Genetic and Evolutionary Computation

Wolfgang Banzhaf
Lee Spector
Leigh Sheneman *Editors*

Genetic Programming Theory and Practice XVI

Springer

Genetic and Evolutionary Computation

Wolfgang Banzhaf • Lee Spector • Leigh Sheneman
Editors

Genetic Programming Theory and Practice XVI

Springer

Editors
Wolfgang Banzhaf
Computer Science and Engineering
John R. Koza Chair, Michigan State University
East Lansing, MI, USA

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA, USA

Leigh Sheneman
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, USA

ISSN 1932-0167 ISSN 1932-0175 (electronic)
Genetic and Evolutionary Computation
ISBN 978-3-030-04734-4 ISBN 978-3-030-04735-1 (eBook)
https://doi.org/10.1007/978-3-030-04735-1

Library of Congress Control Number: 2018965727

We dedicate this book to the memory of the co-founder of the workshop series on Genetic Programming Theory and Practice, Rick Riolo, who passed away on August 25, 2018.

Foreword

I first met Rick Riolo in the late 1990s. I'd followed a well-beaten path by contacting John Holland to talk about GP. John politely referred me to Rick, and when I got to Rick's office in the Program for the Study of Complex Systems, I found a tall, slightly balding man in a tie-dye shirt with a face like a figure out of Leonardo's sketchbook. I asked if he was Rick Riolo. He agreed that he was, and we talked for about an hour about a scheme that I had for a Genetic Programming system. He was courteous but to the point saying what I was proposing was something he hadn't heard of before but was interested in seeing if it would work. This was the start of a friendship that lasted until his death this year. Over the years, we did some consulting work together, our families socialized, and we worked on several of the GPTP Workshops over the years.

He introduced himself as being in charge of the hardware at CSCS—I took this to mean he was a tech. I quickly learned that he was much more than that! Rick was one of the members of the BACH group at CSCS and was known in particular for his use of GAs to study the prisoner's dilemma. He had been working with GAs for decades with John Holland and the other members of the BACH group and was one of the early people working and teaching at the Santa Fe Institute.

Elsewhere I've told the story of how GPTP came to be, but while I made some suggestions of the organization of GPTP, Rick was the person who was the one constant throughout the years until he became too ill to manage the workshop. With the gracious staff of CSCS and Rick's quiet skill at making hard things look easy, GPTP was always on an even keel for the days of the workshop. More and more over the years, GPTP grew in importance under Rick's quiet stewardship.

However, though Rick was quiet, he was funny and his humor was often acerbic. One time, when we were putting together one of the early books, we had an author whose chapter was way over the page limit we set. We had sent it back and asked him to cut it in half, and when he returned it, he had cut maybe two pages. I expressed my frustration as time was getting short, but all Rick said was that for a smart man, he couldn't count very well.

When we started GPTP, we had no idea that it would last as long as it did. In fact, we thought it was a one-off event. As the years past, I would often sit next to Rick for at least one of the days, and after a particularly exciting talk, I leaned over to Rick and asked him if GPTP reminded him of the early years of GAs. He paused a moment and said that it reminded him more of the early days of the Santa Fe Institute.

When his disease kept him from joining us at GPTP, we took GPTP to him by live streaming and also visiting him at his home. In 2015 John Holland, who was one of the godfathers of GPTP, died. John was a close friend of Rick, and in the normal course of things, Rick would have written the dedication to John in the Foreword of that year's GPTP book, but his disease stopped him from doing so. He asked me to write them a thing that was both heartbreaking and daunting, but since Rick asked, I did my best. When I had done, I read it to him and told him that I tried to find the words that he would have used and wished with all my heart that he could have written them. We both cried a little, and I left that night knowing that Rick was slipping away from us.

When I heard from Carl Simon that Rick had died, I sat and thought about the years I'd known Rick, the things we'd done together, particularly at GPTP. Our children had grown up and were adults. CSCS had changed; Carl Simon, who had started the ball rolling for GPTP, had retired. But in thinking of Rick, I am grateful for all that he did for us, and I know how much we will all miss him.

Ann Arbor, MI, USA Bill Worzel
October 2018

Preface

The 16th instance of the *Workshop on Genetic Programming Theory and Practice* (GPTP) was held in Ann Arbor, Michigan, from May 17 to May 20, 2018. It was held at the University of Michigan and was organized and supported by the University's Center for the Study of Complex Systems.

This book contains the written contributions of the workshop's participants. Each contribution was drafted, read, and reviewed by other participants prior to the workshop. Each was then presented at the workshop and subsequently revised after the workshop on the basis of feedback received during the event.

GPTP has long held a special place in the genetic programming community, as an unusually intimate, interdisciplinary, and constructive meeting. It brings together researchers and practitioners who are eager to engage with one another deeply, in thoughtful, unhurried discussions of the major challenges and opportunities in the field.

Participation in the workshop is by invitation only, and an effort is made to invite a group of participants each year that is diverse in several ways, including participants both from academia and industry. Efforts are also made to include participants in "adjacent" fields such as evolutionary biology.

GPTP is a single-track workshop, with a schedule that provides ample time for presentations and for discussions, both in response to specific presentations and on more general topics. Participants are encouraged to contribute observations from their own, unique perspectives and to help one another to engage with the presented work. Often, new ideas are developed in these discussions, leading to collaborations after the workshop.

Aside from the presentations of regular contributions, the workshop also features keynote presentations that are chosen to broaden the group's perspective on the theory and practice of genetic programming. This year, the workshop began with a keynote presented by longtime GPTP participant Katya Vladislavleva, now the CEO of DataStories, on "Moonshot thinking and abundance mentality for better data science." On the second day, the keynote was presented by Walter Fontana, Professor of Systems Biology at Harvard Medical School, on "Actual causality in rule-based models." The third and final keynote was delivered by Marco Tomassini, Professor Emeritus in the Department of Information Systems at the University of Lausanne, on “Strategic games: theory and human behavior.” As can be gathered from their titles, none of these talks focused explicitly on genetic programming per se. But each presented fascinating developments that connect to open issues in genetic programming theory and practice in intriguing ways.

While most readers of this volume will not have had the pleasure of attending the workshop’s presentations and discussions, our hope is that they will nonetheless be able to appreciate and engage with the ideas that were presented. We also hope that all readers will gain an understanding of the current state of the field and that those who seek to do so will be able to use the work presented herein to advance their own work and to make additional contributions to the future of the field.

Acknowledgments

We would like to thank all of the participants for again making GP Theory and Practice a successful workshop 2018. As is always the case, it produced a lot of interesting and high-energy discussions, as well as speculative thoughts and new ideas for further work. The keynote speakers did an excellent job at raising our awareness and provided thought-provoking ideas about the potential of Genetic Programming and its place in the world.

We would also like to thank our financial supporters for making the existence of GP Theory and Practice possible for the past 15 years and counting. For 2018, we did not have a major fund-raising drive but instead made good use of what had been left from earlier workshops. Those funds were contributed by:

- The Center for the Study of Complex Systems at the University of Michigan
- John Koza
- Jason H. Moore
- Babak Hodjat at Sentient
- Michael Korns, Lantern LLC
- Mark Kotanchek at Evolved Analytics
- Stuart Card
- Thomas Kern
- The Heuristic and Evolutionary Algorithms Laboratory at the Upper Austria University of Applied Sciences

A number of people made key contributions to the organization and assisted our participants during their stay in Ann Arbor. Foremost among them are Linda Wood and Mita Gibson who made the workshop run smoothly with their diligent efforts behind the scene before, during, and after the workshop. Special thanks go to the Springer Publishing Company for providing the editorial assistance for producing this book. We are particularly grateful for the contractual assistance by Melissa Fearon at Springer and all their staff has done to make this book possible.

We would like to express our gratitude especially to Carl Simon and Charles Doering, the champions of the workshop series at the Center for the Study of Complex Systems. Finally and foremost, we want to thank Rick Riolo for his dedication to the workshop series since its beginning. With the passing of Rick, one of the founders of the workshop series, we are entering a new era for GPTP. This 16th edition represents a transition with some organizational changes that will need to be considered to secure the future of this very successful workshop series.

East Lansing, MI, USA	Wolfgang Banzhaf
Amherst, MA, USA	Lee Spector
East Lansing, MI, USA	Leigh Sheneman
October 2018	

Contents

1 Exploring Genetic Programming Systems with MAP-Elites 1
Emily Dolson, Alexander Lalejini, and Charles Ofria
1.1 Introduction 1
1.2 Methods 3
1.2.1 Computational Substrate 3
1.2.2 Evolution 6
1.2.3 Experimental Design 8
1.3 Results and Discussion 10
1.4 Conclusion 14
References 15

2 The Evolutionary Buffet Method 17
Arend Hintze, Jory Schossau, and Clifford Bohm
2.1 Introduction 17
2.2 Methods 18
2.2.1 Markov Brains: An Introduction 19
2.2.2 Genetic Encoding 20
2.2.3 Direct Encoding 22
2.2.4 Multi-Step Functions 22
2.2.5 Gate Types 22
2.2.6 Tasks 23
2.2.7 Experimental Parameters 30
2.3 Results 30
2.4 Discussion and Conclusion 32
2.5 Future Work 34
References 35

3 Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial 37
Stephen Kelly, Robert J. Smith, and Malcolm I. Heywood
3.1 Introduction 37
3.2 Related Work 38
3.2.1 Evolving Graphs 38
3.2.2 Evolution of Multiple Programs Without Graphs 40
3.3 Visual Reinforcement Learning 41
3.4 Tangled Program Graphs 42
3.4.1 Developmental Cycle 45
3.4.2 Variation 46
3.4.3 Agent Evaluation 47
3.5 Case Study: Arcade Learning Environment 49
3.6 Case Study: VizDoom 51
3.7 Discussion 54
References 55

4 Strong Typing, Swarm Enhancement, and Deep Learning Feature Selection in the Pursuit of Symbolic Regression-Classification 59
Michael F. Korns and Tim May
4.1 Introduction 59
4.2 Comparison Algorithms 61
4.2.1 AMAXSC in Brief 61
4.2.2 MDC in Brief 61
4.2.3 M_2GP in Brief 62
4.2.4 LDA Background 62
4.2.5 LDA Matrix Formalism 64
4.2.6 LDA Assisted Fitness Implementation 65
4.2.7 Bees Swarm Optimization 69
4.3 User-Defined Typing System 69
4.3.1 User-Defined Templates with Constraints 70
4.3.2 Strong Typing 72
4.4 Deep Learning Enhancements 72
4.5 Artificial Test Problems 75
4.6 Real World Banking Problem 76
4.7 Performance on the Theoretical Problems 78
4.8 Performance on the Real World Problem 78
4.9 Conclusion 80
Appendix: Artificial Test Problems 80
References 83

5 Cluster Analysis of a Symbolic Regression Search Space 85
Gabriel Kronberger, Lukas Kammerer, Bogdan Burlacu, Stephan M. Winkler, Michael Kommenda, and Michael Affenzeller
5.1 Introduction 85
5.2 Methodology 87
5.2.1 Grammar Enumeration 88
5.2.2 Phenotypic Similarity 90
5.2.3 Genotypic Similarity 91
5.2.4 Clustering and Visualization 91
5.2.5 Mapping GP Solution Candidates 93
5.3 Results 93
5.3.1 Phenotypic Mapping 94
5.3.2 Genotypic Mapping 96
5.3.3 Cluster Qualities for Benchmark Problems 97
5.3.4 Mapping of GP Solution Candidates 98
5.4 Discussion 99
5.5 Conclusion 101
References 101

6 What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP 103
Alexander Lalejini and Charles Ofria
6.1 Introduction 103
6.2 SignalGP 105
6.3 The Value of Imprecision in Evolvable Names 106
6.3.1 The Changing Environment Problem 107
6.3.2 Results and Discussion 109
6.4 The Value of Not Listening 112
6.4.1 The Distracting Environment Problem 113
6.4.2 Results and Discussion 114
6.5 What Else Is in an Evolved Name? Broadened Applications of Tag-Based Naming in SignalGP 117
6.5.1 SignalGP Function Regulation 117
6.5.2 Multi-Representation SignalGP 117
6.5.3 Major Transitions in SignalGP 118
6.6 Conclusion 119
References 120

7 Lexicase Selection Beyond Genetic Programming 123
Blossom Metevier, Anil Kumar Saini, and Lee Spector
7.1 Introduction 123
7.2 Lexicase Selection 124
7.3 Problems 126
7.3.1 Boolean Constraint Satisfaction 126
7.3.2 Random Problem Generation 126

7.4 Experimental Methods 127
7.4.1 Genetic Algorithm 127
7.4.2 Variation 128
7.4.3 Parent Selection 128
7.5 Results 129
7.5.1 Success Rates by Parent Selection Method 129
7.5.2 Success Rates by Tournament Size 130
7.5.3 Errors over Evolutionary Time 131
7.5.4 Mean Least Error 131
7.5.5 Success Generations 132
7.5.6 Diversity over Evolutionary Time 133
7.6 Discussion 133
References 135

8 Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems 137
Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanc
8.1 Introduction 137
8.2 Related Work 138
8.3 The Neuron Model 140
8.3.1 Model Parameters 144
8.3.2 Developing the Brain and Evaluating the Fitness 145
8.3.3 Updating the Brain 146
8.3.4 Running and Updating the Soma 147
8.3.5 Updating the Dendrites and Building the New Neuron 147
8.4 Cartesian GP 149
8.5 Benchmark Problems 151
8.6 Experiments and Results 151
8.7 Tables of Results 153
8.8 Comparisons and Statistical Significance 154
8.9 Evolved Developmental Programs 156
8.10 Developed ANNs for Each Classification Problem 156
8.11 Evolving Neural Learning Programs 158
8.12 Further Work 163
8.13 Conclusions 164
Appendix: Detailed Algorithms 165
References 176

9 The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming 179
Michael O'Neill and David Fagan
9.1 Introduction 179
9.2 A Journey with Genetic Programming and Automatic Programming 180
9.3 A Journey in Software-Defined Communications Networks 181
9.3.1 Network Scheduling 184
9.3.2 Network Configuration 186

9.3.3 Combining Network Configuration and Scheduling 187
9.3.4 Summary 188
9.4 Discussion and Concluding Remarks 188
References 190

10 Untapped Potential of Genetic Programming: Transfer Learning and Outlier Removal 193
Leonardo Trujillo, Luis Muñoz, Uriel López, and Daniel E. Hernández
10.1 Introduction 193
10.2 Transfer Learning 195
10.2.1 Case Study 196
10.2.2 Experiments and Results 197
10.2.3 Discussion 200
10.3 Detecting Outliers 200
10.3.1 Case Study 202
10.3.2 Experiment and Results 202
10.3.3 Discussion 204
10.4 Conclusions and Future Outlook 205
References 205

11 Program Search for Machine Learning Pipelines Leveraging Symbolic Planning and Reinforcement Learning 209
Fangkai Yang, Steven Gustafson, Alexander Elkholy, Daoming Lyu, and Bo Liu
11.1 Introduction 209
11.2 Related Work 211
11.3 Background 213
11.3.1 Symbolic Planning 213
11.3.2 Reinforcement Learning 214
11.4 Methodology 215
11.4.1 Pipeline Generation 215
11.4.2 Parameter Tuning and Pipeline Evaluation 218
11.4.3 AutoML Framework 219
11.5 Empirical Evaluation and Discussion 221
11.5.1 Dataset and Problem Instance 221
11.5.2 Hypotheses 222
11.5.3 Pipeline Profiling 222
11.5.4 Optimal Pipeline Generation 224
11.5.5 Pipeline Ranking 225
11.5.6 Extracted Knowledge About Problem 228
11.6 Conclusion and Future Work 229
References 230

Index 233

Contributors

Michael Affenzeller Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Clifford Bohm Department of Integrative Biology and BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA

Bogdan Burlacu Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria

Sylvain Cussat-Blanc University of Toulouse, IRIT - CNRS - UMR5505, Toulouse, France

Emily Dolson BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA

Alexander Elkholy Maana, Inc., Bellevue, WA, USA

David Fagan Natural Computing Research & Applications Group, School of Business, University College Dublin, Dublin, Ireland

Steven Gustafson Maana, Inc., Bellevue, WA, USA

Daniel E. Hernández Instituto Tecnológico de Tijuana, Tijuana, B.C., México

Malcolm I. Heywood Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada

Arend Hintze Department of Integrative Biology and Department of Computer Science and Engineering and BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA

Lukas Kammerer Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria

Stephen Kelly Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada

Michael Kommenda Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria

Michael F. Korns Lantern Credit LLC, Henderson, NV, USA

Gabriel Kronberger Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria

Alexander Lalejini BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA

Bo Liu Auburn University, Auburn, AL, USA

Uriel López Instituto Tecnológico de Tijuana, Tijuana, B.C., México

Daoming Lyu Auburn University, Auburn, AL, USA

Tim May Insight Sciences Corporation, Henderson, NV, USA

Blossom Metevier College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA

Julian F. Miller University of York, Heslington, York, UK

Luis Muñoz Instituto Tecnológico de Tijuana, Tijuana, B.C., México

Charles Ofria BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA

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

Anil Kumar Saini College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA

Jory Schossau Department of Integrative Biology and BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA

Robert J. Smith Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada

Lee Spector School of Cognitive Science, Hampshire College, Amherst, MA, USA

College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA

Leonardo Trujillo Instituto Tecnológico de Tijuana, Tijuana, B.C., México

Dennis G. Wilson University of Toulouse, IRIT - CNRS - UMR5505, Toulouse, France

Stephan M. Winkler Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Fangkai Yang Maana, Inc., Bellevue, WA, USA

Chapter 1
Exploring Genetic Programming Systems with MAP-Elites

Emily Dolson, Alexander Lalejini, and Charles Ofria

1.1 Introduction

When programmers write code, they ideally want to structure it to be fast to implement, easy to extend, and clear for others to understand. Of course, these properties aren't usually compatible: program architectures that are fastest to write are usually not simple to extend or understand. We face a similar problem when we evolve programs with genetic programming (GP); the solutions that evolve most easily are typically a tangled mess. They are not useful as building blocks for solving more complex problems (i.e., they are not evolvable), and they are challenging to tease apart what is going on. Because program architecture is so important for evolvability and decomposability, substantial effort goes into developing genetic programming systems that promote the evolution of programs with more evolvable architectures. For example, modularity is an important principle of software design and is also known to be an important component of evolvability [9], which has led many to design genetic programming systems with the goal of promoting the evolution of modular code [11, 24, 26].

Indeed, within the GP community, we have an abundance of ways to represent programs that we expect will be evolvable or interpretable, each with its own unique set of available programmatic elements and ways of organizing, interpreting, and mutating programs. Given the diversity of GP representations, understanding how to choose the most appropriate representation or configuration of a representation for a particular problem is an open issue in the field [17]. Making headway on this issue

E. Dolson (✉) · A. Lalejini · C. Ofria
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA
e-mail: dolsonem@msu.edu; lalejini@msu.edu; ofria@msu.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_1

requires expanding the existing toolkit of formal analyses for GP representations. While many different high-level properties of code can be considered, for the rest of this chapter we will focus on code evolvability.

In particular, it would be helpful to have a way to get insight into the range of program architectures that a given representation is capable of evolving. Doing so will help us disentangle issues related to the representation itself from issues with the way the rest of an evolutionary algorithm is set up (selection for evolvability is notoriously challenging to facilitate). Moreover, having access to examples of programs with different architectures is critical to setting up experiments that will tell us when these architectures are useful and how they interact with other features of a given representation. Ultimately, a tool for exploring program architectures would aid us in drawing generalizable insights that may be useful for others in the field.

The issue of wanting access to programs with a range of different architectures is an instance of a common problem. Often in evolutionary computation, we would like to evolve good solutions that are diverse with respect to a set of phenotypic traits [18]. For example, we might want to provide a variety of options to stakeholders making a decision [4]. Alternatively, we might want to provide a robot with alternative locomotion strategies to use if it gets damaged [5]. MAP-Elites has been demonstrated to be an effective technique for evolving a diverse set of solutions to a problem [15].

In MAP-Elites, the user selects some number of phenotypic axes that they expect will be relevant to solving the problem but might not be directly correlated with fitness. Each axis is then discretized into some pre-determined number of bins, resulting in a multi-dimensional grid where a location in the grid corresponds to a distinct combination of phenotypic traits. When a new solution is generated, its phenotype is assessed, and it is placed into a bin corresponding to that phenotype. If that bin is empty or occupied by a lower-fitness individual, the new solution replaces it. Otherwise, it is discarded.

Thus far, MAP-Elites had been used to evolve robot arms [6], robot gaits [5], soft-bodied robots [15], and neural networks for a computer vision task [15]. Interestingly, in this last task, the phenotypic axes (connection cost and modularity) related to the morphology of the neural networks themselves. Using these axes, MAP-Elites not only found a range of good solutions, but provided insight into the topology of the underlying fitness landscape as it relates to these two traits. The heat map produced by MAP-Elites shows which types of networks are capable of succeeding at the task and what constraints network traits place on each other. Here, we attempt to do the same for GP. Are there multiple programmatic paths to solving a problem? Can we identify inherent trade-offs between different traits (e.g., modularity, instruction composition, etc.) in evolving programs? We see MAP-Elites as a tool that can be used to answer these questions by illuminating interactions between different aspects of a GP representation when applied to a problem. This increased understanding can help in building an intuition for what types of programs might be most appropriate for a given problem, which can be used to inform representation choice, population initialization, or mutation operators.

In this chapter, we demonstrate the use of MAP-Elites to explore a simple linear GP representation. By selecting phenotypic axes for MAP-Elites that correspond to program architecture and instruction composition, we can show how relevant different features of our GP representation are to the evolutionary process across a variety of problems. We compare the forms of programs evolved under MAP-Elites, lexicase selection, tournament selection, and random drift, demonstrating that MAP-Elites produces more varied solutions. Further, we discuss additional program traits that could be used as phenotypic axes in MAP-Elites that appear promising, but we have not yet explored in this work.

1.2 Methods

1.2.1 Computational Substrate

For this study, we evolve linear genetic programs where each program is a linear sequence of instructions, and each instruction has up to three arguments that may modify its behavior. Most notably, our linear genetic programming (LGP) computational substrate supports subroutines, allowing programs with modular architectures to evolve.

Making efficient use of modular subroutines has long been thought to be important to facilitating the evolution of genetic programs that solve complex problems. Indeed, incorporating modules into GP has been extensively explored, and their benefits have been well documented (e.g., [1, 8, 10, 11, 20, 21, 24, 26]). We designed our LGP representation to facilitate the evolution of modular and reusable code while minimally affecting the way in which traditional linear genetic programs are organized (i.e., linear sequences of instructions). We include a number of instructions in the language that automatically create programming structures like loops and subroutines, and we enable these features through the concept of "scopes", which are functionally similar to "environments" in Push [22, 25].

1.2.1.1 Virtual CPU Hardware

Our linear genetic programs are executed in the context of a virtual CPU with the following components:

- **Instruction Pointer:** A marker to indicate the position in the genome currently being executed. Many instructions will influence how the instruction pointer (IP) moves through the genome.
- **Registers:** Each virtual CPU has 16 registers. Programs can store a single floating-point value in each register. Registers are initialized with numbers corresponding to their ID (e.g., register 0 is initialized to the value 0.0, register 1 is initialized to the value 1.0, and so on).

- **Stacks:** Each virtual CPU has 16 stacks. Programs can push floating point numbers onto these stacks and pop them off later.
- **Inputs:** Each virtual CPU can accept an arbitrary number of input values. These values do not need to be in any particular order, but each value needs to be associated with a unique numerical label that the program can use to access it. For the purposes of this paper, we always use sequential integers starting at 0.
- **Outputs:** Outputs function the same way as inputs. The only difference between inputs and outputs is the way instructions interact with them; whereas instructions can read from inputs but not write to them, instructions can write to outputs but not read from them.
- **Scopes:** Each virtual CPU has 16 scopes (plus the global scope), described in the next section.

1.2.1.2 Scopes

In software development, the *scope* of a variable specifies the region of code in which that element may be used. In a sense, a scope is like a programmatic membrane, capable of encapsulating all manner of programmatic elements, such as variables, functions, *et cetera*, and allowing regions to be looped through or skipped entirely. Our LGP representation gives evolving programs control over instruction- and memory-scoping, allowing programs to easily manage flow control and variable lifetime.

In our LGP representation, scopes provide the backbone on top of which all of the other modularity-promoting features, such as loops and functions, are built. All instructions in a program exist within a scope, be it the default outermost scope or one of the 16 other available scopes (making 17 possible scopes) that can be accessed via various instructions. These 16 inner scopes have a hierarchy to them, such that higher-numbered scopes are always nested inside lower-numbered scopes.

At the beginning of the program, all instructions before the first change of scope are in the outermost scope. After a scope-changing instruction occurs in the genome, subsequent instructions are added to the new scope until another scope-changing instruction is encountered, and so on. These scopes are ordered numerically. Higher-numbered scopes are always nested inside lower-numbered scopes. Scopes can be exited with the `break` instruction or by any instruction that moves control to a lower-numbered scope.

Scopes are also the foundation of program modules (functions). The `define` instruction allows the program to put instructions into a scope and associate the contents of that scope with one of 16 possible function names. Later, if that function is called (using the `call` instruction), the program enters the scope in which that function was defined and executes the instructions within that scope in sequence, including any internal (nested) scopes.

Similarly, scopes are the foundation of loops. Two kinds of loops exist in the instruction set used here: while loops and countdown loops. Loops of both types have a corresponding scope, which contains the sequence of instructions that make up the body of the loop. Both types of loops repeat their body (i.e., the contents of their associated scope) until the value in an argument-specified register is 0. Countdown loops automatically decrement this register by one on every iteration. When any instruction is encountered that would cause the program to leave the current scope, the current iteration is ended and the next one begins.

1.2.1.3 Instructions

In this work, evolving programs can contain the following library of 26 different instructions. For each, instruction arguments are limited to 16 values (0 through 15) and are used to specify any of the following: registers, scopes, functions, inputs, or outputs. Arguments for each instruction and their types are shown in the parentheses after each instruction name.

- **`Inc`** *(Register A)*: Increment the value in register A by 1.
- **`Dec`** *(Register A)*: Decrement the value in register A by 1.
- **`Not`** *(Register A)*: If Register A equals 0.0, set to 1.0. Otherwise set to 0.0.
- **`SetReg`** *(Register A, Value B)*: Store numerical value of B into register A.
- **`Add`** *(Register A, Register B, Register C)*: Add the value in register A to the value register B and store the result in register C.
- **`Sub`** *(Register A, Register B, Register C)*: Subtract the value in register B from the value in register A and store the result in register C.
- **`Mult`** *(Register A, Register B, Register C)*: Multiply the value in register A by the value in register B and store the result in register C.
- **`Div`** *(Register A, Register B, Register C)*: Divide the value in register A by the value in register B and store the result in register C.
- **`Mod`** *(Register A, Register B, Register C)*: Calculate the value in register A modded by the value in register B and store the result in register C.
- **`TestEqu`** *(Register A, Register B, Register C)*: Store the value 1.0 in register C if the value in register A is equal to value in register B. Otherwise store the value 0.0 in register C.
- **`TestNEqu`** *(Register A, Register B, Register C)*: Store the value 0.0 in register C if the value in register A is equal to value in register B. Otherwise store the value 1.0 in register C.
- **`TestLess`** *(Register A, Register B, Register C)*: Store the value 1.0 in register C if the value register A is less than the value in register B. Otherwise store the value 0.0 in register C.
- **`If`** *(Register A, Scope B)*: If the value in register A is not 0.0, continue to scope B, otherwise skip scope B.
- **`While`** *(Register A, Scope B)*: Repeat the contents of scope B until the value in register A is equal to 0.
- **`Countdown`** *(Register A, Scope B)*: Repeat the contents of scope B, decrementing the value in register A each time, until the value in register A is 0.
- **`Break`** *(Scope A)*: Break out of scope A.

- `Scope` *(Scope A)*: Enter scope A.
- `Define` *(Function A, Scope B)*: Define this position as the starting point of function A, with its contents defined by scope B. The function body is skipped after being defined; when called, the function automatically returns when scope B is exited.
- `Call` *(Function A)*: Call function A (must have already been defined).
- `Push` *(Register A, Stack B)*: Push the value in register A onto stack B.
- `Pop` *(Stack A, Register B)*: Pop the top value off of stack A and store it in register B.
- `Input` *(Input A, Register B)*: Store the value in input A in register B.
- `Output` *(Register A, Output B)*: Write the value in register A to output B.
- `CopyVal` *(Register A, Register B)*: Copy the value in register A into register B.
- `ScopeReg` *(Register A)*: Backup the value in register A. When the current scope is exited, it will be restored.
- `Dereference`*(Register A, Register B)*: Store the value of the register specified by the value of register A in register B.

1.2.2 Evolution

1.2.2.1 Selection Operators

MAP-Elites
The Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm is designed to illuminate search spaces and has been demonstrated as an effective technique for evolving a diverse set of solutions to a problem [15]. In MAP-Elites, a population is structured based on a set of chosen phenotypic traits. Each chosen trait defines an axis on a grid of cells where each cell represents a distinct combination of the chosen traits. Cells maintains only the most fit (elite) solution discovered with that cell's associated combination of traits. A MAP-Elites grid is initialized by randomly generating solutions and placing them into their appropriate cell in the grid (based on the random solution's traits). After initialization, occupied cells are randomly selected to reproduce. When a solution is selected for reproduction, we generate a mutated offspring and determine where that offspring belongs in the grid. If the cell is unoccupied, the new solution is placed in that cell; otherwise, we compare the new solution's fitness to that of the current occupant, keeping the fitter of the two. Over time, this process produces a grid of solutions that span the range of traits we used to define our grid axes.

Tournament Selection
To understand whether it's valuable to use MAP-Elites to explore the range of potential program architectures (rather than simply looking at the architectures evolved under the selection scheme that is already being used), we need to compare

it to a more standard approach to selection. Here, tournament selection with a tournament size of two will serve as that control. Any time we need to generate an offspring program using tournament selection, we select two random programs from the population and allow the fitter one to reproduce. We intentionally selected the lowest possible tournament size to minimize the strength of selection, facilitating as much diversification as possible.

Lexicase Selection
Tournament selection is known to be bad at maintaining diversity in a population. Lexicase selection [23] is known to maintain phenotypic diversity, although little is known about the diversity of program architectures within these populations. To better understand whether our proposed use of MAP-Elites is more effective at exploring the range of possible program architectures than simply maintaining a diverse population, we compare it to the results of using standard lexicase selection.

In lexicase selection, all of the test cases for a problem are randomly re-ordered each time another program is being selected to reproduce. We then go through the test cases in order and, for each test case, remove all but the top performing programs from the set of candidates for selection. When there is only one program remaining, we allow it to reproduce. In case of a tie we choose randomly.

Random Drift
What types of program architectures arise in the absence of selection pressure (i.e., from purely random drift)? We additionally compare the range of evolved program architectures produced by MAP-Elites with those produced under random drift where we select programs to reproduce at random. Although we do not expect any of these programs to actually solve any of our test problems, they will provide insight into large scale statistical trends that we might expect in the absence of selection.

1.2.2.2 Mutation Operators

In this work, we propagated programs asexually and applied consistent rates of mutations to offspring across all treatments. We used four different operators to introduce mutations on reproduction: instruction substitutions, argument substitutions, point insertions, and point deletions [3]. Instruction substitutions, in which one instruction was randomly replaced with another instruction, had a 0.005 chance of occurring at each site in the genome. Argument substitutions, in which one argument to an instruction was randomly replaced with another, had a 0.005 chance of occurring for each argument in the genome. In point insertions, a random instruction was added after a given site, increasing the length of the genome. Conversely, point deletions removed the instruction at a given site, decreasing the length of the genome. Point insertions and deletions both had a 0.005 chance of happening at every site in the genome.

1.2.3 Experimental Design

Is MAP-Elites an effective technique for exploring how different aspects of evolving program architectures interact to affect performance in GP? In this work, we evolve linear genetic programs with MAP-Elites to solve four simple programming synthesis problems, selecting phenotypic axes that correspond to program architecture and instruction composition. For each problem, we compare the types of programs evolved with MAP-Elites and with lexicase and tournament selection; additionally, we compare the types of programs evolved with MAP-Elites to programs produced via random drift.

1.2.3.1 MAP-Elites Phenotype Axes

As a proof-of-concept, we have selected two phenotypic axes that we expected to promote a useful exploration of the features of our linear GP representation. In Sect. 8.13, we discuss additional axes that may also prove to be generally useful for exploring GP representations.

Program Composition
A representation's instruction set has a huge impact on what programs are able to do. However, predicting the importance of an instruction *a priori* can be challenging. MAP-Elites can help in understanding instructions' importance in various contexts. Theoretically, a wide variety of phenotypic traits related to instructions could be used. For example, for any individual instruction, the number of times it is used could be an axis.

For the purposes of getting a high-level understanding of the range of programs that can evolve, however, we have chosen to use the overall diversity of instructions in a program as an axis. Here, we quantify the diversity of instructions in a program using Shannon entropy. This measurement provides high-level information about the genotype as whole. Importantly, it cannot be easily altered through small numbers of neutral mutations, meaning it should be informative about practical differences between genomes. We discretize this value into 20 bins between 0 and the maximum possible entropy for a program.

Module Use
Our representation is centered around modules in the form of scopes. As we attempt to understand whether this programming paradigm is useful to evolution, it is critical to understand the extent to which scopes are used. Thus, we chose the number of scopes used by a program as our second phenotypic axis. Importantly, we only counted scopes that were actually used; when a program is run, it must execute at least one instruction in a given scope to get credit for using that scope. This, however, does not guarantee that scopes are used *meaningfully*, only that they are used. Since this measurement is already an integer value, we used 17 bins along this axis so that each bin corresponds to a different possible number of scopes.

1.2.3.2 Test Problems

All problems except the logic problem were defined by a set of test cases in which programs were given specified input data and were scored on how close their output was to the correct output. For MAP-Elites and tournament selection, we calculated fitness as the sum of scores on these test cases.

- **Logic:** Programs receive two integers in binary form and must output the results of doing bitwise logic operations on them. We reward 10 2-input (with the exception of ECHO, which is 1-input) logic operations: ECHO, NOT, NAND, OR-NOT, AND, OR, AND-NOT, NOR, XOR, and EQUALS. To facilitate the evolution of these computations, we added a `Nand` instruction to the instruction set, which converts inputs to integers and then performs a bitwise not-and operation, structured in the same way as the `Add` instruction. Every unique logic operation that a program outputs the solution to over the course of its execution increases that program's score by 1. Once a program has solved all of the logic problems, it gets bonus points for the speed with which it solved them. Specifically, the bonus is equal to the total number of allowed instruction executions minus the number of instructions the program actually executed before performing all 10 logic tasks. For lexicase selection, each logic operation was treated as a different test case.
- **Squares:** Programs receive an integer as input and must output its square. Because this problem is known to be easy, we evaluated programs on just 11 test cases.
- **Sum:** Programs receive a list of five integers as input that they must add together and output their total. Programs were evaluated on a set of 200 test cases.
- **Smallest:** Programs receive a list of four integers as input and must output the smallest one (from [7]). Programs were evaluated on a set of 200 test cases.

1.2.3.3 Experimental Parameters

We ran 30 replicates per condition for 50,000 generations. In conditions where tournament selection, lexicase selection, or random drift is used to determine which programs reproduce, we maintained a population size of 1000 programs. The maximum population size in MAP-Elites, however, depends on the number and resolution of phenotypic trait axes used to define the MAP-Elites grid. Thus, in conditions that use MAP-Elites, the maximum population size is 340. However, in our MAP-Elites conditions, we define a single generation to be equal to 1000 reproduction events, which ensures that all conditions experience the same total number of reproduction events.

We initialized the population by generating random programs of random lengths. Programs could not be less than 1 instruction long or greater than 1024 instructions long. Each program was evaluated by executing its instructions in sequence until an upper limit was hit (128 instruction executions for the squares and logic problems; 512 instruction executions for the sum and smallest problems).

1.2.3.4 Data Analysis

To quantify the different ranges of program architectures explored by our different selection operators we look at the population in the final generations of all of our replicates and filter out all programs that do not fully solve the problem (i.e., those that do not score perfectly on all test cases or, in the context of the logic problem, do not perform all of the logic operations). However, in the random drift condition, we do not filter any programs from the final population, as it is unreasonable to expect that they would have solved the problems. We then look at the distribution of values for each of our phenotypic axes and compare them across selection schemes using a Kolmogorov-Smirnov test to tell us whether MAP-Elites produces a significantly different distribution of program architectures than other selection schemes. To correct for the number of Kolomogorov-Smirnov tests we perform (one for each alternative selection operator that we compare MAP-Elites to, for both scope count and instruction entropy), we use a Bonferonni correction. All data analysis was performed using the R Statistical Computing Platform [19], and all data visualization was done using the ggplot2 package [27]. We used the implementation of the Kolmogorov-Smirnov test in the dgof package, as it is able to properly handle discrete variables, such as scope count [2].

