algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 651–656, Sidney, Australia, 1991.

[67] José del R. Millán and Carme Torras. Learning sensor-based navigation. In M. Kaiser K. Morik and V. Klingspor, editors, *Making Robots Smarter: Combining Sensing and Action through Robot Learning*, pages 85–108. Kluwer Academic Publishers, 1999.

[68] L. Desjarlais and Stephanie Forrest. Linked learning in classifier systems: A control architecture for mobile robots. In *Collected Abstracts for the First International Workshop on Learning Classifier System (IWLCS-92)* [1]. October 6–8, NASA Johnson Space Center, Houston, Texas.

[69] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis.* McGraw-Hill, 1994.

[70] Jean-Yves Donnart. *Cognitive Architecture and Adaptive Properties of an Motivationally Autonomous Animat.* PhD thesis, Université Pierre et Marie Curie. Paris, France, 1998.

[71] Jean-Yves Donnart and Jean-Arcady Meyer. Learning Reactive and Planning Rules in a Motivationally Autonomous Animat. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 26(3):381–395, 1996.

[72] Marco Dorigo. New perspectives about default hierarchies formation in learning classifier systems. In E. Ardizzone, E. Gaglio, and S. Sorbello, editors, *Proceedings of the 2nd Congress of the Italian Association for Artificial Intelligence (AI*IA) on Trends in Artificial Intelligence*, volume 549 of *LNAI*, pages 218–227, Palermo, Italy, October 1991. Springer Verlag.

[73] Marco Dorigo. Alecsys and the AutonoMouse: Learning to Control a Real Robot by Distributed Classifier Systems. *Machine Learning*, 19:209–240, 1995.

[74] Marco Dorigo and Hugues Bersini. A Comparison of Q-Learning and Classifier Systems. In *Proceedings of From Animals to Animats III, Third International Conference on Simulation of Adaptive Behavior (SAB94).* Morgan Kaufmann, 1994.

[75] Marco Dorigo and Marco Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 2:321–370, 1994.

[76] Marco Dorigo and Marco Colombetti. *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press/Bradford Books, 1998.

[77] Marco Dorigo, V. Maniezzo, and D. Montanari. Classifier-based robot control systems. In *IFAC/IFIP/IMACS International Symposium on Artificial Intelligence in Real-Time Control*, pages 591–598, Delft, Netherlands, 1992.

[78] Marco Dorigo, Mukesh J. Patel, and Marco Colombetti. The effect of Sensory Information on Reinforcement Learning by a Robot Arm. In M. Jamshidi et al., editor, *Proceedings of ISRAM'94, Fifth International Symposium on Robotics and Manufacturing, August 14–18, Maui, HI*, pages 83–88. ASME Press, 1994.

[79] Marco Dorigo and U. Schnepf. Organisation of Robot Behaviour Through Genetic Learning Processes. In *Proceedings of ICAR'91 – Fifth IEEE International Conference on Advanced Robotics, Pisa, Italy*, pages 1456–1460. IEEE Press, 1991.

[80] Marco Dorigo and U. Schnepf. Genetics-based Machine Learning and Behaviour Based Robotics: A New Synthesis. *IEEE Transactions on Systems, Man and Cybernetics*, 23(1):141–154, 1993.

[81] K. Doya. Temporal difference learning in continuous time and space. In Touretzky et al. [275], pages 1073–1079.

[82] Daniel Eckert and Johann Mitlöhner. Modelling individual and endogenous learning in games: the relevance of classifier systems. In *Complex Modelling for Socio-Economic Systems, SASA, Vienna*, 1997.

[83] G. Enee and C. Escazut. Classifier systems evolving multi-agent system with distributed elitism. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation CEC99*, volume 3, pages 1740–1745, Washington (DC), 1999. IEEE Press.

[84] Cathy Escazut, Philippe Collard, and Jean-Louis Cavarero. Dynamic Management of the Specificity in Classifier Systems. In Albrecht et al. [5], pages 484–491.

[85] Thomas Fent. *Applications of Learning Classifier Systems for Simulating Learning Organizations*, volume 10 of *Fortschrittsberichte Simulation*. ARGESIM / ASIM–Verlag, Wien, September 2001.

[86] Peter Fletcher. Simulating the use of 'fiat money' in a simple commodity economy. Master's thesis, Schools of Psychology and Computer Science, University of Birmingham, 1996.

[87] Terence C. Fogarty, Larry Bull, and Brian Carse. Evolving Multi-Agent Systems. In J. Periaux and G. Winter, editors, *Genetic Algorithms in Engineering and Computer Science*, pages 3–22. John Wiley & Sons, 1995.

[88] Terence C. Fogarty, Brian Carse, and Larry Bull. Classifier Systems – recent research. *AISB Quarterly*, 89:48–54, 1994.

[89] Terence C. Fogarty, N. S. Ireson, and Larry Bull. Genetic-based Machine Learning – Applications in Industry and Commerce. In Vic

Rayward-Smith, editor, *Applications of Modern Heuristic Methods*, pages 91–110. Alfred Waller Ltd, 1995.

[90] Stephanie Forrest. *A study of parallelism in the classifier system and its application to classification in KL-ONE semantic networks.* PhD thesis, University of Michigan, Ann Arbor, MI, 1985.

[91] Peter W. Frey and David J. Slate. Letter Recognition Using Holland-Style Adaptive Classifiers. *Machine Learning*, 6:161–182, 1991.

[92] Attilio Giordana and Filippo Neri. Search-Intensive Concept Induction. *Evolutionary Computation*, 3:375–416, 1995.

[93] David E. Goldberg. *Computer-Aided Gas Pipeline Operation using Genetic Algorithms and Rule Learning.* PhD thesis, The University of Michigan, 1983.

[94] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[95] David E. Goldberg. Probability Matching, the Magnitude of Reinforcement, and Classifier System Bidding. *Machine Learning*, 5:407–425, 1990. (Also TCGA tech report 88002, U. of Alabama).

[96] David E. Goldberg, Jeffrey Horn, and Kalyanmoy Deb. What Makes a Problem Hard for a Classifier System? In *Collected Abstracts for the First International Workshop on Learning Classifier System (IWLCS-92)* [1]. Also technical report 92007 Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

[97] David E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms. motivation, analysis and first results. *Complex Systems*, 3:493–530, 1989.