1.2.3.5 Code Availability

All code used to generate and analyze the data presented here is open source and publicly available [12]. This code makes heavy use of the Empirical library, which is also open source and publicly available [16].

1.3 Results and Discussion

The distributions of scope count and instruction entropy values for programs evolved using MAP-Elites were dramatically different from those of programs evolved using other selection schemes (Kolomogorov-Smirnov test, $p < 0.0001$). As is qualitatively evident in Figs. 1.1 and 1.2, the range of each of the metrics in programs evolved with MAP-Elites is much wider than the range for programs evolved under other selection methods (with the exception in some cases of random drift, which was not subject to the requirement that programs actually solve the problem). While this result is not surprising, it provides confirmation that using MAP-Elites as a tool for exploring GP representations provides information we would not otherwise obtain.

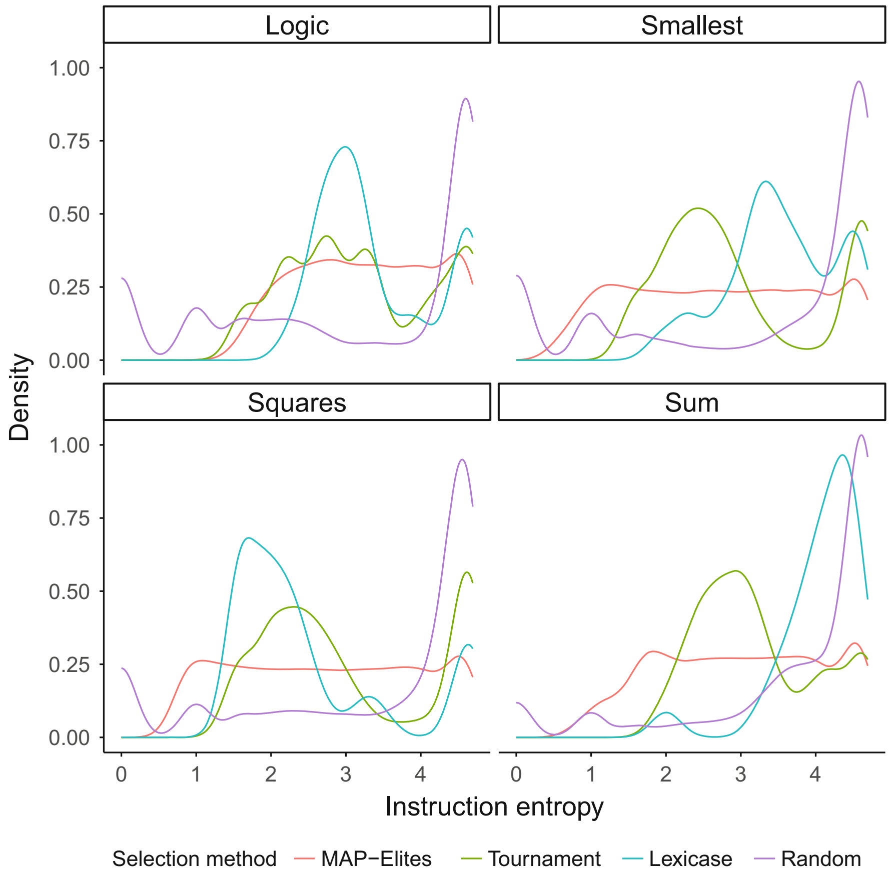


Fig. 1.1 Distribution (as density plot) of instruction entropy metric across all perfect solutions from the final generation of all replicates for each selection scheme

The distribution of metrics evolved under MAP-Elites can suggest the presence of constraints. For example, there generally seems to be some cut-off instruction entropy below which solutions are hard (or impossible) to find. The location of this cut-off varies by problem. This result makes intuitive sense; to achieve the minimum possible instruction entropy, 0, a program would have to consist of a single type of instruction. Any successful solution in this experiment would minimally need to include both the `Input` instruction and the `Output` instructions, as well as some instructions that perform actual calculations. Thus, no successful program could possibly have an instruction entropy of 0. The same logic applies to other low values of instruction entropy. The lack of other empty spots in the distributions of instruction entropy and scope count generated by MAP-Elites indicates that there are not other constraints on these aspects of program architecture; in essence, MAP-Elites has provided an existence proof for programs with these various properties.

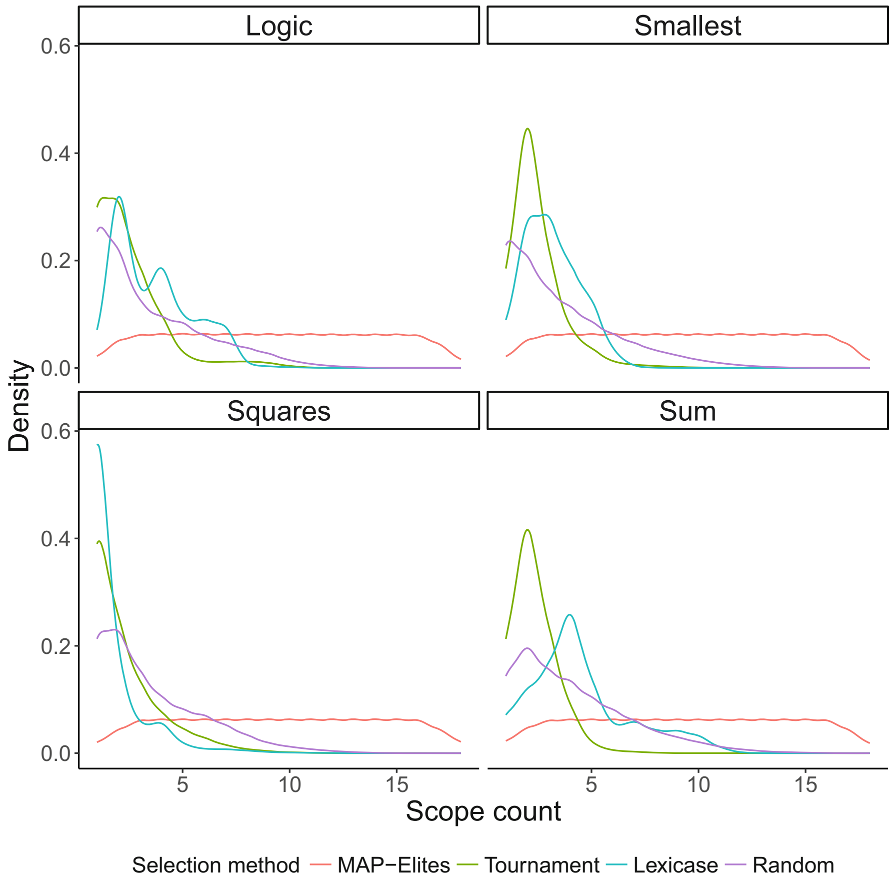


Fig. 1.2 Distribution (as density plot) of scope count metric across all perfect solutions from the final generation of all replicates for each selection scheme

Note that if we had tried to make the same inference based on one of the other selection operators, we may have been mislead. Of course, even MAP-Elites is not guaranteed to find all possible successful program structures. It just comes closer to doing so.

Are there interactions between our two phenotypic traits? We can illuminate possible interactions between phenotypic traits by making heat maps showing the number of solutions found with each distinct combination of traits across all replicates of a condition. To compare the interactions discovered by MAP-Elites to the interactions we might have inferred from looking at the programs generated by other selection schemes, we made a set of heat maps for each selection operator on each problem (see Fig. 1.3). The fact that the heat maps for MAP-Elites are

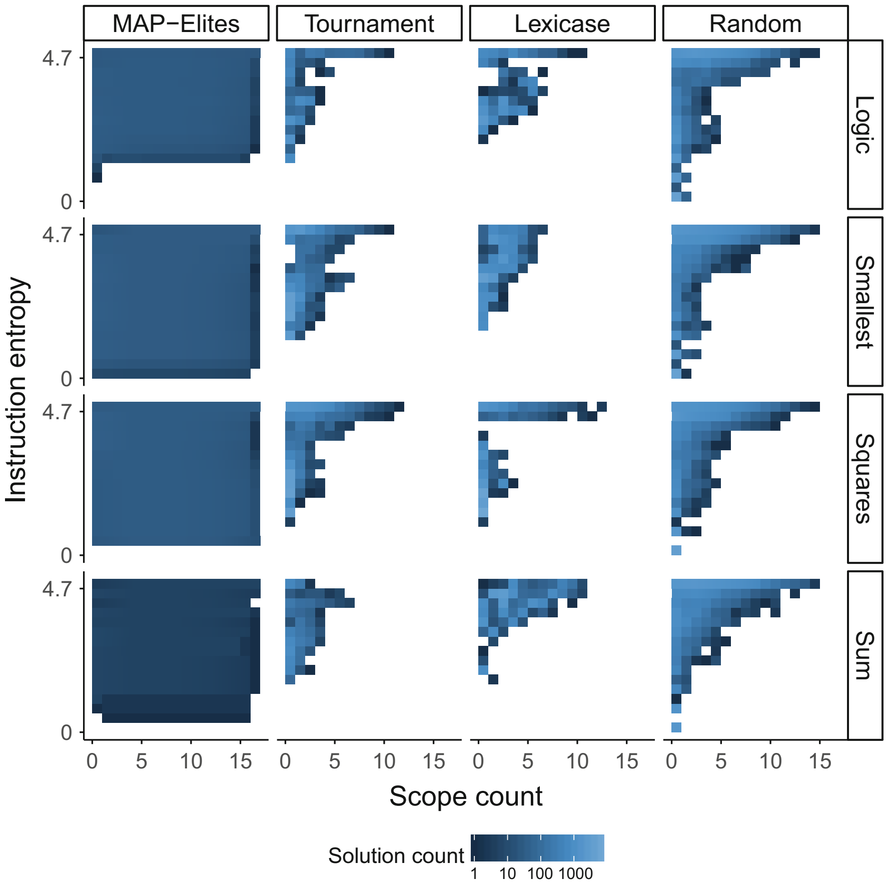


Fig. 1.3 Heat maps showing the number of perfect solutions from the final generation across all replicates for each problem and selection scheme falling into each phenotypic bin

almost completely filled in (with the exception of bins at low instruction entropy) suggests that there are no substantial interactions between instruction entropy and scope count. Note that this is counter to the conclusion we would draw by looking at the results of any of the other selection operators, all of which would seem to suggest that programs with high scope count and low instruction entropy are hard to find. Since random drift displays the same pattern, this is likely some sort of statistical artifact of the types of programs that are most likely to be assembled by chance. Using MAP-Elites to explore the space of program representations allows us to distinguish this artifact from an actual constraint. For an example of using this technique to uncover actual constraints in program architectures, see [13].

1.4 Conclusion

We have demonstrated the use of MAP-Elites as a tool for exploring simple linear GP representations. By selecting phenotypic axes for MAP-Elites that correspond to aspects of program architecture, we can build an intuition for how relevant different features of a GP representation are to the evolutionary process across a variety of problems. These types of analyses are important as the GP community continues to develop and characterize increasingly expressive representations.

In this work, we limited our selection of MAP-Elites phenotypic axes to instruction entropy and module use; however, there are many possible informative axes. Further, MAP-Elites is not limited to just two axes. We could select any number of traits with which to define axes, allowing us to explore how many different aspects of a GP representation interact in the context of a given problem. There are a wide range of possible metrics that we could use in this context.

In genetic programming, it makes sense to evaluate the composition of instructions (or operations in the context of tree or graph-based GP) in the genome. We can either evaluate the composition of all instructions in the genome, or only those instructions that are actually executed. Such analyses can be performed using the following metrics:

- Total number of instructions in the program (length)
- Total number of unique instruction types in the program
- Entropy of instructions (as used in this paper)
- The number of times a given instruction type was used
- The entropy of numbers of times that instruction types were used.
- The average effective dependence distance of instructions [3]

We might also care about more abstract attributes of program architecture. For example, there are many quantities related to modularity that it may be informative to measure, particularly in light of the fact that modularity is thought to promote evolvability. The simplest of these is to measure the modularity of the program using metrics such as the physical and functional modularity metrics described in [14]. In cases where modules are easy to identify, we can probe further with the following metrics:

- Total number of modules in the program
- Total number of times any module is used
- Total number of unique modules that are used (equivalent to scope count in this paper)
- Entropy of time spent in each module

For representations with a concept of memory positions, it may be useful to measure the way the program makes use of them:

- Number of effective memory locations [3]
- Entropy of memory locations used

- The number of times a given individual memory location was referenced/accessed
- Modularity of memory used.

There are also a wide range of potentially useful metrics that will depend on the specifics of the genetic programming representation being used. For example, trees and graph-based programs (e.g., Cartesian genetic programming) can be assessed with metrics that describe their topology. Representations that have linear genomes, on the other hand, can likely borrow a variety of useful metrics from biology.

We have demonstrated that using MAP-Elites with phenotypic axes based on program architecture can illuminate constraints on program architecture that we would have been unaware of simply from examining the programs generated by traditional selection operators. Understanding these constraints can help us understand why certain genetic programming representations are more or less successful under certain circumstances, an important goal for the long-term advancement of the field [17]. Thus, we expect that the approach presented here will be a useful addition to the toolkit we use to study genetic programming.

Acknowledgements We thank members of the MSU Digital Evolution Lab for helpful comments and suggestions on this manuscript. This research was supported by the National Science Foundation (NSF) through the BEACON Center (Cooperative Agreement DBI-0939454), Graduate Research Fellowships to ED and AL (Grant No. DGE-1424871), and NSF Grant No. DEB-1655715 to CO. Michigan State University provided computational resources through the Institute for Cyber-Enabled Research and the Digital Scholarship Lab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or MSU.

References

1. Angeline, P.J., Pollack, J.B.: The evolutionary induction of subroutines. Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society pp. 236–241 (1992)
2. Arnold, T.A., Emerson, J.W.: Nonparametric Goodness-of-Fit Tests for Discrete Null Distributions. The R Journal **3**, 34–39 (2011)
3. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer US, Boston, MA (2007)
4. Chikumbo, O., Goodman, E., Deb, K.: Approximating a multi-dimensional Pareto front for a land use management problem: A modified MOEA with an epigenetic silencing metaphor. In: 2012 IEEE Congress on Evolutionary Computation, pp. 1–9 (2012)
5. Cully, A., Clune, J., Tarapore, D., Mouret, J.B.: Robots that can adapt like animals. Nature **521**, 503 (2015)
6. Cully, A., Demiris, Y.: Quality and Diversity Optimization: A Unifying Modular Framework. IEEE Transactions on Evolutionary Computation **22**, 245–259 (2018)
7. Helmuth, T., Spector, L.: General Program Synthesis Benchmark Suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, pp. 1039–1046. ACM, New York, NY, USA (2015)
8. Keijzer, M., Ryan, C., Murphy, G., Cattolico, M.: Genetic Programming, *Lecture Notes in Computer Science*, vol. 3447. Springer, Berlin, Heidelberg (2005)
9. Kirschner, M., Gerhart, J.: Evolvability. Proceedings of the National Academy of Sciences **95**, 8420–8427 (1998)

10. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
11. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, MA, USA (1994)
12. Lalejini, A., Dolson, E.: amlalejini/GPTP-2018-Exploring-Genetic- Programming-Systems-with-MAP-Elites: Initial Release (2018). URL https://doi.org/10.5281/zenodo.1345799
13. Lalejini, A., Ofria, C.: What else is in an evolved name? Exploring evolvable specificity with SignalGP. PeerJ Preprints 6:e27122v1 pp. 1–21 (2018)
14. Misevic, D., Ofria, C., Lenski, R.E.: Sexual reproduction reshapes the genetic architecture of digital organisms. Proceedings of the Royal Society B: Biological Sciences **273**, 457–464 (2006)
15. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs, q-bio] (2015). ArXiv: 1504.04909
16. Ofria, C., Dolson, E., Lalejini, A., Fenton, J., Jorgensen, S., Miller, R., Moreno, M., Stredwick, J., Zaman, L., Schossau, J., leg2015, cgnitash, V, A.: amlalejini/Empirical: GPTP 2018 - Exploring Genetic Programming Systems with MAP-Elites (2018). URL https://doi.org/10.5281/zenodo.1346397.
17. O'Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. Genetic Programming and Evolvable Machines **11**, 339–363 (2010)
18. Pugh, J.K., Soros, L.B., Szerlip, P.A., Stanley, K.O.: Confronting the Challenge of Quality Diversity. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, pp. 967–974. ACM, New York, NY, USA (2015)
19. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2017). URL https://www.R-project.org/
20. Roberts, S.C., Howard, D., Koza, J.R.: Evolving modules in Genetic Programming by subtree encapsulation. Genetic Programming, Proceedings of EuroGP'2001 **LNCS 2038**, 160–175 (2001)
21. Spector, L.: Simultaneous Evolution of Programs and their Control Structures. Advances in Genetic Programming 2 pp. 137–154 (1996)
22. Spector, L.: Autoconstructive Evolution: Push, PushGP, and Pushpop. GECCO-2001, pp. 137–146 (2001)
23. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, pp. 401–408. ACM (2012)
24. Spector, L., Martin, B., Harrington, K., Helmuth, T.: Tag-based modules in genetic programming. GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation pp. 1419–1426 (2011)
25. Spector, L., McPhee, N.F.: Expressive genetic programming: concepts and applications. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '18, pp. 977–997. ACM Press, Kyoto, Japan (2018)
26. Walker, J.A., Miller, J.F.: The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. IEEE Transactions on Evolutionary Computation **12**, 397–417 (2008).
27. Wickham, H.: ggplot2: elegant graphics for data analysis. Springer, New York (2009)

Chapter 2
The Evolutionary Buffet Method

Arend Hintze, Jory Schossau, and Clifford Bohm

2.1 Introduction

Within the field of Genetic Algorithms (GA) and Artificial Intelligence (AI) a variety computational substrates with the power to find solutions to a large variety of problems have been described. Research has specialized on different computational substrates that each excel in different problem domains. For example, Artificial Neural Networks (ANN) [28] have proven effective at classification, Genetic Programs (by which we mean mathematical tree-based genetic programming and will abbreviate with GP) [18] are often used to find complex equations to fit data, Neuro Evolution of Augmenting Topologies (NEAT) [35] is good at robotics control problems [7], and Markov Brains (MB) [8, 12, 21] are used to test hypotheses about evolutionary behavior [25] (among many other examples). Given the wide range of problems and vast number of computational substrates practitioners of GA and AI face the difficulty that every new problem requires an assessment to find an appropriate computational substrates and specific parameter tuning to achieve optimal results.

Methods have been proposed that combine different computational substrates. “AutoML” [27, 36] is a method designed to select a computational substrate most appropriate for a given problem, and then generate a solution in that substrate.

A. Hintze (✉)
Department of Integrative Biology and Department of Computer Science and Engineering and BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA
e-mail: hintze@msu.edu

J. Schossau · C. Bohm
Department of Integrative Biology and BEACON Center for the Study of Evolution in Action, Michigan State University, East Lansing, MI, USA

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_2

Another compound method is the "mixture of experts" concept, where an artificial neural network is allowed to be constructed from a heterogeneous set of sub-networks, originally pioneered by Jacobs et al. [14] and similar to more recent work [31]. These methods choose from existing substrates or create a network of existing substrates.

In this manuscript we propose a compound method that borrows elements from various known computational substrates and uses them to create a heterogeneous genetic algorithm that allows for direct low-level integration between components from the different substrates. We call this approach the *Buffet Method*, in deference to the No Free Lunch theorem [29, 39, 40, 42], which loosely "state[s] that any two optimization algorithms are equivalent when their performance is averaged across all possible problems" [41]. In this paper we choose components from four computational substrates, and combined these components with a MB framework to construct one possible implementation of the Buffet Method. Each substrate was selected because it has been observed to be successful in some domain. We will show that a MB that incorporates widely heterogeneous computational elements from other systems (in this case GP, NEAT, ANNs, and MB) can combine the advantages of these different systems, resulting in the ability to automatically discover high quality, often hybrid, solutions while avoiding pitfalls suffered by the individual systems.

2.2 Methods

We will be using terms that require definition, and/or specification. When we talk about "computational substrate" we refer to the definition of a particular method of computation. In this light, ANN, MB, etc., all define computational substrates, but so does a biological brain (though we are not proposing the integration of biological neurons into a buffet method... yet). It is also important here that when we talk about ANN, GP and NEAT, we are talking about particular implementations (usually the initial or canonical implementation). In particular, when we use GP we specifically mean mathematical tree-based Genetic Programming and by no means dismiss the large body of work that exists investigating other forms of Genetic Programming. We also wish to note that we are using Markov Brains in two ways in this paper. Firstly, MB are being used as the underlying method that allows for the interconnection between elements from different computational substrates. Secondly, we will be using two types of MB gates that have been in use since MB were first proposed. When talking about Markov Brains in the first sense, we will use MB, and in the second sense we will use CMB (for canonical Markov Brain).

Every computational substrate specifies unique sub-component behavior, the ways sub-components connect with each other, the actions available to the substrate, how inputs are received, how outputs are delivered, and how the substrate stores internal states to allow for memory and recurrence. For example, GP is constructed from nodes arranged as a tree whereas ANNs have a fixed layered topology. CMB logic gates work on digital inputs and not on the continuous values such as those used by NEAT. It is therefore impossible to create one system that integrates unmodified elements from all systems. Instead, we identified essential qualities of each system and devised a way to incorporate these characteristics in a new system.

2.2.1 Markov Brains: An Introduction

Markov Brains describe a computational substrate comprised of three primary elements: Nodes, Gates and Wires. Nodes are simply values (either input, output, or hidden/recurrent). Gates are logical units which execute computations between nodes. Wires connect input nodes to gates and gates to output nodes. When a MB is executed, all of its gates are processed in parallel.

If more then one gate output is connected to the same output node then the gate output values are summed (other methods such as overwrite and average have been tested, but their discussion is outside the scope of this article). In many cases, MB are used with binary inputs and produce binary outputs, in these cases the output values are discretized as `1 if value > 0, else 0`.

Usually, MB are used to find solutions to problems which require multiple updates (i.e. navigation of a robot). On each update the inputs are set (conversion of sensor state to input nodes), the MB is executed, and then the “world” is updated based on the state of the output nodes (i.e. output nodes are used to control motors). Memory between updates is achieved with hidden nodes. These are extra nodes are added by reserving extra space in the input and output buffers. The values written to the output hidden nodes are copied to the input hidden nodes after each MB execution. In some configurations of MB additional input nodes are reserved so that output node values may be copied in the same manner as hidden nodes, providing direct access to the last outputs.

Since MB architecture is only one layer deep, output values are limited to operations that only use single gate execution or a summation of such executions. This limitation can be overcome by allowing hidden nodes and executing the MB multiple times with a single set of inputs. More elaborate deeper topologies of nodes and gates are possible. For example, if output nodes are available as inputs to gates, then the output of one gate can be accessed by another gate in a single MB execution. Of course, in this serial configuration, the gates cannot be run in parallel and a gate execution order must be established (usually as additional information extracted from the genome when the gates are being constructed).

The number of input and output nodes is determined by the task and the number of hidden nodes is set by the user. The gates and wires are initially randomly generated, but then are subject to selection, reproduction, and mutation (see encoding methods below).

Gates may have any number of inputs and outputs. The basic gate type is a deterministic logic gate using between 1 and 4 inputs that converts the inputs to bits (with the discretization function mentioned above) and then delivers between 1 and 4 outputs derived from a genetically determined look-up table. Neuron gates are a more complex example gate type that take one or more inputs, sum these inputs, then deliver an evolvable output value if the input total exceeds an evolvable threshold and 0 if it does not. The summed value in a neuron gate can persist over multiple updates providing the gate with its own local memory (apart from the hidden nodes). Neuron gates have additional configuration options that, for example, allow them to interactively alter their threshold value. Other gate types that have been explored include counters, timers [13], mathematical operations, and feedback [32]. It is not our intention here to describe the full range of possible (or even existing) gate types, but rather to convey that the range of possible behaviors is not limited and could even include nested MB networks. Adding new gate types only requires the implementation of the gate's internal process (gate update) and construction and/or mutation behavior (depending on the encoding type). For a more detailed description of Markov Brains, see [12].

In actuality, the logic contained within gates can define any computational operation. Because of I/O standardization of MB gates any collection of gate types will be compatible, regardless of internal computational processes. The modular inter-operable structure of MB [12] lays the foundation for creating a heterogeneous computational substrate that adopts elements from multiple sources and this is what allows for the Buffet Method.

There are many processes that can be used to construct a MB. While the construct method will likely effect evolvability, it can be considered separately from the MB substrate. In order to generate more robust results (i.e. to insure that the encoding method is not critical to the results) we replicated all experiments using a Genetic Encoding method and a Direct Encoding method.

2.2.2 Genetic Encoding

Genetic Encoding uses a genome (a string of numbers) and a translation method. The occurrence of predefined start codons (sub-strings of numbers, e.g. '43, 212') identify regions (genes) that define gates. Each gate is associated with a particular start codon. The sequence following the start codon provides information needed to define the function of that gate and how it is wired. Thus, every sequence of '43, 212' in the genome will initiate the subsequent translation of a gate (of the type associated with '43, 212') when encountered during a linear read. Note that this allows for overlapping gate genes. Since each gate type requires different data for its construction, the information that must be extracted from the genome after the start codon will be different and must be defined by the gate type. Gate types can be allowed or disallowed by adding or removing their start codons from the translation method.

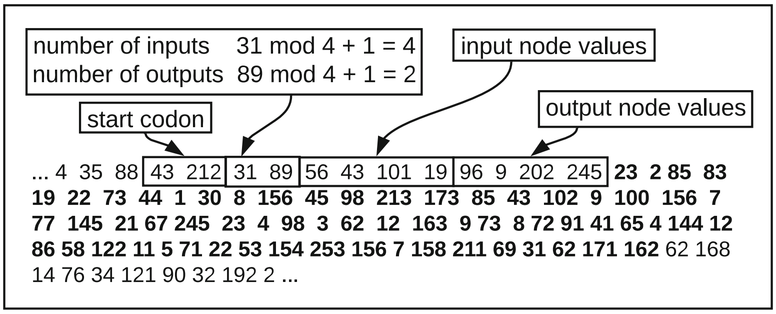


Fig. 2.1 Illustration of Indirect encoding for Markov Brains. A sub-string of values found in a genome encodes each gate of a Markov Brain, each site on this string specifies different aspects of the gate, such as the number of inputs and outputs, or how the inputs and outputs connect to nodes

For example, consider the genome sub-string in Fig. 2.1. If this were part of a genome being translated into a MB the first thing that would happen is that the '43, 212' sub-string would be located. As it happens, '43, 212' is the start codon for a deterministic gate. The next two values 31 and 89 would be used to determine the number of inputs and number of outputs. Since deterministic gates have between 1 and 4 inputs and 1 and 4 outputs each of these values would be processed with ((`value mod` 4) + 1); resulting in 4 inputs and 2 outputs. The next 8 values determine the input and output addresses. This gate will use all 4 input address values but only the first 2 output address values. Since a mutation could alter the number of inputs or outputs, the additional genome locations representing the 3rd and 4th inputs go unread so that such mutations will not result in a frame shift. In order to process the input and output address values the input genome values are modded by the number of input nodes (including hidden) and the output values are modded by the number of output nodes (including hidden). The following 64 values (bold text in figure) are modded by 2 to generate 64 binary values for the look-up table. Since this gate only has 4 inputs and 2 outputs a significant number of these look-up table values will not be used, but they are unread (like the input and output address values) to avoid frame shifts in the case of mutations, which alter the number of inputs or outputs.

Reproduction and mutation are simple when using this form of genetic encoding—the genome is copied, random mutations are applied, and then the resulting genome is translated into a new MB. We allow for point mutations (the random alteration of one random site in the genome), copy mutations (where a section of the genome is selected and copied to another location) and deletion mutations (where a section of the genome is deleted). The mutation rates are established by the user and are defined as a per site percent chance. Sexual reproduction is achieved by crossover between parent genomes (although all experiments in this paper were asexual).

2.2.3 Direct Encoding

The Direct encoding method we used in this paper generates the initial MB populations using randomly generated genomes and translates the genomes using the method described above, ensuring that experiments using Direct or Indirect encoding have the same starting condition. But thereafter, organisms perform reproduction and mutation by copying the MB and applying mutations to each component of the MB directly. This method adds the requirement of specifying mutational operators for each gate type based on the unique structure of that gate type. In addition, the mutation rates of every possible mutation must be determined explicitly. In this paper the direct encoding method allows for mutations that can add a new randomly generated gate, copy an existing randomly selected gate, remove an existing randomly selected gate, alter input or output wires of a gate and alter gate internal parameters.

2.2.4 Multi-Step Functions

MBs execute all gates in parallel within a single update. If a computation requires the participation of multiple gates in sequence, then there must be multiple updates. If a task requires multiple updates and the MB performs multiple updates (i.e. a navigation task where the input/update/output loop is repeated) then multi-step processes can occur over time. On the other hand, some tasks pose a problem and expect an answer or are time-critical. In these cases we allow the MB to update multiple times between setting inputs and reading outputs. This is similar in concept to allowing evolution to use portions of Jordan recurrent architecture, Elman recurrent architecture, and evolvable connections as in NEAT [9, 16, 35].

2.2.5 Gate Types

For this paper we used four architectures: Canonical Markov Brain (CMB), GP, ANN, and NEAT. We chose these because each has been shown to excel in different problem domains. Elements from other systems, including neural Turing machines [22], HyperNEAT [34], and POMDP [17] were considered and could be incorporated later.

Our intent here is not to compare these different computational substrates, which would not be possible or meaningful given that we are re-implementing them in the MB framework, but rather to use these architectures for investigation.

To represent MB we selected two gate types: deterministic and probabilistic, which were the first and most commonly used gates. When we are referring to the use of these gate types we will use the abbreviation CMB (i.e. Canonical Markov

Table 2.1 Computational elements from genetic programming

Command	Description
CONST	Constant value output
ADD	Addition operator
SUB	Subtraction operator
MUL	Multiplication operator
DIV	Division operator
SIN	Sine operator in radians
COS	Cosine operator in radians
LOG	Logarithm
EXP	Exponent function

Brain) rather then MB to avoid confusion. These gates take 1–4 inputs and generate 1–4 outputs. The update function is a look-up table that maps inputs to outputs. In the probabilistic gate, every input pattern can result in any output pattern. For each possible input pattern, every possible output pattern is assigned a probability. Determining which output is generated by a given input requires the generation of a random number during gate update.

From GP, we co-opted the idea of unary and binary math operations. A GP gate takes 1 or 2 inputs and then performs an operation generating a single output (see Table 2.1).

From ANN, we adopted the transfer function (summed weighted inputs) and the hyperbolic tangent as the threshold function. These ANN gates have 1–4 inputs serving as the input layer, and 1–4 outputs serving as the output layer, identical to a small ANN without a hidden layer. The specific function of such an ANN gate is controlled by a weight matrix.

From NEAT we borrowed more complex weighting and integration methods [33]. These gates are a hybrid between ANN and GP in that they take multiple inputs, apply weights, aggregate them (product, sum, max, min, or mean), and then pass them though a mathematical operation. These gates have 1–2 inputs, and a single output (for the specific operators used, see Table 2.2).

2.2.6 Tasks

We chose a range of tasks that includes examples for which each of the gate types described above has demonstrated competence. The purpose here is not to benchmark these gate types or the Buffet Method as a whole, but to demonstrate that the Buffet Method allows evolution to leverage the computational abilities of the different gate types, which on average, yields better results than the use of a single gate type. Moreover, our results show that evolution often finds solutions comprised of combinations of gate types.

Table 2.2 Computational elements from NEAT, the inputs become aggregated into a

Command	Description
CONST	Constant value output
ABS	Absolute operator
CLAMP	Clamps a so that $-1.0 < a < 1.0$
CUBE	a^3
EXP	$\exp a$
GAUSS	$\exp(-5.0a^2)$
HAT	Hat function of a
EQU	a
INV	$\frac{1}{a}$
LOG	$\log a$
RELU	0 for $a < 0.0$ otherwise a
SIG	$\frac{1.0}{1.0+\exp(-a))}$
SQRT	$\sqrt{a}$
TANH	$\tanh a$
SOFTPLUS	$0.2 \log 1.0 + \exp a$

2.2.6.1 Xor

For this task, two binary inputs are given, and fitness is awarded if the logical XOR operation is correctly computed [15]. This is an extremely simple task for a MB, since the initial population is made from randomly generated logic gates that likely implement this function. However, evolving an ANN to solve this task is not trivial, and this task has been used before as a benchmark example for NEAT [15, 37]. The fitness (performance) is evaluated by presenting 100 pairs of binary inputs one pair at a time, and comparing agent output to the expected output. Correct answers are tallied resulting in a fitness between 0 and 100. Each agent is given 10 updates to allow multi-step computation to be performed before evaluating the output.

2.2.6.2 Symbolic Regression

We organized a small set of functions for symbolic regression which all seem to be equally complicated for the GA to find regardless of method (data not shown). Here, we show the result for one function with two inputs x_1 and x_2: $f(x_1, x_2) = (x_1 * x_2) * (x_2 - x_1)$. The fitness of the agent is determined by the difference between this function and the agent's response (sampled 100 times with random input values between -2.0 and 2.0), which is summed and squared. Each agent is given 10 updates to allow multi-step computation.

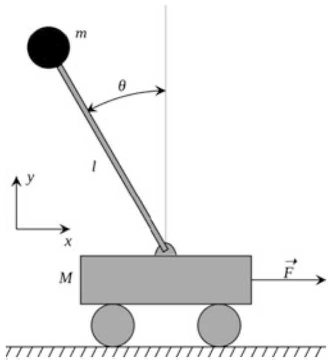


Fig. 2.2 Schematic overview of the inverted pendulum [38]. A weight with the mass $m = 0.1$, kg is mounted on top of a beam (pendulum) of length $l = 0.5$ m. The cart must move forward and backward to balance the pendulum upright. The simulation is stopped if the angle Θ increases above 12°. The cart has a mass M of 1.0 kg, but can be accelerated with a force $\overrightarrow{F}$. We model gravity to be earth-like: 9.8 m/s^2

2.2.6.3 Inverted Pendulum

This task [4] involves balancing a vertical beam on a cart that can move only left or right as if on rails. Each agent was evaluated 10 times. The beam (pendulum) is mounted on top of the cart such that it can freely rotate around its mounting point in 1 perpendicular axis (for an overview see Fig. 2.2. The agent can accelerate the cart left or right, and the time the pendulum is above the cart is recorded during 100 simulation updates. The inputs are the current angle of the pendulum Θ, its first derivative $\dot{\Theta}$, the location of the cart x, and the current acceleration of the cart $\dot{x}$. For each simulation update the agent experiences 8 multi-step computations, and the output is the cart acceleration $\overrightarrow{F}$ (limited between -1.0 and 1.0). Both inputs and outputs in this task are continuous (floating point) values. The code for this was ported from openAI [26].

2.2.6.4 Value Judgment

This task originates from decision-making in psychology. An agent is confronted with two noisy signals and must identify the stronger signal [19]. Imagine two flickering lights of which you have to identify the one which is lit more often. In this task each agent has two binary inputs and two binary outputs. Each evaluation consists of 100 updates. At the beginning of each evaluation, a random value of 0 or 1 is generated that determines whether the first or second input will be more likely. During each update there is a 55% probability that the more likely input will be 1 and the other input will be 0, and a 45% probability of the opposite. For the first 80 updates, the outputs are ignored. the agent then has 20 updates to provide an answer.

Outputs of 0, 0 or 1, 1, are ignored, but 0, 1 and 1, 0 trigger an end of evaluation. If an output of 0, 1 is provided and the second input was more likely or 1, 0 and the first input was more likely, then the agent receives one point. This test is repeated 100 times and the results are summed, resulting in a fitness value between 0 and 100. Note: a) guessing will result in an average score of 50 and b) a perfect score is unlikely because of the small variation in blink probabilities 0.55 vs. 0.45.

2.2.6.5 Block-Catching Task

This task involves an embodied agent that must either catch small blocks or avoid large blocks that are moving obliquely towards the agent [5]. This is a task we thoroughly investigated before [3, 21, 30] and it can be solved by MBs and ANNs (for an illustration of the task see Fig. 2.3). The task is not trivial because information must be integrated over time. Agents in this task are only allowed one brain update per world update so to be successful they should integrate and remember information while new information is presented.

2.2.6.6 Associative Memory

In the associative learning task the agent is rewarded for each unique location visited along a predefined path. The agent is given one input representing whether the agent is on a straight part of the path or not. Two other inputs are also provided, one if the path progresses to the Left and another if the path progresses to the Right. Fitness is increased for each unique location along the path the agent visits, and decreased for every time step that the agent is not on the path. Between each fitness evaluation the set of inputs for "turn left" and "turn right" are randomized. No information is provided indicating the end of the path. At the beginning of a fitness evaluation the agent first should discover the mapping between signs and turn directions, the behavior of which, if performed, looks like exploration. The agent then may utilize that information to follow the path properly, which appears like an exploitation phase [11] (Fig. 2.4). The original work was performed in AVIDA [1] and this particular extension of the task to associative learning was proposed by Anselmo Pontes in yet unpublished dissertation work. From prior experiments (data not shown) we know that MBs as well as ANNs are well-suited to solve this task.

2.2.6.7 Noisy Foraging

This task uses an embodied agent that must search for randomly placed food on a discrete grid; once the food is collected the agent must return to a previously defined home location to increase fitness. This foraging trip can be repeated many times over the lifetime of the agent, on each repetition the randomly placed food is moved farther away from home. The home and food locations are marked by a beacon that can be seen from great distance by the agent. The agent has eight detectors,

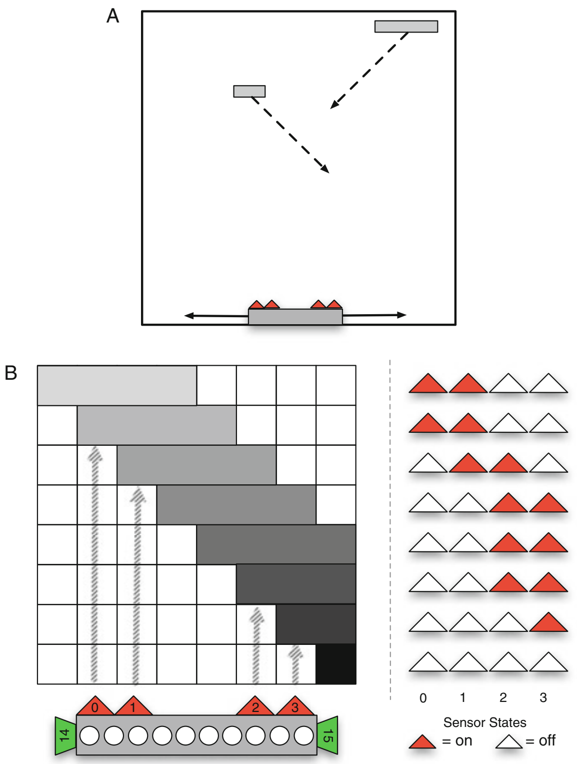


Fig. 2.3 (**a**) In the simulation, large or small blocks fall toward the bottom row of a 20×20 discrete world, only one at a time. As a block falls, it moves diagonally in a fixed direction either to the right or left (e.g. on each time step a given block will move down one row and the left one column). The agent is situated at the bottom row and can move left or right. For the purpose of illustrating the task, a large brick may fall to the left, then the next block might be a small block that will fall to the right. In this example the agent is rewarded for catching small blocks and punished for catching large blocks, but the task may be reversed, or made more complicated with more rules (catch left-falling blocks). (**b**) A depiction of the agent's states (bottom left: triangles depict sensors, circles illustrate brain (internal) states, trapezoids denote actuators) and the sequence of activity patterns on the agent's 4-bit retina (right), as a large brick falls to the right. Reproduced from [21], with permission

each covering a 45° arc that can detect the food and home beacons. Agents can turn 45° left and right or move forward. Additional sensors inform the agent about having reached the home location or a location where food can be found. Each agent has between 9900 and 10,155 time steps to collect food. The variation in lifetime prevents agents from deploying a repetitive search pattern (data not shown). Fitness is defined according to the following function:

$$w = \sum_{j=0}^{j<s_i} 1.05^{\frac{1}{t_{i,j}} d_{i,j}^2} \tag{2.1}$$

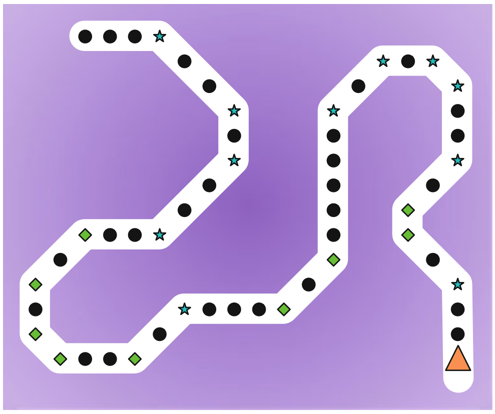


Fig. 2.4 Overview of the associative learning task. The agent (orange triangle) navigates a path (white) on which there are four signals that the agent can only see when standing on top of them. One signal indicates that the path continues in a straight line (black circle). Two other signals (green diamond and blue star) indicate that the path will turn left or right, with the meaning of the signals randomized at the start of every experiment. Thus, the agent must first explore what the signals mean and then exploit that information. The final signal is given when navigating off the path into poison (purple) and has negative consequences, the severity of which is set by the experimenter

where fitness is w, the distance of food to home is d, the time each trip takes is t (only considering the last time an organism leaves either food or home until it reaches the other), and the number of successful trips per trial is s. This equation rewards efficient resource collection by penalizing time-consuming search ($\frac{1}{t}$). Additionally, collecting additional resources is rewarded exponentially (d^2). We then summed over all successful foraging trips completed in one evaluation of the agent.

2.2.6.8 Maze Solving Task

In this navigation task [8], the agent is rewarded for navigating a binary maze consisting of a sequence of long parallel walls, each containing a single door somewhere along the wall (Fig. 2.5). When passing through a door the agent receives an input (1) if the next door is to the right of the current door or an input (0) if the next door is not to the right of the current door. The agent can only perform three actions: step forward, sidestep left, or sidestep right. This task requires the agent to identify sporadic signals and remember them (at least until the next door) in order to navigate efficiently.

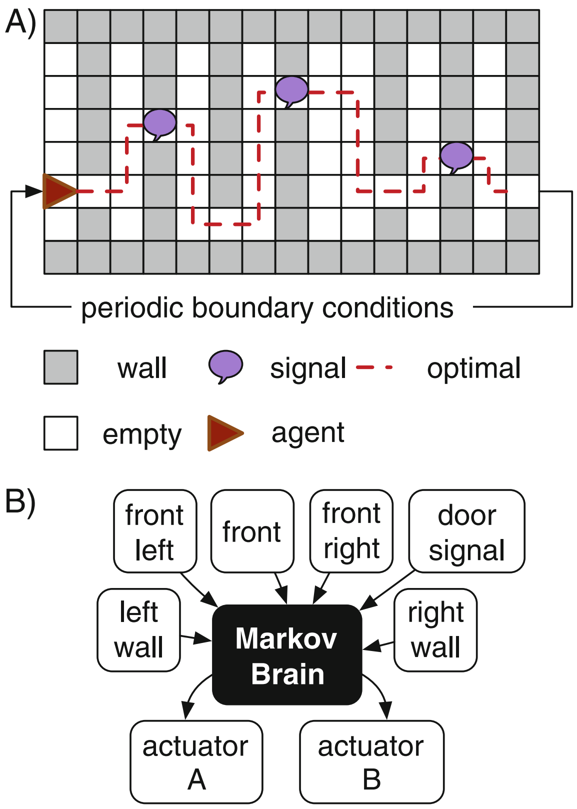


Fig. 2.5 Overview of the navigation task. A typical maze (panel A), with gray boxes indicating walls and white boxes indicating empty space to navigate. The agent (red triangle) finds the shortest path through the maze by attending to the signals received when passing through the doors. A signal (purple speech bubble) indicates that the next door is to the right (from the perspective of the agent), and the absence of a signal indicates that the next door is either straight ahead or to the left. When the agent navigates successfully through the maze, ideally following the optimal path (dashed red line), it is moved back to the start. A schematic view of inputs feeding into the MB and its connection to outputs is shown in panel B. If both actuators receive a positive signal then the agent attempts to step forward. If actuator A receives a positive signal and B does not then the agent attempts to step to the left, and vice versa for a step to the right. An attempt to step sideways into a wall is conveyed by left/right wall inputs. The agent experiences single updates, disallowing multi-step computation

2.2.6.9 Behavioral Optimization in “Berry-World”

This task was designed to test speciation and specialization of behavior [6], however, it also allows us to test how agents integrate past information to optimize future decisions. The environment is a small grid (6 × 6) surrounded by a wall and filled with two types of collectible resources (commonly referred to as red and blue berries, hence ‘berry world’). The agent is evaluated for 200 time steps. On each time step the agent can turn 45° left or right, move forward, or consume the berry at its current location. If a berry is consumed and the agent moves, then the empty spot is replenished with a new random berry (red or blue). The agent can sense resources at its current location, in front of it, to the front left, and to the front right; empty, red, blue, or wall. Consumption of each berry is rewarded with one point, however if the type of berry consumed differs from the previous one, then a task-switching cost (1.4 points) is subtracted. The task-switching penalty discourages random consumption and encourages foraging strategies that minimize switching. On each world update (time step), agents receive five binary inputs. The first two describe the state of the location occupied by the agent (either 1, 0 red food, 0, 1 blue food or, 0, 0 no food here). The other three inputs depend on the state of the location in front of the agent (red food, blue food or wall). the agent provides 2 binary output (0, 0 = no action, 0, 1 = turn right, 1, 0 = turn left, 1, 1 = move forward).

2.2.7 Experimental Parameters

For each possible combination of brains (CMB, GP, NEAT, ANN, and buffet), environments (XOR, symbolic regression, inverted pendulum, value judgment, berry world, spatial temporal integration, maze, associative learning, and noisy foraging) and encoding (direct and indirect) we investigated 100 replicates of each condition. Each condition was seeded with 10,000 randomly generated agents in the first generation (to increase the likelihood to start with an initial viable agent). After that the population size was reduced to 100 agents and evolution progressed for 5000 generations. In each generation roulette wheel selection was used to choose parents and via asexual reproduction, mutated offspring were generated. We expect, our results should also apply to other search methods such as map-elites [24], novelty search [20], and sophisticated hill climbers [10].

All experiments were implemented and performed using the MABE framework [6]. MABE is a general purpose digital evolution framework designed to allow for the arbitrary combination of modules in order to construct agent-based evolution experiments. Here we leveraged MABE's ability to "swap" world modules (fitness functions) between experiments while leaving computational substrates, genomes, mutational operations, and the selection method fixed.

2.3 Results

While most combinations of brains and environments performed well, some combinations failed to find a solution or preformed sub-optimally (see Fig. 2.6). For example, NEAT, GP, and ANNs struggled with the Berry World and Foraging environments, and CMB was unable to solve the pendulum task in the time allotted. The Buffet Method was able to solve all tasks near perfectly.

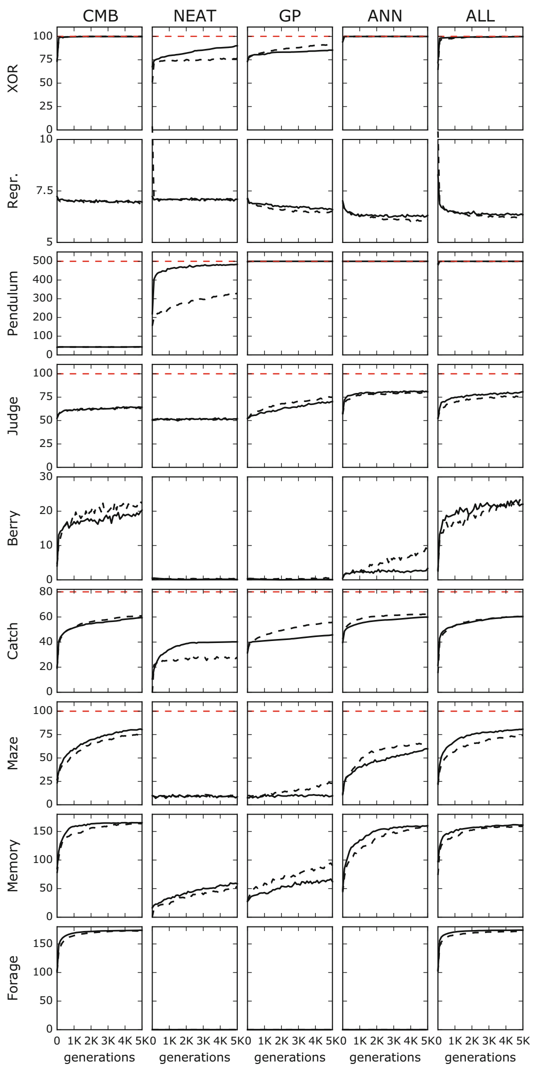


Fig. 2.6 Comparisons of different experimental conditions of the buffet method only run allowing CMB, NEAT, GP, or ANN gates, respectively, in comparison to allowing them ALL types of gates at the same time (columns). Each condition was tested on nine different tasks (XOR, symbolic regression, inverted pendulum, value judgment, berry world, block catching, maze navigation, associative memory, and noisy foraging) represented by the rows. Average performance for the indirect encoding shown as a solid black line, direct encoding represented by a dashed line. The red dashed line indicates the maximum performance on each task where applicable, except for the symbolic regression, which tries to minimize the error

When comparing CMB, GP, ANN, and NEAT to the Buffet Method we find that the Buffet Method generally evolves populations to higher fitness regardless of direct or indirect encoding (see Fig. 2.7), but in some cases is not the best. In particular, ANNs performed better on the Inverted Pendulum task and CMBs perform better on the associative memory task.

To investigate which components evolution selects, we reconstructed gate usage over evolutionary time for CMB, NEAT, GP, and ANN gates. See Fig. 2.8. In some environments (symbolic regression and inverted pendulum) ANN gates are predominantly used, while in the XOR and Berry World CMB gates were dominant. In all other environments we find more than one gate type with slight bias toward one

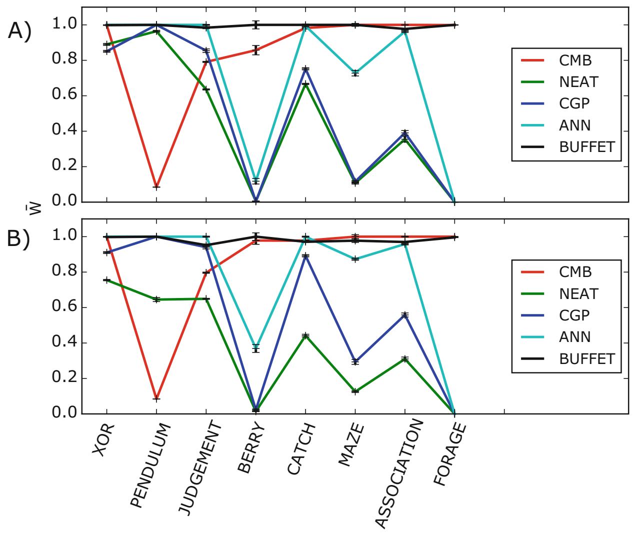


Fig. 2.7 Final normalized scores after 5000 generations for encoding methods. See the legend for the gate conditions. The x-axis is the eight different environments without the symbolic regression task because that task is a score-minimizing task. Y-axis $\bar{W}$ is average fitness. Standard error is shown. (**a**) Results after genetic encoding was used. (**b**) Results after direct encoding was used

type or another (depending on task). Inspecting individual brains (data not shown) confirms that evolved brains are composed of different gate types. This indicates that the Buffet Method is not just allowing evolution to select a single optimal gate type, but to also generate heterogeneous brains composed of different gate types.

2.4 Discussion and Conclusion

We showed that the Buffet Method performs generally well across all tasks, while each of the subsets (CMB, NEAT, GP, ANN) failed at least one task and underperformed on others.

While we cannot say for certain why different computational substrates struggle with some tasks and not others, it is worth noting that the two tasks that were problematic for NEAT, GP, and ANN involve directional navigation. We do know that CMBs' difficulty with the pendulum task is a result of the fact that MB gates are not capable of producing negative outputs (because we chose to represent CMB with deterministic and probabilistic gates which are binary and non-negative), and thus can only accelerate the cart in one direction. We could have resolved the problem of negative numbers for CMBs in the Inverted Pendulum task by changing the meaning of the outputs for that task. We could have discretized the input to a string of bits and instead of a force, we could have used two binary outputs, where both 00 and 11

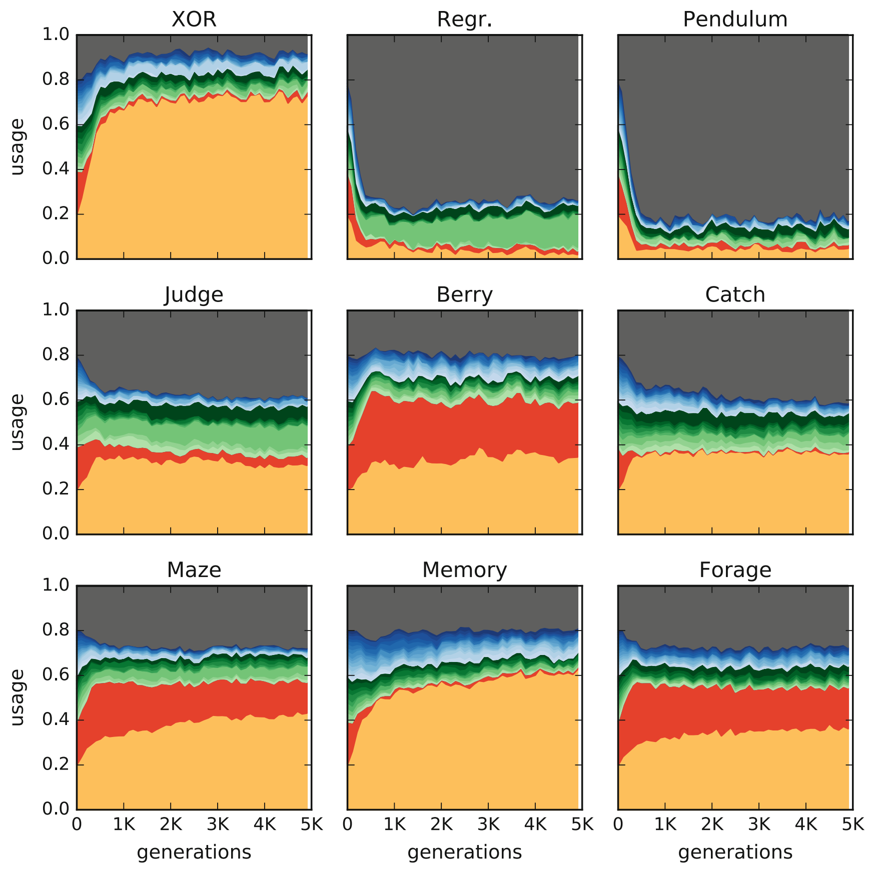


Fig. 2.8 Gate usage for the different experimental conditions using genetic encoding (direct encoding results are similar, data not shown), in orange deterministic logic gates, in red probabilistic logic gates, in shades of green GP gates, in shades of blue NEAT gates, and dark gray ANN gates

mean nothing, and 01 or 10 indicate an acceleration to the left or right. The reason that we did not re-implement the input and output to the Inverted Pendulum task such that non-continuous value substrates could solve it was to highlight that not all computational substrates will always be able to solve all problems. Our approach allowed us to test whether the Buffet Method was capable of discovering solutions using the provided elements given that the problem could be solved by at least some part of the provided elements. One of the advantages of the buffet Method is that it allows for the combination of significantly different computational substrates. In this case, the substrates had different limitations on their input and output specifics. Previous work has shown that the representation of a task can affect evolutionary outcomes [2]. With the Buffet Method we were able to include tasks without needing to consider how, or even if, the task will interface with the substrates.

Considering that the Buffet Method has access to all of the elements of the individual computational substrates it is noteworthy that it could not always find the optimal solution. For example, a MB using only ANN gates is superior to the Buffet Method on the pendulum task. Why did the Buffet Method not simply discard all other gates while retaining the ANN gates? The same logic applies to the association task where using MB gates alone produced better results than the Buffet Method. We expect that this effect may be related to historical contingency. That is, if a strategy that provided some fitness using a sub-optimal gate type was discovered, this gate and strategy may have been "locked in" prohibiting the discovery of a more optimal solution. But this is simply conjecture and requires more investigation.

2.5 Future Work

Here we incorporated ANN, NEAT, and GP gates into a MB substrate, but integration of probabilistic and deterministic MB logic gates and ANN weighted threshold elements into GP or NEAT should be investigated. This idea is not entirely novel. For instance, some implementations of Cartesian GP include binary logic elements [23]. Future exploration into integrated computational substrates, the methods that allow for arbitrary integration, and methods for testing these emerging systems provide ample opportunity for research and development.

The work presented here suggests that the typical approach of finding the correct computational substrate for a given problem is sub-optimal to the Buffet Method in which evolution can be used to discover not only which computational substrate is optimal for a given problem, but can generate new hybrid systems in an automated manner.

Far from insisting that everyone should abandon what they are doing to work on the Buffet Method, we hope that work continues exploring separate domains, so that whenever a new idea is shown to perform well (even if the domain is narrow) this idea can be integrated into the buffet.

One area we did not explore is task classification depending on gate usage. The Buffet Method could be applied to a greater number of tasks and a comparison made between the times to achieve optimal solutions given access to a subset of gates. Or, the distribution of gates used to make up solutions when all gates are provided could be compared. Such profiles could provide an objective task classification method.

Lastly, we found that the Buffet Method creates new heterogeneous solutions made from different components. One cannot help but wonder how components never intended to work together might suddenly form functional computational machines. We will explore these hybrid substrates in the future.

Acknowledgements This work was in part funded by the NSF BEACON Center for the Study of Evolution in Action, DBI-0939454. We thank Ken Stanley, Joel Lehman, and Randal Olson for insightful discussions on HyperNEAT and Markov Brain crossovers.

References

1. Adami, C., Brown, C.T.: Evolutionary learning in the 2d artificial life system avida. In: Artificial Life IV, vol. 1194, pp. 377–381. Cambridge, MA: MIT Press (1994)
2. Adami, C., Schossau, J., Hintze, A.: Evolutionary game theory using agent-based methods. Physics of Life Reviews **19**, 1–26 (2016)
3. Albantakis, L., Hintze, A., Koch, C., Adami, C., Tononi, G.: Evolution of integrated causal structures in animats exposed to environments of increasing complexity. PLoS Computational Biology **10**, e1003,966 (2014)
4. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics **13**, 834–846 (1983)
5. Beer, R.D., et al.: Toward the evolution of dynamical neural networks for minimally cognitive behavior. From Animals to Animats **4**, 421–429 (1996)
6. Bohm, C., CG, N., Hintze, A.: MABE (modular agent based evolver): A framework for digital evolution research. Proceedings of the European Conference of Artificial Life (2017)
7. Cully, A., Clune, J., Tarapore, D., Mouret, J.B.: Robots that can adapt like animals. Nature **521**, 503 (2015)
8. Edlund, J.A., Chaumont, N., Hintze, A., Koch, C., Tononi, G., Adami, C.: Integrated information increases with fitness in the evolution of animats. PLoS Computational Biology **7**, e1002,236 (2011)
9. Elman, J.L.: Finding structure in time. Cognitive Science **14**, 179–211 (1990)
10. Goldman, B.W., Punch, W.F.: Parameter-less population pyramid. In: GECCO '14: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, pp. 785–792. ACM, Vancouver, BC, Canada (2014).
11. Grabowski, L.M., Bryson, D.M., Dyer, F.C., Ofria, C., Pennock, R.T.: Early evolution of memory usage in digital organisms. In: ALIFE, pp. 224–231. Citeseer (2010)
12. Hintze, A., et al.: Markov Brains: A Technical Introduction. arXiv preprint arXiv:1709.05601 (2017)
13. Hintze, A., Miromeni, M.: Evolution of autonomous hierarchy formation and maintenance. Artificial Life **14**, 366–367 (2014)
14. Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. Neural Computation **3**, 79–87 (1991)
15. James, D., Tucker, P.: A comparative analysis of simplification and complexification in the evolution of neural network topologies. In: Proc. of Genetic and Evolutionary Computation Conference (2004)
16. Jordan, M.I.: Serial order: A parallel distributed processing approach. In: Advances in Psychology, vol. 121, pp. 471–495. Elsevier (1997)
17. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101**, 99–134 (1998)
18. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Statistics and Computing **4**, 87–112 (1994)
19. Kvam, P., Cesario, J., Schossau, J., Eisthen, H., Hintze, A.: Computational evolution of decision-making strategies. arXiv preprint arXiv:1509.05646 (2015)
20. Lehman, J., Stanley, K.O.: Exploiting open-endedness to solve problems through the search for novelty. In: ALIFE, pp. 329–336 (2008)
21. Marstaller, L., Hintze, A., Adami, C.: The evolution of representation in simple cognitive networks. Neural Computation **25**, 2079–2107 (2013)
22. Merrild, J., Rasmussen, M.A., Risi, S.: Hyperentm: Evolving scalable neural turing machines through hyperneat. arXiv preprint arXiv:1710.04748 (2017)
23. Miller, J.F.: Cartesian genetic programming. In: Cartesian Genetic Programming, pp. 17–34. Springer (2011)

24. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv preprint arXiv:1504.04909 (2015)
25. Olson, R.S., Hintze, A., Dyer, F.C., Knoester, D.B., Adami, C.: Predator confusion is sufficient to evolve swarming behaviour. Journal of The Royal Society Interface **10**, 20130,305 (2013)
26. openAI.com: OpenAI Gym Toolkit (2018). URL https://gym.openai.com/envs/. [Online; accessed 1-Jan-2018]
27. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q., Kurakin, A.: Large-scale evolution of image classifiers. arXiv preprint arXiv:1703.01041 (2017)
28. Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M., Edwards, D.D.: Artificial Intelligence: A Modern Approach, vol. 2. Prentice Hall Upper Saddle River (2003)
29. Schaffer, C.: A conservation law for generalization performance. In: Proceedings of the 11th International Conference on Machine Learning, pp. 259–265 (1994)
30. Schossau, J., Adami, C., Hintze, A.: Information-theoretic neuro-correlates boost evolution of cognitive systems. Entropy **18**, 6 (2015)
31. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J.: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv preprint arXiv:1701.06538 (2017)
32. Sheneman, L., Hintze, A.: Evolving autonomous learning in cognitive networks. Scientific Reports **7**, 16,712 (2017)
33. Smith, A.W.: Neat-python (2015). URL http://neat-python.readthedocs.io/en/latest/index.html. [Online; accessed 10-31-2017]
34. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artificial Life **15**, 185–212 (2009)
35. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10**, 99–127 (2002)
36. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13, pp. 847–855. ACM, New York, NY, USA (2013).
37. Trujillo, L., Muñoz, L., Naredo, E., Martínez, Y.: Neat, there's no bloat. In: European Conference on Genetic Programming, pp. 174–185. Springer (2014)
38. Wikipedia: Inverted pendulum — Wikipedia, the free encyclopedia (2018). URL https://en.wikipedia.org/wiki/Inverted_pendulum. [Online; accessed 1-Jan-2018]
39. Wolpert, D.H.: The lack of a priori distinctions between learning algorithms. Neural Computation **8**, 1341–1390 (1996)
40. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**, 67–82 (1997)
41. Wolpert, D.H., Macready, W.G.: Coevolutionary free lunches. IEEE Transactions on Evolutionary Computation **9**, 721–735 (2005)
42. Wolpert, D.H., Macready, W.G., et al.: No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute (1995)

Chapter 3
Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial

Stephen Kelly, Robert J. Smith, and Malcolm I. Heywood

3.1 Introduction

Visual reinforcement learning represents the direct application of reinforcement learning algorithms to frame (pixel) data from camera or video sources. The learning agent is therefore able to interact with the environment more directly than previously possible, i.e. there are no a priori decisions made regarding what features are useful/important, potentially reducing sources of bias. To date, such approaches have been dominated by results from deep learning that have successfully minimized the amount of pre-processing necessary to the source images (typically down sampling with sequential image averaging) while demonstrating the ability to better the performance of humans [28].

Some of the rationale for the ability of deep learning to provide state-of-the-art performance in visual reinforcement learning tasks has been attributed to the explicit manner in which the initially high-dimensional sensory input (pixels) is encoded into an efficient low(er) dimensional representation. Having found an appropriate encoding though a highly modular hierarchical (deep learning) architecture, a decision making component is simultaneously trained to provide the agent's policy (typically a multi-layer perceptron). Also of importance was achieving these results with a common topology and hyperparameters regardless of task (e.g. 49 Atari gaming titles). Most research since the initial pioneering report by Mnih et al. [28] has concentrated on either improving on the initial formulation of reinforcement learning employed with deep learning (of which there are many, see for example [24]) or suggesting approaches for evolving different parts of the deep learning

S. Kelly · R. J. Smith · M. I. Heywood (✉)
Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada
e-mail: kelly@cs.dal.ca; smith@cs.dal.ca; mheywood@cs.dal.ca

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_3

architecture. The latter would imply that the complexity of the deep learning architecture would more closely reflect the underlying complexity of the task, e.g. [29].

In this work, we review a recent result in which a completely different approach is taken, one in which program modularity plays a central role; hereafter Tangled Program Graphs (TPG) [12, 13, 15, 31]. Thus, instead of learning to encode a lower dimensional representation, we learn how to connect together multiple teams of programs (modules). Each team individually attempts to make decisions based on a low dimensional policy, thus teams decompose the task based on very small subsets of pixel information. In this way the composition of teams and the connectivity between teams is all emergent. Finally, only a fraction of an agent's complement of teams are actually utilized per decision. Thus, as an agent reacts to different states, different subsets of the teams respond.[1] All of this results in an exceptionally efficient framework for decision making under visual reinforcement learning tasks.

In the following, we review related work (Sect. 3.2) and characterize visual reinforcement learning (Sect. 3.3). A tutorial overview to the TPG framework is presented in Sect. 3.4, before a case study is presented on decision making using TPG in the Atari video game and VizDoom environments (Sects. 3.5 and 3.6 respectively). A concluding discussion is made in Sect. 3.7.

3.2 Related Work

TPG emphasizes the development of interrelationships between programs, ultimately resulting in the emergence of a (tangled) graph. However, this came about as a generalization of earlier research into teaming metaphors in genetic programming. With this in mind, we provide a brief survey of related work from these two perspectives. Naturally, there are many other interesting developments involving graphs and programs, for example Cartesian GP [27], gene regulatory networks [2], and graph programming [1]. However, these developments have tended to focus on using graphs to express the interrelation between individual instructions, whereas the contribution of TPG lies in providing for the emergent organization of the interrelation *between* programs.

3.2.1 Evolving Graphs

Fogal pioneered the evolution of Finite State Machines for sequence learning tasks [8]. However, such models required transition rules for all combinations of

[1]In contrast, even once trained, the convolutional operation central to deep learning results in orders of magnitude higher computational cost.

states and inputs, limiting their scalability. PADO appeared in 1996 and expressed solutions as a graph of programs [35]. The graph had a start node and finish node. Each node also had a program that manipulated its own stack, as well as supporting the indexing of global memory. The path through the graph was determined by a single conditional instruction associated with each node. Execution began with the program at the start node and continued until a time out or the finish node was encountered. Given that the input remains unchanged, it appears that the state of programs local to each node is retained. It is not clear how much of the graph was developed as an emergent property (i.e. if the number of nodes predefined/constant throughout evolution), but it did appear that most if not all the nodes of the graph were visited during each execution.

More recently Genetic Network Programming (GNP) has been proposed in which a fixed number of graph nodes are declared a priori and each node has to be one of two ‘types’: conditional or action [23]. There is a fixed number of nodes (constant across evolution and common to all individuals), and a finite set of (application specific) conditional operators and actions. GNP is initialized at a specific start node, and then allowed to execute up to a minimum of 5 ‘ticks’. Each node type has a specific tick cost (1 for conditionals and 5 for actions), where this limits how much of the graph is visited per state from the environment. In effect, a GA is used to define the connectivity between nodes and node type relative to a predefined library of conditionals or actions.

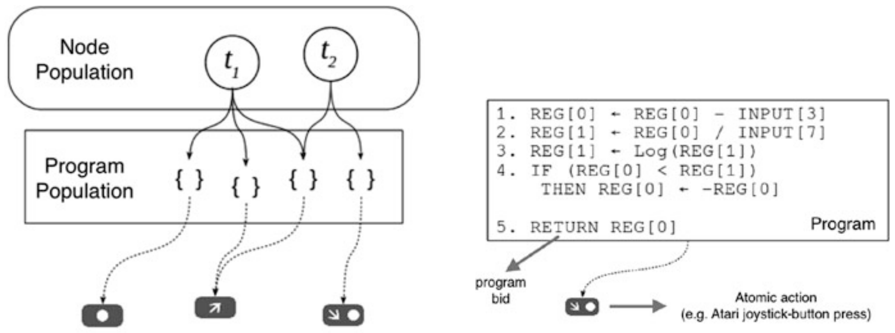
Neural Evolution of Augmented Topologies (NEAT) represents a framework for developing neural networks with arbitrary topology, beginning with a population of perceptrons (each initially fully connected to the inputs). As such, the genotype expresses a graph of connections between different types of node (input, output and neurone). Moreover, NEAT introduced a genotypic marking scheme for connections in order to establish context for crossover [33]. The same marking scheme forms the basis for a genotypic diversity measure that maintains a fixed number of niches in the population during evolution. The framework is capable of describing recurrent connections, as well as defining weight values. The NEAT framework has been widely adopted, and even benchmarked under the Atari visual reinforcement learning domain [9]. Later developments, such as HyperNEAT, concentrated on expressing very large arrays of neurones in a particularly compact genotype, but resulted in all members of the population retaining a fixed number of neurones [9].

Several other frameworks for evolving neural networks (and therefore graphs) have since been proposed, including neuroevolution through Cartesian GP [37] and the use of linear genetic encoding for expressing graphs [25]. To date, such approaches have not been applied to visual reinforcement learning tasks. More recently, the DeepNEAT framework was proposed [26], in which each node of the genome represents a *layer* as opposed to a single neuron (as in NEAT). Specifically, each genome is a table of parameters used to characterize layers of the deep learning architecture and edges of the genome express connectivity between layers. Evaluation was performed with image classification and language modelling tasks.

3.2.2 *Evolution of Multiple Programs Without Graphs*

Teaming metaphors in genetic programming have previously been proposed and define cooperation in terms of an ensemble of programs that all operate simultaneously. Early work assumed a fixed length genome, thus the number of programs was always declared a priori and never varied by the evolutionary process [4]. In other cases a variable length genome was assumed, but fitness had to be specified at the 'level' of programs ***and*** teams, so limiting the application base to classification tasks [36, 40].

In order to avoid these issues, a symbiotic framework was previously assumed in which one population conducts a search for useful team members and a second provides the population of programs [20]. In addition, a trick from learning classifier systems was adopted in which the decision of when to act and what action to take were explicitly separated, or bid-based GP (Fig. 3.1) [19]. Hence, given a team of bid-based GP individuals, the program for each is executed given the current state, $\mathbf{s}(t)$, of the task. Which ever individual from the team has the maximum output is said to have 'won' the right to suggest its action. The action is just a scalar, a, taken from the set of scalar task specific (atomic) actions, $\mathscr{A}$. Thus, for a three class classification problem $\mathscr{A} = \{0, 1, 2\}$. Each bid-based GP individual can only ever have a single action, implying that the program context evolves against a



```
REG[0] ← REG[0] - INPUT[3]
REG[1] ← REG[0] / INPUT[7]
REG[1] ← Log(REG[1])
IF (REG[0] < REG[1])
  THEN REG[0] ← -REG[0]

RETURN REG[0]
```

(a) Symbiotic relation between Nodes (Teams) and Programs

(b) Example (bid-based GP) program

Fig. 3.1 Symbiotic coevolutionary relation between population of nodes and programs (**a**). Each program expresses a bid-value and scalar action (**b**), as per bid-based GP. Programs assume a linear representation (**b**) as this provides support for other algorithmic speedups such as intron instruction skipping [5]. Instruction set is limited to arithmetic operators, 3 non-linear operators (cosine, natural log, exponential), and a conditional (if $R[x] < R[y]$ then $R[x] = -R[x]$)

static action, potentially clarifying credit assignment. Moreover, team complement incrementally evolves in an emergent way until the relevant task decomposition is achieved, i.e. there might be multiple programs with the same action within the same team. Application examples to date include, multi-class classification [20, 21], decomposition of very large attribute spaces into simple classification rules [7, 21], and operation under non-stationary streams [18, 38].

A second development of the above approach introduced the ability to have (bid-based GP) programs learn the context for deploying other (bid-based GP) programs [6, 16, 22]. This was particularly useful in reinforcement learning tasks in which the ultimate policy is constructed hierarchically from earlier policies (as in task transfer) [11, 14, 30]. A limitation of the approach is that all aspects of each task need encountering at the first 'layer' of development (e.g. source tasks), as all later layers (of teams) express their policy 'through' some subset of earlier policies. Naturally, there are many reinforcement learning tasks for which an agent only incrementally uncovers properties of the task as the agent interacts with the environment, e.g. game playing agents or evolutionary robotics. Conversely, the TPG approach explicitly addresses this limitation by providing much more flexibility in how hierarchical relationships between teams of programs develop.