[98] David Perry Greene and Stephen F. Smith. COGIN: Symbolic induction using genetic algorithms. In *Proceedings 10th National Conference on Artificial Intelligence*, pages 111–116. Morgan Kaufmann, 1992.

[99] David Perry Greene and Stephen F. Smith. Using Coverage as a Model Building Constraint in Learning Classifier Systems. *Evolutionary Computation*, 2(1):67–91, 1994.

[100] John J. Grefenstette. Credit Assignment in Rule Discovery Systems Based on Genetic Algorithms. *Machine Learning*, 3:225–245, 1988.

[101] John J. Grefenstette. Lamarckian Learning in Multi-Agent Environments. In Booker and Belew [38], pages 303–310. http://www.ib3.gmu.edu/gref/publications.html.

[102] John J. Grefenstette. Learning decision strategies with genetic algorithms. In *Proc. Intl. Workshop on Analogical and Inductive Inference*, volume 642 of *Lecture Notes in Artificial Intelligence*, pages 35–50. Springer-Verlag, 1992. http://www.ib3.gmu.edu/gref/.

[103] John J. Grefenstette. The Evolution of Strategies for Multi-agent Environments. *Adaptive Behavior*, 1:65–89, 1992.

[104] John J. Grefenstette. Using a genetic algorithm to learn behaviors for autonomous vehicles. In *Proceedings American Institute of Aeronautics and Astronautics Guidance, Navigation and Control Conference*, pages 739–749. AIAA, 1992. http://www.ib3.gmu.edu/gref/.

[105] John J. Grefenstette. Evolutionary Algorithms in Robotics. In M. Jamshedi and C. Nguyen, editors, *Robotics and Manufacturing: Recent Trends in Research, Education and Applications, v5. Proc. Fifth Intl. Symposium on Robotics and Manufacturing, ISRAM 94*, pages 65–72. ASME Press: New York, 1994. http://www.ib3.gmu.edu/gref/.

[106] John J. Grefenstette and Alan C. Schultz. An evolutionary approach to learning in robots. In *Machine Learning Workshop on Robot Learning*, New Brunswick, NJ, 1994. http://www.ib3.gmu.edu/gref/.

[107] Axel Großmann. Adaptive state-space quantisation and multi-task reinforcement learning using constructive neural networks. In Jean-Arcady Meyer, Alain Berthoz, Dario Floreano, Herbert L. Roitblat, and Stewart W. Wilson, editors, *Proceedings of the Sixth International Conference on the Simulation of Adaptive Behaviour (SAB2000). Supplement Book*, Honolulu, HI, USA, September 2000. International Society for Adaptive Behaviour.

[108] Axel Großmann. *Continual Learning for Mobile Robots.* PhD thesis, University of Birmingham, 2000.

[109] Hisashi Handa, Takashi Noda, Tadataka Konishi, Osamu Katai, and Mitsuru Baba. Coevolutionary fuzzy classifier system for autonomous mobile robots. In Takadama [266].

[110] Adrian Hartley. Genetics Based Machine Learning as a Model of Perceptual Category Learning in Humans. Master's thesis, University of Birmingham, 1998.

[111] Adrian Hartley. Accuracy-based fitness allows similar performance to humans in static and dynamic classification environments. In Banzhaf et al. [9], pages 266–273.

[112] Jörg Heitkötter and David Beasley (editors). The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ). USENET: comp.ai.genetic ftp://rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/.

[113] Luis Miramontes Hercog and Terence C. Fogarty. Analysis of inductive intelligence in xcs-based multi-agent system (maxcs). In J. Periaux, P. Joly, and E. Onate, editors, *Innovative Tools for Scientific Computation in Aeronautical Engineering*, pages 351–366. CIMNE, Barcelona, 2001. ISBN: 84-90025-78-X.

[114] Luis Miramontes Hercog and Terence C. Fogarty. Co-evolutionary classifier systems for multi-agent simulation. In *Proceedings of the Congress on Evolutionary Computation*, 2002.

[115] Luis Miramontes Hercog and Terence C. Fogarty. Social simulation using a Multi-Agent Model based on Classifier Systems: The Emergence of Vacillating Behaviour in the "El Farol" Bar Problem. In W. Stolzman, editor, *Proceedings of the Fourth International Workshop in Learning Classifier Systems 2001*. Springer, 2002.

[116] John H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. Technical Report ORA Projects 01252 and 08226, Department of Computer and Communication Sciences. University of Michigan, 1968.

[117] John H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. In A. W. Burks, editor, *Essays on cellular automata*, pages 320–353. Urbana: University of Illinois Press, 1970.

[118] John H. Holland. Processing and processors for schemata. In E. L. Jacks, editor, *Associative Information Processing*, pages 127–146. New York: American Elsevier, 1971.

[119] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975. Republished by the MIT press, 1992.

[120] John H. Holland. Adaptation. In R. Rosen and F. M. Snell, editors, *Progress in Theoretical Biology.* New York: Plenum, 1976.

[121] John H. Holland. Escaping brittleness. In *Proceedings Second International Workshop on Machine Learning*, pages 92–95, 1983.

[122] John H. Holland. Properties of the bucket brigade. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, pages 1–7, Pittsburgh, PA, July 1985. Lawrence Erlbaum Associates.

[123] John H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In T. Mitchell, R. Michalski, and J. Carbonell, editors, *Machine Learning, an Artificial Intelligence Approach. Volume II*, chapter 20, pages 593–623. Morgan Kaufmann, 1986.

[124] John H. Holland. Concerning the emergence of tag-mediated lookahead in classifier systems. *Physica D*, 42:188–201, 1990.

[125] John H. Holland, Lashon B. Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. What is a Learning Classifier System? In Lanzi et al. [167], pages 3–32.

[126] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction. Processes of Inference, Learning and Discovery.* The MIT Press, 1986.

[127] John H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-directed Inference Systems.* New York: Academic Press, 1978. Reprinted in: Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN: 0-7803-3481-7.

[128] John H. Holmes. *Evolution-Assisted Discovery of Sentinel Features in Epidemiologic Surveillance.* PhD thesis, Drexel University, 1996.

[129] John H. Holmes. Quantitative Methods for Evaluating Learning Classifier System Performance in Forced Two-Choice Decision Tasks. In Wu [318], pages 250–257.

[130] John H. Holmes. Learning Classifier Systems Applied to Knowledge Discovery in Clinical Research Databases. In Lanzi et al. [167], pages 243–261.