3.3 Visual Reinforcement Learning

In the following we will assume visual reinforcement learning tasks with discrete actions. For example, the Atari (arcade) Learning Environment (ALE) [3] defines state in terms of the frame buffer at time t, with $t = \{0, 1, 2, \dots, n\}$ denoting a sequence of $n + 1$ consecutive frames.[2] The goal of the decision making agent is to suggest an action for each frame such that game score is maximized, i.e. reward is delayed until some end criterion is encountered, such as the maximum number of agent to game interactions ($t = t_{max}$) or an end of game state. Under ALE the set of available (atomic) actions, $\mathscr{A}$, are defined in terms of up to 18 atomic actions[3] corresponding to the enumeration of the 8 discrete directions of the joystick, both with and without a button 'press',[4] plus a button press alone and 'no action' (NA).

The interest therefore lies in finding a single machine learning framework that is capable of playing multiple game titles directly from the visual state information. No attempt is made to a priori identify appropriate input features or decompose the task into a sequence of incrementally more difficult training scenarios. The learning

[2]In practice the sequence of frames as experienced by the agent might represent a stochastic sampling of the actual true frame sequence [24].

[3]Game titles might not use all atomic actions.

[4]The action of a button press is game dependent and might make the avatar 'jump' in some games and 'fire' in others.

agent therefore has to identify an appropriate policy for playing the game directly from the same information that a human player would perceive (albeit without any sound information).

To date, such visual reinforcement learning tasks have been dominated by developments from Deep Learning (see the review in [24]). Evolutionary approaches have been proposed that required prior information to preprocess the original frame information into separate ‘channels’ [9] or limited to optimizing the parameters of a prior Deep Learning architecture [29, 34]. A GP formulation has also been proposed in which GP defines a sequence of image processing operators [10]. The resulting GP processed image then requires interpretation through a set of heuristics in order to determine the action. Such an approach is therefore limited to subsets of games for which appropriate heuristics can be designed. Most recently, Cartesian GP was demonstrated on ALE, albeit while assuming a large instruction set that included operators for explicitly manipulating vector (i.e. image) data [39].

TPG was previously demonstrated to be particularly effective at the 20 ALE game titles that Deep Learning was poor at playing [12] and then benchmarked over all 49 ALE titles [15]. TPG was also demonstrated to be capable of developing single policies that played *multiple* game titles [13, 15]. Most recently TPG was demonstrated under the VizDoom first person shooter environment [31], where this indicates that TPG may also scale to much larger state spaces than under ALE, as well as operating under much higher levels of partial observability than typically present in ALE.

3.4 Tangled Program Graphs

TPG represents a framework for organizing *multiple* programs into structures such that they solve some larger task in a highly modular way. Our starting point is a symbiotic evolutionary framework (Fig. 3.1a) consisting of: (1) a Node population that defines nodes in the TPG graph and (2) a Program population. Nodes define which programs will cooperate in order to make a decision at that particular node. The program population defines each individual in terms of program, p, and atomic action $a \in \mathscr{A}$ (Fig. 3.1b). The two populations coevolve under a symbiotic relationship in which the Node population conducts a search for ‘good’ modules (teams of programs) and the Program population concentrates on sourcing ‘useful’ programs.

In the initial population all nodes consist of between 2 and ω (bid-based) programs, initialized randomly, with action, a, assigned with uniform probability from the set of atomic actions, $\mathscr{A}$, under the constraint that:

- there must be at least two different actions present in the set of programs associated with the same node.
- each node must have a unique complement of programs.

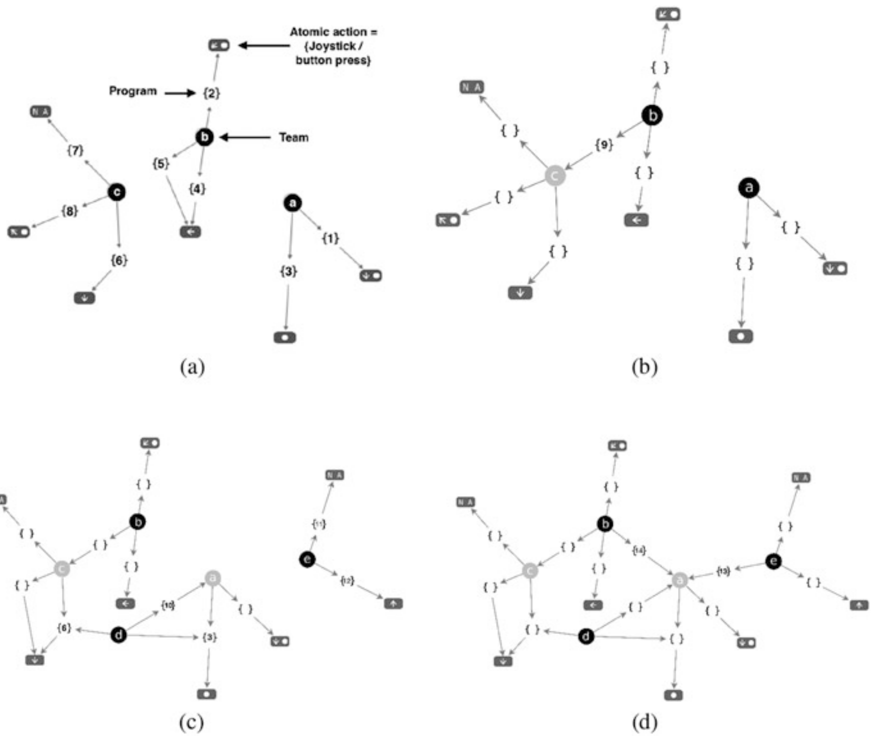


Fig. 3.2 Snapshot of emergent process of TPG graph construction using a hypothetical illustrative example. Black circles represent root nodes, and each root node constitutes the starting node for evaluating an agent. Grey nodes represent members of the Node population that have became subsumed into another agent through the action of the program variation operator pair (p_{mn}, p_{atomic}). (**a**) Initial (single node) agents. (**b**) Transition from a single node to a bi-node agent. (**c**) Two bi-node agents emerge. (**d**) An agent with 4 nodes emerges

Such a starting point implies that: (1) all graphs initially only possess a single node (Fig. 3.2a), and (2) the same program can appear in multiple Nodes (Fig. 3.1a).

After fitness evaluation the agents are ranked (Step 2d, Algorithm 1) and the worst performing Gap deleted from the Node population (Step 2e). Variation operators will sample until R_{size} new agents are introduced into the Node population (Sect. 3.4.2). There is no special significance to the adoption of a breeder model, other than it is elitist.

At this point it is worth noting that:

- Although a two population framework is assumed, fitness is only explicitly defined for nodes in the Node population.[5]

[5]Actually as nodes are subsumed into graphs, it will be come apparent that only a subset of nodes require explicit fitness evaluation (Sect. 3.4.1).

Algorithm 1 TPG algorithm. The term 'agent' denotes the subset of nodes in the Node population that represent root nodes (Sect. 3.4.1). Although an agent typically consists of multiple nodes, only the root node is subject to variation (Sect. 3.4.2)

```
• Initialize $Node(0)$
• Initialize $Prog(0)$
• For ($g = 0$; !EndOfEvolution; $g = g + 1$)          # Generation loop
  1. $AgentList = \emptyset$
  2. For all (node $\in Node(g)$) AND (node = root)  # Identify valid agent
     a. agent = node;
     b. update($AgentList$, node)
     c. For all (evaluations)                       # Evaluation loop (Sect. 3.4.3)
        i. Evaluate(agent)
        ii. update(agent.Fitness)
     d. Rank(agents $\in AgentList$)                 # Select parents
     e. Prune worst ranked $Gap$ agents in $AgentList$ and corresponding nodes in $N(g)$
     f. Prune all $prog \in Prog(g)$ without a node
     g. Select (Parents $\in AgentList$)
     h. Clone ($NewAgents$, Parents)
     i. DO                                          # Create offspring (Sect. 3.4.2)
        i. agent $\in NewAgents$
        ii. newRoot = DeleteProgFromNode(root $\in$ agent, $p_d$)      # Node variation
        iii. newRoot = AddProgToNode(root $\in$ agent, $p_a$)
        iv. IF (ModifyProgram($p_m$) THEN            # Program variation
            A. Select(prog $\in$ newRoot)
            B. Clone (newProg, prog)
            C. ModInstr(newProg, $p_{del}, p_{add}, p_{mut}, p_{swp}$)
            D. ModAction(newProg, $p_{mn}, p_{atomic}$)              # Action variation
        v. IF (!unique(newProg)) THEN repeat 'ModifyProgram'        # Neutrality test
        vi. $Prog(g) = Prog(g) \cup$ newProg          # Update Program population
        vii. $Node(g) = Node(g) \cup$ newRoot         # Update Node population
     j. WHILE count($newRoot$) $< Gap$ AND $|AgentList| < R_{size}$
```

- Members of the program population are tested after the Gap worst performing agents are deleted. If any program is not associated with a surviving individual from the Node population, it is deleted (Step 2f). This implies that the size of the Program population actually fluctuates as a function of the selection and variation operators.
- Nodes in the Node population might be root nodes or nodes that are internal to a graph. As will become apparent, the number of agents is equal to the number of root nodes. It is therefore generally the case that the number of agents is less than the size of the Node population, the precise composition also being an emergent property. With this in mind, a test is introduced to ensure that a minimal number of agents, R_{size}, is maintained at each generation (Step 2j).

Before introducing further details of the TPG algorithm Sect. 3.4.1 provides some intuition as to how the symbiotic representation develops solutions into a tangled graph from teams of programs (each node is a unique team). Section 3.4.2 will then define the variation operators and relate their function to the overall TPG algorithm (Algorithm 1). Finally, Sect. 3.4.3 provides a walk through for how a TPG agent is evaluated, given a frame buffer input.

3.4.1 Developmental Cycle

Figure 3.1a established that at initialization each node (of the Node population) identifies a unique subset of programs from the program population. We can make the connection to graphs more obvious by ignoring the relationship to the two populations (a genotypic property) and instead just concentrate on the representation of agents (initially each node is an agent), a phenotypic property.

Figure 3.2a illustrates the case of an initial population consisting of three agents a, b, c, each of which only consist of a single node. Agents b and c consist of three programs and agent a only 2. Taking agent a as an example, it comprises of two programs, 1 and 3. Arc direction always follows from the node at which programs have team membership and ends at the corresponding action. Given that this is the initial population, actions can only ever be atomic actions, Fig. 3.2a, and all members of the Node population are therefore root nodes. Each root node represents the node at which evaluation/execution begins (Sect. 3.4.3), thus each agent may only have a single root node.

Figure 3.2b illustrates the effect of a modification to the action of program '5'. Action modifications can either result in a different atomic action being selected ($a \in \mathscr{A}$), or the program pointing to another node ($a \in N(g)$). As program '5' was modified, its identifier changes (to '9') to reflect the fact that the original program '5' might still exist elsewhere, i.e. as used by a different node (hence the use of cloning before the application of variation operators, Sect. 3.4.2). Node 'c' now no longer represents a root node, as it is a child of node 'b'. Hence the number of agents has actually decreased.

In Fig. 3.2c node 'a' from Fig. 3.2b was sampled twice as the parent and cloned in both cases resulting in nodes 'd' and 'e'. Node 'e' inherited the programs of node 'a', but had its atomic actions changed. Conversely, Node 'd' retained one program unchanged ('3') but also gained program '6' and a new program '10'. As the action of program '10' is a pointer to node 'a', node 'a' is no longer a root node. At this point we have three agents: $\langle b : c \rangle$, $\langle d : a \rangle$, $\langle e : \emptyset \rangle$ where $\langle x : y, z \rangle$ denotes agent 'x' with root node 'x' and non-root nodes 'y' and 'z'.

Figure 3.2d illustrates the point at which an agent with three nodes emerges. In this case Node 'b' gained a 4th program ('14') with action identifying Node 'a'. In addition, Node 'e' also had a program added ('13') which, in this case, also happened to use Node 'a' as its action. We now have one three node agent, $\langle b : a, c \rangle$, and two agents with 2 nodes, $\langle d : a \rangle$, $\langle e : a \rangle$. Note, that for clarity we have focused the above commentary on the offspring agents introduced as a function of selection and variation. The parents would be retained and compete with the offspring for the right to survive.

3.4.2 Variation

Following the identification of parents (Step 2g, Algorithm 1), each parent is cloned. More specifically, only the *root node* of each parent is subject to cloning and variation (at initialization all nodes are root nodes). This implies that competition between the simpler parent and generally more complex offspring is enforced. Thus, in order to survive the more complex offspring have to perform better than the (simpler) parent.

The first set of variation operators operate on the cloned (root) nodes and take the form:

- delete (Step 2(i)ii) or add (Step 2(i)iii) a reference to a program (p_a, p_d) or,
- modify a program currently in the node (p_m).[6]

The second set of variation operators operate on programs associated with a root node offspring (only called upon when p_m test true, Step 2(i)iv), in which case the effected program is first cloned before applying variation operators to the cloned program:

- delete or add an instruction (p_{del}, p_{add}),
- mutate an instruction (p_{mut}),
- swap two instructions within the same program (p_{swp}), or
- modify the action of a program within the team (p_{mn}).

Should p_{mn} test true, then an additional test is applied, p_{atomic}, which establishes the *type* of action change (Step 2(i)ivD). Thus, for p_{atomic} true, the cloned program's action, a_i is selected from the set of atomic actions $a_i \in \mathscr{A}$, whereas for p_{atomic} false, the new action is a pointer to any node in the Node population of generation g, or $a_i \in N(g)$.

One constraint is enforced during action variation. When the program is inserted in its corresponding node, there must be at least one atomic action present across the subset of programs local to that node. This property will later be employed to guarantee that infinite loops do not result during evaluation, i.e. all states will always result in an atomic action (Sect. 3.4.3).

Finally, given the high cost of fitness evaluation we assess the uniqueness of programs (Step 2(i)v). To do so, a collection of the 50 last state observations as executed under fitness evaluation are retained. Each new program is then tested to

[6]The variation operators are actually applied multiplicatively, possibly resulting in any single operator being applied several times, see [21].

determine whether the bid values for the new program are at least τ different from any of the other programs in $Prog(g)$. If not, the variation operators for program modification are reapplied. In effect this represents a test on the neutrality of the variation operators using a minimum threshold of bidding behaviour.

3.4.3 Agent Evaluation

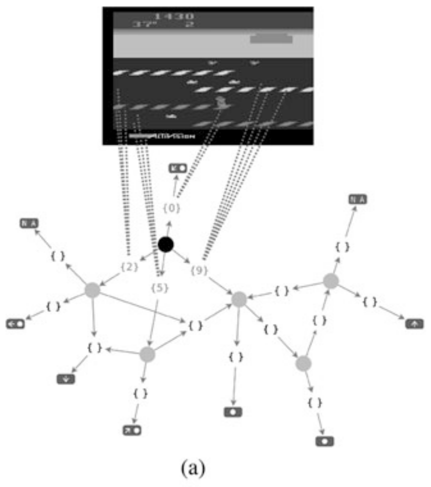
In the following we will assume that state information, $\mathbf{s}(t)$, corresponds to the current content of the frame buffer. Evaluation of the agent always commences from the root node. Thereafter, the path through the agent's graph is dynamic, generally resulting in only a fraction of the agent's program graph being evaluated before an action is identified. This makes TPG exceptionally efficient to evaluate as compared to current Deep Learning or neuro-evolutionary approaches (both of which evaluate the contribution from all of the topology for every decision).

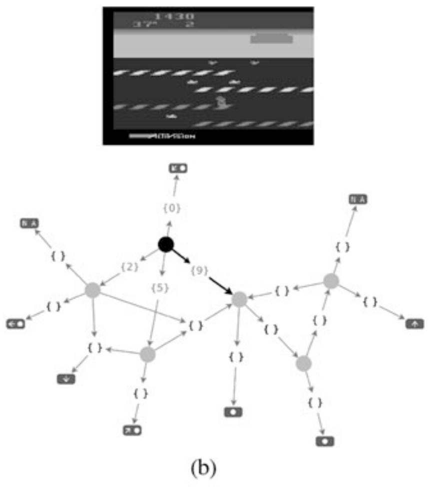
Evaluation of a node is a two step process in which only the subset of programs associated with the node are executed. Out of this subset, only the program with the largest bid 'wins' the right to suggest its corresponding action (providing it is not 'marked', see below). Thus, evaluating a node identifies a specific arc. There are then two scenarios, either the arc references an atomic action or it references another node in the graph. We process these cases as follows:

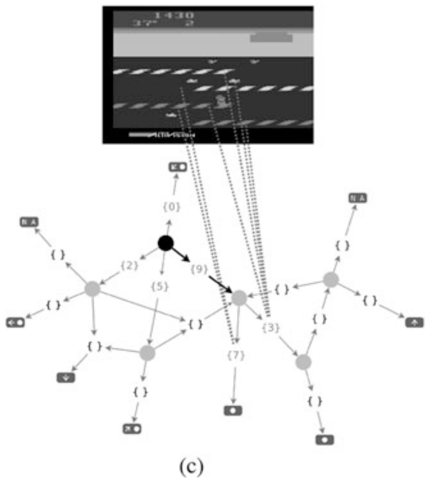
- **Action is atomic:** this represents the decision of this agent. The state of the world would then be updated and the process of evaluation repeated.
- **Action is non-atomic:** the arc is marked and the node pointed to is subject to evaluation, as above. The environmental state, $\mathbf{s}(t)$, is unchanged.

In the special case of a *marked* arc 'winning' a Node evaluation, this implies that the node in question has been previously visited and a loop detected. The loop is broken by removing this arc from the set of candidate programs at this node evaluation and then returning the arc with the winning bid. If this is also marked, the process of removing the marked arc and selecting the next available winning bid at this node repeats. Because every node must have at least one atomic action, there is always a way to break out of a loop, i.e. each time the same node is visited, it has to 'exit' using a different arc.

Figure 3.3 illustrates the process of evaluation for a hypothetical TPG agent and state. Subplot (a) represents the execution of programs associated with the agent's root node (programs 0, 2, 5, 9). Note that each program is free to index any part of the state space (frame), adding to the ability to decompose the task. Program 9 had the 'winning' bid and identifies the next node for evaluation (this arc is also marked), Subplot (b). This node only consists of two (outgoing) arcs, corresponding to programs 3 and 7. Execution of these two programs potentially implies that completely different state information is utilized, Subplot (c). The winning program is identified as program 7, with an atomic action, so evaluation is complete, Subplot (d). This action would result in the game state changing in







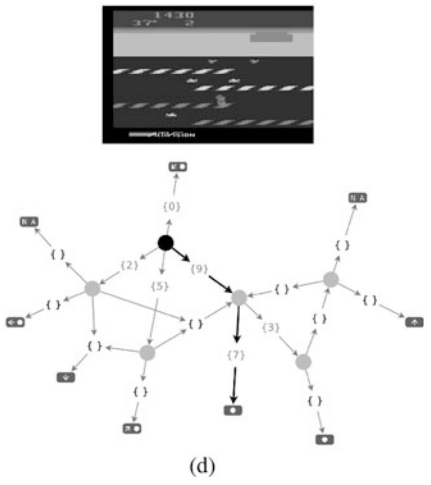


Fig. 3.3 Example of a TPG agent during evaluation. Execution always commences relative to the agent's root node (Subplot (**a**)). Winning program identifies next hop through the TPG individual (Subplot (**b**)). Process of node evaluation repeats (Subplot (**c**)), until an atomic action is identified (Subplot (**d**)). (**a**) Execution begins at the root node. (**b**) Winning program identified from root node. (**c**) Execution passes to node, evaluate programs at node. (**d**) Winning program associated with an atomic action

response to the avatar assuming this action, thus game state is advanced one step, $\mathbf{s}(t) \rightarrow \mathbf{s}(t+1)$. Naturally, if $\mathbf{s}(t+1)$ corresponds to an end-of-game condition this would complete the fitness evaluation for the agent. Otherwise all marked arcs are reset and agent evaluation again commences from the root node.

3.5 Case Study: Arcade Learning Environment

Section 3.3 provided background to the Arcade Learning Environment (ALE) [3, 24], which represents one of the most widely employed benchmarks for visual reinforcement learning. The basic goal is to return an agent capable of playing different gaming titles from the ALE library under a common parameterization. The only information provided to the agent is the visual information from the frame buffer, and the subset of joystick actions specific to the title in question. We also note that the ALE provides various sources of uncertainty including: random sampling of the frames provided to the agent, sticky actions, inter frame variation in sprites described, and depending on the game title, partial observability, i.e. the first person perspective (see [24] for a discussion of these properties).

The approach taken by deep learning assumes: (1) 'frame stacking', a mechanism for encoding the movement of objects, (2) screen down sampling (to 84×84 pixels) and greyscale pixels, (3) and turning 'sticky actions' off [24, 28]. For the basis of this comparison, we will assume 13 titles common to recent evaluations of evolutionary computation approaches to visual reinforcement learning under ALE. Specifically, Salimans et al. describe an approach based on Evolutionary Strategies for identifying weights of an a priori specified deep learning architecture [29]. Such et al., use a genetic algorithm to provide a very compact description of a deep learning architecture consisting of four million weights [34]. In both cases, the authors emphasize that the evolutionary computation approach is faster to train (with extensive GPU support) than the original deep learning architecture, DQN [28], while providing competitive agent policies.

Initial results for TPG in the ALE employed a frame preprocessing procedure in which each pixel was limited to 8 potential colour values (i.e. the SECAM colour encoding, [3]) and each frame was quantized by a factor of 5, resulting in an input space of 1344 decimal state variables in the range 0–255 [12, 13]. In the results reported here, *no attempt* was made to reduce the dimensionality of the initial state information. This implies that the screen as perceived by TPG agents includes all $210 \times 160 = 33{,}600$ pixels with 128 possible colour values for each pixel (i.e. the NTSC colour encoding, [3]). The instruction set is unchanged from the 8 instructions adopted in earlier research when team GP was applied to classification tasks [21], i.e. no application/image specific operators are employed. Figure 3.1b illustrates a typical program employing this instruction set. As per previous TPG results, 'sticky actions' are also present, implying that the ALE environment experienced by TPG agents is stochastic [24].

Table 3.1 summarizes the average game score for each agent on the 13 ALE titles (averaged over 200 evaluations with the no-op game initialization [28]). It is clear that specific algorithms do particularly well on different subsets of games. Also apparent is that DQN has both the most number of games at rank 1 (best) and most number of games at rank 4 (worst). Conversely, TPG appeared to be the most consistent across this set of titles, with only one worst ranked title (Atlantis) and the best average rank across all titles. In short, TPG is competitive in terms of the

Table 3.1 Game scores for 13 Atari titles under no-op game initialization

Game title	TPG	GA [34]	ES [29]	DQN [34]
Amidar	365 (3)	377 (2)	112 (4)	978 (1)
Assault	2027 (2)	814 (4)	1674 (3)	4280 (1)
Asterix	2967 (2)	2255 (3)	1440 (4)	4359 (1)
Asteroids	2575 (2)	2700 (1)	1562 (3)	1365 (4)
Atlantis	110,247 (4)	127,167 (3)	1,267,410 (1)	279,987 (2)
Enduro	108 (2)	80 (4)	95 (3)	727 (1)
Frostbite	6059 (2)	6220 (1)	370 (4)	797 (3)
Gravitar	1068 (1)	764 (3)	805 (2)	473 (4)
Kangaroo	14,026 (1)	11,254 (2)	11,200 (3)	7259 (4)
Sequest	1083 (3)	850 (4)	1390 (2)	5861 (1)
Skiiing	−5734.1 (2)	−5541 (1)	−15,443 (4)	−13,062 (3)
Venture	740 (3)	1422 (1)	760 (2)	163 (4)
Zaxxon	6523 (2)	7864 (1)	6380 (3)	5363 (4)
Avg. rank	2.15	2.54	2.77	2.54

Integer in parenthesis denotes the rank of the algorithm under each game title. Last row details Average rank of each algorithm across the 12 titles

quality of the agent policies identified. Moreover, TPG is providing these results while operating at the original ALE resolution and no frame staking.

Table 3.2 summarizes the complexity of TPG solutions, where for comparison the GA individuals describe a deep learning architecture with four million weights [34]. Moreover, the number of computations actually performed *per decision* is significantly more than this on account of the convolution operation present in deep learning architectures. It is readily apparent that the worst case cost of decision making in TPG ($\approx$1200 instructions) is at least 3 orders of magnitude less than that experienced in deep learning (and can be up to 5 orders of magnitude less). In short, TPG can evolve solutions to visual reinforcement learning tasks *without* specialized hardware support such as GPUs because it explicitly breaks a task down through a coevolutionary process of divide and conquer.

In general, TPG policies developed an exceedingly sparse coverage of the available frame pixels, typically less that 5%, while each decision required less than 2% of the frame pixels, Table 3.2. This is a reflection of the fact that Atari video games (and visual information in general) have a lot of redundant information. In particular, a large portion of the screen content is designed for entertainment value rather that being important for decision-making. Furthermore, while important game entities are most often larger than a single pixel, the agent may only need to perceive a very small number of pixels in order to detect and respond to such entities. The capacity to automatically scale to these properties of the environment contributes to the overall efficiency of the resulting policies.

Table 3.2 Cost of decision making in TPG agent post training

Game title	#Nodes	Av. Nodes	Av. #Instr.	%Pixels	Av.%Pixels
Amidar	65	4	680	6.36	0.9
Assault	53	4	768	4.03	0.88
Asterix	24	2	643	2.73	0.75
Asteroids	51	4	1259	4.86	1.38
Atlantis	53	6	1201	3.84	1.36
Enduro	7	2	170	0.53	0.24
Frostbite	63	4	1063	4.22	1.08
Gravitar	48	4	1150	4.93	1.21
Kangaroo	109	7	1213	9.43	1.38
Seaquest	70	4	980	4.98	1.03
Skiing	173	3	642	8.67	0.76
Venture	20	2	294	2.18	0.38
Zaxxon	35	3	414	2.23	0.48

#Nodes is the total number of nodes in the agent; Av. Nodes is the average number of nodes actually evaluated in order to make a decision; Av. #Instr. is the average number of instructions executed per decision; %Pixels is the percent pixels indexed by the entire TPG graph; Av.%Pixels is the average percent of pixels indexed per decision

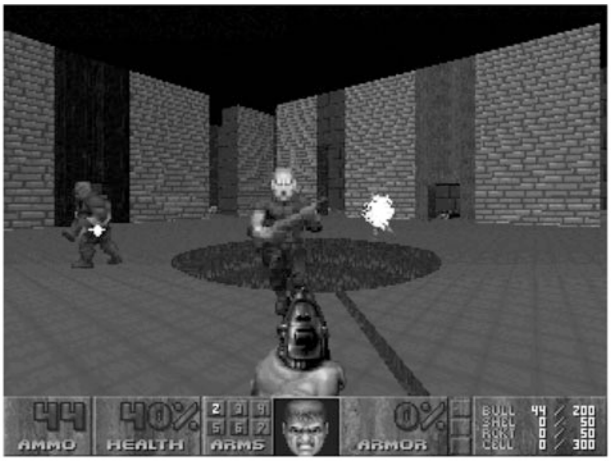
3.6 Case Study: VizDoom

In the following we provide a further illustration of some of the possible outcomes and process of decision making when using TPG, in this case under the first person shooter environment of VizDoom [17]. VizDoom represents an environment with three dimensional state information and (care of the first person perspective) a lot of partial observability. Smith and Heywood demonstrated TPG agents operating under the raw VizDoom frame resolution of $320 \times 240 = 76{,}800$ [31].

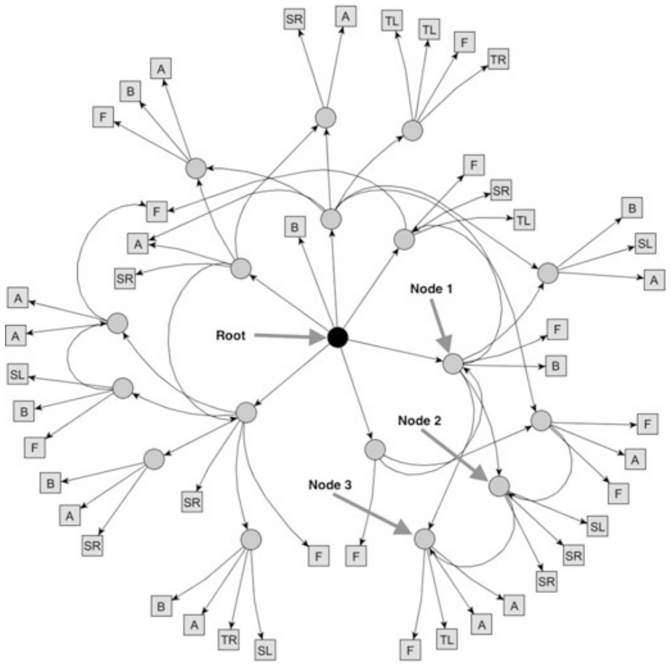
Figure 3.4a illustrates a typical view that a player might encounter under VizDoom when playing in single agent mode. That is to say, the goal is to navigate a cavern like environment, while annihilating a range of predefined opponents, before your own health reaches zero. Statistics characterizing the agent specific information is present at the bottom of the screen (e.g., available ammunition, current health), whereas current game state is captured in the remaining frame real-estate. Information that a decision making agent can employ must come from the frame buffer. There are no special inputs that express/summarize agent health or armour.

Figure 3.4b illustrates a TPG individual post training, and the nodes visited in order to make a decision on the specific game state of Fig. 3.4a.[7] In addition, we can observe what state information is actually indexed in order to make each decision at each TPG node and then express this as a distribution of pixels indexed w.r.t. the x- and y-axis (Fig. 3.5).

[7]This TPG individual actually represents a policy able to operate under ten different VizDoom tasks [31].



(a)



(b)

Fig. 3.4 Example source frame and TPG agent. Sequence of node evaluations for this frame content: Root → Node 1 → Node 2 → Node 3. A total of 7 atomic actions exist: F/B—move forward/backward; TL/TR—turn left/right; SL/SR—Strafe Left/Right, A—Attack (shoot). (**a**) Source frame from defend the circle task. (**b**) TPG agent

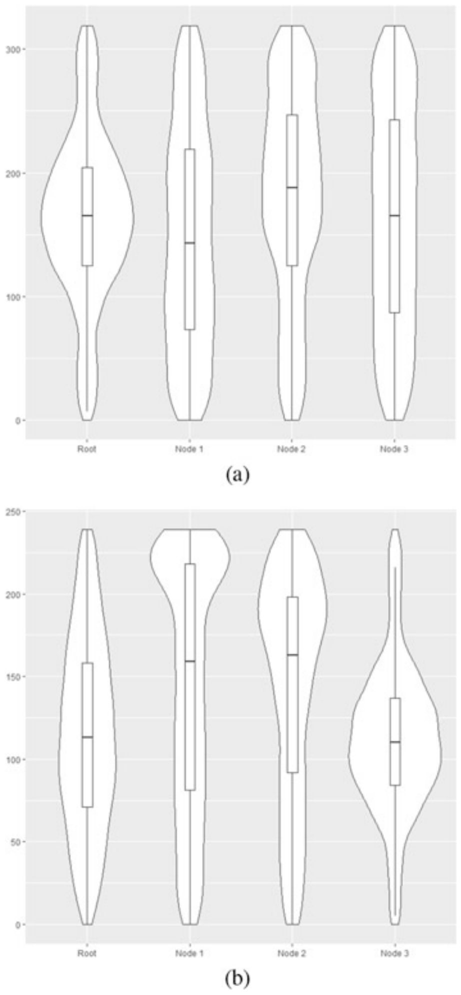


Fig. 3.5 Distribution of pixels indexed by nodes identified in Fig. 3.4b for the frame of Fig. 3.4a, as projected on to the *x*- and *y*-axis. Pixel co-ordinate (0, 0) is at the top LHS of the frame, (320, 0) is the top RHS, (0, 240) is the bottom LHS. Box plot denotes the quartiles. (**a**) *x*-axis distribution (0–320). (**b**) *y*-axis distribution (0–240)

It is now apparent that each node takes a different approach to ‘querying’ the frame. The root node (Fig. 3.4b) indexes pixels using a circular distribution, centred on the middle of the image (‘Root’ violin distributions, Fig. 3.5). Node 1, however, only indexes pixels at the very bottom of the frame, broadly corresponding to the information pertaining to the internal state of the agent (i.e. the ‘AMMO’/‘HEALTH’/‘ARMS’/‘ARMOR’ status bar in Fig. 3.4a). This seems to suggest that Node 1 is using the agent’s internal state information to select the next TPG node. The pixel indexing of Node 2 is again unique, this time selecting pixels predominantly from the RHS of the visual field of the agent. Indeed, when reviewing animations of the agent behaviour, the agent does have a preference for ‘turning’ right, if it is going to turn. Finally, Node 4 indexes pixels corresponding to a horizontal ‘slice’ right through the middle of the *y*-axis. This is interesting because Node 3 represents a node with atomic actions alone; so the agent either shoots, moves forward or turns left, purely on the basis of this final slice of visual information.

In summary, it appears that TPG decision making results in a *hierarchical decomposition* of the input space in which an ordered conditional sequence of decisions (path through the TPG graph) are being deployed relative to very specific regions of the visual space. Moreover, no attempts are made to bias the order of decisions, the regions indexed, or the specific formulation of the decision made at each TPG node.

3.7 Discussion

Few machine learning paradigms are able to auto-construct modular topologies at multiple levels of abstraction. Genetic programming represents a variable length representation, but is rarely deployed to evolve anything other than monolithic solutions, i.e. single programs in which all the instructions are executed.[8] Conversely, teaming metaphors organize programs into a single ‘group’, but might require (task specific) fitness assignment at the level of individual (program) and team or face issues regarding diversity maintenance.

TPG instead pursues an emergent process for spatially decomposing the task. Thus, new topologies are identified by an offspring pointing to something else in the population. A competition between simpler parents and more complex children is the norm. Evaluation is always initiated from the root node and limited to the programs explicitly associated with that node. There can only ever be a single ‘winning’ arc at each node evaluation, and loop detection ensures that a different path must be taken when a previously visited node is detected. All of these properties help ensure that graph structure is emergent, execution efficient, and task decomposition explicit. Indeed, these results were demonstrated under the native resolutions of each platform (33,600 and 76,800 pixels under Atari and VizDoom respectively), implying that TPG continues to identify very efficient solutions, even under high dimensional inputs. Additional results from the Atari platform demonstrate that TPG is still competitive when evaluated against a wider range of game titles and machine learning algorithms [12, 15]. Moreover, it appears that TPG agents are also capable of playing multiple game titles simultaneously [13, 15].

Naturally, this represents a snapshot of very early developments in the TPG framework and consequently a lot of unknowns exist. For example, can refinements also be introduced ‘bottom up’ during development as well as ‘top down’, or how might memory be introduced into geno/phenotypes that emerge dynamically?

[8]Part of this might be due to the types of tasks that researchers choose to deploy GP on. For example, ‘expressive GP’ demonstrates its more interesting properties under tasks such as software synthesis [32].

It is certainly clear that the deep learning methodology (in concentrating on finding appropriate encodings) scales to a wide range of tasks. Conversely, TPG learns to develop very compact decompositions of a task relative to the original high-dimensional state space. It remains to be seen what task preferences and/or strengths/weaknesses might result as a function of this approach to model building.

Acknowledgements Stephen Kelly gratefully acknowledges support from the Nova Scotia Graduate Scholarship program. Malcolm Heywood gratefully acknowledges support from the NSERC Discovery and CRD programs (Canada).

References

1. Atkinson, T., Plump, D., Stepney, S.: Evolving graphs by graph programming. In: European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 10781, pp. 35–51. Springer (2018)
2. Banzhaf, W.: Artificial regulatory networks and genetic programming. In: R. Riolo, B. Worzel (eds.) Genetic Programming Theory and Practice, pp. 43–62. Springer (2003)
3. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research **47**, 253–279 (2013)
4. Brameier, M., Banzhaf, W.: Evolving teams of predictors with linear genetic programming. Genetic Programming and Evolvable Machines **2**, 381–407 (2001)
5. Brameier, M., Banzhaf, W.: Linear Genetic Programming, Springer (2007)
6. Doucette, J.A., Lichodzijewski, P., Heywood, M.I.: Hierarchical task decomposition through symbiosis in reinforcement learning. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2012), pp. 97–104 (2012)
7. Doucette, J.A., McIntyre, A.R., Lichodzijewski, P., Heywood, M.I.: Symbiotic coevolutionary genetic programming: a benchmarking study under large attribute spaces. Genetic Programming and Evolvable Machines **13**, 71–101 (2012)
8. Fogal, L., Owens, A., Walsh, M.: Artificial intelligence through a simulation of evolution. In: Proceedings of the Cybernetic Sciences Symposium, pp. 131–155 (1965)
9. Hausknecht, M., Lehman, J., Miikkulainen, R., Stone, P.: A neuroevolution approach to general Atari game playing. IEEE Transactions on Computational Intelligence and AI in Games **6**, 355–366 (2014)
10. Jia, B., Ebner, M.: Evolving game state features from raw pixels. In: European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 10196, pp. 52–63. Springer (2017)
11. Kelly, S., Heywood, M.I.: On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In: European Conference on Genetic Programming 2014, *Lecture Notes in Computer Science*, vol. 8599, pp. 75–86. Springer (2014)
12. Kelly, S., Heywood, M.I.: Emergent tangled graph representations for Atari game playing agents. In: European Conference on Genetic Programming 2017, *Lecture Notes in Computer Science*, vol. 10196, pp. 64–79. Springer (2017)
13. Kelly, S., Heywood, M.I.: Multi-task learning in Atari video games with emergent tangled program graphs. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2017), pp. 195–202 (2017)
14. Kelly, S., Heywood, M.I.: Discovering agent behaviors through code reuse: Examples from Half-Field Offense and Ms. Pac-Man. IEEE Transactions on Games **10**, 195–208 (2018)

15. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multi-task reinforcement learning. Evolutionary Computation **26**(3) (2018)
16. Kelly, S., Lichodzijewski, P., Heywood, M.I.: On run time libraries and hierarchical symbiosis. In: IEEE Congress on Evolutionary Computation, pp. 1–8 (2012)
17. Kempka, M., Wydmuch, M., Runc, G., Toczek, J., Jaśkowski, W.: ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In: IEEE Conference on Computational Intelligence and Games, pp. 1–8 (2016)
18. Khanchi, S., Vahdat, A., Heywood, M.I., Zincir-Heywood, A.N.: On botnet detection with genetic programming under streaming data label budgets and class imbalance. Swarm and Evolutionary Computation **39**, 123–140 (2018)
19. Lichodzijewski, P., Heywood, M.I.: Coevolutionary bid-based genetic programming for problem decomposition in classification. Genetic Programming and Evolvable Machines **9**, 331–365 (2008)
20. Lichodzijewski, P., Heywood, M.I.: Managing team-based problem solving with symbiotic bid-based genetic programming. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2008), pp. 363–370 (2008)
21. Lichodzijewski, P., Heywood, M.I.: Symbiosis, complexification and simplicity under GP. In: Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO-2010), pp. 853–860 (2010)
22. Lichodzijewski, P., Heywood, M.I.: The Rubik's Cube and GP temporal sequence learning. In: R. Riolo, T. McConaghy, E. Vladislavleva (eds.) Genetic Programming Theory and Practice VIII, pp. 35–54. Springer (2011)
23. Mabu, S., Hirasawa, K., Hu, J.: A graph-based evolutionary algorithm: Genetic network programming and its extension using reinforcement learning. Evolutionary Computation **15**, 369–398 (2007)
24. Machado, M.C., Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M.J., Bowling, M.: Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. Journal of Artificial Intelligence Research **61**, 523–562 (2018)
25. Metzen, J.H., Edgington, M., Kassahun, Y., Kirchner, F.: Analysis of an evolutionary reinforcement learning method in multiagent domain. In: ACM International Conference on Autonomous Agents and Multiagent Systems, pp. 291–298 (2008)
26. Miikkulainen, R., Liang, J.Z., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., Hodjat, B.: Evolving deep neural networks. CoRR **abs/1703.00548** (2017)
27. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: European Conference on Genetic Programming 2000, *Lecture Notes in Computer Science*, vol. 1802, pp. 121–132. Springer (2000)
28. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
29. Salimans, T., Ho, J., Chen, X., Sutskever, I.: Evolution strategies as a scalable alternative to reinforcement learning. CoRR **abs/1703.03864** (2017)
30. Smith, R.J., Heywood, M.I.: Coevolving deep hierarchies of programs to solve complex tasks. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2017), pp. 1009–1016 (2017)
31. Smith, R.J., Heywood, M.I.: Scaling tangled program graphs to visual reinforcement learning in ViZDoom. In: European Conference on Genetic Programming 2018, *Lecture Notes in Computer Science*, vol. 10781, pp. 135–150. Springer (2018)
32. Spector, L., McPhee, N.F.: Expressive genetic programming: concepts and applications. In: ACM Genetic and Evolutionary Computation Conference (Tutorial) (2016)
33. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10** (2002)

34. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J.: Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. CoRR **abs/1712.06567** (2018)
35. Teller, A., Veloso, M.: Pado: A new learning architecture for object recognition. In: Symbolic visual learning. Oxford University Press (1996)
36. Thomason, R., Soule, T.: Novel ways of improving cooperation and performance in ensemble classifiers. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2007), pp. 1708–1715 (2007)
37. Turner, A.J., Miller, J.F.: Neuroevolution: Evolving heterogeneous artificial neural networks. Evolutionary Intelligence **7**, 135–154 (2014)
38. Vahdat, A., Morgan, J., McIntyre, A.R., Heywood, M.I., Zincir-Heywood, A.N.: Evolving GP classifiers for streaming data tasks with concept change and label budgets: A benchmarking study. In: A.H. Gandomi, A.H. Alavi, C. Ryan (eds.) Handbook of Genetic Programming Applications, pp. 451–480. Springer (2015)
39. Wilson, D.G., Cussat-Blanc, S., Luga, H., Miller, J.F.: Evolving simple programs for playing Atari games. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2018), pp. 229–236 (2018)
40. Wu, S.X., Banzhaf, W.: Rethinking multilevel selection in genetic programming. In: ACM Genetic and Evolutionary Computation Conference (GECCO-2011), pp. 1403–1410 (2011)

Chapter 4
Strong Typing, Swarm Enhancement, and Deep Learning Feature Selection in the Pursuit of Symbolic Regression-Classification

Michael F. Korns and Tim May

4.1 Introduction

Symbolic Classification (SC), an offshoot of Genetic Programming (GP), can play an important role in any well-rounded predictive analytics tool kit, especially because of its so called "*WhiteBox*" properties. In these recent papers [4, 9, 13], algorithms were developed to push SC to the level of basic classification accuracy competitive with existing commercially available classification tools, including the introduction of GP assisted Linear Discriminant Analysis (LDA) [10]. In this chapter we add a number of important enhancements to our basic SC system and demonstrate their accuracy improvements on a set of theoretical problems and on a banking industry problem. We enhance GP assisted linear discriminant analysis with a modified version of Platt's Sequential Minimal Optimization algorithm [14] which we call (MSMO), and with swarm optimization techniques [5]. We add a user-defined typing system, and we add deep learning feature selection to our basic SC system. This extended algorithm (LDA^{++}) is highly competitive with the best commercially available M-Class classification techniques on both a set of theoretical benchmarks and on a real world banking industry problem. This new LDA^{++} algorithm moves genetic programming classification solidly into the top rank of commercially available classification tools.

Each of the first four genetic programming SC algorithms is briefly explained below, then a more detailed description of the proposed extended Linear Discriminant Analysis algorithm (LDA^{++}) is presented in this paper. The Platt inspired

M. F. Korns (✉)
Lantern Credit LLC, Henderson, NV, USA
e-mail: mkorns@korns.com

T. May
Insight Sciences Corporation, Henderson, NV, USA

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*, Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_4

MSMO algorithm is described in detail and the manner in which the LDA matrix math and Swarm optimizations are tightly integrated is also explained. The user-defined typing system is described in detail herein, and the deep learning feature selection methodology is discussed.

For theoretical testing, a set of ten artificial classification problems are constructed with no noise such that absolutely accurate classifications are theoretically possible. The discriminant formulas for these ten artificial problems are listed. The problems vary from linear to nonlinear multimodal and from 25 to 1000 features such that each classification algorithm will be stressed on well understood problems from the simple to the very difficult. All theoretical problems have 5000 training points and a separate 5000 testing points. The scores on the out of sample testing data, for each of the ten classification algorithms are reported here.

No assertion is made that these five genetic programming SC algorithms are the best in the literature. In fact, we know of an additional enhanced algorithm, *which we have not had time to implement for this study*, M_3GP [13]. No assertion is made that the five KNIME classification algorithms are the best commercially available, only that KNIME is a trusted component of Lantern Credit predictive analytics. This study is simply meant to provide one reference point for how far genetic programming symbolic classification has improved relative to a set of reasonable commercially available classification algorithms.

This paper includes a comparison study of the five new SC algorithms and five well-known commercially available classification algorithms to determine just where SC now ranks in competitive comparison. The five SC algorithms are: simple genetic programming using argmax referred to herein as AMAXSC; the M_2GP algorithm [4]; the MDC algorithm [9], Linear Discriminant Analysis (LDA) [10], and Linear Discriminant Analysis extended with MSMO and Swarm (LDA^{++}). The five commercially available classification algorithms are available in the KNIME system [1], and are as follows: Multiple Layer Perceptron Learner (MLP); Decision Tree Learner (DTL); Random Forest Learner (RFL); Tree Ensemble Learner (TEL); and Gradient Boosted Trees Learner (GBTL).

For real world testing, we use an actual banking data set for loan scoring as it was received. The training data contains 337 features with 36,223 entries, while the testing data contains the same 337 features with an additional 85,419 entries. The testing and training data are distinct. We include a comparison study of the five new SC algorithms and five well-known commercially available classification algorithms to determine just where SC now ranks in competitive comparison on this real world problem. Also included is the bank's benchmark score, achieved over a multiple month period by the bank's in-house data science team with proprietary tools.

In conclusion we show that, on the theoretical problems, the two best classification algorithms are Gradient Boosted Decision Trees (GBTL) and this paper's extended Linear Discriminant Analysis (LDA^{++}). Furthermore we show, on the real world banking problem, the three best classification algorithms are Gradient Boosted Decision Trees (GBTL), the bank's in-house data science approach, and this paper's extended Linear Discriminant Analysis (LDA^{++}).

Most of the formalism discussed here has been published in a previous volume [11], but to make it easier for readers the formalism is repeated here.

4.2 Comparison Algorithms

4.2.1 AMAXSC in Brief

The simplest naive genetic programming approach to multiclass classification is arguably a standard genetic programming approach, such as a modification of the baseline algorithm [6], using the **argmax** function to classify as follows,

$$y = argmax(gp_1, gp_2, \ldots, gp_C) \tag{4.1}$$

where C is the number of classes

Each gp_k represents a separate discriminant function evolved via standard genetic programming. The argmax() function chooses the class (1 to C) which has the highest value, and is strongly related to the Bayesian probability that the training point belongs to the c-th class. No other enhancements are needed other than the standard argmax() function and a slightly modified genetic programming system—modified to evolve one formula for each class instead of the usual single formula.

4.2.2 MDC in Brief

The Multilayer Discriminant Classification (MDC) algorithm is an evolutionary approach to enhancing the simple AMAXSC algorithm.

$$y = argmax(w_{10} + w_{11} * gp_1, w_{20} + w_{21} * gp_2, \ldots, w_{C0} + w_{C1} * gp_C) \tag{4.2}$$

where C is the number of classes, gp_k are the GP evolved formulas, and w_{ij} are real weight coefficients (there are $2C$ weights).

Each gp_k represents a separate discriminant function evolved via standard genetic programming. The argmax() function chooses the class (1 to C) which has the highest value, and is strongly related to the Bayesian probability that the training point belongs to the c-th class. Given a set of GP evolved discriminant formulas $\{gp_1, gp_2, \ldots, gp_C\}$, the objective of the MDC algorithm is to optimize the choice of coefficient weights $\{w_{10}, w_{11}, w_{20}, w_{21}, \ldots, w_{C0}, w_{C1}\}$ such that Eq. (4.2) is optimized for all X and Y.

The first step in the MDC algorithm is to perform a Partial Bipolar Regression on each discriminant entry i.e. $w_{k0} + (w_{k1} \times gp_k) = Y_k + e$. This produces starting weights for w_{k0} and w_{k1} which are not very good but are much better than random. The second step in the MDC algorithm is to run a Modified Sequential Minimization on selected discriminant entries. This produces much better weight candidates for all discriminant functions, but is still not perfect. Finally, the MDC algorithm employs the Bees Algorithm [5] to fully optimize the coefficient weights. The MDC algorithm is discussed in much greater detail in [9].

4.2.3 M$_2$GP in Brief

The M$_2$GP algorithm is described in detail in [4]. Briefly the M$_2$GP algorithm generates a D-dimensional GP tree instead of a 1-dimensional GP tree. Assuming that there are C classes, the algorithm attempts to minimize the Mahalanobis distance between the n-th training point and the centroid of the k-th class. The basic training algorithm is as follows.

Algorithm A1: M$_2$GP Training

```
1. Input: $X$, $Y$, $D$—where $X$ is an $M \times N$ real matrix, $Y$ is an $N$ vector, $D$ is a scalar
2. For $g$ from 1 to $G$ do
3. Generate: $F = \{f_1, f_2, \ldots, f_D\}$ set of $D$ solutions
4. Evaluate: $Z_s = \text{Eval}(f_s(X))$ for $s$ from 1 to $D$—a $D$-dimensional point
5. Cluster: $Z^k$ in $Z$ for all $k$ from 1 to $C$—group all the $Z$ which belong to each class
6. For $k$ from 1 to $C$ do
7. $C^k = \text{cov}(Z^k)$—a $D$ by $D$ covariance matrix for each class
8. $W^k = \text{centroid}(Z^k)$—a 1 by $D$ centroid vector
9. $D_k(X_n) = \text{sqrt}((Z_n - W^k) \times (C^k)^{-1} \times (Z_n - W^k)^T)$—for $n$ from 1 to $N$ (the number of training points)
10. For $n$ from 1 to $N$ do $EY_n = \text{argmin}(D_1(X_n), D_2(X_n), \ldots, D_C(X_n))$
11. For $n$ from 1 to $N$ do $E_n = 1$ IFF $EY_n \neq Y_n$, 0 otherwise
12. Minimize average($EY$)
13. Return $F$, $C$, $M$
```

The M$_2$GP algorithm is discussed in much greater detail in [4].

4.2.4 LDA Background

Linear Discriminant Analysis (LDA) is a generalization of Fischer's linear discriminant, which is a method to find a linear combination of features which best separates K classes of training points [2, 3, 12]. LDA is used extensively in Statistics, Machine Learning, and Pattern Recognition.

Similar to the arguments leading up to the M$_2$GP algorithm [4], we argue that any symbolic regression system can be converted into a symbolic classification system.

In this paper we start with the baseline algorithm published in [6]. Our baseline SR system inputs an N by M matrix of independent training points, X, and an N vector of dependent values, Y. The SR system outputs a predictor function, $F(X) \sim Y$ where F is the best least squares estimator for Y which the SR system could find in the allotted training time. The format of F is important, and consists of one or more basis functions Bf_b with regression constants c_b. There are always B basis functions and $B + 1$ coefficients. The following is the format of F.

$$y = c_0 + c_1 * Bf_1 + c_2 * Bf_2 + \cdots + c_B * Bf_B \tag{4.3}$$

There are from 1 to B basis functions with 2 to $B+1$ real number coefficients. Each basis function is an algebraic combination of operators on the M features of X, such that $Bf_b(X)$ is a real number. The following is a typical example of an SR produced predictor, $F(X)$.

$$y = 2.3 + 0.9 * cos(x_3) + 7.1 * x_6 + 5.34 * (x_4/tan(x_8)) \tag{4.4}$$

The coefficients c_0 to c_B play an important role in minimizing the least squares error fit of F with Y. The coefficients can be evolved incrementally, but most industrial strength SR systems identify the optimal coefficients via an assisted fitness training technique. In the baseline SR algorithm this assisted fitness training technique is simple linear regression ($B = 1$) or multiple linear regression ($B > 1$).

In symbolic classification problems the N by M matrix of independent training points, X, is unchanged. However, the N vector of dependent values contains only categorical unordered values between 1 and K. Furthermore the least squares error fitness measure (LSE) is replaced with classification error percent (CEP) fitness. Therefore we cannot use regression for assisted fitness training in our new SC system. Instead, we can use LDA as an assisted fitness training technique in our new SC system.

Our new SC system now outputs not one predictor function, but instead outputs K predictor functions (*one for each class*). These functions are called discriminants, $D_k(X) \sim Y_k$, and there is one discriminant function for each class. The format of the SC's discriminant function output is always as follows.

$$y = argmax(D_1, D_2, \ldots, D_K) \tag{4.5}$$

The argmax function returns the class index for the largest valued discriminant function. For instance if $D_i = \max(D_1, D_2, \ldots, D_K)$, then $i = \text{argmax}(D_1, D_2, \ldots, D_K)$.

A central aspect of LDA is that each discriminant function is a linear variation of every other discriminant function and reminiscent of the multiple basis function estimators output by the SR system. For instance if the GP symbolic classification system produces a candidate with B basis functions, then each discriminant function has the following format.

$$
\begin{aligned}
D_0 &= c_{00} + c_{01} \times Bf_1 + c_{02} \times Bf_2 + \cdots + c_{0B} \times Bf_B \\
D_1 &= c_{10} + c_{11} \times Bf_1 + c_{12} \times Bf_2 + \cdots + c_{1B} \times Bf_B \\
D_k &= c_{k0} + c_{k1} \times Bf_1 + c_{k2} \times Bf_2 + \cdots + c_{kB} \times Bf_B
\end{aligned}
\tag{4.6}
$$

The $K \times (B + 1)$ coefficients are selected so that the i-th discriminant function has the highest value when the $y = i$ (*i.e. the class is i*). The technique for selecting these optimized coefficients c_{00} to c_{KB} is called linear discriminant analysis and in the following section we will present the Bayesian formulas for these discriminant functions.

4.2.5 LDA Matrix Formalism

We use Bayes rule to minimize the *classification error percent* (CEP) by assigning a training point $X_{[n]}$ to the class k if the probability of $X_{[n]}$ belonging to class k, $P(k|X_{[n]})$, is higher than the probability for all other classes as follows:

$$EY_{[n]} = k, \text{ iff } P(k|X_{[n]}) \geq P(j|X_{[n]}) \text{ for all } 1 \leq j \leq K \tag{4.7}$$

The CEP is computed as follows:

$$CEP = \sum (EY_{[n]} \neq Y_{[n]}| \text{ for all } n)/N \tag{4.8}$$

Therefore, each discriminant function D_k acts a Bayesian estimated percent probability of class membership in the formula.

$$y = argmax(D_1, D_2, \ldots, D_K) \tag{4.9}$$

The technique of LDA makes three assumptions, (a) that each class has a multivariate Normal distribution, (b) that all class covariances are equal, and (c) that the class covariance matrix is nonsingular. Once these assumptions are made, the mathematical formula for the optimal Bayesian discriminant function is as follows:

$$D_k(X_n) = \mu_k(C_k)^{-1}(X_n)^T - 0.5\mu_k(C_k)^{-1}(\mu_k)^T + \ln P_k \tag{4.10}$$

where X_n is the n-th training point, μ_k is the mean vector for the k-th class, $(C_k)^{-1}$ is inverse of the covariance matrix for the k-th class, $(X_n)^T$ is the transpose of the n-th training point, $(\mu_k)^T$ is the transpose of the mean vector for k-th class, and $\ln P_k$ is the natural logarithm of the naive probability that any training point will belong to class k.

In the following section we will present step by step implementation guidelines for LDA assisted fitness training in our new extended baseline SC system, as indicated by the above Bayesian formula for $D_k(X_n)$.

4.2.6 LDA Assisted Fitness Implementation

The baseline SR system [6] attempts to score thousands to millions of regression candidates in a run. These are presented for scoring via the fitness function which returns the least squares error (LSE) fitness measure.

$$LSE = fitness(X, Y, Bf_1, \dots, Bf_B, c_0, \dots, c_B) \tag{4.11}$$

The coefficients $c_0, \dots, c_B$ can be taken as is, and the simple LSE returned. However, most industrial strength SR systems use regression as an assisted fitness technique to supply optimal values for the coefficients before returning the LSE fitness measure. This greatly speeds up accuracy and allows the SR to concentrate all of its algorithmic resources on the evolution of an optimal set of basis functions $Bf_1, \dots, Bf_B$.

Converting to a baseline symbolic classification system will require returning the classification error percent (CEP) fitness measure, *which is defined as the count of erroneous classifications divided by the size of Y*, and extending the coefficients to allow for linear discriminant analysis as follows:

$$CEP = fitness(X, Y, Bf_1, \dots, Bf_B, c_{00}, \dots, c_{KB}) \tag{4.12}$$

Of course the coefficients $c_{00}, \dots, c_{KB}$ can be taken as is, and the simple CEP returned. However, our new baseline SC system will use LDA as an assisted fitness technique to supply optimal values for the coefficients before returning the CEP fitness measure. This greatly speeds up accuracy and allows the SC to concentrate all of its algorithmic resources similarly on the evolution of an optimal set of basis functions $Bf_1, \dots, Bf_B$.

4.2.6.1 Converting to Basis Space

The first task of our new SC fitness function must be to convert from N by M feature space, X, into N by B basis space XB. Basis space is the training matrix created by assigning basis function conversions to each of the B points in XB as follows:

$$XB_{[n][b]} = Bf_b(X_{[n]}) \tag{4.13}$$

So for each row n of our N by M input feature space training matrix, $(X_{[n]})$, we apply all B basis functions, yielding the B points of our basis space training matrix for row n, $(XB_{[n][b]})$. Our new training matrix is the N by B basis space matrix, XB.

4.2.6.2 Class Clusters and Centroids

Next we must compute the K class cluster matrices for each of the K classes as follows:

$$\textbf{Define } CX_{[k]} \text{ where } XB_{[n]} \in CX_{[k]} \text{ iff } Y_{[n]} = k \tag{4.14}$$

Each matrix $CX_{[k]}$ contains only those training rows of XB which belong to the class k. We also compute the simple Bayesian probability of membership in each class cluster matrix as follows:

$$P_{[K]} = length(CX_{[k]})/N \tag{4.15}$$

Next we compute the K cluster mean vectors, each of which is a vector of length B containing the average value in each of the B columns of each of the K class cluster matrices, CX, as follows:

$$\mu_{[k][b]} = \text{ column mean of the b-th column of } CX_{[K]} \tag{4.16}$$

We next compute the class centroid matrices for each of the K classes, which are simply the mean adjusted class clusters as follows:

$$CXU_{[k][m][b]} = (CX_{[k][m][b]} - \mu_{[k][b]}) \text{ for all } k, m, \text{ and } b \tag{4.17}$$

Finally we must compute the B by B class covariance matrices, which are the class centroid covariance matrices for each class as follows:

$$COV_{[k]} = covarianceMatrix(transpose(CXU_{[k]})) \tag{4.18}$$

Each of the K class covariance matrices is a B by B covariance matrix for that specified class.

In order to support the core LDA assumption that the class covariance matrices are all equal, we compute the final covariance matrix by combining each class covariance matrix according to their naive Bayesian probabilities as follows:

$$C_{[m][n]} = \sum (COV_{[k][m][n]} \times P_{[k]}) \text{ for all } 1 \leqslant k \leqslant K \tag{4.19}$$

The final treatment of the covariance matrix allows the LDA optimal coefficients to be computed as shown in the following section.

4.2.6.3 LDA Coefficients

Now we can easily compute the single axis coefficient for each class as follows.

$$c_{[k][0]} = -0.5\mu_k(C_k)^{-1}(\mu_k)^T + \ln P_k \tag{4.20}$$

The B basis function coefficients for each class are computed as follows.

$$c_{[k][1,\ldots B]} = \mu_k (C_k)^{-1} \tag{4.21}$$

All together these coefficients form the discriminants for each class:

$$y = c_{k0} + c_{k1} * Bf_1(X_n) + c_{k2} * Bf_2(X_n) + \cdots + c_{kB} * Bf_B(X_n) \tag{4.22}$$

The estimated value for Y is defined as follows:

$$y = argmax(D_1(X_n), D_2(X_n), \ldots, D_K(X_n)) \tag{4.23}$$

4.2.6.4 Addressing the Problems with LDA Coefficients

The LDA matrix formalism is made easily computable via an important set of a priori assumptions. These assumptions include that the distribution of the training data is Gaussian, that all covariance matrices are equivalent, and that the covariance matrix for all classes is not singular. Unfortunately these ideal assumptions are not always valid in the course of LDA training, especially since in an SC run, there are tens of thousands to hundreds of thousands of individual LDA training attempts on various nonlinear GP discriminant candidates. Whenever these assumptions are invalid, the resulting LDA coefficients will not be entirely accurate.

To address the problems with LDA coefficients, the LDA^{++} algorithm changes the argument for the LDA matrix from a statistical justification to a heuristic justification where only approximately accurate results are expected.

Since the LDA *heuristic* is no longer expected to obtain optimal results, we must add optimization steps after the LDA to correct for any possible errors produced. The second (post LDA) step is a set of heuristic matrix adjustments designed to correct for any singular covariance matrix conditions. The third step adds Modified Sequential Minimal Optimization (MSMO), and the fourth step adds a Swarm intelligence layer consisting of the Bees algorithm [5].

In the unhappy event that the covariance matrix, C_k, in Eqs. (4.20) and (4.21) is singular the matrix inversion function, $(C_k)^{-1}$ will fail with a divide by zero error. Our heuristic alters the computer code of the covariance matrix inversion function such that, should a divide by zero error be detected, the diagonal of the covariance matrix, C_k, is multiplied by the scalar 1.0001. In the vast majority of cases multiplying the covariance matrix diagonal by this scalar forces the matrix to be non-singular. After diagonal adjustment the matrix inversion is attempted anew. The resulting LDA coefficient will, of course, be inaccurate but nevertheless approximately accurate. After diagonal adjustment, should a divide by zero error still be encountered, then the adjusted computer code will divide by the scalar 0.0001 instead of zero. Once again the resulting LDA coefficient will, of course, be inaccurate but nevertheless approximately accurate.

The LDA coefficients, produced by the adjusted matrix inversion function, will be anywhere from accurate to approximately accurate. In order to optimize the coefficients further, we add a layer of Modified Sequential Minimal Optimization (MSMO), described in the next section.

4.2.6.5 Modified Sequential Minimal Optimization (MSMO)

The third heuristic layer of the LDA^{++} algorithm is an opportunistic modification of Platt's sequential minimization optimization algorithm often used to train support vector machines [14]. At the start of the MSMO heuristic, the percent of misclassification errors (CEP) in the training data set is calculated for the LDA coefficients. If the CEP is greater than 10%, then the MSMO heuristic is skipped on the theory that it is not worthwhile to waste resources on a heuristics which may improve the accuracy of the CEP by a maximum of 2% or 3%. Thus MSMO resources are only expended for better candidates. Better candidates receive more evolutionary activity. Worse candidates receive less evolutionary activity.

At the start of the Modified Sequential Minimal Optimization (MSMO) layer, the candidate contains a swarm pool of a single set of coefficient constants which was produced by the LDA heuristic with its singular matrix modifications. Also the CEP for LDA coefficients in this single entry in the swarm pool is available.

For each repetition of our MSMO algorithm, on the current candidate, the most fit coefficient entry in the swarm pool is chosen. If the CEP for the best coefficient entry is 0%, then the MSMO algorithm is terminated, as in the case when all allocated evolutionary iterations have been exhausted. If the CEP is greater than 0% then a single erroneous training point is chosen at random, n.

Since the chosen training point, n, is in error, we know that it's estimated dependent variable, e, will not match the actual dependent variable, y (i.e. $e \neq y$). If K is the number of classes, then $0 \leq e < K$ and $0 \leq y < K$. Since LDA uses the **argmax** function to select the discriminant function with the highest Bayesian probability, we know that two discriminant formulas have the following relationship.

$$e = argmax(D_1(X_n), D_2(X_n), \ldots, D_K(X_n)) \tag{4.24}$$

And therefore

$$D_e(X_n) \geq D_y(X_n) \tag{4.25}$$

And therefore

$$\begin{aligned} c_{e0} + c_{e1} * Bf_1(X_n) + c_{e2} * Bf_2(X_n) + \cdots + c_{eB} * Bf_B(X_n) \geq \\ c_{y0} + c_{y1} * Bf_1(X_n) + c_{y2} * Bf_2(X_n) + \cdots + c_{yB} * Bf_B(X_n) \end{aligned} \tag{4.26}$$

Our goal at this step in the MSMO algorithm is to force

$$y = argmax(D_1(X_1), D_2(X_2), \dots, D_K(X_n)) \tag{4.27}$$

Since the B basis functions in the K discriminants are all the same, our only option is to modify the coefficients. Using Eq. (4.26), we select the basis function, $B_{fm}(X_n)$, such that the absolute difference $abs((c_{em} * B_{fm}(X_n)) - (c_{ym} * B_{fm}(X_n)))$ is greater than all other possible choices from the B basis functions. We then make minimalist random changes to the coefficients c_{em} and c_{ym} such that Eq. (4.27) is forced to be true.

At this point in the MSMO algorithm, the modified coefficients are used to score the candidate obtaining a new CEP. The modified coefficients along with their associated new CEP are inserted into the candidate's swarm pool, sorted in order of most fit CEP. Then the most fit coefficient entry in the swarm pool is chosen, and the MSMO algorithm repeats itself until all allocated evolutionary iterations have been exhausted.

4.2.7 *Bees Swarm Optimization*

The fourth step adds a Swarm intelligence layer consisting of the Bees algorithm [5]. The BEES algorithm is too complex to describe in this paper. In summary the BEES algorithm involves a fitness driven evolutionary modification of all coefficients at once using a well-defined Swarm intelligence approach. More details can be found in [5].

At the start of the BEES heuristic, the percent of misclassification errors (CEP) in the training data set are calculated for the most fit coefficients. If the CEP is greater than 5%, then the BEES heuristic is skipped on the theory that it is not worthwhile to waste resources on a heuristic which may improve the accuracy of the CEP by a maximum of 0.5% or 1%. Thus BEES resources are only expended for the better candidates. Better candidates receive more evolutionary activity. Worse candidates receive less evolutionary activity.

4.3 User-Defined Typing System

Unconstrained expressions (models) from Symbolic Classification Systems are often rejected by decision makers regardless of their advantageous fitness scores. Decision makers most often require that SC models make intuitive sense and be logically defensible within the problem domain. This is especially true with SC models for high impact decisions. In general, the greater the impact of trusting an SC model, the more the decision maker needs to understand and believe in the model.

SC systems can be modified with template based logic to constrain the evolution of resulting models to match decision maker specifications [6–8]. Template constrained SC systems produce models which are readily accepted by the decision maker since those models always conform to the templates supplied by the decision maker. Unfortunately such template constraint systems are insufficient to accommodate user strong typing rules.

Strong typing rules are important in SC applications where unit types should not be mixed. For instance in a banking system the decision maker might want to prohibit all SC models which add ‘personal income’ to ‘number of defaults’, or which multiply ‘credit score’ by ‘race’. SC models which are confused about appropriate typing will be rejected out of hand by decision makers, *even if they are highly accurate*, and may engender distrust of the SC system as a whole.

4.3.1 User-Defined Templates with Constraints

Given any selected maximum depth K, it is an easy process to construct a maximal binary tree fixed constraint template U_K, which can be overlaid on the GP system without violating the selected maximum depth limit K nor limiting the s-expressions which can be produced. As long as we are reminded that each f-node represents a function node while each t-node represents a terminal node (*either a feature or a real number constant*), the template construction algorithm, for building the template U_K, is both simple and recursive as follows.

- (U_0): t
- (U_1): $(f\ t\ t)$
- (U_2): $(f\ (f\ t\ t)\ (f\ t\ t))$
- (U_3): $(f\ (f\ (f\ t\ t)\ (f\ t\ t))\ (f\ (f\ t\ t)\ (f\ t\ t)))$
- (U_K): $(f\ U_{k-1}\ U_{k-1})$

For the arbitrary fixed template U_K, each f-node represents a Lisp function call such as cos, sin, $+$, /, or ln, and each t-node represents a terminal feature or constant such as x_0, 34.56, x_{10}, or -45.1. The basic GP symbolic regression system [6] contains a set of functions F, and a set of terminals T. We let $t \in T$, where $T = x_0, \ldots, x_M, \ldots$ IEEE real numbers. . . . We let $f \in F = \{+, -, *, /, \text{sin}, \text{cos}, \text{tan}, \text{square}, \text{etc}\}$. Now U_K becomes almost another template representation of the irregular s-expressions in the SC. We say almost because U_K is fixed and regular whereas the SC s-expressions are irregular. To complete the template, we must add a special noop function, ξ, to $F = F \cup \xi$. The special ξ function allows U_K to express irregular s-expressions be defining $\xi(a, b, \ldots) = \xi(a) = a$. Now any basis function produced by the basic GP system will be represented by at least one element of U_K. Adding the ξ function allows U_K to express all possible basis functions generated by the basic GP system to a depth of K. Note to the reader, the ξ function performs the job of a pass-through function. The ξ function allows a fixed-maximal-depth expression in U_K to express trees of varying depth, such as might be produced from a GP system. For instance, the varying depth GP expression

- $x_2 + (x_3 - x_5) = \xi(x_2, 0.0) + (x_3 - x_5) = +(\xi(x_2, 0.0) - (x_3 - x_5))$

which is a fixed-maximal-depth expression in U_2.

In addition to the special pass through function ξ, in our system we also make additional slight alterations to improve template coverage, reduce unwanted errors, and restrict results from wandering into the complex number range. All unary functions, such as cos, are extended to ignore any extra arguments so that, for all unary functions, cos(a, b) = cos(a). The sqroot and ln functions are extended for negative arguments so that sqroot(a) = sqroot(abs(a)) and ln(a) = ln(abs(a)).

Given this formalism of the search space, it is easy to compute the size of the search space, and it is easy to see that the search space is huge even for rather simple basis functions. For our use in this chapter the function set will be the following functions: F = (+ − * / abs inv cos sin tan tanh sqroot square cube quart exp ln ξ) (where inv(x) = 1.0/x). The terminal set is the features x_0 through x_{M-1} and the real constant c, which we shall consider to be standard IEEE double long at 2^{64} in size.

User specified constraints can be added to our SC system by explicitly enumerating each node in the template U_K and by adding constraints to each of the explicitly enumerated nodes. For instance an explicit enumeration of nodes in U_2 would be as follows.

- (U_2): $(f\ (f\ t\ t)\ (f\ t\ t)) = f_0(f_1(t_0, t_1),\ f_2(t_2, t_3))$

User specified constraints can be added to SC search commands by explicit user declarations as follows.

$$\begin{aligned}\text{regress } (f_0(f_1(t_0, t_1),\ f_2(t_2, t_3)))\\ \text{where } \{f_0(cos, sin, tan, *) f_2(+, -, *, /) t_0(x_4, x_6) t_3(x_{10}, x_{19}, 1.0, 0.0)\}\end{aligned} \tag{4.28}$$

The above SC search command will produce models constrained by the explicit user declarations as in the following examples (*remember* f_0, f_2, t_0, *and* t_3 *are constrained as declared above, while* f_1, t_1, *and* t_2 *are unconstrained*) (Table 4.1).

User constraints enhance the probability that a resulting SC model will be accepted by the decision makers; but, alone they are not enough. Adding user specified strong typing to the SC system will also be required for many applications.

Table 4.1 Productions from template $f_0(f_1(t_0, t_1),\ f_2(t_2, t_3))$ as constrained in Eq. (4.28)

f_0	f_1	f_2	t_0	t_1	t_2	t_3	Lisp expression
*	sin	+	x_6	x_{30}	45.7	x_{10}	(* (sin x_6) (+45.7 x_{10}))
*	ξ	*	x_4	x_{40}	x_1	x_{19}	(* x_4 (* x_{10} x_{19}))
sin	ln	*	x_6	0.0	x_{21}	1.0	(sin (ln x_6))
cos	square	/	x_4	x_{63}	x_1	x_{10}	(cos (square x_4))

4.3.2 Strong Typing

A user template facility is greatly enhanced if accompanied by the ability to add strong typing rules to the SC search command. These rules may be formed as in the following examples.

- Number = $\{x_0, x_1, x_{10}, x_{20}\}$
- Number = (Number + Number), (Number ∗ Number), (Number/Number) (Number − Number)
- Income = $\{x_{21}, x_{31}, x_{11}, x_{22}\}$
- Income = (Income + Income), (Income ∗ Income), (Income/Income), (Income − Income)
- Income = cos(Income), tan(Income), sin(Income)

The type rules consist of a “type” followed by a set of features, or operator-type combinations which result in the specified type. The type rules must not contain conflicts as in the rules below which contain a type conflict because feature x_{10} cannot be both a Number and a Widget.

- Number = $\{x_0, x_1, x_{10}, x_{20}\}$
- Income = $\{x_{21}, x_{31}, x_{11}, x_{22}\}$

The type rules can be consolidated by type and subdivided into the features, unary operators, and binary operators which result in the specified type as in the following example.

- Number

$$\begin{cases} \{x_0, x_1, x_{10}, x_{20}\} \\ (Number{+}Number), (Number{*}Number), (Number/Number)(Number- \\ Number) \\ cos(Number), abs(Number) \end{cases}$$

- Income: (Number + Income), (Number ∗ Income), (Income/Number)(Income − Number)

Taken together, a user template system with constraints and strong typing can greatly enhance the acceptability to the final SC model in the eyes of the domain expert. This can be a critical component of model acceptance.