[131] John H. Holmes. Personal communication. September 16, 2000.

[132] John H. Holmes. The Learning Classifier Systems Mailing List. The University of Pennsylvania, 2002. cslist@cceb.med.upenn.edu.

[133] John H. Holmes, Dennis R. Durbin, and Flaura K. Winston. The learning classifier system: an evolutionary computation approach to knowledge discovery in epidemiologic surveillance. *Artificial Intelligence In Medicine*, 19(1):53–74, 2000.

[134] Jeffrey Horn and David E. Goldberg. Towards a Control Map for Niching. In *Foundations of Genetic Algorithms (FOGA)*, pages 287–310, 1998.

[135] Mark Humphrys. *Action Selection Methods using Reinforcement Learning*. PhD thesis, Cambridge University, 1997.

[136] N. Ireson, Y. J. Cao, L. Bull, and R. Miles. A Communication Architecture for Multi-Agent Learning Systems. In *Proceedings of the EvoNet Workshops - EvoTel 2000*, pages 255–266, 2000.

[137] Hisao Ishibuchi and Tomoharu Nakashima. Linguistic Rule Extraction by Genetics-Based Machine Learning. In Whitely et al. [290], pages 195–202.

[138] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[139] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1988.

[140] Richard Johnsonbaugh. *Discrete Mathematics*. Macmillan, 3rd edition, 1993.

[141] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4, 1996.

[142] Jonas Karlsson. *Learning to Solve Multiple Goals*. PhD thesis, University of Rochester, 1997.

[143] Daisuke Katagami and Seiji Yamada. Real robot learning with human teaching. In Takadama [266].

[144] Tim Kovacs. Evolving Optimal Populations with XCS Classifier Systems. Master's thesis, University of Birmingham, 1996.

[145] Tim Kovacs. Steady State Deletion Techniques in a Classifier System. Unpublished document. School of Computer Science, University of Birmingham, 1997.

[146] Tim Kovacs. XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In Roy, Chawdhry, and Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 59–68. Springer–Verlag, London, 1997.

[147] Tim Kovacs. Deletion schemes for classifier systems. In Banzhaf et al. [9], pages 329–336.

[148] Tim Kovacs. Strength or Accuracy? Fitness Calculation in Learning Classifier Systems. In Lanzi et al. [167], pages 143–160.

[149] Tim Kovacs. Trends in Learning Classifier Systems Publication. Technical Report CSRP-00-21, School of Computer Science, University of Birmingham, 2000.

[150] Tim Kovacs. Towards a theory of strong overgeneral classifiers. In Worthy Martin and William M. Spears, editors, *Foundations of Genetic Algorithms Volume 6*, pages 165–184. Morgan Kaufmann, 2001.

[151] Tim Kovacs. What should a classifier system learn? In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC01)*, pages 775–782. IEEE Press, 2001.

[152] Tim Kovacs. A Learning Classifier Systems Bibliography. Department of Computer Science, University of Bristol, 2002. http://www.cs.bris.ac.uk/~kovacs/lcs.

[153] Tim Kovacs. The Electronic Learning Classifier Systems Archive. Department of Computer Science, University of Bristol, 2002. http://www.cs.bris.ac.uk/~kovacs/lcs/lcs.archive.html.

[154] Tim Kovacs and Manfred Kerber. What makes a problem hard for XCS? In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, number 1996 in LNAI, pages 80–99. Springer–Verlag, 2001.

[155] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 1989.

[156] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, Mass., 1992.

[157] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[158] John R. Koza, Forrest H Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, 1999.

[159] William B. Langdon and John R. Koza. The genetic programming bibliography, 2001. http://www.cs.bham.ac.uk/~wbl/biblio/.

[160] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[161] Pier Luca Lanzi. A Study of the Generalization Capabilities of XCS. In Thomas Bäck, editor, *Proceedings Seventh International Conference on Genetic Algorithms (ICGA-97)*, pages 418–425, East Lansing (MI), 1997. Morgan Kaufmann.

[162] Pier Luca Lanzi. Adding Memory to XCS. In *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98).* IEEE Press, 1998. http://ftp.elet.polimi.it/people/lanzi/icec98.ps.gz.

[163] Pier Luca Lanzi. Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding. In Banzhaf et al. [9], pages 337–344.

[164] Pier Luca Lanzi. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In Banzhaf et al. [9], pages 345–352.

[165] Pier Luca Lanzi and Marco Colombetti. An Extension of XCS to Stochastic Environments. In Banzhaf et al. [9], pages 353–360.

[166] Pier Luca Lanzi and Rick L. Riolo. A Roadmap to the Last Decade of Learning Classifier System Research (from 1989 to 1999). In Lanzi et al. [167], pages 33–62.

[167] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*. Springer-Verlag, Berlin, 2000.

[168] Blake Lebaron, W. Brian Arthur, and R. Palmer. The Time Series Properties of an Artificial Stock Market. *Journal of Economic Dynamics and Control*, 23:1487–1516, 1999.

[169] Martin Lettau and Harald Uhlig. Rules of Thumb and Dynamic Programming. Technical report, Department of Economics, Princeton University, 1994.

[170] Martin Lettau and Harald Uhlig. Rules of thumb versus dynamic programming. *American Economic Review*, 89:148–174, 1999.

[171] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, New York, USA, second edition, 1997.

[172] Gunar E. Liepins and Lori A. Wang. Classifier System Learning of Boolean Concepts. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA-91)*, pages 318–323, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[173] Long-Ji Lin. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8:293–321, 1992.

[174] C. B. Lucasius, M. J. Blommers, L. M. Buydens, and G. Kateman. A genetic algorithm for conformational analysis of DNA. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 18, pages 251–281. Van Nostrand Reinhold: New York, 1991.

[175] Chikara Maezawa and Masayasu Atsumi. Collaborative Learning Agents with Structural Classifier Systems. In Banzhaf et al. [9], page 777. One page poster paper.

[176] Ramon Marimon, Ellen McGrattan, and Thomas J. Sargent. Money as a Medium of Exchange in an Economy with Artificially Intelligent Agents. *Journal of Economic Dynamics and Control*, 14:329–373, 1990. Also Technical Report 89-004, Santa Fe Institute, 1989.

[177] E. J. McCluskey. Minimization of Boolean Functions. *Bell System Technical Journal*, 35:1417–1444, 1956.