4.4 Deep Learning Enhancements

We use the RQL language to develop a deep learning strategy to enhance the LDA^{++} algorithm [7, 8]. The first deep learning layer (RQL Island) handles feature selection by performing a series of individual LDA (*including the SMO, and Bees*

Algorithm enhancements) learning runs on each individual independent feature using the following RQL search specification.

$$lda(v_0, inv(v_0), abs(v_0), sqroot(v_0), square(v_0), cube(v_0), curoot(v_0), ln(v_0), cos(v_0), sin(v_0), tan(v_0), tanh(v_0))$$

Each independent feature is given this same nonlinear treatment with the above 12 nonlinear basis functions. The best ranking features (CEP) are selected as the *'of-special-interest'* feature group.

Special Interest Feature Selection

1. Input: F^X, X, Y—where F^X is a set of M features, X is an MxN real matrix, Y is an N Vector
2. For v_0 from x_1 to x_M do
3. Run: $lda(v_0, inv(v_0), abs(v_0), sqroot(v_0), square(v_0), cube(v_0), curoot(v_0), ln(v_0), cos(v_0), sin(v_0), tan(v_0), tanh(v_0))$
4. Sort each of LDA classification run by CEP accuracy
5. Extract the feature $v_0 = x_i$ from each of the 50 most accurate runs into the special feature set, F^S.
6. F^S now contains at most 50 features x_i which returned the most accurate CEP scores in Step 3.
7. Return F^S

When feature selection is complete, a second deep learning layer is run, whose only connected inputs are the *'of-special-interest'* feature group. This general deep learning layer is a standard 20 basis function Pareto front LDA evolutionary GP run *(including the SMO, and Bees Algorithm enhancements)* which halts when it has reached a learning plateau.

Initial Pareto Front Layer

1. Input: F^S, X, Y—where F^S is special feature set features, X is an MxN real matrix, Y is an N Vector
2. For $Gens$ from 1 to G evolve
3. $lda(Bf_1, Bf_2, \ldots, Bf_{20})$
4. Where each basis function, Bf_i, selects features ONLY from the special feature set F^S
5. Extract the 20 basis functions from the most fit individual into the best-of-breed basis function set, B^B
6. B^B now contains 20 best-of-breed basis functions $\{B_1^B, \ldots, B_{20}^B\}$ which contain expressions using ONLY the special features in F^S
7. Return B^B

Once the general Pareto front LDA layer is complete, another 41 strongly linked model recurrent deep learning layers are launched. The first of these is another standard 20 basis function Pareto front LDA layer *(including the SMO, and Bees Algorithm enhancements)* but which has ALL features connected.

The next 20 deep learning layers are standard 20 basis function Pareto front LDA layers *(including the SMO, and Bees Algorithm enhancements)* whose only connected inputs are the *'of-special-interest'* feature group, where each layer has only one of its basis functions participating in evolution. The remaining basis functions are all fixed. Each of the 20 deep learning layers has a different basis function undergoing active evolution while the remaining basis functions are fixed.

The next 20 deep learning layers are also standard 20 basis function Pareto front LDA layers *(including the SMO, and Bees Algorithm enhancements)* which have ALL features connected, where each layer has only one of its basis functions participating in evolution. The remaining basis functions are all fixed. Each of the 20 deep learning layers has a different basis function undergoing active evolution while the remaining basis functions are fixed.

Strongly Linked Model Recurrent Layers

```
1. Input: $F^X$, $F^S$, $B^B$, $X$, $Y$—where $X$ is an $MxN$ real matrix, $Y$ is an $N$ Vector
2. For $Gens$ from 1 to $G$ evolve the following 41 layers simultaneously one iteration at a time
3. $lda(Bf_1, Bf_2, \ldots, Bf_{20})$ evolve $Bf_1$ thru $Bf_{20}$ feature set from $F^X$
4. $lda(Bf_1, B^B_2, \ldots, B^B_{20})$ evolve $Bf_1$ only where each $B^B_j$ is fixed with feature set from $F^X$
5. $lda(B^B_1, Bf_2, B^B_3, \ldots, B^B_{20})$ evolve $Bf_2$ only where each $B^B_j$ is fixed with feature set from $F^X$
6. $\cdots$
7. $lda(B^B_1, B^B_2, \ldots, Bf_{20})$ evolve $Bf_{20}$ only where each $B^B_j$ is fixed with feature set from $F^X$
8. $lda(Bf_1, B^B_2, \ldots, B^B_{20})$ evolve $Bf_1$ only where each $B^B_j$ is fixed with feature set from $F^S$
9. $lda(B^B_1, Bf_2, B^B_3, \ldots, B^B_{20})$ evolve $Bf_2$ only where each $B^B_j$ is fixed with feature set from $F^S$
10. $\cdots$
11. $lda(B^B_1, B^B_2, \ldots, Bf_{20})$ evolve $Bf_{20}$ only where each $B^B_j$ is fixed with feature set from $F^S$
12. IF we have discovered a new global best fit champion then
13. Extract the 20 basis functions from the new global best individual into the best-of-breed basis function set, $B^B$
14. GOTO Step 3 and repeat the next evolution step1 3 thru 13 for all layers
15. Return $B^B$
```

Each of the above 41 deep learning layers are *strongly linked and model recurrent*. At the start, the best model discovered by the second 20 basis function learning layer is fed into each of the 41 strongly linked model recurrent deep learning layers. For example:

$$lda(Bf_1, Bf_2, \ldots, Bf_{20}) \tag{4.29}$$

This best scoring model is fed into each of the 41 deep learning layers, where layer 1 performs a general evolutionary search on all 20 basis functions with ALL features connected. Layers 2 through 21 each perform a general evolutionary search on all 20 basis functions with only the *'of-special-interest'* feature connected. And, where each layer evolves only one basis function holding all other basis functions fixed. Layers 22 through 41 each perform a general evolutionary search on all 20 basis functions with ALL the features connected. And, where each layer evolves only one basis function holding all other basis functions fixed.

Since each of these 41 deep learning layers are strongly linked, the moment any one of these layers discovers a model, *which is a global fitness improvement*, all processing stops AND the new model is fed back into all 41 deep learning layers (*hence the term model recurrent*). For instance, after some evolutionary effort, the n-th layer discovers a model which is a global fitness improvement, for example: $lda(Bg_1, Bg_2, \ldots, Bg_{20})$. This *new improved globally fit* model is fed recurrently into each of the 41 deep learning layers, and the whole strongly linked model recurrent process begins anew until another better globally fit model is discovered.

For purposes of keeping the models *"WhiteBox"* we arbitrarily set the maximum number of basis functions to 20, and we arbitrarily set the maximum depth of each basis function to 3 (i.e. 2^3 = *8 terminal nodes*). If a basis function is not needed for accuracy the algorithm will set its coefficient to 0.0, and each a basis function will evolve to the optimal shape (*limited by our arbitrary maximum depth of 3*). Obviously these parameters can be adjusted for the particular problem domain.

4.5 Artificial Test Problems

A set of ten artificial classification problems are constructed, with no noise, to compare the five proposed SC algorithms and five well-known commercially available classification algorithms to determine just where SC now ranks in competitive comparison. The test problems were described in [11] and are reproduced in the appendix for convenience.

The five SC algorithms are: simple genetic programming using argmax referred to herein as AMAXSC; the M$_2$GP algorithm [4]; the MDC algorithm [9], Linear Discriminant Analysis (LDA) [10], and Linear Discriminant Analysis extended with MSMO and Swarm (LDA^{++}). The five commercially available classification algorithms are available in the KNIME system [1], and are as follows: Multiple Layer Perceptron Learner (MLP); Decision Tree Learner (DTL); Random Forest Learner (RFL); Tree Ensemble Learner (TEL); and Gradient Boosted Trees Learner (GBTL).

Each of the artificial test problems is created around an X training matrix filled with random real numbers in the range $[-10.0, +10.0]$. The number of rows and columns in each test problem varies from 5000×25 to 5000×1000 depending upon the difficulty of the problem. The number of classes varies from $Y = 0, 1$ to $Y = 0, 1, 2, 3, 4$ depending upon the difficulty of the problem. The test problems are designed to vary from extremely easy to very difficult. The first test problem is linearly separable with 2 classes on 25 columns. The tenth test problem is nonlinear multimodal with 5 classes on 1000 columns.

Standard statistical best practices out of sample testing are employed. First training matric X is filled with random real numbers in the range $[-10.0, +10.0]$, and the Y class values are computed from the argmax functions specified below. A champion is trained on the training data. Next a testing matric X is filled with random real numbers in the range $[-10.0, +10.0]$, and the Y class values are computed from the argmax functions specified below. The previously trained champion is run on the testing data and scored against the Y values. Only the out of sample testing scores are shown in the results in Table 4.2.

The Symbolic Classification system for all four SC algorithms (AMAXSC, M_2GP, MDC, LDA) avail themselves of the following operators:

- Binary Operators: $+ - / \times$ minimum maximum
- Relational Operators: $<, \leq, =, \neq, \geq, >$
- Logical Operators: lif land lor
- Unary Operators: inv abs sqroot square cube curoot quart quroot exp ln binary sign sig cos sin tan tanh

The unary operators sqroot, curoot, and quroot are square root, cube root, and quart root respectively. The unary operators inv, ln, and sig are the inverse, natural log, and sigmoid functions respectively.

4.6 Real World Banking Problem

For real world testing, we use an actual banking data set for loan scoring as it was received by Lantern Credit. The training data contains 337 features with 36,223 rows, while the testing data contains the same 337 features with an additional 85,419 rows. The training and testing data are distinct, and the training and testing data are completely anonymized. The 337 independent features contain both categorical and continuous data. But do not contain any FICO or similar agency ranking scores. The dependent variable is 1 for ‘good’ and 0 for ‘bad’ loan.

An extensive user-defined typing file of thousands of strong typing rules was constructed, with the help of domain experts.

The objective is to develop a model for scoring incoming 337 feature loan applications. The model must pass the bank’s in-house compliance management, and be accepted by the bank’s in-house lending management team.

Table 4.2 Test problem CEP testing results after deep learning enhancements

Test	AMAXSC	LDA	M_2GP	MDC	DTL	GBTL	MLP	RFL	TEL	**LDA^{++}**
T_1	0.0808	0.0174	0.0330	0.0330	0.0724	0.0308	0.0072	0.0492	0.0496	**0.0138**
T_2	0.1108	0.0234	0.0656	0.0402	0.0740	0.0240	0.0360	0.0664	0.0648	**0.0116**
T_3	0.1436	0.0182	0.1010	0.0774	0.0972	0.0332	0.0724	0.1522	0.1526	**0.0132**
T_4	0.1954	0.0188	0.0180	0.0162	0.0174	0.0170	0.0472	0.0260	0.0252	***0.0194***
T_5	0.1874	0.1026	0.1052	0.1156	0.0858	0.0530	0.3250	0.0920	0.0946	***0.0712***
T_6	0.6702	0.5400	0.4604	0.5594	0.5396	0.3198	0.6166	0.6286	0.6284	***0.5420***
T_7	0.4466	0.4002	0.4060	0.4104	0.2834	0.2356	0.4598	0.2292	0.2284	**0.1522**
T_8	0.8908	0.4176	0.4006	0.4124	0.2956	0.2340	0.4262	0.2250	0.2248	**0.2302**
T_9	0.8236	0.7450	0.7686	0.6210	0.6058	0.4286	0.6904	0.4344	0.4334	**0.4188**
T_{10}	0.8130	0.7562	0.7330	0.6440	0.5966	0.4286	0.5966	0.4296	0.4352	**0.4186**
Avg	0.4362	0.3039	0.3091	0.2930	0.2668	0.1805	0.3277	0.2333	0.2337	**0.1891**

Best performance printed in bold.

4.7 Performance on the Theoretical Problems

Here we compare the out of sample CEP testing scores of the five proposed SC algorithms and five well-known commercially available classification algorithms to determine where SC ranks in competitive comparison. The five SC algorithms are: simple genetic programming using argmax referred to herein as AMAXSC; the M_2GP algorithm [4]; the MDC algorithm [9], Linear Discriminant Analysis (LDA) [10], and Linear Discriminant Analysis extended with SMO and Swarm (LDA^{++}). The five commercially available classification algorithms are available in the KNIME system [1], and are as follows: Multiple Layer Perceptron Learner (MLP); Decision Tree Learner (DTL); Random Forest Learner (RFL); Tree Ensemble Learner (TEL); and Gradient Boosted Trees Learner (GBTL). The following table lists each classification algorithm in descending order of average CEP scores on all ten theoretical test problems. The lower the CEP the more accurate the classification results.

On a positive note, the four new proposed symbolic classifications algorithms are a substantial improvement over the simple AMAXSC algorithm, and the newly proposed LDA^{++} SC algorithm is extremely competitive with the best performer. The top performer overall by a very small margin is the Gradient Boosted Trees Learner (GBTL). The penultimate performer is the newly proposed LDA^{++} SC algorithm.

It is interesting to note that all four newly proposed SC algorithms perform better overall than the Multiple Layer Perceptron Learner (MLP). Of the four newly proposed SC algorithms, the LDA^{++} algorithm was the best overall performer, and is extremely competitive with the commercially available Gradient Boosted Trees Learner (GBTL). In fact the LDA^{++} algorithm did better than GBTL on all but three of the test problems.

4.8 Performance on the Real World Problem

We include a comparison study of the five new SC algorithms and five well-known commercially available classification algorithms to determine just where SC now ranks in competitive comparison on this real world banking problem. Also included is the bank's benchmark score, achieved over a multiple month period by the bank's in-house data science team with proprietary tools. The bank's benchmark score represents the best that human domain experts can achieve with months of effort and state-of-the-art data science tools.

Table 4.3 Banking problem CEP testing results after deep learning enhancements

AMAXSC	LDA	M_2GP	MDC	DTL	GBTL	MLP	RFL	TEL	LDA^{++}	**Benchmark**
0.9101	0.4282	0.4472	0.4174	0.3165	0.2621	0.4523	0.2658	0.3084	0.2856	**0.2841**

Best performance printed in bold.

The bank's benchmark score is compared to the five proposed SC algorithms and five well-known commercially available classification algorithms to determine just where SC now ranks in competitive comparison. The five SC algorithms are: simple genetic programming using argmax referred to herein as AMAXSC; the M_2GP algorithm [4]; the MDC algorithm [9], Linear Discriminant Analysis (LDA) [10], and Linear Discriminant Analysis extended with MSMO and Swarm (LDA^{++}). The five commercially available classification algorithms are available in the KNIME system [1], and are as follows: Multiple Layer Perceptron Learner (MLP); Decision Tree Learner (DTL); Random Forest Learner (RFL); Tree Ensemble Learner (TEL); and Gradient Boosted Trees Learner (GBTL) (Table 4.3).

Each CEP score represents the percent of applications which were misclassified (*lower is better*). On a positive note, the four new proposed symbolic classifications algorithms are a large improvement over the AMAXSC algorithm, and the newly proposed LDA^{++} SC algorithm is extremely competitive with the best performer. The top performer overall by a 2% margin is the Gradient Boosted Trees Learner (GBTL). It is interesting to note that the LDA^{++} result included user-defined types—thus rendering basis functions which meet the preconditions established by the bank. This greatly improves the probability of model acceptance.

It is interesting to note that of the four newly proposed SC algorithms, the LDA^{++}, algorithm was the best performer, and is extremely competitive with the bank's in-house benchmark. Both the Random Forest Learner (RFL) and the Gradient Boosted Trees Learner (GBTL) are extremely competitive top performers. However, despite achieving a 2% advantage over the Bank's benchmark, both the RFL and GBTL models will face a difficult uphill battle for acceptance by the bank's in-house compliance management. Each of the RFL and GBTL models contain hundreds to thousands of embedded decision trees. They are very difficult to understand and doubly difficult to support cogently within the logic of the banking domain.

On the other hand, the newly proposed LDA^{++} SC algorithm produces an easily understood *"WhiteBox"* model with 20 or less very simple user type compliant basis functions. In its first training run, the LDA^{++} algorithm has produced a model that is highly competitive with the bank's in-house model (*which took their in-house data science team months to construct*). Furthermore, the LDA^{++} model is already in a form that is easily understood by the bank's in-house data science team, easy for them to work with, and compatible with their pre-specified constraints.

4.9 Conclusion

Several papers have proposed GP Symbolic Classification algorithms for multi-class classification problems [4, 9, 10, 13]. Comparing these newly proposed SC algorithms with the performance of five commercially available classifications algorithms shows that progress has been made. All four newly proposed SC algorithms performed better overall than the Multiple Layer Perceptron Learner (MLP). Of the four newly proposed SC algorithms, the LDA^{++} algorithm was the best overall performer by a good margin, and is extremely competitive with the commercially available Gradient Boosted Trees Learner (GBTL). In fact the LDA^{++} algorithm did better than GBTL on all but three of the theoretical test problems.

Clearly progress has been made in the development of commercially competitive SC algorithms. We now have an SC classification algorithm which is highly competitive with GBTL. But, a great deal more work has to be done before SC can radically outperform the Gradient Boosted Trees Learner (GBTL) on the basis of raw accuracy alone. We must perform comparative tests on a much wider range of theoretical problems, and we must perform comparative tests on a wide range of real world industry problems.

Acknowledgements Our thanks to: Thomas May from Lantern Credit for assisting with the KNIME Learner training/scoring on all ten artificial classification problems.

Appendix: Artificial Test Problems

- T_1: $y = argmax(D_1, D_2)$ where $Y = 1, 2$, X is 5000×25, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57*x_0), (-39.34*x_1), (2.13*x_2), (46.59*x_3), (11.54*x_4)) \\ D_2 = sum((-1.57*x_0), (39.34*x_1), (-2.13*x_2), (-46.59*x_3), (-11.54*x_4)) \end{cases}$$

- T_2: $y = argmax(D_1, D_2)$ where $Y = 1, 2$, X is 5000×100, and each D_i is as follows:

$$\begin{cases} D_1 = sum((5.16*x_0), (-19.83*x_1), (19.83*x_2), (29.31*x_3), (5.29*x_4)) \\ D_2 = sum((-5.16*x_0), (19.83*x_1), (-0.93*x_2), (-29.31*x_3), (5.29*x_4)) \end{cases}$$

- T_3: $y = argmax(D_1, D_2)$ where $Y = 1, 2$, X is 5000×1000, and each D_i is as follows:

$$\begin{cases} D_1 = sum((-34.16*x_0), (2.19*x_1), (-12.73*x_2), (5.62*x_3), (-16.36*x_4)) \\ D_2 = sum((34.16*x_0), (-2.19*x_1), (12.73*x_2), (-5.62*x_3), (16.36*x_4)) \end{cases}$$

- T_4: $y = argmax(D_1, D_2, D_3)$ where $Y = 1, 2, 3$, X is 5000×25, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * cos(x_0)), (-39.34 * square(x_{10})), (2.13 * (x_2/x_3)), \\ (46.59 * cube(x_{13})), (-11.54 * log(x_4))) \\ D_2 = sum((-0.56 * cos(x_0)), (9.34 * square(x_{10})), (5.28 * (x_2/x_3)), \\ (-6.10 * cube(x_{13})), (1.48 * log(x_4))) \\ D_3 = sum((1.37 * cos(x_0)), (3.62 * square(x_{10})), (4.04 * (x_2/x_3)), \\ (1.95 * cube(x_{13})), (9.54 * log(x_4))) \end{cases}$$

- T_5: $y = argmax(D_1, D_2, D_3)$ where $Y = 1, 2, 3$, X is 5000×100, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * sin(x_0)), (-39.34 * square(x_{10})), (2.13 * (x_2/x_3)), \\ (46.59 * cube(x_{13})), (-11.54 * log(x_4))) \\ D_2 = sum((-0.56 * sin(x_0)), (9.34 * square(x_{10})), (5.28 * (x_2/x_3)), \\ (-6.10 * cube(x_{13})), (1.48 * log(x_4))) \\ D_3 = sum((1.37 * sin(x_0)), (3.62 * square(x_{10})), (4.04 * (x_2/x_3)), \\ (1.95 * cube(x_{13})), (9.54 * log(x_4))) \end{cases}$$

- T_6: $y = argmax(D_1, D_2, D_3)$ where $Y = 1, 2, 3$, X is 5000×1000, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * tanh(x_0)), (-39.34 * square(x_{10})), (2.13 * (x_2/x_3)), \\ (46.59 * cube(x_{13})), (-11.54 * log(x_4))) \\ D_2 = sum((-0.56 * tanh(x_0)), (9.34 * square(x_{10})), (5.28 * (x_2/x_3)), \\ (-6.10 * cube(x_{13})), (1.48 * log(x_4))) \\ D_3 = sum((1.37 * tanh(x_0)), (3.62 * square(x_{10})), (4.04 * (x_2/x_3)), \\ (1.95 * cube(x_{13})), (9.54 * log(x_4))) \end{cases}$$

- T_7: $y = argmax(D_1, D_2, D_3, D_4, D_5)$ where $Y = 1, 2, 3, 4, 5$, X is 5000×25, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * cos(x_0/x_{21})), (9.34 * ((square(x_{10})/x_{14}) * x_6)), \\ (2.13 * ((x_2/x_3) * log(x_8))), (46.59 * (cube(x_{13}) * (x_9/x_2))), \\ (-11.54 * log(x_4 * x_{10} * x_{15}))) \\ D_2 = sum((-1.56 * cos(x_0/x_{21})), (7.34 * ((square(x_{10})/x_{14}) * x_6)), \\ (5.28 * ((x_2/x_3) * log(x_8))), (-6.10 * (cube(x_{13}) * (x_9/x_2))), \\ (1.48 * log(x_4 * x_{10} * x_{15}))) \\ D_3 = sum((2.31 * cos(x_0/x_{21})), (12.34 * ((square(x_{10})/x_{14}) * x_6)), \\ (-1.28 * ((x_2/x_3) * log(x_8))), (0.21 * (cube(x_{13}) * (x_9/x_2))), \\ (2.61 * log(x_4 * x_{10} * x_{15}))) \\ D_4 = sum((-0.56 * cos(x_0/x_{21})), (8.34 * ((square(x_{10})/x_{14}) * x_6)), \\ (16.71 * ((x_2/x_3) * log(x_8))), (-2.93 * (cube(x_{13}) * (x_9/x_2))), \\ (5.228 * log(x_4 * x_{10} * x_{15}))) \\ D_5 = sum((1.07 * cos(x_0/x_{21})), (-1.62 * ((square(x_{10})/x_{14}) * x_6)), \\ (-0.04 * ((x_2/x_3) * log(x_8))), (-0.95 * (cube(x_{13}) * (x_9/x_2))), \\ (0.54 * log(x_4 * x_{10} * x_{15}))) \end{cases}$$

- T_8: $y = argmax(D_1, D_2, D_3, D_4, D_5)$ where $Y = 1, 2, 3, 4, 5$, X is 5000×100, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * sin(x_0/x_{11})), (9.34 * ((square(x_{12})/x_4) * x_{46})), \\ (2.13 * ((x_{21}/x_3) * log(x_{18}))), (46.59 * (cube(x_3) * (x_9/x_2))), \\ (-11.54 * log(x_{14} * x_{10} * x_{15}))) \\ D_2 = sum((-1.56 * sin(x_0/x_{11})), (7.34 * ((square(x_{12})/x_4) * x_{46})), \\ (5.28 * ((x_{21}/x_3) * log(x_{18}))), (-6.10 * (cube(x_3) * (x_9/x_2))), \\ (1.48 * log(x_{14} * x_{10} * x_{15}))) \\ D_3 = sum((2.31 * sin(x_0/x_{11})), (12.34 * ((square(x_{12})/x_4) * x_{46})), \\ (-1.28 * ((x_{21}/x_3) * log(x_{18}))), (0.21 * (cube(x_3) * (x_9/x_2))), \\ (2.61 * log(x_{14} * x_{10} * x_{15}))) \\ D_4 = sum((-0.56 * sin(x_0/x_{11})), (8.34 * ((square(x_{12})/x_4) * x_{46})), \\ (16.71 * ((x_{21}/x_3) * log(x_{18}))), (-2.93 * (cube(x_3) * (x_9/x_2))), \\ (5.228 * log(x_{14} * x_{10} * x_{15}))) \\ D_5 = sum((1.07 * sin(x_0/x_{11})), (-1.62 * ((square(x_{12})/x_4) * x_{46})), \\ (-0.04 * ((x_{21}/x_3) * log(x_{18}))), (-0.95 * (cube(x_3) * (x_9/x_2))), \\ (0.54 * log(x_{14} * x_{10} * x_{15}))) \end{cases}$$

- T_9: $y = argmax(D_1, D_2, D_3, D_4, D_5)$ where $Y = 1, 2, 3, 4, 5$, X is 5000×1000, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * sin(x_{20} * x_{11})), (9.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (2.13 * ((x_{321} - x_3) * tan(x_{18}))), (46.59 * (square(x_3)/(x_{49} * x_{672}))), \\ (-11.54 * log(x_{24} * x_{120} * x_{925}))) \\ D_2 = sum(((-1.56) * sin(x_{20} * x_{11})), (7.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (5.28 * ((x_{321} - x_3) * tan(x_{18}))), ((-6.10) * (square(x_3)/(x_{49} * x_{672}))), \\ (1.48 * log(x_{24} * x_{120} * x_{925}))) \\ D_3 = sum((2.31 * sin(x_{20} * x_{11})), (12.34 * (tanh(x_{12}/x_4) * x_{46})), \\ ((-1.28) * ((x_{321} - x_3) * tan(x_{18}))), (0.21 * (square(x_3)/(x_{49} * x_{672}))), \\ (2.61 * log(x_{24} * x_{120} * x_{925}))) \\ D_4 = sum(((-0.56) * sin(x_{20} * x_{11})), (8.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (16.71 * ((x_{321} - x_3) * tan(x_{18}))), ((-2.93) * (square(x_3)/(x_{49} * x_{672}))), \\ (5.228 * log(x_{24} * x_{120} * x_{925}))) \\ D_5 = sum((1.07 * sin(x_{20} * x_{11})), ((-1.62) * (tanh(x_{12}/x_4) * x_{46})), \\ ((-0.04) * ((x_{321}-x_3) * tan(x_{18}))), ((-0.95) * (square(x_3)/(x_{49} * x_{672}))), \\ (0.54 * log(x_{24} * x_{120} * x_{925}))) \end{cases}$$

- T_{10}: $y = argmax(D_1, D_2, D_3, D_4, D_5)$ where $Y = 1, 2, 3, 4, 5$, X is 5000×1000, and each D_i is as follows:

$$\begin{cases} D_1 = sum((1.57 * sin(x_{20} * x_{11})), (9.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (2.13 * ((x_{321} - x_3) * tan(x_{18}))), (46.59 * (square(x_3)/(x_{49} * x_{672}))), \\ (-11.54 * log(x_{24} * x_{120} * x_{925}))) \\ D_2 = sum(((-1.56) * sin(x_{20} * x_{11})), (7.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (5.28 * ((x_{321} - x_3) * tan(x_{18}))), ((-6.10) * (square(x_3)/(x_{49} * x_{672}))), \\ (1.48 * log(x_{24} * x_{120} * x_{925}))) \\ D_3 = sum((2.31 * sin(x_{20} * x_{11})), (12.34 * (tanh(x_{12}/x_4) * x_{46})), \\ ((-1.28) * ((x_{321} - x_3) * tan(x_{18}))), (0.21 * (square(x_3)/(x_{49} * x_{672}))), \\ (2.61 * log(x_{24} * x_{120} * x_{925}))) \\ D_4 = sum(((-0.56) * sin(x_{20} * x_{11})), (8.34 * (tanh(x_{12}/x_4) * x_{46})), \\ (16.71 * ((x_{321} - x_3) * tan(x_{18}))), ((-2.93) * (square(x_3)/(x_{49} * x_{672}))), \\ (5.228 * log(x_{24} * x_{120} * x_{925}))) \\ D_5 = sum((1.07 * sin(x_{20} * x_{11})), ((-1.62) * (tanh(x_{12}/x_4) * x_{46})), \\ ((-0.04) * ((x_{321}-x_3) * tan(x_{18}))), ((-0.95) * (square(x_3)/(x_{49} * x_{672}))), \\ (0.54 * log(x_{24} * x_{120} * x_{925}))) \end{cases}$$

References

1. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Kötter, T., Meinl, T., Ohl, P., Thiel, K., Wiswedel, B.: Knime-the konstanz information miner: version 2.0 and beyond. ACM SIGKDD Explorations Newsletter **11**, 26–31 (2009)

2. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of Human Genetics **7**, 179–188 (1936)
3. Friedman, J.H.: Regularized discriminant analysis. Journal of the American Statistical Association **84**, 165–175 (1989)
4. Ingalalli, V., Silva, S., Castelli, M., Vanneschi, L.: A multi-dimensional genetic programming approach for multi-class classification problems. In: European Conference on Genetic Programming 2014, pp. 48–60. Springer (2014)
5. Karaboga, D., Akay, B.: A survey: algorithms simulating bee swarm intelligence. Artificial intelligence review **31**(1–4), 61 (2009)
6. Korns, M.F.: A baseline symbolic regression algorithm. In: Genetic Programming Theory and Practice X. Springer (2012)
7. Korns, M.F.: Extreme accuracy in symbolic regression. In: Genetic Programming Theory and Practice XI, pp. 1–30. Springer (2014)
8. Korns, M.F.: Highly accurate symbolic regression with noisy training data. In: Genetic Programming Theory and Practice XIII, pp. 91–115. Springer (2016)
9. Korns, M.F.: An evolutionary algorithm for big data multiclass classification problems. In: Genetic Programming Theory and Practice XIV. Springer (2017)
10. Korns, M.F.: Evolutionary linear discriminant analysis for multiclass classification problems. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 233–234. ACM (2017)
11. Korns, M.F.: Genetic programming symbolic classification: A study. In: Genetic Programming Theory and Practice XV, pp. 39–52. Springer (2017)
12. McLachlan, G.: Discriminant analysis and statistical pattern recognition, vol. 544. John Wiley & Sons (2004)
13. Munoz, L., Silva, S., Trujillo, L.: M3gp–multiclass classification with gp. In: European Conference on Genetic Programming 2015, pp. 78–91. Springer (2015)
14. Platt, J.: Sequential minimal optimization: A fast algorithm for training support vector machines. Tech. Rep. MSR-TR-98-14, Microsoft Research (1998)

Chapter 5
Cluster Analysis of a Symbolic Regression Search Space

Gabriel Kronberger, Lukas Kammerer, Bogdan Burlacu, Stephan M. Winkler, Michael Kommenda, and Michael Affenzeller

5.1 Introduction

Knowledge discovery systems such as Genetic Programming (GP) for symbolic regression often have to deal with a very large search space of mathematical expressions, which only grows exponentially larger with the number of input variables.

Genetic programming guides the search via selection and discovers new model structures via the action of crossover and mutation. Population diversity plays an

G. Kronberger (✉)
Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria
e-mail: Gabriel.Kronberger@fh-hagenberg.at

L. Kammerer · B. Burlacu · M. Kommenda
Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Josef Ressel Center for Symbolic Regression, University of Applied Sciences Upper Austria, Hagenberg, Austria
e-mail: Lukas.kammerer@fh-hagenberg.at; Bogdan.Burlacu@fh-hagenberg.at; michael.kommenda@fh-hagenberg.at

S. M. Winkler · M. Affenzeller
Heuristic and Evolutionary Algorithms Laboratory (HEAL), University of Applied Sciences Upper Austria, Hagenberg, Austria

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
e-mail: Stephan.Winkler@fh-hagenberg.at; michael.affenzeller@fh-hagenberg.at

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_5

important role in this process, as it affects the algorithm's ability to assemble existing building blocks into increasingly-fit solutions. The relationship between diversity at both the genotypic and phenotypic level has been previously explored [1], leading to a number of important insights:

- Strong exploitation of structures occurs in almost all runs
- Diversity at the structural level is quickly lost
- Encouraging different amounts of diversity can lead to better performance
- The interplay between genetic operators induces a neighborhood structure in the search space
- Fitness is positively-correlated with fitness-based (phenotypic) diversity and negatively correlated with genotypic diversity

In light of the above, we set our goal to investigate GP's ability to explore different areas of the search space by superimposing a neighborhood structure obtained via clustering of symbolic regression models generated via grammar enumeration.

In this contribution we concentrate on the distribution of models in symbolic regression search spaces. Our motivation for this work is that we hope to be able to reduce the computational effort which is required to find well-fitting symbolic regression models by precomputing a clustering of all symbolic regression models in the search space. In particular, we aim to precompute a similarity network or (equivalently) a hierarchical clustering for symbolic regression models, that is independent from a concrete dataset.

Our research questions in this work are:

1. *What is the distribution of models in a symbolic regression search space?* We are interested in identifying clusters of similar models whereby similarity could either be determined based on the model outputs (phenotypic) or similarity of the evolved expressions (genotypic).
2. *What is the distribution of solutions visited by genetic programming?* Here we are interested in the systematic search biases of GP. In particular, (1) whether there are areas of the search space that are completely ignored by GP, and (2) how the individuals in a GP population are distributed in the search space and how this distribution changes from the beginning to the end of a GP run.

Our assumptions about the goals of symbolic regression modeling indexregression, symbolic are the following. We use these assumptions as a guide for our research in symbolic regression solution methods.

- The aim of symbolic regression modeling is primarily to find compact expressions that are open for interpretation.
- Shallow expressions are preferred over deeply nested expressions.
- Models for real-world regression problems are often made up of multiple terms which capture independent effects. The independent terms can be modeled one after another.

- Interesting real-world regression problems often necessitate to capture non-linear correlations in the model. A non-linear effect is often driven by only one independent variable.
- Interactions of two or three variables are common. Interactions of more than four variables are often not relevant.
- The set of potentially relevant variables is much larger than the set of actually relevant variables. It is usually not known which individual variables or which interactions of variables are relevant.
- Measurements of input variables as well as target variables are noisy.

5.2 Methodology

In our journey to find answers for the research questions stated above we enumerate the complete search space (see Sect. 5.2.1) and evaluate all expressions for fixed input data. To limit the size of the search space we consider only uni-variate functions and limit the maximum length of expressions. Additionally, we use a grammar which constrains the complexity of expressions and which does not allow numeric parameters (i.e. random constants). This further reduces the search space.

For the analysis of the distribution of expressions in the symbolic regression search space (*RQ1*), our idea is to create a visual map of all expressions which hopefully allows us to identify larger clusters of similar expressions. Thus, we use the set of all evaluated expressions, identify the set of phenotypically distinct expressions and determine the phenotypically nearest neighbors for each expression (see Sect. 5.2.3). This allows us to map expressions to a two-dimensional space while preserving the local neighborhood structure of the high-dimensional space. The graph of nearest neighbors also allows us to create a clustering of all expressions. As we are interested in both—the map of phenotypically similar expressions as well as the map of genotypically similar expressions—we create a similar map based on a measure for genotypic similarity (see Sect. 5.2.2). Ideally, we expect to see similar cluster structure on both levels, assuming that expressions that are genotypically similar should also be phenotypically similar and vice-versa.[1]

For the analysis of the search bias of GP (*RQ2*), the idea is to re-use the map and clusters that we generated in the first phase and analyze whether GP explores the complete map and all clusters. Our idea is to find the phenotypically most similar expression in the pre-calculated map for each solution candidate visited by GP. For this we determine the visitation frequency for the pre-computed clusters for each generation of GP. We expect that GP visits many different clusters in the beginning and converges to the clusters with well-fitting expressions at the end of the run.

[1]We actually found that this assumption is wrong. We found that the search space can be split into clusters of phenotypically and genotypically similar expressions. However, we could not show that phenotypically similar expressions also are phenotypically similar and/or vice versa. This is intuitive because two highly similar expressions become dissimilar on the phenotypic level just by a multiplication with zero. Symmetrically, many different expressions can be found which produce the same output.

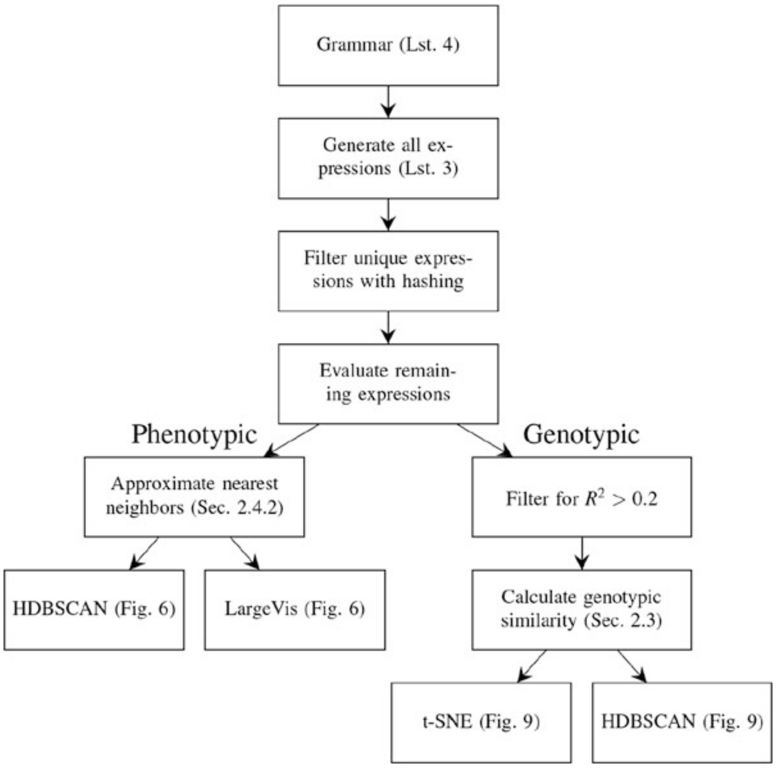

Grammar (Lst. 4)
Generate all expressions (Lst. 3)
Filter unique expressions with hashing
Evaluate remaining expressions
Phenotypic
Genotypic
Approximate nearest neighbors (Sec. 2.4.2)
Filter for $R^2 > 0.2$
HDBSCAN (Fig. 6)
LargeVis (Fig. 6)
Calculate genotypic similarity (Sec. 2.3)
t-SNE (Fig. 9)
HDBSCAN (Fig. 9)

Fig. 5.1 Overview of the flow of information for the search space visualization and clustering (RQ1)

Figure 5.1 shows an overview of the flow of information for search space visualization and clustering. Figure 5.2 shows how we map GP solution candidates to the pre-computed mapping of the search space and the clusters.