[178] Nicholas F. McPhee, Riccardo Poli, and Jonathan Rowe. A schema theory analysis of mutation size biases in genetic programming with linear representations. Technical Report CSRP-00-24, School of Computer Science, University of Birmingham, 2000.

[179] Nicolas Meuleau and Paul Bourgine. Exploration of multi-state environments: Local measure and back-propagation of uncertainty. *Machine Learning*, 35(2), 1999.

[180] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1996. Contains introductory chapter on LCS.

[181] Ryszard S. Michalski. A theory and methodology of inductive learning. In R. Michalski, G. Carbonell, and T. Mitchell, editors, *Maching Learning. An Artificial Intelligence Approach*, pages 83–134. Springer–Verlag, 1983.

[182] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, 1996.

[183] Tom Mitchell. The need for bias in learning generalizations. Technical Report CBM-TR-117, Rutgers University, 1980.

[184] Tom M. Mitchell. *Machine Learning*. McGraw–Hill, 1997.

[185] Johann Mitlöhner. Classifier systems and economic modelling. In *APL '96. Proceedings of the APL 96 Conference on Designing the Future*, volume 26 (4), pages 77–86, 1996.

[186] Chilukuri K. Mohan. *Expert Systems: A Modern Overview*. Kluwer, 2000. Contains an introductory survey chapter on LCS.

[187] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999.

[188] David E. Moriarty and Risto Mikkulainen. Efficient Reinforcement Learning through Symbiotic Evolution. *Machine Learning*, 22:11–32, 1996.

[189] Rémi Munos and Andrew Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[190] E. M. Oblow. Implementing Valiant's Learnability Theory using Random Sets. Technical Report ORNL/TM-11512R, Oak Ridge National Laboratory, 1990.

[191] Norihiko Ono and Adel T. Rahmani. Self-Organization of Communication in Distributed Learning Classifier Systems. In Albrecht et al. [5], pages 361–367.

[192] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–100, 1990.

[193] Mukesh J. Patel and Marco Dorigo. Adaptive Learning of a Robot Arm. In Terence C. Fogarty, editor, *Evolutionary Computing, AISB Workshop Selected Papers*, number 865 in Lecture Notes in Computer Science, pages 180–194. Springer-Verlag, 1994.

[194] Steven E. Phelan. *Using Artificial Adaptive Agents to Explore Strategic Landscapes*. PhD thesis, School of Business, Faculty of Law and Management, La Trobe University, Australia, 1997.

[195] A. G. Pipe and Brian Carse. Autonomous Acquisition of Fuzzy Rules for Mobile Robot Control: First Results from two Evolutionary Computation Approaches. In Whitely et al. [290], pages 849–856.

[196] Riccardo Poli. Recursive conditional schema theorem, convergence and population sizing in genetic algorithms. In *Proceedings of the Foundation of Genetic Algorithm (FOGA) Workshop*, pages 143–163. Morgan Kaufmann, 2000.

[197] Riccardo Poli. Why the schema theorem is correct also in the presence of stochastic effects. In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC00)*, pages 487–492. IEEE Press, 2000.

[198] Riccardo Poli and William B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In *Proceedings of the Second International Conference on Genetic Programming (GP'97)*, pages 278–285. Morgan Kaufmann, 1997.

[199] Riccardo Poli and William B. Langdon. A review of theoretical and experimental results on schemata in genetic programming. In M. Schoenauer W. Banzhaf, R. Poli and T. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming (EuroGP'98)*, pages 1–15. Springer-Verlag, 1998.

[200] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[201] Riccardo Poli, William B. Langdon, and Una-May O'Reilly. Analysis of schema variance and short term extinction likelihoods. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Third International Conference on Genetic Programming (GP'98)*, pages 284–292. Morgan Kaufmann, 1999.

[202] Riccardo Poli and Guido Valli. Neural inhabitants of MR and echo images segment cardiac structures. In *Proceedings of Computers in Cardiology*, pages 193–196. IEEE Computer Society press, 1993.

[203] Tony J. Prescott. *Explorations in Reinforcement and Model-based Learning.* PhD thesis, University of Sheffield, 1993.

[204] Martin L. Puterman. *Markov Decision Problems.* Wiley, 1994.

[205] J. R. Quinlan. An empirical comparison of genetic and decision-tree classifiers. In *Proceedings of the Fifth International Machine Learning Conference*, pages 135–141, 1988.

[206] Stuart I. Reynolds. Decision boundary partitioning: Variable resolution model-free reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, San Fransisco, 2000. Morgan Kaufmann.

[207] Stuart I. Reynolds. Adaptive representation methods for reinforcement learning. In *Advances in Artificial Intelligence, Proceeding of AI-2001, Ottawa, Canada*, Lecture Notes in Artificial Intelligence (LNAI 2056), pages 345–348. Spring Verlag, June 2001.

[208] Stuart I. Reynolds. *Reinforcement Learning with Exploration.* PhD thesis, School of Computer Science, University of Birmingham, 2002.

[209] Mark Bishop Ring. *Continual Learning in Reinforcement Environments.* PhD thesis, The University of Texas at Austin, 1994.

[210] Rick L. Riolo. Bucket Brigade Performance: I. Long Sequences of Classifiers. In *Proceedings Second International Conference on Genetic Algorithms (ICGA-87)*, pages 184–195. Lawrence Erlbaum Associates, 1987.

[211] Rick L. Riolo. Bucket Brigade Performance: II. Default Hierarchies. In John J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87)*, pages 196–201, Cambridge, MA, July 1987. Lawrence Erlbaum Associates.

[212] Rick L. Riolo. CFS-C: A Package of Domain-Independent Subroutines for Implementing Classifier Systems in Arbitrary User-Defined Environments. Technical report, University of Michigan, 1988.

[213] Rick L. Riolo. The emergence of coupled sequences of classifiers. In Schaffer [225], pages 256–264.

[214] Rick L. Riolo. Lookahead planning and latent learning in a classifier system. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB-90)*, pages 316–326. The MIT Press, 1991.

[215] Rick L. Riolo. Modelling Simple Human Category Learning with a Classifier System. In Booker and Belew [38], pages 324–333.

[216] A. Robert, F. Chantemargue, and M. Courant. Grounding Agents in EMud Artificial Worlds. In *Proceedings of the First International Conference on Virtual Worlds, Paris (France), July 1-3*, 1998.

[217] Gary Roberts. Dynamic planning for classifier systems. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA-93)*, pages 231–237. Morgan Kaufmann, 1993.

[218] Gavin Rummery. *Problem Solving with Reinforcement Learning.* PhD thesis, Cambridge University Engineering Department, 1995. http://svr-www.eng.cam.ac.uk/reports/people/gar.html.

[219] Gavin Rummery and Mahesan Niranjan. On-line Q-learning using Connectionist Systems. Technical Report 166, Engineering Deparment, University of Cambridge, 1994.

[220] Stuart Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach.* Prentice Hall International, 1995.

[221] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.

[222] A. L. Samuel. Some studies in machine learning using the game of chequers. II – recent progress. *IBM Journal on Research and Development*, 11:601–617, 1967.

[223] Shaun Saxon and Alwyn Barry. XCS and the Monk's Problems. In Lanzi et al. [167], pages 223–242.

[224] Cullen Schaffer. A conservation law for generalization performance. In Haym Hirsh and Willian W. Cohen, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 259–265, San Francisco, CA, 1994. Morgan Kaufmann.

[225] J. David Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA-89)*, George Mason University, June 1989. Morgan Kaufmann.

[226] Sonia Schulenburg and Peter Ross. An Adaptive Agent Based Economic Model. In Lanzi et al. [167], pages 263–282.

[227] Alan C. Schultz and John J. Grefenstette. Evolving Robot Behaviors. Poster at the 1994 Artificial Life Conference. (NCARAI Report: AIC-94-017) http://www.ib3.gmu.edu/gref/.

[228] Sandip Sen. Classifier system learning of multiplexer function. The University of Alabama, Tuscaloosa, Alabama. Class Project, 1988.

[229] Sandip Sen. Modeling human categorization by a simple classifier system. In *WSC1: 1st Online Workshop on Soft Computing*, 1996.

[230] Sandip Sen and Mahendra Sekaran. Multiagent Coordination with Learning Classifier Systems. In Gerhard Wei and Sandip Sen, editors, *Proceedings of the IJCAI Workshop on Adaption and Learning in Multi-Agent Systems*, volume 1042 of *LNAI*, pages 218–233. Springer Verlag, 1996.

[231] F. Seredynski, Pawel Cichosz, and G. P. Klebus. Learning classifier systems in multi-agent environments. In *Proceedings of the First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95)*, 1995.

[232] Olivier Sigaud. On the usefulness of a semi-automated Classifier System: the engineering perspective. In *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2000), in the Joint Workshops of SAB 2000 and PPSN 2000* [2]. Extended abstract.

[233] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pages 974–980. MIT Press, 1997.

[234] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, 38(3):287–308, 2000.

[235] Satinder Singh, P. Norvig, and D. Cohn. How to Make Software Agents Do the Right Thing: An Introduction to Reinforcement Learning. *Dr. Dobb's Journal*, March 1997.

[236] Aaron Sloman. Explorations in design space. In A. G. Cohn, editor, *Proceedings European Conference on Artificial Intelligence 1994*, pages 578–582. John Wiley, 1994.

[237] Aaron Sloman. Exploring design space and niche space. In *Proceedings 5th Scandinavian Conference on AI*. IOS Press, Amsterdam, 1995.

[238] Aaron Sloman. The "Semantics" of Evolution: Trajectories and Trade-offs in Design Space and Niche Space. In Helder Coelho, editor, *Progress in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, pages 27–38. Springer–Verlag, 1998.

[239] Aaron Sloman. Interacting Trajectories in Design Space and Niche Space: A philosopher speculates about evolution. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN VI*, number 1917 in Lecture Notes in Computer Science, pages 3–16. Springer–Verlag, 2000.

[240] George D. Smith. Economic Applications of Genetic Algorithms. In Vic Rayward-Smith, editor, *Applications of Modern Heuristic Methods*, pages 71–90. Alfred Waller Ltd, 1995. Contains 2 pages on LCS.

[241] Robert E. Smith. *Default Hierarchy Formation and Memory Exploitation in Learning Classifier Systems*. PhD thesis, University of Alabama, 1991.

[242] Robert E. Smith. A Report on The First International Workshop on Learning Classifier Systems (IWLCS-92). Unpublished docuement, 1992.

[243] Robert E. Smith. Derivative Methods: Learning Classifier Systems. In Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors,

Handbook of Evolutionary Computation, pages B1.2:6–B1.5:11. Institute of Physics Publishing and Oxford University Press, 1997. http://www.iop.org/Books/Catalogue/.

[244] Robert E. Smith and H. Brown Cribbs. Is a Learning Classifier System a Type of Neural Network? *Evolutionary Computation*, 2(1):19–36, 1994.

[245] Robert E. Smith, B. A. Dike, R. K. Mehra, B. Ravichandran, and A. El-Fallah. Classifier Systems in Combat: Two-sided Learning of Maneuvers for Advanced Fighter Aircraft. In *Computer Methods in Applied Mechanics and Engineering*. Elsevier, 1999.

[246] Robert E. Smith, B. A. Dike, B. Ravichandran, A. El-Fallah, and R. K. Mehra. The Fighter Aircraft LCS: A Case of Different LCS Goals and Techniques. In Lanzi et al. [167], pages 283–300.

[247] Robert E. Smith and David E. Goldberg. Variable default hierarchy separation in a classifier system. In Gregory J. E. Rawlins, editor, *Proceedings of the First Workshop on Foundations of Genetic Algorithms*, pages 148–170, San Mateo, July 15–18 1991. Morgan Kaufmann.

[248] Robert E. Smith and H. B. Cribbs III. Combined biological paradigms. *Robotics and Autonomous Systems*, 22(1):65–74, 1997.

[249] Robert E. Smith and Manuel Valenzuela-Rendón. A Study of Rule Set Development in a Learning Classifier System. In Schaffer [225], pages 340–346.

[250] S. F. Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, 1980.

[251] S. F. Smith. Flexible Learning of Problem Solving Heuristics through Adaptive Search. In *Proceedings Eight International Joint Conference on Artificial Intelligence*, pages 422–425, 1983.

[252] S. F. Smith. Adaptive learning systems. In R. Forsyth, editor, *Expert Systems: Principles and Case Studies*, pages 169–189. Chapman and Hall, 1984.