A major challenge in our methodological approach is the sheer size of the search space. We have handled this challenge using state-of-the-art algorithms for dimensionality reduction and clustering which still work for datasets with millions of observations and hundreds of features (see Sect. 5.2.4). The core idea of these algorithms is the approximation of nearest neighbors using random projection trees [3].

5.2.1 Grammar Enumeration

We create symbolic regression models by deriving sentences from a formal grammar for expressions as shown in the listing in Fig. 5.3. By defining a maximum sentence length and omitting numerical constants as a start, a large but finite set

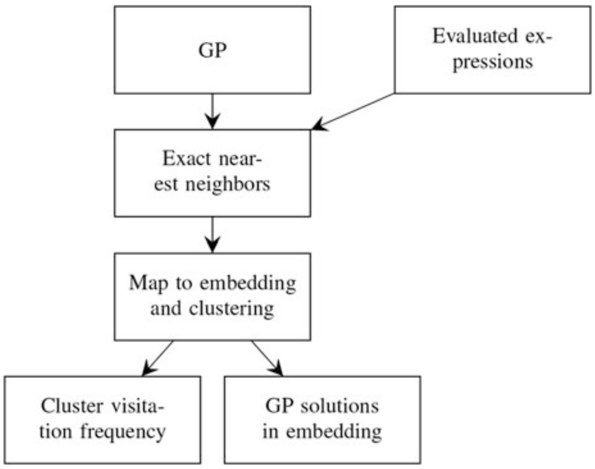

GP
Evaluated expressions
Exact nearest neighbors
Map to embedding and clustering
Cluster visitation frequency
GP solutions in embedding

Fig. 5.2 Overview of the flow of information for tracking which parts of the search space are explored by GP (RQ2). GP solution candidates are mapped to the visualization and the clusters by finding the most similar representative

```
Algorithm for grammar enumeration:
    stack.push(start)

    while stack not empty:
       phrase = stack.pop()
       symbol = fetch nonterminal symbol from phrase
       for each production rule of symbol:
          create new phrase with substituted symbol

          if new phrase is a sentence:
             evaluate(new phrase)
             save(new phrase)
          else:
             stack.push(new phrase)
```

Fig. 5.3 Pseudo-code for generating all sentences of a language defined via a context-free grammar

of all possible models—the search space—can be generated for a problem. These sentences without actual constant values can be seen as a general structure of an actual model [5, 14].

Given a formal grammar many mathematical identities and other phenotypically equal but genotypically different expressions are generated. This includes for example different orders of arguments in commutative operators or different representations of binomial identities. To keep the search space size manageable, we want to avoid semantic duplicates. Although it is computationally not feasible to fully prevent all semantic duplicates in such a large search space, their number can be largely reduced in two simple steps: First, the grammar is restricted, so that only one representation of relevant mathematical identities can be derived. Second, identities which cannot be prevented in the grammar are identified by hashing: semantic duplicates should have the same hash value.

```
G(Expr):
Expr   -> Term "+" Expr | Term
Term   -> Factor "*" Term | Factor | "1/(" InvExpr ")"
Factor -> VarFac | ExpFac | LogFac | SinFac
VarFac -> <variable>
ExpFac -> "exp(" SimpleTerm ")"
LogFac -> "log(" SimpleExpr ")"
SinFac -> "sin(" SimpleExpr ")"

SimpleExpr -> SimpleTerm "+" SimpleExpr | SimpleTerm
SimpleTerm -> VarFac "*" SimpleTerm | VarFac

InvExpr -> InvTerm "+" InvExpr | InvTerm
InvTerm -> Factor "*" InvTerm | Factor
```

Fig. 5.4 The formal grammar used for grammar enumeration. In the design of the grammar, we want to allow a large set of potentially interesting expressions on the one hand, on the other hand we want to restrict the search space to disallow overly complex expressions as well as many different forms of semantically equal expressions

Expressions derived from the restricted grammar (Fig. 5.4) are sums of terms, which again contain variables and unary functions, such as e.g. the sine function or the inverse function. The latter can only occur once per term. Also the structures of function arguments are individually restricted.

Using a context-free grammar, semantic duplicates such as differently ordered terms cannot be prevented. Therefore, we additionally use a semantic hash function to identify semantically equivalent expressions. For each derived symbolic expression we calculate a hash value symbolically without evaluating the expression. Semantic duplicates are recognized by comparing the hash values of previously derived sentences. In case of a match, the derived sentence is a likely to be a semantic duplicate and therefore discarded.

The hashing function calculates the hash value recursively from a syntax tree. Each terminal symbol in the tree is assigned to a constant hash value. To cover commutativity, binary operators like multiplication or addition are flattened to n-ary operators and their arguments are ordered.

5.2.2 Phenotypic Similarity

For the phenotypic similarity we use Pearson's correlation coefficient of the model outputs. This allows us to determine the output similarity regardless of the offset and scale of the function values [4]. When we evaluate the expressions it is necessary to assume a range of valid input values. We use 100 points distributed on a grid in the range $(-5.0..5.0)$. All output vectors are scaled to zero mean and unit variance; undefined output values and infinity values are replaced by the average output. This preprocessing allows us to use cosine-similarity for the clustering and visualization which is supported by many implementations for approximate nearest neighbors and equivalent to Pearson's correlation coefficient for zero-mean vectors.

5.2.3 *Genotypic Similarity*

We define the genotypic similarity between two tree-based solution candidates using the Sørensen-Dice index

$$\text{GenotypicSimilarity}(T_1, T_2) = \frac{2 \cdot |M|}{|T_1| + |T_2|} \tag{5.1}$$

Here, M is the bottom-up mapping between T_1 and T_2 calculated using the algorithm described in [13]. We describe below the main steps of the algorithm:

1. Build a forest $F = T_1 \dot{\bigcup} T_2$ consisting of the disjoint union between the two trees
2. Map F to a directed acyclic graph G. Two nodes in F are mapped to the same vertex in G if they are at the same height in the tree and their children are mapped to the same sequence of vertices in G. The bottom-up traversal ensures that nodes are mapped before their parents, leading to $O(|T_1| + |T_2)|$ build time for G.
3. Use the map $K : F \to G$ obtained in the previous step to build the final mapping $M : T_1 \to T_2$. This step iterates over the nodes of T_1 in level order and uses K to determine which nodes correspond to the same vertices in G. Level-order iteration guarantees that every largest unmapped subtree of T_1 will be mapped to an isomorphic subtree of T_2.

The algorithm has a runtime complexity linear in the size of the trees regardless whether the trees are ordered or unordered.

5.2.4 *Clustering and Visualization*

One of the challenges in visualizing the search space using phenotypic or genotypic similarity measures is to find a mapping of expressions to a two-dimensional space which preserves pairwise similarities as well as possible.

We use the t-SNE algorithm [7] with the distance matrices which are calculated on the basis of the previously described phenotypic or genotypic similarity measures.

The main idea behind t-SNE is to map a high-dimensional space X to a low-dimensional space Y where the distribution of pairwise similarities is preserved as much as possible. Similarity between data points $x_i, x_j \in X$ is defined as the probability that x_i would pick x_j as its neighbor. A similar probability distribution is found in Y by minimizing the Kullback-Leibler divergence between the two distributions using gradient descent. While t-SNE does not preserve distances, the visualization can potentially provide new insight into the structure of the search space for symbolic regression.

In the following we describe the visualization and clustering procedure in more detail.

5.2.4.1 Clustering and Visualization Based on Genotypic Similarity

The base set of $\approx 1.6 \cdot 10^5$ unique expressions obtained via grammar enumeration is unfeasibly big for the calculation of the full similarity matrix. Therefore, we further reduce this set to a feasible quantity by filtering expressions based on their R^2 value in relation to the Keijzer-4 function [4]. Figure 5.5 shows the distribution of R^2 values over the full set of all unique expressions. For the genotypic mapping we took only expressions with $R^2 > 0.2$.

We used the HDBSCAN algorithm [2] for clustering. We tried two approaches: (1) clustering based directly on the pre-computed similarity matrix, and (2) clustering in the mapped 2-d space. Both approaches produced similar results.

5.2.4.2 Clustering and Visualization Based on Phenotypic Similarity

For the mapping based on phenotypic similarity we decided to use the complete set of the unique expressions and applied the *LargeVis* implementation [11] to produce the visualization. LargeVis relies on approximate nearest neighbor algorithms to

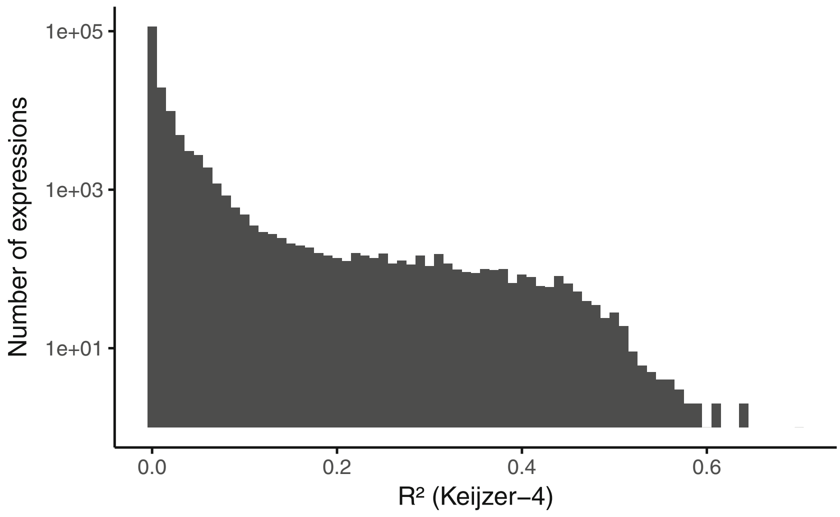


Fig. 5.5 Distribution of R^2 values for the Keijzer-4 function

make visualization of large-scale and high-dimensional datasets feasible. For our analysis we used the R library for LargeVis.[2] LargeVis implements a variant of t-SNE in which the exact determination of nearest neighbors is replaced by the approximate nearest neighbors list. This has only linear runtime complexity in the number of data points. As a consequence, the asymptotic runtime of clustering and the embedding becomes linear in the number of data points.

The algorithm works in three steps. First, a the lists of approximate nearest neighbors for each data point are determined using random projection trees [3]. In the second step, a sparse weighted edge matrix is calculated which encodes the nearest neighbor graph. Finally, the approximate nearest neighbor lists and the edge matrix can be used for t-SNE. LargeVis provides a variant of the HDBSCAN algorithm [8, 9] which uses the approximate nearest neighbor list.

5.2.5 Mapping GP Solution Candidates

Based on the results of the phenotypic clustering we study how GP explores the search space. For this we add a step in the GP algorithm after all individuals in the population have been evaluated. In this step, we identify the closest expression or cluster of expressions in the enumerated search space for each evaluated solution candidate. With this we are able to calculate a visitation density for the enumerated search space.

We expect that in the early stages of evolution GP explores many different areas of the search space, whereby over time GP should converge to the area of the search space which contains expressions which are most similar to the target function.

5.3 Results

In the following sections we first present our results of the clustering and visualization based on the phenotypic similarity and compare with the results of clustering and visualization based on the genotypic similarity. Then we present the results of our analysis of cluster qualities for five uni-variate functions. Finally, we present the results of the GP visitation frequency analysis.

[2]https://github.com/elbamos/largeVis.

5.3.1 Phenotypic Mapping

Figure 5.6 shows a result for the visualization and clustering based on phenotypic similarity where we have used LargeVis directly on all output vectors and using cosine-similarity. Each dot represents an expression. The color indicates to which cluster an expression has been assigned. The visualization clearly shows that several clusters of similar expressions can be identified in the search space.

As a post-processing step we prepared plots for all clusters which show the outputs of all expressions within the clusters. We found that the search space includes many rather complex functions and that several clusters of interesting and similar functions are identified. Some selected plots as well as the position of the cluster center on the mapped search space are shown in Fig. 5.6. It should be noted that the visualization does not show unique expressions which have been identified by HDBSCAN as outliers.

Figure 5.7 again shows a phenotypic map. The difference to Fig. 5.6 is that here all expressions are shown and we have used a coloring scheme based the similarity of expression outputs with the Keijzer-4 function (squared correlation R^2). The visualization clearly shows that only certain areas of the search space contain expressions which are similar to the target function. Notably, there are several areas which contain expressions with large R^2 values. There are at least two potential explanations for this. First, it could be an artifact of the approximation of t-SNE. Another reason could be the fact that we have used cosine similarity for

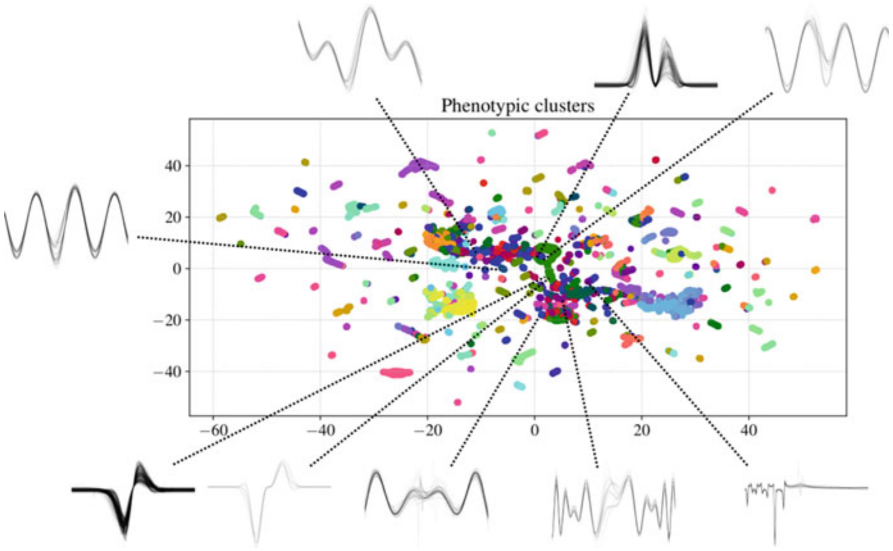


Fig. 5.6 Visualization of the embedding and clustering result based on phenotypic similarity. The phenotypic mapping leads to several clearly defined clusters

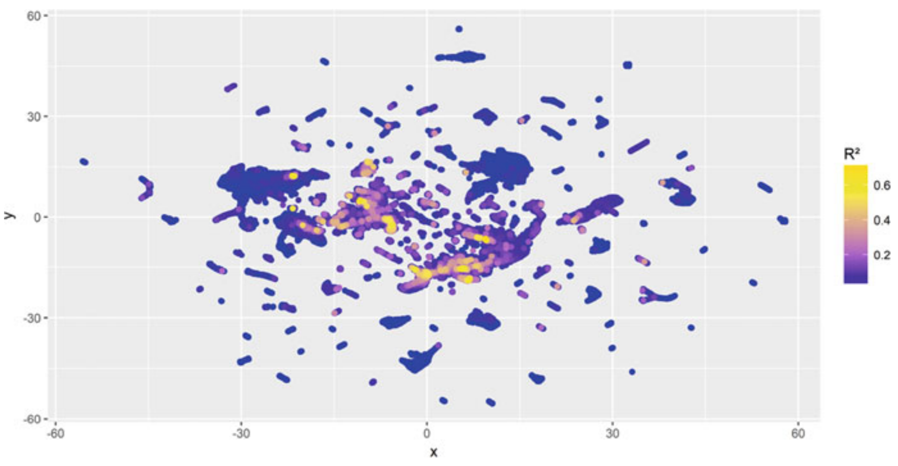


Fig. 5.7 All expressions in the mapped search space. We use squared correlation with the Keijzer-4 function for the coloring

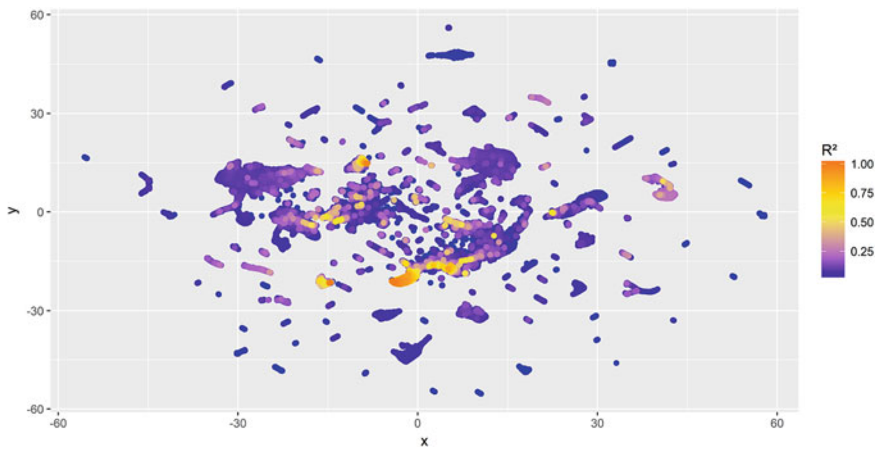


Fig. 5.8 All expressions in the mapped search space colored based on the squared correlation with the Pagie-1d function

the embedding and the R^2 value for the coloring scheme. With the cosine similarity measure, two vectors that are negatively correlated are dissimilar.

Figure 5.8 shows the same visualization for a different function (Pagie-1d). Compared to Fig. 5.7 other areas of the search space are highlighted.

The results produced for the phenotypic mapping are motivating for further research. At least for the two considered examples we should be able to use a hill-climbing algorithm on the mapped search space to find well-fitting expressions. The mapping of the search space must be pre-computed only once and can be reused for potentially any target function.

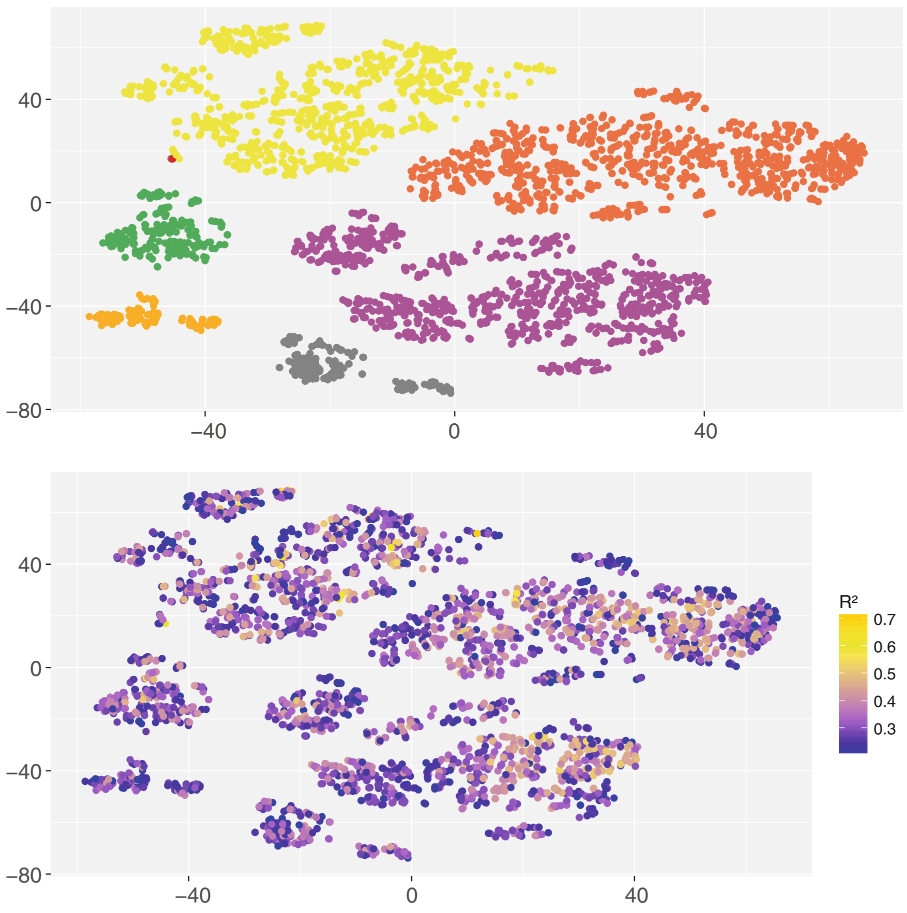


Fig. 5.9 Results of HDBSCAN and t-SNE with the genotypic similarity (top: coloring based on clusters; bottom: coloring based on the R^2 value with the Keijzer-4 function). The visualization of the two-dimensional mapping shows clusters of genotypically similar solutions. However, the genotypic clusters do not correlate strongly with qualities

5.3.2 Genotypic Mapping

Figure 5.9 shows the results of t-SNE for dimensionality reduction and HDBSCAN for clustering when we use the genotypic similarity (see Sect. 5.2.3). We used HDBSCAN on the t-SNE mapped points. In comparison to the phenotypic mapping of the search space, the genotypic mapping does not produce such clearly defined clusters. In particular, in the lower sub-plot of Fig. 5.9 no strong correlation with the quality values is visible.

5.3.3 Cluster Qualities for Benchmark Problems

To check whether the phenotypic clustering of our search space is generalizable over multiple problem instances, we use the following five uni-variate benchmark functions:

- Keijzer-4 [4]: $f(x) = x^3 e^{-x} \cos(x) \sin(x)(\sin(x)^2 \cos(x) - 1)$ for $x \in [0, 10]$
- Keijzer-9 [4]: $f(x) = \ln(x + \sqrt{x^2 + 1})$ for $x \in [0, 100]$
- Pagie-1d [10][3]: $f(x) = \frac{1}{1+x^{-4}}$ for $x \in [-5, 5]$
- Nguyen-5 [12]: $f(x) = \sin(x^2) \cos(x) - 1$ for $x \in [-1, 1]$
- Nguyen-6 [12]: $f(x) = \sin(x) + \sin(x + x^2)$ for $x \in [-1, 1]$

We average the resulting R^2 values within each cluster and rank the clusters by the average R^2 for each benchmark function independently. Figure 5.10 shows the average R^2 values over cluster ranks for the five benchmark functions. We find that for each of the benchmark functions a cluster with well-fitting expressions is be found. Figure 5.11 shows plots of all functions in the best four clusters for each of the considered benchmark functions. The plots show that for the Keijzer-4 and the Pagie-1d functions the search space contains well-fitting expressions. However, for the other three functions the best clusters do not fit the target function well. This is in contrast to the calculated average R^2 values for the clusters which are relatively high (≥ 0.95) for all functions except for Keijzer-4. Here we want to stress again, that the search space clustering has been calculated independently from the target functions.

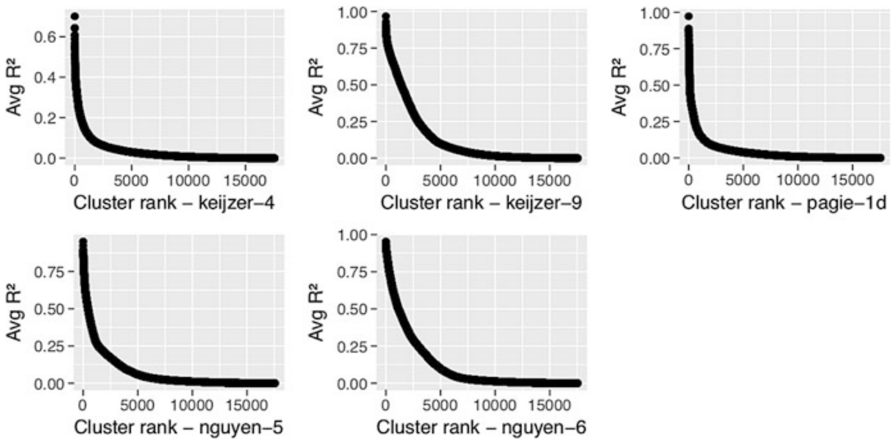


Fig. 5.10 Ranking of clusters by average R^2 values of expressions within all clusters. For each of the benchmark functions there are clusters which contain well-fitting expressions

[3]We have used a uni-variate variant of the benchmark function described by Pagie and Hogeweg.

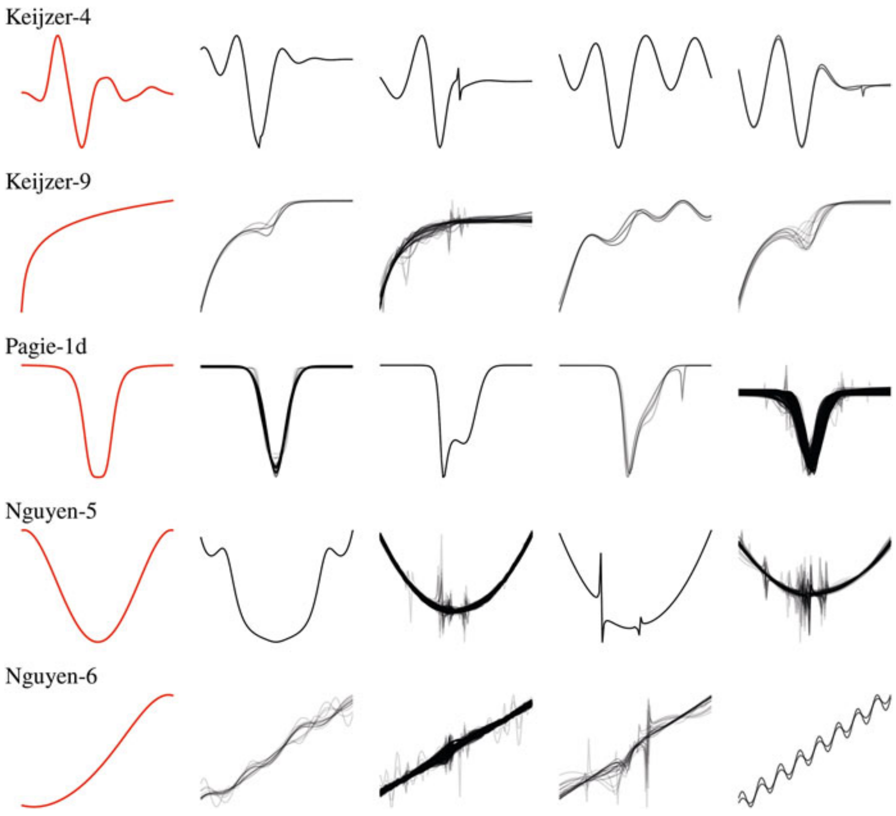


Fig. 5.11 The five benchmark functions and the best four clusters with the highest average R^2 for each of the functions from left to right. The average R^2 values are: 0.70 (Keijzer-4), 0.97 (Keijzer-9), 0.97 (Pagie-1d), 0.95 (Nguyen-5), and 0.95 (Nguyen-6)

5.3.4 Mapping of GP Solution Candidates

We use the 2d-mapping of the search space to analyze the search bias of GP. For this we use a rather canonical GP implementation and map each evaluated solution candidate to the search space by identifying the closest representative expression in the enumerated search space. Each expression in the search space is assigned to a cluster. Therefore, we can determine which clusters of the search space are visited by GP. We have again used the Keijzer-4 function to demonstrate the concept.

Figure 5.12 shows the results of the analysis. In the left sub-plot the number of different clusters visited by GP are shown over generations; on the right hand side the median cluster rank (ordered by cluster quality) is shown. This clearly shows that in the beginning GP visits many different solution candidates and later concentrates on only a view high quality clusters.

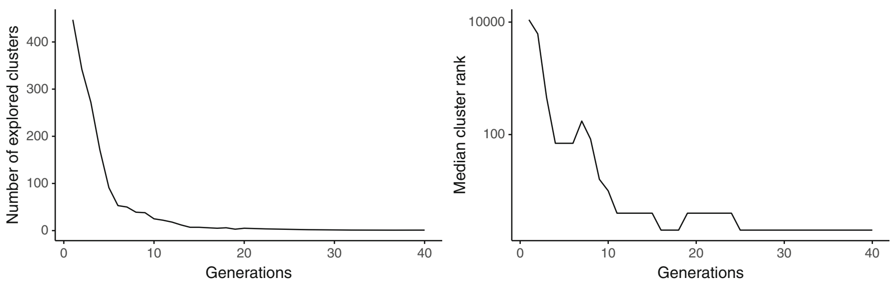


Fig. 5.12 The number of clusters that are visited by GP as well as the median rank (lower rank in better quality) of clusters over GP generations

In Fig. 5.13 we show the distribution of solution candidates visited by GP in more detail for the first few generations (1, 2, 3, 4, 5, and 10). It is clearly visible that within the first ten generations GP explores almost each of the 16,000 clusters (population size=500 and PTC2 tree creator [6]) and quickly finds the clusters with highest quality.

5.4 Discussion

Our analysis so far has some limitations that merit a more detailed discussion.

- We have only looked at uni-variate models.
- The grammar is very restricted.
- Even with the limit of seven variable references, the computational effort is already rather high. When increasing the length limits the computational effort quickly becomes too big.
- The phenotypic clustering depends on the range of input values that are used when evaluating the model.
- We have completely ignored the effect of numeric constants in models, i.e. we have not used numeric constants.

In this work we have considered a search space of approximately 160,000 semantically different expressions. However, in Fig. 5.11 we see that the search space does not contain well-fitting expressions for all of the considered target functions. For a more in-depth analysis the size of the search space should be extended even for uni-variate problems.

We also found that the grammar we used produced many discontinuous functions (e.g. because of division by zero or extreme arguments to $\exp(x)$). If we assume that we are not interested in such functions then the search space could potentially be reduced massively by removing expressions leading to discontinuous functions.

If we extend the analysis to include multi-variate models, the size of the search space would increase significantly even if we use the same size restrictions. This is

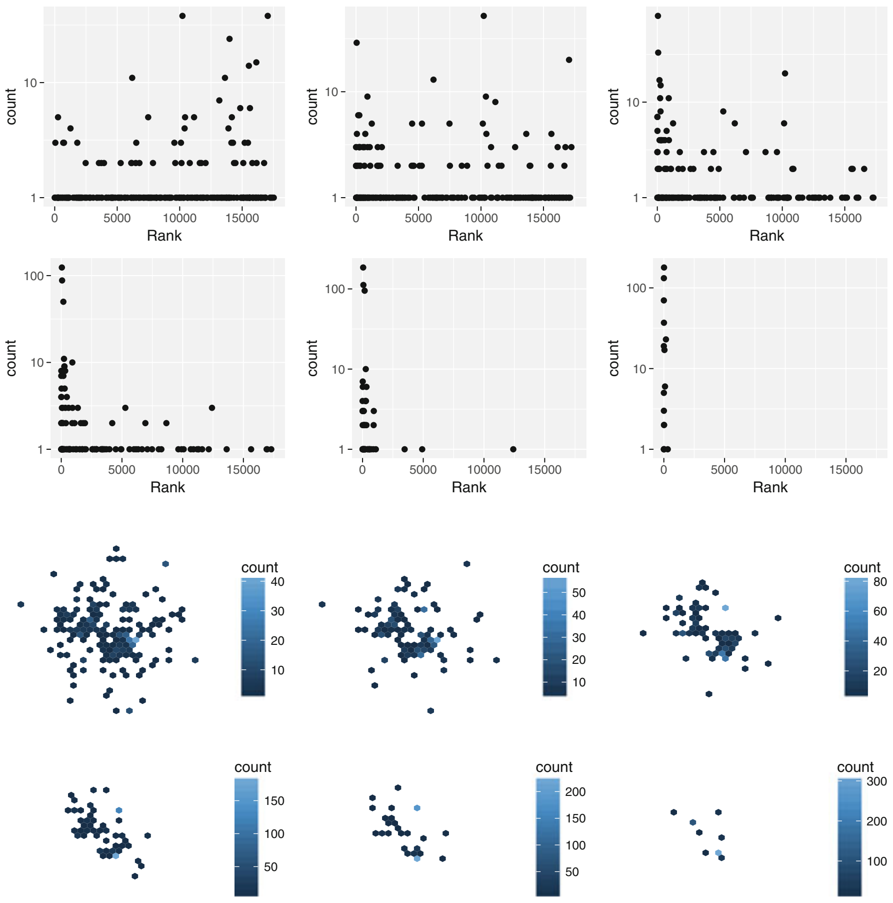


Fig. 5.13 More detailed visualization of the search space explored by a GP run. The plots in the first and second rows show the visitation frequency for each cluster in generations 1, 2, 3, 4, 5, 10. Clusters with lower ranks are high quality clusters. The plots in the third and fourth row show the visitation frequency in the two-dimensional phenotypic map. Within the first 10 generations GP converges to a focused area of the search space

simply a consequence of the fact that there are more different models that can be expressed. Based on preliminary experiments, even with two or three independent variables it is possible to enumerate the search space with the same size restrictions as we used above. For more than three variables it would be necessary to use even smaller size limits.

We also need to consider that the variety of function outputs becomes much larger with increased dimensionality which could lead to a larger set of clusters.

We hypothesize that for practical problems it is usually sufficient to be able to represent two- or three-way interactions of variables as separable terms can be modeled independently. However, we cannot expect this in general.

Regarding the complexity of the grammar, we have purposefully limited the number of alternatives to be able to enumerate the full search space. It would however be rather easy to add more functions (e.g. a power or root function) as long as the complexity of the argument to the added functions is limited similarly as we have limited arguments for $\log(x)$.

A different approach that could potentially be worthwhile is to calculate the phenotypic similarity in the frequency-domain of the function.

5.5 Conclusion

This contribution aims to analyze the search behavior of GP in a space of hypotheses that is visualized as a 2D mapping of the search space. The idea of this approach is to enumerate the complete search space in the off-line phase independently of the regression problem to be solved. In order to do so we had to define some restrictions like the consideration of only uni-variate functions under restricting grammar assumptions. The still huge search space can be further restricted by filtering unique expressions by hashing.

The ideas presented in this chapter have to be considered as very first findings of a novel approach having in mind that the restrictions on the search space delimit the generality of claimed findings. In order to achieve a deeper and more universal understanding it will be necessary to extend the approach to multi-variate models. Also it will be interesting to analyze the search behavior of different flavors of GP and other hypothesis search techniques.

Furthermore, the massive set of off-line generated models could be used to filter an initial population for a GP run as soon as the regression problem to be tackled is available. Similar to what is done in the ANN community recently with pre-trained neural networks we could establish somehow pre-evolved initial populations for evolutionary search: In this way we could for example filter a genotypically diverse subset of models for a GP problem.

Acknowledgements The authors thank the participants of the Genetic Programming in Theory and Practice (GPTP XVI) workshop for their valuable feedback and ideas which helped to improve the work described in this chapter. The authors gratefully acknowledge support by the Christian Doppler Research Association and the Federal Ministry for Digital and Economic Affairs within the *Josef Ressel Center for Symbolic Regression*.