[253] Graham Spencer. Automatic generation of programs for crawling and walking. In Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 335–354, Cambridge, 1994. MIT Press.

[254] Wolfgang Stolzmann. Learning classifier systems using the cognitive mechanism of anticipatory behavioral control, detailed version. In *Proceedings of the First European Workshop on Cognitive Modelling*, pages 82–89. Berlin: TU, 1996.

[255] Wolfgang Stolzmann. *Antizipative Classifier Systeme.* PhD thesis, Fachbereich Mathematik/Informatik, University of Osnabrück, 1997.

[256] Wolfgang Stolzmann. Two Applications of Anticipatory Classifier Systems (ACSs). In *Proceedings of the 2nd European Conference on Cognitive Science*, pages 68–73. Manchester, U.K., 1997.

[257] Wolfgang Stolzmann. Anticipatory classifier systems. In *Proceedings of the Third Annual Genetic Programming Conference*, pages 658–664. Morgan Kaufmann, 1998.

[258] Wolfgang Stolzmann. Latent Learning in Khepera Robots with Anticipatory Classifier Systems. In Wu [318], pages 290–297.

[259] Wolfgang Stolzmann. An Introduction to Anticipatory Classifier Systems. In Lanzi et al. [167], pages 175–194.

[260] Wolfgang Stolzmann and Martin Butz. Latent Learning and Action-Planning in Robots with Anticipatory Classifier Systems. In Lanzi et al. [167], pages 301–317.

[261] Wolfgang Stolzmann, Martin Butz, J. Hoffmann, and D. E. Goldberg. First cognitive capabilities in the anticipatory classifier system. In J. A. Meyer et al., editor, *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*, pages 287–296, 2000. Also Technical Report 2000008 of the Illinois Genetic Algorithms Laboratory.

[262] Richard Sutton. RL FAQ., 2002. http://www-anw.cs.umass.edu/~rich/RL-FAQ.html.

[263] Richard S. Sutton. Reinfocement learning architectures for animats. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB-90)*, pages 288–296. The MIT Press, 1991.

[264] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. The MIT Press, 1996.

[265] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[266] Keiki Takadama, editor. *Exploring New Potentials in Learning Classifier Systems. A Session of the 4th Japan-Australia Joint Workshop on Intelligent and Evolutionary Systems*. Ashikaga Institute of Technology, 2000.

[267] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[268] G. J. Tesauro. TD-Gammon, a self-teaching backgammon player, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[269] G. J. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[270] Sebastian Thrun. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Rheinhold, 1992.

[271] Sebastian Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, London, 1996.

[272] Andy Tomlinson. *Corporate Classifier Systems*. PhD thesis, University of the West of England, 1999.

[273] Andy Tomlinson and Larry Bull. A Corporate Classifier System. In A. E. Eiben, T. Bäck, M. Shoenauer, and H.-P. Schwefel, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving From Nature – PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 550–559. Springer Verlag, 1998.

[274] Hui Tong and Timothy X. Brown. Adaptive Call Admission Control Under Quality of Service Constraints: A Reinforcement Learning Solution. *IEEE Journal on Selected Areas in Communications*, 18(2):209–221, 2000.

[275] D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors. *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*. MIT Press, 1996.

[276] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202, 1994.

[277] Patrick Tufts. Evolution of a Clustering Scheme for Classifier Systems: Beyond the Bucket Brigade. PhD Thesis proposal. http://www.cs.brandeis.edu/~zippy/papers.htm, 1994.

[278] Patrick Tufts. Dynamic Classifiers: Genetic Programming and Classifier Systems. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 114–119, MIT, Cambridge, MA, USA, 1995. AAAI.

[279] Kirk Twardowski. Credit assignment for pole balancing with learning classifier systems. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA-93)*, pages 238–245. Morgan Kaufmann, 1993.

[280] J. Urzelai, Dario Floreano, Marco Dorigo, and Marco Colombetti. Incremental Robot Shaping. *Connection Science*, 10(3–4):341–360, 1998.

[281] Manuel Valenzuela-Rendón. Boolean analysis of classifier sets. In *Proceedings Third International Conference on Genetic Algorithms*, pages 351–358. Morgan Kaufmann, 1989.

[282] Manuel Valenzuela-Rendón. The Fuzzy Classifier System: a Classifier System for Continuously Varying Variables. In Booker and Belew [38], pages 346–353.

[283] Gilles Venturini. *Apprentissage Adaptatif et Apprentissage Supervisé par Algorithme Génétique.* PhD thesis, Université de Paris-Sud., 1994.

[284] Nickolas Vriend. Self-Organization of Markets: An Example of a Computational Approach. *Computational Economics*, 8(3):205–231, 1995.

[285] L. A. Wang. Classifier System Learning of the Boolean Multiplexer Function. Master's thesis, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[286] Christopher Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[287] Christopher Watkins and P. Dayan. Technical note: Q-Learning. *Machine Learning*, 8:279–292, 1992.

[288] Geoffrey I. Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research (JAIR)*, 4:397–417, 1996.

[289] P. J. Werbos. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 493–525. Van Nostrand Rheinhold, 1992.

[290] Darrell Whitely, David Goldberg, Erick Cantú-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*. Morgan Kaufmann, 2000.

[291] Jason R. Wilcox. Organizational Learning within a Learning Classifier System. Master's thesis, University of Illinois, 1995. Also Technical Report No. 95003 IlliGAL.

[292] Stewart W. Wilson. Knowledge Growth in an Artificial Animal. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 16–23, Pittsburgh, PA, July 1985. Lawrence Erlbaum Associates.

[293] Stewart W. Wilson. Classifier Systems and the Animat Problem. *Machine Learning*, 2:199–228, 1987.

[294] Stewart W. Wilson. Bid competition and specificity reconsidered. *Complex Systems*, 2:705–723, 1989.

[295] Stewart W. Wilson. Perceptron Redux: Emergence of Structure. *Physica D*, pages 249–256, 1990.

[296] Stewart W. Wilson. The Animat Path to AI. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB-90)*, pages 15–21. The MIT Press/Bradford Books, 1991.

[297] Stewart W. Wilson. ZCS: A Zeroth Level Classifier System. *Evolutionary Computation*, 2(1):1–18, 1994.

[298] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[299] Stewart W. Wilson. Explore/exploit strategies in autonomy. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour (SAB 4)*, pages 325–332. MIT Press/Bradford Books, 1996.

[300] Stewart W. Wilson. Personal communication, 1996.

[301] Stewart W. Wilson. Personal communication. November 21, 1996.

[302] Stewart W. Wilson. Response to a question by Lashon B. Booker in NetQ, 1996. http://world.std.com/~sw/.

[303] Stewart W. Wilson. Response to a question in NetQ, 1996. http://world.std.com/~sw/.

[304] Stewart W. Wilson. Generalization in the XCS classifier system. In John Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[305] Stewart W. Wilson. Personal communication. July 2, 1998.

[306] Stewart W. Wilson. Personal communication. June 1, 1998.

[307] Stewart W. Wilson. Personal communication. June 11, 1998.

[308] Stewart W. Wilson. Personal communication. June 30, 1998.

[309] Stewart W. Wilson. Get Real! XCS with Continuous-Valued Inputs. In Lanzi et al. [167], pages 209–219.

[310] Stewart W. Wilson. Mining Oblique Data with XCS. In *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2000), in the Joint Workshops of SAB 2000 and PPSN 2000* [2]. Extended abstract.

[311] Stewart W. Wilson. Personal communication. May 4, 2000.

[312] Stewart W. Wilson. State of XCS Classifier System Research. In Lanzi et al. [167], pages 63–82.

[313] Stewart W. Wilson. Personal communication. March 4, 2001.

[314] Stewart W. Wilson. Home Page, 2002. http://world.std.com/~sw/.

[315] Stewart W. Wilson and David E. Goldberg. A Critical Review of Classifier Systems. In Schaffer [225], pages 244–255.

[316] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1992.

[317] Sewell Wright. Evolution in Mendelian Populations. *Genetics*, 16:97–159, 1931.

[318] Annie S. Wu, editor. *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, 1999.

[319] Jeremy L. Wyatt. *Exploration and Inference in Learning from Reinforcement*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1997.

[320] Jeremy L. Wyatt. Reinforcement learning: a brief overview. In I. O. Stamatescu, editor, *Perspectives on Adaptivity and Learning*. Springer, 2002.

[321] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[322] W. Zhang. *Reinforcement Learning for Job-shop Scheduling*. PhD thesis, Oregon State University, 1996.

[323] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann, 1995.

[324] W. Zhang and T. G. Dietterich. High-performance job-shop scheduling with a time-delay TD(λ) network. In Touretzky et al. [275], pages 1024–1030.

Index

action set, 37
action-value functions, *see* reinforcement learning, action-value functions
agent, *see* reinforcement learning, agent
algorithms
  evolutionary, generic, 283
  Q-learning, *see* reinforcement learning, Q-learning
  reward function approximator, 257
  Sarsa, *see* reinforcement learning, Sarsa
  XCS, *see* XCS
animats, 283

back-up, *see* reinforcement learning, value, back-up
best action maps, 82–84
biased functions, 149
bucket brigade, 70–71, 298

chromosomes, *see* evolutionary algorithms, chromosomes
complete maps, 75, 84
complex systems, 22, 68–69, 216
control tasks, *see* reinforcement learning, control
covering, *see* XCS, covering
crossover, *see* evolutionary algorithms, crossover
crowding, *see* evolutionary algorithms, crowding
curse of dimensionality, 246

default hierarchies, 82, 138
design-based approach, 15
discounting return, *see* reinforcement learning, discounting return
Dynamic Programming, 252

endogenous fitness LCS, 116
environments, *see* learning tasks
evolution, 280–282
evolutionary algorithms
  chromosomes, 282
  comparison with GPI, 294–296
  crossover, 285
  crowding, 289
  elitism, 286
  evaluation methods for, 294
  fitness, 280, 282–284
  fitness sharing, *see* fitness sharing
  forms of, 279
  generalisation problem and, 288
  generational, 286
  genetic operators, 285
    in XCS, *see* XCS, genetic algorithm
  introductory material, 279
  mating restriction, 288–290
  mutation, 285
  niche GA, *see* LCS, niche GA
  niching, 288–290
  reinforcement learning and, 290–294

roulette wheel, 284
search, 286–287
selection, 284
steady-state, 286

fit overgeneral rules, *see* rule types
fitness, *see* evolutionary algorithms, fitness
fitness sharing
introduction, 288
Q-learning and classifier systems, 197
strong overgenerals and, 139, 189, 214
XCS and, *see* XCS, fitness sharing
function approximation
AE-tables, 265–266
classifier systems as, 100
evolutionary algorithms and, 288
reinforcement learning and, 269
state aggregation, 248–249, 265–269
storing value estimates, 268–269

GA-view of LCS
introduction, 14–15
revisited, 199–205
generalisation hypothesis, 71–73
genetic algorithms, *see* evolutionary algorithms
global optimum, 287
GPI, *see* reinforcement learning, Generalised Policy Iteration
greedy rule creation, 135

Holland, John, 22

implicit bucket brigade, 298

LCS
applications, 22
bibliography, 23
electronic paper archive, 23
GA-view of, *see* GA-view of LCS
history, 23
introductory references, 23
Michigan style, 23
model-based, 27
niche GA, 47
numerosity, 31
on-line resources, 23
Pittsburgh style, 23
adaptive power of, 207–208
policy and generalisation mechanisms, 202
renaissance, 22
RL-view of, *see* RL-view of LCS
rule notation, 28
rule representation, 25–28
SB–XCS, *see* SB–XCS
surveys, 23
ternary language, 25
traditional LCS, 24
types, 23
XCS, *see* XCS
learning paradigms
reinforcement learning, 233–234
supervised learning, 232, 247
unsupervised learning, 235
learning rate, 256
learning tasks
1-max, 284, 290–292, 294
biased variance task, 165
continuing, 238
control tasks, *see* reinforcement learning, control
episodic, 238
formalisation of, 241–243
function optimisation example, 286
hidden parity suite, 85
ice cream example, 258–259

MDP, *see* reinforcement learning, MDP
minimal strong overgenerals for SB–XCS, 159, 161
minimal strong overgenerals for XCS, 153
mountain car, 184–186
multiplexer
37 multiplexer, 85
definition, 57
evolved rules, 76–81, 85–87
initial results, 59–60
standard settings, 59
non-sequential, 236–239, 241
partial map gridworlds, 93–96
prediction tasks, *see* reinforcement learning, prediction
Russell and Norvig's maze, 264
evolutionary approach, 292–294
search, 286–287
sequential, 236–239, 242
strong undergenerals, 170
woods1, 182, 184
woods2
definition, 61–62
difficulty of, 62, 184, 182−−184
initial results, 63–65
revisited, 179–184
standard settings, 63
XCS-hard and SB–XCS-easy, 99–101, 164–168
local optimum, 287

macroclassifiers, 31, 219–221, 230
MAM technique, 39–40
Markov Decision Process, *see* reinforcement learning, MDP
match set, 36
Monte Carlo, *see* reinforcement learning, Monte Carlo methods
multiplexer, *see* learning tasks, multiplexer
mutation, *see* evolutionary algorithms, mutation

niche GA, *see* LCS, niche GA
non-sequential tasks, *see* learning tasks, non-sequential
numerosity, 31

off-policy evaluation, *see* reinforcement learning, off-policy evaluation
on-policy evaluation, *see* reinforcement learning, on-policy evaluation
optimality hypothesis, 75
overgeneral rules, *see* rule types

partial maps, 82–84
performance metrics, 119
policies, *see* reinforcement learning, policies
population state metrics, 121
prediction array, *see* XCS, prediction array
prediction tasks, *see* reinforcement learning, prediction
problems, *see* learning tasks

Q-learning, *see* reinforcement learning, Q-learning
Q-value, *see* reinforcement learning, Q-value

reinforcement learning, 231–244
action-value functions, 257
agent, 242–243
control, 252, 271, 274
discounting return, 240–241
Dynamic Programming, 252
evolutionary algorithms and, 290–294

explore/exploit dilemma, 234–236
function approximation and, *see* function approximation
Generalised Policy Iteration, 271–276
  comparison with evolution, 294–296
greedy policy, 273
immediate reward, 237
MDP, 242
model-based, 269
Monte Carlo methods, 260–261
  estimator density and, 269
Neuro-Dynamic Programming, 269
non-sequential prediction algorithm, 256–257
off-policy evaluation, 263
on-policy evaluation, 263
paradigm, *see* learning paradigms
policies, 242–243
  greedy policy, 273
  optimal policy, 273
policy evaluation, 251–270
policy improvement, 272–273
prediction, 252, 271, 274
Q-learning, 263
  classifier systems and, 196–199, 206
  evolution and, 203
Q-value, 253
return, 239–241
reward, 233
reward function, 242
RL-view of LCS
  introduction, 14–15
  revisited, 199–205
Sarsa, 262–263
Temporal Difference methods, 261–263
  estimator density and, 269
  value, 237
    back-up, 258
    dependency on policy, 259
    policy, 260
    state actions, 252–253
    storing value estimates, 268–269
  value functions, 257
representations
  Σ notation, 106
  best action maps, 82–84
  Boolean functions, 105–107
  comparison of complete and partial maps, 88–98
  complete maps, 84
  default hierarchies, 82, 138
  Disjunctive Normal Form, 106
  gaps in covering, 136
  off-set, 106
  on-set, 106
  partial maps, 82–84
  rule notation, 28
  rule representation, 25–28
  subsumed rules, *see* subsumed rules
  ternary language, 25
  truth tables, 105
restricted mating, *see* evolutionary algorithms, mating restriction
return, *see* reinforcement learning, return
reward, *see* reinforcement learning, reward
reward function, *see* reinforcement learning, reward function
RL-view of LCS
  introduction, 14–15
  revisited, 199–205
rule types

fit overgenerals, 147
SB–XCS, 168
survival in GA, 161–163
XCS, 164–167
overgeneral rules, 78
overgenerals, 81, 135, 140–141, 143, 145
strong and fit undergenerals, 168–169
strong overgenerals, 136, 146–147
other contributing factors, 190
SB–XCS, 160
XCS, 153–157
subsumed rules, 78, 80

Sarsa, *see* reinforcement learning, Sarsa
SB–XCS
comparison with ZCS, 56
polices, determining, 200–201, 205
Q-learning and, 198
rationale for adaptation, 76
specification, 52–57
system strength, 53
utility of, 188
schema theory, 69, 74–75
sequential tasks, *see* learning tasks, sequential
standard ternary language, *see* ternary language
state aggregation, *see* function approximation, state aggregation
strong overgeneral rules, *see* rule types
subsumed rules, *see* rule types, subsumed rules
subsumption deletion, 51–52, 54, 229
supervised learning, *see* learning paradigms, supervised learning
system prediction, *see* XCS, system prediction
system strength, *see* SB–XCS, system strength

tasks, *see* learning tasks
Temporal Difference, *see* reinforcement learning, Temporal Difference methods
ternary language, 25

unsupervised learning, 235, *see* learning paradigms, unsupervised learning

value functions, *see* reinforcement learning, value functions

woods2, *see* learning tasks, woods2

XCS
accuracy
estimating, 267
action selection, 37
action set, 37
complete maps, 75
covering, 46–47, 51, 58, 59, 63
credit assignment algorithm, 40
credit assignment system, 38–45
deletion, 50–51
diagram, 36
example cycle, 221–230
explore/exploit framework, 33
features of, 30
fitness sharing in, 45
generalisation hypothesis, 71–73
genetic algorithm, 47–51
macroclassifiers, *see* macroclassifier

MAM technique, *see* MAM technique
match set, 36
numerosity, 31
optimality hypothesis, 75
overview, 30–33
parameter initialisation, 51
parameter settings
multiplexer, 59
woods2, 63
performance system, 35–38
performance system algorithm, 36
polices, determining, 201, 205
prediction array, 36
Q-learning and, 198
rationale for adaptation, 71–73
rule discovery algorithm, 47
rule discovery system, 45–52
schema theory, 74–75
specification, 28–52
subsumption deletion, *see* subsumption deletion
system prediction, 36
XCS-hard functions, 101, 168

ZCS
Sarsa and, 298
SB–XCS and, 56
strong overgenerals and, 136
woods2 and, 182–184
XCS and, 29

Errata

Errata July 2005.

- The note on p. 41 that (2.3) reduces to (2.2) when $\gamma = 0$ neglects to mention that we must also move the update from $[\mathrm{A}]_{-1}$ to $[\mathrm{A}]$ (which we do for non-sequential tasks) for this reduction to occur.
- The errata page says "Index" in the top-left corner.