References

1. Burke, E.K., Gustafson, S., Kendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation **8**(1), 47–62 (2004). https://doi.org/10.1109/TEVC.2003.819263

2. Campello, R.J.G.B., Moulavi, D., Sander, J.: Density-based clustering based on hierarchical density estimates. In: J. Pei, V.S. Tseng, L. Cao, H. Motoda, G. Xu (eds.) Advances in Knowledge Discovery and Data Mining, pp. 160–172. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
3. Dasgupta, S., Freund, Y.: Random projection trees and low dimensional manifolds. In: Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC'08), pp. 537–546. ACM (2008)
4. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: European Conference on Genetic Programming, pp. 70–82. Springer (2003)
5. Kommenda, M., Kronberger, G., Winkler, S., Affenzeller, M., Wagner, S.: Effects of constant optimization by nonlinear least squares minimization in symbolic regression. In: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 1121–1128. ACM (2013)
6. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation **4**(3), 274–283 (2000)
7. Maaten, L.v.d., Hinton, G.: Visualizing data using t-SNE. Journal of Machine Learning Research **9**(Nov), 2579–2605 (2008)
8. McInnes, L., Healy, J.: Accelerated hierarchical density based clustering. In: 2017 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 33–42 (2017). https://doi.org/10.1109/ICDMW.2017.12
9. McInnes, L., Healy, J., Astels, S.: hdbscan: Hierarchical density based clustering. The Journal of Open Source Software **2**(11) (2017). https://doi.org/10.21105/joss.00205
10. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. Evolutionary Computation **5**(4), 401–418 (1997)
11. Tang, J., Liu, J., Zhang, M., Mei, Q.: Visualizing large-scale and high-dimensional data. In: Proceedings of the 25th International Conference on World Wide Web, WWW '16, pp. 287–297. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2016). https://doi.org/10.1145/2872427.2883041
12. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines **12**(2), 91–119 (2011)
13. Valiente, G.: An efficient bottom-up distance between trees. In: Proc. 8th Int. Symposium on String Processing and Information Retrieval, pp. 212–219. IEEE Computer Science Press (2001)
14. Worm, T., Chiu, K.: Prioritized grammar enumeration: symbolic regression by dynamic programming. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, pp. 1021–1028. ACM (2013)

Chapter 6
What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP

Alexander Lalejini and Charles Ofria

6.1 Introduction

In Genetic Programming Theory and Practice IX, [20] explored the use of tag-based naming in evolving modular programs. In this chapter, we continue exploring tag-based naming with SignalGP [14]; we investigate the importance of inexactness when making tag-based references: How important is imprecision when calling an evolvable name? Additionally, we discuss possible broadened applications of tag-based naming in the context of SignalGP.

What's in an evolved name? How should modules (e.g., functions, sub-routines, data-objects, etc.) be referenced in evolving programs? In traditional software development, the programmer hand-labels modules and subsequently refers to them using their assigned label. This technique for referencing modules is intentionally rigid, requiring programmers to precisely name the module they aim to reference; imprecision often results in syntactic incorrectness. Requiring evolving programs to follow traditional approaches to module referencing is not ideal: mutation operators must do extra work to guarantee label-correctness, else mutated programs are likely to make use of undefined labels, resulting in syntactic invalidity [20]. Instead, is genetic programming (GP) better off relying on more flexible, less exacting referencing schemes?

Inspired by John Holland's use of "tags" [8–11] as a mechanism for matching, binding, and aggregation, Spector et al. [20–22] introduced and demonstrated a tag-based naming scheme for GP where tags are used to name and reference

A. Lalejini (✉) · C. Ofria
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA
e-mail: lalejini@msu.edu; ofria@msu.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*, Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_6

program modules. Tags are evolvable labels that are mutable, and the similarity (or dissimilarity) between any two possible tags is quantifiable. Tags allow for *inexact* referencing. Because the similarity between tags can be calculated, a referring tag can always link to the program module with the *most similar* tag; further, this ensures that all possible tags are valid references. Because tags are mutable, evolution can incrementally shape tag-based references within evolving code. Spector et al. demonstrated the value of an evolvable name, showing that the tag-based naming scheme supports the evolution of programs with complex, modular architectures by allowing programs to more easily reference and make use of program modules [20].

We previously extended Spector et al.'s tag-based naming scheme, broadening the application of tags to develop SignalGP [14], a GP technique designed to provide direct access to the event-driven programming paradigm. In Spector et al.'s original implementation [22], tags were used as an evolvable mechanism to label and later refer to code fragments. At there core, tags provide general-purpose, evolvable specificity—an evolvable way to specify zero or more tagged entities. SignalGP broadens the application of tags, using them to specify the relationships between events and event handlers (i.e., program modules that process events). However, the application of tag-based naming can be *further* broadened. For example, tag-based naming could be used to label and refer to particular instructions, other agents, or other virtual hardware components (e.g., registers, locations in memory, etc.). In this broader context, tags are still mutable labels with well-defined tag-tag similarity measures, allowing for inexact referencing. The context of a referring tag can limit the valid set of tagged entities with which it can match. For example, in the context of a function call, a referring tag might only match to function tags, whereas in the context of a memory access, a referring tag might only match to a tagged location in memory.

In this chapter, we investigate the importance of inexactness when making tag-based references, and we propose possible extensions to SignalGP that use broader applications of tag-based naming. In Sect. 6.2, we give a brief overview of SignalGP. In Sect. 6.3, we use an environment coordination toy problem to investigate the effectiveness of different thresholds of allowed imprecision when performing tag-based referencing. We compare the fitness effects of requiring different levels of tag similarity when matching referring tags to referents, ranging from requiring exact matches between tags for a successful reference to placing no restrictions on tag similarity for a successful reference. We find that, indeed, allowing for some inexactness when performing tag-based referencing is crucial. In Sect. 6.4, we demonstrate that requiring a minimum threshold of similarity for tags to match is important when programs must evolve to ignore irrelevant or misleading environmental signals. In addition to providing access to the event-driven programming paradigm, the way SignalGP programs are organized is well-suited for several interesting extensions. In Sect. 6.5, we speculate on several possibilities for how SignalGP can be extended to support module regulation, multi-representation programs, and major transitions in individuality.

6.2 SignalGP

SignalGP defines a way of organizing and interpreting genetic programs to provide computational evolution direct access to the event-driven programming paradigm. The event-driven programming paradigm is a software-design philosophy where software development focuses on the processing of events (often in the form of messages from other processes, sensor alerts, or user actions) [2, 4, 5]. Events are processed by segments of code called event handlers. In traditional event-driven programming, some identifying characteristic associated with the event (e.g., its name or type) determines the most appropriate event handler to trigger for processing the event, and the programmer is responsible for labeling event handlers such that they process the appropriate types of events. Software development environments that support the event-driven paradigm often abstract away the logistics of monitoring for events and triggering event handlers. This technique simplifies the code that must be designed and implemented by the programmer in domains that require on-the-fly reactions to signals from the environment or other agents.

SignalGP provides similarly useful abstractions to *evolving* programs. In SignalGP, signals (events) trigger the execution of program modules (functions) to respond to those signals. SignalGP applies tag-based referencing techniques to specify which function is triggered by each signal, allowing the relationships between signals and functions to evolve over time.

Here, we give a general overview of SignalGP in the context of linear GP, wherein programs are represented as sequences of instructions; however, the underlying organization and interpretation of SignalGP programs is generalizable across a variety of evolvable representations of computation (see Sect. 6.5.2). Figure 6.1 is provided to visually guide our discussion of SignalGP. A more detailed discussion can be found in [14].

SignalGP programs (agents) are explicitly modular, composed of a set of functions, each of which associates a tag with a linear sequence of instructions.

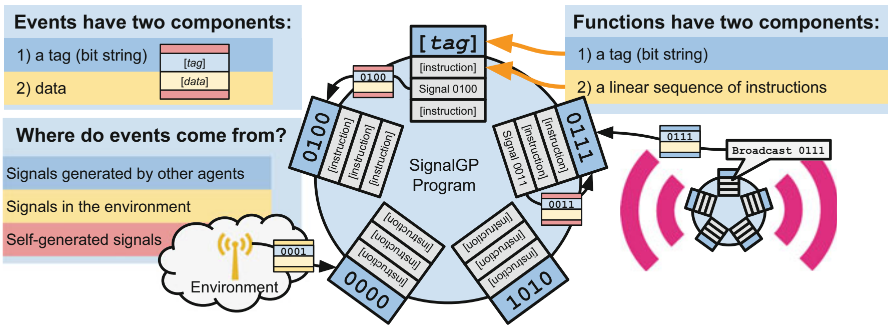


Fig. 6.1 A high-level overview of SignalGP. Programs are defined by a set of functions. Events trigger functions with the most similar tag, allowing programs to respond to signals. SignalGP agents handle many signals simultaneously by processing them in parallel

SignalGP makes explicit the concept of events. Each event is associated with a tag (indicating the event type) as well as additional event-specific data. In our work, we represent tags as fixed-length bit strings where tag similarity is quantified as the proportion of matching bits between two tags (simple matching coefficient). Because both events and functions are tagged, SignalGP uses tag-based referencing to determine the most appropriate function to process an event: events trigger the function with the closest matching tag as long as its within a fixed threshold. When an event triggers a function, the function is run with the event's associated data as input. In this way, functions act as event handlers, and tag-based referencing is used as an evolvable mechanism to determine the most appropriate function to trigger in response to an event. SignalGP agents handle many events simultaneously by processing them in parallel. Events may be generated internally, by the environment, or by other agents, making SignalGP particularly well-suited for domains that require programs to respond quickly to their environment or other agents.

The underlying instruction set is crafted to allow programs to easily trigger internal events, broadcast external events, and to otherwise work in a tag-based context. In our implementation of SignalGP, instructions are argument based, and as in traditional linear GP representations, arguments modify the effect of an instruction, often specifying memory locations or fixed values. In addition to evolvable arguments, each instruction has an evolvable tag, which may also modify the effect of an instruction. For example, instructions that refer to functions do so using tag-based referencing, and when an instruction generates an event (e.g., to be used internally or broadcast to other agents), the instruction's tag is used as the event's tag. The set of SignalGP instructions used in this work are documented in our supplemental material, which can be accessed via GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP [13].

In Spector et al.'s original conception of tag-based referencing, as long as a program had at least one tagged module, all referential tags could successfully reference *something* [22]. The tag-based referencing employed by SignalGP, however, can be configured to only match tags whose similarity exceeds a threshold, allowing programs to ignore events by avoiding the use of similar tags. This similarity threshold allows us to adjust the degree of exactness required for tag-based references to succeed.

6.3 The Value of Imprecision in Evolvable Names

How important is imprecision when calling an evolvable name? Tag-based referencing has built-in flexibility, not requiring tags to *exactly* match to successfully reference one another. In Spector et al.'s initial implementation of tag-based referencing [22], referring tags *always* matched to the most similar receptor tag. Spector et al. speculated that tag-based referencing performed well because of this inexactness: any tag-based reference is able to find a referent as long as one exists. We can, however, imagine different degrees of allowed imprecision when performing tag-based referencing, ranging from only identical tags being allowed to reference one another, to any two tags being allowed to match as long as they are the most similar pair. Indeed, any minimal level of tag-similarity for successful referencing can be imposed (e.g., requiring tags to be at least 50% similar before they can be considered as the best match).

Here, we explore the importance of imprecision in tag-based referencing using SignalGP. We evolve SignalGP agents to solve an environment coordination problem under a range of similarity thresholds, spanning from 0% (no similarity requirement) to 100% (requiring perfect matches).

6.3.1 The Changing Environment Problem

The changing environment problem is a toy problem that we designed to test GP programs' capacity to respond appropriately to environmental signals. We have previously used this problem to demonstrate the value of the event-driven paradigm using SignalGP [14].

The changing environment problem requires agents to continually match their internal state with the current state of a stochastically changing environment. The environment is initialized to a random state, and at every subsequent time step, the environment has a 12.5% chance of randomly changing to any of 16 possible states. To be successful, agents must monitor the environment for changes, adjusting their internal state as appropriate.

Environmental changes produce signals (events) with environment-specific tags that will trigger an appropriate SignalGP function; in this way, SignalGP agents can respond to environmental changes. Each of the 16 environment states is associated with a distinct tag that is randomly generated at the beginning of a run. Agents adjust their internal state by executing one of 16 state-altering instructions (one for each possible environmental state). Thus, the optimal solution to this problem is a 16-function program where each function is triggered by a different environment signal, and functions, when triggered, adjust the agent's internal state appropriately. An example solution to the changing environment problem is documented in our supplemental material, which can be accessed via GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP [13].

To explore the value of imprecision in tag-based referencing, we evolved 30 replicate populations of SignalGP agents under nine treatments, each requiring a different similarity threshold for events to trigger functions: 0%, 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, 87.5%, and 100%. Note that when performing a tag-based reference, if the closest matching tag is not greater than or equal to the required similarity threshold, the reference fails.

6.3.1.1 Hypothesis

A 100% similarity threshold is equivalent to exact-name referencing; thus, we expected it to perform poorly. A 0% similarity threshold is equivalent to what [22] used in their original demonstration of tag-based referencing; thus, we expected to it perform well. However, are intermediate thresholds just as effective? They provide varying degrees of allowed imprecisions while allowing programs to passively ignore some incoming signals. In prior work using SignalGP [14], a 50% similarity threshold performed well on the changing environment problem; thus, we expected treatments with intermediate thresholds to perform better than runs requiring exact tag-matching for references to succeed.

6.3.1.2 Experimental Parameters

For each treatment, we evolved 30 replicate populations of 1000 agents for 10,000 generations, starting from a simple ancestor program consisting of a single function with eight no-operation instructions. We initialized all replicates with a unique random number seed. Each generation, we evaluated all agents in the population three times (three trials). Each trial was composed of 256 time steps, and an agent's score for a single trial was equal to the number of time steps the agent's internal state matched the environment state. Thus, possible scores ranged from 0 to 256. An agent's fitness was the minimum score achieved after three trials, thus selecting agents that performed consistently. We used a combination of elite and tournament (size four) selection to determine which agents reproduced asexually each generation.

Offspring were mutated using SignalGP-aware mutation operators. We used whole-function duplication and deletion operators, applied at a per-function rate of 0.05; these operators allowed evolution to tune the number of functions in a SignalGP program. We mutated instruction- and function-tags at a per-bit mutation rate of 0.005. We applied instruction and argument substitutions at a per-instruction/argument rate of 0.005. We applied single-instruction insertion and deletion operators at a per-instruction rate of 0.005; when a single-instruction insertion occurred, we inserted a random instruction with random arguments and a random tag. In addition to single-instruction insertions and deletions, instruction sequences could be inserted or removed via slip-mutation operators [15]. When triggered, slip-mutations can either duplicate or delete multi-instruction sequences within a function. We applied slip-mutations at a per-function rate of 0.05.

Agents were limited to a maximum of 16 total functions, each of which were limited to a maximum length of 32 instructions. Agents were limited to a maximum of 32 parallel-executing threads. Agents were further limited to 128 call states per call stack. All tags were represented as length-16 bit strings.

6.3.1.3 Data Analysis

We analyzed evolving populations at two time points during the evolutionary process: generation 1000 and generation 10,000. For every population analyzed, we extracted the best-performing program and evaluated it 100 times (to account for environmental stochasticity), using its average performance as its representative fitness. For each time point (generation 1000 and 10,000) analyzed, we compared the performances of evolved programs across treatments. To determine if any of the treatments were significant ($p < 0.05$) within a set, we performed a Kruskal-Wallis test. For a time point in which the Kruskal-Wallis test was significant, we performed a post-hoc pairwise Wilcoxon rank-sum test, applying a Bonferroni correction for multiple comparisons. All statistical analyses were conducted in R 3.3.2 [18].

All visualizations of our results were generated using the seaborn Python library [23]. The code to run our experiments, perform statistical analyses, and generate visualizations is publicly available on Github [13].

6.3.2 Results and Discussion

Figure 6.2 gives the results for the changing environment problem early during our experiment (generation 1000) and at the end of our experiment (generation 10,000). At both generation 1000 and generation 10,000, programs evolved under different similarity thresholds had significantly different performance (Gen. 1000: Kruskal-Wallis test, Chi-squared = 161.27, $p < 2.2\text{e}{-16}$; Gen. 10,000: Kruskal-Wallis test, Chi-squared = 221.72, $p < 2.2\text{e}{-16}$). Table 6.1 gives the results of a post-hoc pairwise Wilcoxon rank-sum test for our results at both generation 1000 and generation 10,000.

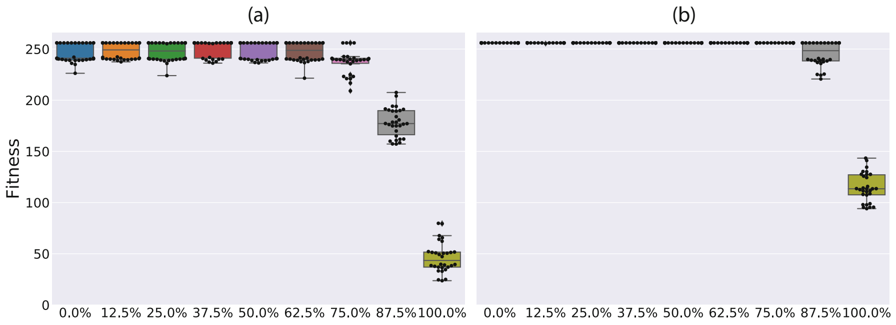


Fig. 6.2 Changing environment problem results at: (**a**) generation 1000 and (**b**) generation 10,000. The box plots indicate the fitnesses (each an average over 100 trials) of the best performing programs from each replicate across a range of minimum similarity thresholds

Table 6.1 Pairwise Wilcoxon rank-sum test results for the changing environment problem at generation 1000 and generation 10,000

		Tag Similarity Threshold									
		0.0%	12.5%	25%	37.5%	50.0%	62.5%	75.0%	87.5%	100.0%	
Tag Similarity Threshold	0.0%	■	1.0	NONE	NONE	NONE	NONE	NONE	**0.00027**	**2.5e-11**	Generation 10,000
	12.5%	1.0	■	1.0	1.0	1.0	1.0	1.0	**0.00079**	**3.6e-11**	
	25%	1.0	1.0	■	NONE	NONE	NONE	NONE	**0.00027**	**2.5e-11**	
	37.5%	1.0	1.0	1.0	■	NONE	NONE	NONE	**0.00027**	**2.5e-11**	
	50.0%	1.0	1.0	1.0	1.0	■	NONE	NONE	**0.00027**	**2.5e-11**	
	62.5%	1.0	1.0	1.0	1.0	1.0	■	NONE	**0.00027**	**2.5e-11**	
	75.0%	0.3559	**0.0109**	0.0956	**0.0013**	0.0654	0.2005	■	**0.00027**	**2.5e-11**	
	87.5%	**8.6e-10**	**7.6e-10**	**8.1e-10**	**5.2e-10**	**7.0e-10**	**7.6e-10**	**1.1e-09**	■	**4.4e-10**	
	100.0%	**8.6e-10**	**7.6e-10**	**8.1e-10**	**5.2e-10**	**7.0e-10**	**7.6e-10**	**1.1e-09**	**1.1e-09**	■	
		Generation 1,000									

Each row/column corresponds to a tag similarity threshold treatment. Each entry in the table indicates the Bonferroni-adjusted p value for a given comparison between two treatments. Statistically significant relationships ($p < 0.05$) are bolded. 'NONE' indicates that two treatments have identical distributions of data. Results for generation 1000 are in red, below the table's diagonal (in black). Results for generation 10,000 are in yellow, above the table's diagonal

At After 10,000 generations of evolution, programs evolved in the 87.5% and 100.0% similarity threshold treatments still perform significantly worse than those evolved in treatments with lower similarity thresholds. However, by 10,000 generations, some replicates evolved under the 87.5% similarity threshold treatment were able to produce optimal programs. No optimal programs evolved in the 100.0% similarity threshold treatment (exact name matching). Fully detailed statistical results can be found in our supplemental material [13].

Allowing for *Some* Imprecision is Crucial When Calling a Tag-Based Name
Because we limited agents to a maximum of 16 total functions, optimally solving the 16-state changing environment problem required programs to dedicate each of their 16 possible functions to responding to a particular environment state. Each function must be tagged such that only a single environment state change could trigger it, and when triggered, the function must immediately update the agent's internal state appropriately. If exact tag-matching (100% similarity threshold) is required for events to trigger functions, each function's tag must evolve to match a single environment state tag bit-for-bit. As expected, our results demonstrate that requiring tags to exactly match for successful references impedes evolution: after 10,000 generations, no optimal programs evolved under the 100.0% similarity threshold treatment.

In treatments that allow for inexactness when performing tag-based referencing, each function's tag must evolve to closely match (above a given similarity threshold) a single environment state; higher minimum required similarity thresholds require evolution to more precisely tune function tags. As demonstrated by the 87.5% similarity threshold treatment, requiring exceedingly high levels of precision can impede evolutionary adaptation.

By 10,000 generations, there was no significant difference in program performance among all treatments with similarity thresholds lower than 87.5%. While allowing for inexactness in tag-based referencing is crucial for evolving programs to solve the changing environment problem, intermediate levels of required precision (12.5%, 25.0%, 37.5%, 50.0%, 62.5%, and 75.0% similarity thresholds) proved just as effective as not imposing any tag similarity constraints (0.0% similarity threshold).

6.3.2.1 Illuminating Solution Space with MAP-Elites

We use the MAP-Elites [17] evolutionary algorithm to further illuminate the importance of inexactness when using tag-based naming schemes. In MAP-Elites, a population is structured based on a set of chosen traits of evolving solutions. Each chosen trait defines an axis on a grid of cells where each cell represents a distinct combination of the chosen traits; further, each cell maintains only the most fit (elite) solution discovered with the cell's associated combination of traits. A MAP-Elites grid is initialized by randomly generating solutions and placing them into their appropriate cell in the grid (based on the random solution's traits). After initialization, occupied cells are randomly selected to reproduce. When a solution is selected for reproduction, we generate a mutated offspring and determine where that offspring belongs in the grid. If the cell is unoccupied, the new solution is placed in that cell; otherwise, we compare the new solution's fitness to the current occupant, keeping the fitter of the two. Over time, this process produces a grid of prospective solutions that span the range of traits we used to define our grid axes.

Dolson et al. extended the use of MAP-Elites to examine GP representations [3]. By selecing MAP-Elites grid axes that correspond to program architecture, we can get a snapshot of what types of programs are capable of succeeding at a task and what tradeoffs might exist between the chosen traits. We use this approach to explore the role of inexactness in SignalGP: we apply the MAP-Elites algorithm to the changing environment problem, using minimum similarity threshold for tag-based referencing and the number of unique functions used by a program during evaluation to define our MAP-Elites grid axes. In our more traditional evolution experiment, we locked in the minimum required similarity threshold for each treatment. In our MAP-Elites analysis, we allow the minimum similarity threshold for a program to evolve between 0.0% and 100.0%. Further, we increased the allowed number of functions in a program from 16 to 32.

We initialized our MAP-Elites grid with 1000 randomly generated SignalGP programs. We ran the MAP-Elites algorithm for 100,000 generations where each generation represents 1000 reproduction events. We ran 50 replicate MAP-Elites runs, giving us 50 grids of diverse solutions for the changing environment problem. At the end of each run, we filtered out any program unable to solve the problem perfectly in each of our 50 runs. The heat map in Fig. 6.3 shows the density of optimal programs (aggregated across runs) within our chosen trait space.

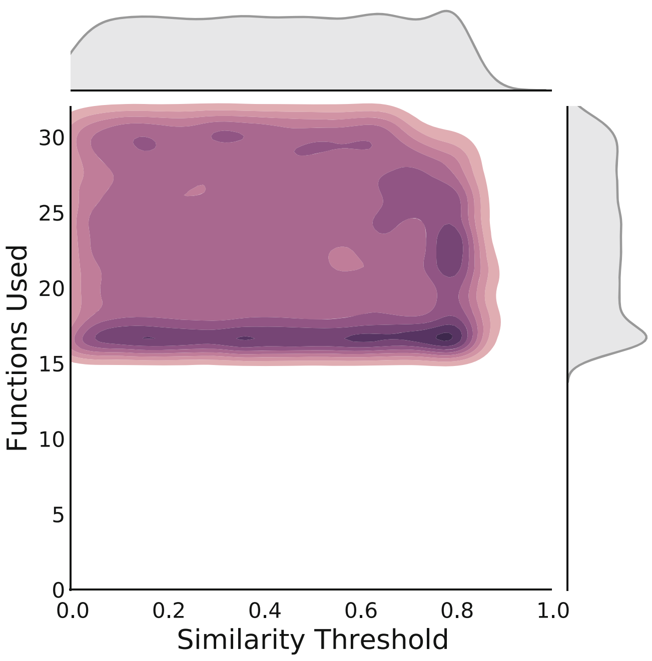


Fig. 6.3 Heat map of SignalGP programs evolved to solve the changing environment problem using MAP-Elites. Locations in the heat map correspond to distinct combinations of the following two program traits: the number of unique functions used by a program during evaluation and the program's minimum tag similarity threshold. Darker areas of the heat map indicate a higher density of perfect solutions found with a particular trait combination

From Fig. 6.3, we can see that all optimal programs use 16 or more functions. This is not surprising, as the changing environment problem cannot be optimally solved with fewer than 16 functions. The highest similarity threshold among all evolved solutions represented in Fig. 6.3 was 87.4657%, supporting the idea that requiring too much precision when performing tag-based referencing can impede evolution.

6.4 The Value of Not Listening

What's the value of ignoring signals in the environment? In some problem domains, the capacity to completely ignore distracting, irrelevant, or misleading signals while monitoring for others is crucial. For example, selective attention at a crowded restaurant allows us to ignore background noise and pay attention to a single conversion.

Here, we incorporate misleading distraction signals into the changing environment problem to demonstrate the value of ignoring signals in the context of SignalGP. In SignalGP, a 0% similarity threshold for tag-based references prevents agents from passively ignoring signals (events) in the environment. SignalGP programs can still be organized to *actively* ignore signals by having appropriately tagged, ineffectual functions to consume signals or by filtering signals based on event-specific data. In SignalGP, a 100% similarity threshold for tag-based references causes SignalGP programs to ignore any event whose tag is not an exact match with one of the agent's function tags, which we have shown to impede evolution (Sect. 6.3). Intermediate similarity thresholds, however, allow SignalGP agents to passively ignore signals in the environment without impeding adaptive evolution. We explore the value of allowing varying degrees of passive signal-discrimination via different similarity thresholds in SignalGP using the distracting environment problem.

6.4.1 The Distracting Environment Problem

The distracting environment problem is identical to the changing environment problem (described in Sect. 6.3.1) but with the addition of randomly occurring distraction signals. Like the changing environment problem, the environment can be in one of 16 states at any time with a 12.5% chance to change each update. Every time step there is also a 12.5% chance of a distraction event occurring, independent of environmental changes. Just as we randomly generate 16 distinct tags associated with each of the 16 environment states, we also generate 16 distinct distraction signal tags, which are guaranteed to not be identical to environment-state tags. Thus, to be successful, agents must monitor the environment for changes (adjusting their internal state as appropriate) while ignoring misleading distraction signals.

We repeated the experiment described in Sect. 6.3 with identical experimental treatments and parameters, but in the context of the distracting environment problem instead of the changing environment problem.

6.4.1.1 Hypothesis

As in the changing environment problem, optimal performance in the distracting environment problem requires 16 functions, each tagged such that it is triggered by a single environment-state signal; once triggered, a function must adjust the agent's internal state appropriately. However, the distracting environment problem also requires agents to ignore distraction signals. If a distraction signal is able to trigger a function, the agent cannot reliably maintain an internal state that matches the current environment state. Given that agents must dedicate 16 functions to adjusting internal state in response to environmental changes, they must be able to passively ignore distraction signals to avoid triggering an erroneous internal state. As such, the

0% similarity threshold treatment cannot produce optimally-performing programs. Further, intermediate similarity thresholds must be high enough to allow agents to passively discriminate between distraction signals and environmental changes. We expect treatments with higher intermediate similarity thresholds to be able to achieve optimality.

6.4.1.2 Statistical Methods

Our statistical methods for analyzing these data are identical to those described in Sect. 6.3.1.3.

6.4.2 Results and Discussion

Figure 6.4 gives the results for the distracting environment problem early during our experiment (generation 1000) and at the end of our experiment (generation 10,000). At both generation 1000 and generation 10,000, programs evolved under different similarity thresholds had significantly different performance (Gen. 1000: Kruskal-Wallis, Chi-squared = 144.3, $p < 2.2\text{e}{-16}$; Gen. 10,000: Kruskal-Wallis, Chi-squared = 193, $p < 2.2\text{e-}16$). Table 6.2 gives the results of a post-hoc pairwise Wilcoxon rank-sum test for our results at both generation 1000 and generation 10,000.

As in the changing environment problem, runs requiring exact name matching (the 100% tag similarity threshold treatment) produce programs that perform significantly worse than those evolved in all other treatments. At generations 1000 and 10,000, programs evolved in the 75% tag similarity threshold treatment significantly outperform programs evolved in all other treatments. By 10,000

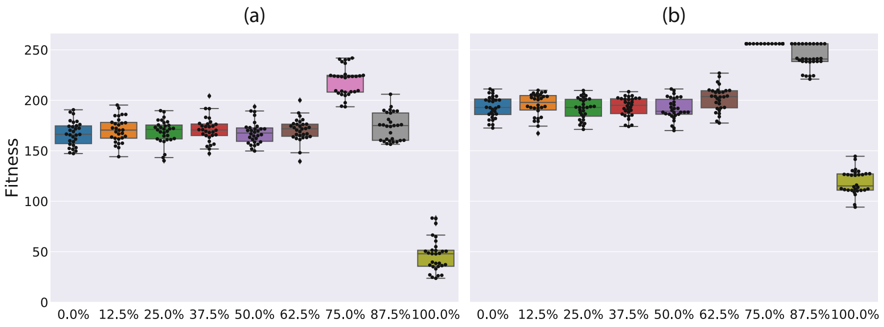


Fig. 6.4 Distracting environment problem results at: (**a**) generation 1000 and (**b**) generation 10,000. The box plots indicate the fitnesses (each an average over 100 trials) of the best performing programs from each replicate across a range of minimum similarity thresholds

Table 6.2 Pairwise Wilcoxon rank-sum test results for the distracting environment problem at generation 1000 and generation 10,000

		Tag Similarity Threshold									
		0.0%	12.5%	25.0%	37.5%	50.0%	62.5%	75.0%	87.5%	100.0%	
Tag Similarity Threshold	0.0%	■	1.0	1.0	1.0	1.0	0.240	**4.4e-11**	**1.4e-09**	**1.1e-09**	Generation 10,000
	12.5%	1.0	■	1.0	1.0	1.0	1.0	**4.4e-11**	**1.4e-09**	**1.1e-09**	
	25.0%	1.0	1.0	■	1.0	1.0	0.072	**4.4e-11**	**1.4e-09**	**1.1e-09**	
	37.5%	1.0	1.0	1.0	■	1.0	0.183	**4.4e-11**	**1.4e-09**	**1.1e-09**	
	50.0%	1.0	1.0	1.0	1.0	■	0.065	**4.4e-11**	**1.4e-09**	**1.1e-09**	
	62.5%	1.0	1.0	1.0	1.0	1.0	■	**4.4e-11**	**2.7e-09**	**1.1e-09**	
	75.0%	**1.1e-09**	**1.3e-09**	**1.1e-09**	**1.5e-09**	**1.2e-09**	**1.5e-09**	■	**7.2e-06**	**4.4e-11**	
	87.5%	1.0	1.0	1.0	1.0	1.0	1.0	**3.3e-09**	■	**1.4e-09**	
	100.0%	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.6e-09**	■	
		Generation 1,000									

Each row/column corresponds to a tag similarity threshold treatment. Each entry in the table indicates the Bonferroni-adjusted p value for a given comparison between two treatments. Statistically significant relationships ($p < 0.05$) are bolded. Results for generation 1000 are in red, below the table's diagonal (in black). Results for generation 10,000 are in yellow, above the table's diagonal

generations, only the 75% and 87.5% tag similarity threshold treatments produced perfectly optimal programs. Fully detailed statistical results can be found in our supplemental material, which can be accessed via GitHub [13].

Requiring *Some* Precision When Calling a Tag-Based Name Can Be Important, Too

These data are not surprising: we designed the distracting environment problem as a toy problem to demonstrate the idea that sometimes requiring some amount of precision when using tag-based referencing can be important. Because we limited programs to 16 functions and all 16 functions were required to monitor for environment changes, solving the distracting environment problem required programs to have the capacity to discriminate between true, meaningful signals and irrelevant, meaningless signals. However, even in this case where signal discrimination was crucial, requiring exact tag-matching for signals to successfully trigger program functions was still too harsh a requirement for well-performing programs to evolve.

For both the changing environment and distracting environment problems, the 75% tag similarity threshold treatments produced optimally performing programs, allowing for sufficient signal discrimination in the distracting environment problem while not too badly impeding evolution's ability to bootstrap program responses to true signals. However, these data do not necessarily imply anything general about a 75% tag similarity threshold. The critical tag similarity threshold for the distracting environment problem depends on the number of distraction signals that must be ignored versus the number of true signals the programs must respond to, the number of bits composing a tag (here, we used 16), as well as the number of functions SignalGP programs are allowed to have.

6.4.2.1 Illuminating Solution Space with MAP-Elites

As we did for the changing environment problem, we again use the MAP-Elites evolutionary algorithm [17] to illuminate the solution space for the distracting environment problem. We apply MAP-Elites to the distracting environment problem exactly as described in Sect. 6.3.2.1, using minimum tag similarity threshold for tag-based referencing and the number of unique functions used during program evaluation as our MAP-Elites grid axes. The heat map in Fig. 6.5 shows the density of optimal programs evolved using MAP-Elites within our chosen trait space.

Figure 6.5 confirms our intuition about the solution space in the distracting environment problem, showing that many strategies with a wide range of minimum tag similarity thresholds exist that use around 32 functions where extra ‘dummy’ functions can consume distraction signals. Indeed, there are optimal solutions that use only 16 functions; however, these solutions seem require high minimum tag similarity thresholds.

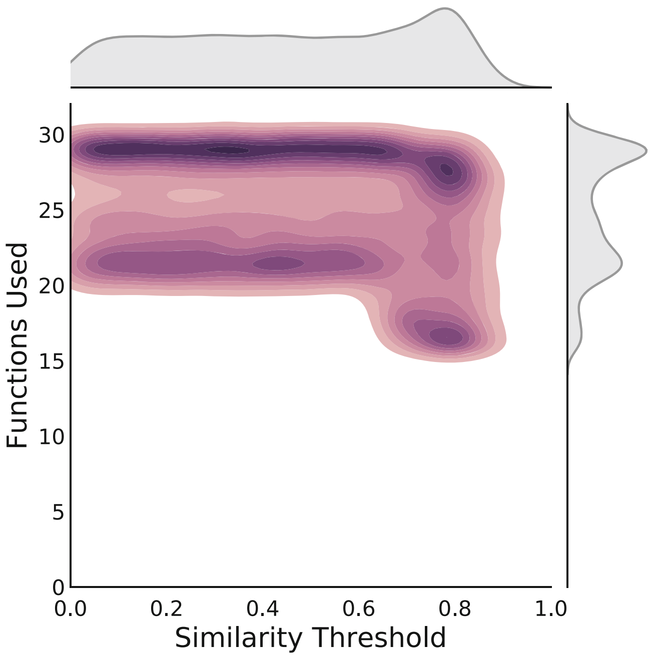


Fig. 6.5 Heat map of SignalGP programs evolved to solve the distracting environment problem using MAP-Elites. Locations in the heat map correspond to distinct combinations of the following two program traits: the number of unique functions used by a program during evaluation and the program’s minimum tag similarity threshold. Darker areas of the heat map indicate a higher density of solutions found with a particular trait combination

6.5 What Else Is in an Evolved Name? Broadened Applications of Tag-Based Naming in SignalGP

Thus far, we have explored the importance of inexactness in evolvable names in the context of SignalGP. In this section, we discuss several extensions to the SignalGP framework that are possible because of the evolvable specificity afforded by its tag-based naming scheme.

6.5.1 SignalGP Function Regulation

Bringing together ideas from GP and gene regulatory networks is not novel [1, 16]. The capacity to regulate genotypic expression is valuable in both biological and computational systems, allowing environmental feedback to alter phenotypic traits within an individual's lifetime.

SignalGP is easily extended to model gene regulatory networks where functions can be up-regulated (i.e., be made more likely to be referenced by a tag) or down-regulated (i.e., be made less likely to be referenced by a tag). For example, a function that would normally not be triggered by an event can be up-regulated to increase its priority over other function that have closer match. We can add regulatory instructions to the instruction set that increase or decrease function regulatory modifiers, using tag-based referencing to determine which function should be regulated by a particular instruction.

Gene regulation provides yet another mechanism for phenotypic flexibility, allowing SignalGP programs to alter referential relationships in response to environmental feedback. Such a mechanism might be useful for problems that require within-lifetime learning or general behavioral plasticity.

6.5.2 Multi-Representation SignalGP

In this work and in prior work, we have exclusively used SignalGP in the context of linear GP: SignalGP functions associate a tag with a linear sequence of instructions. However, in principle, SignalGP is generalizable across a variety of evolutionary computation representations.

SignalGP programs are composed of a set of functions where each function is referred to via its tag. We can imagine these functions to be black-box input-output machines: when called or triggered by an event, they are run with input and can produce output by manipulating memory or by generating signals. We have exclusively used linear GP in SignalGP functions; however, we could have just as easily used other types of representations capable of receiving input and producing output (e.g., other GP representations, artificial neural networks, Markov Brains [6], hard-coded modules, etc.). We could even employ a variety of representations within a single agent.

The evolvable specificity afforded by SignalGP's tag-based naming scheme allows us to use this sort of black-box metaphor. Functions composed of different representations can still refer to one another via tags, and events are agnostic to the underlying representation used to handle them, requiring only that the representation is capable of processing event-specific data. Allowing for these types of multi-representation agents may complicate the SignalGP virtual hardware, program evaluation, and mutation operators, but it would provide evolution with a toolbox of diverse representations.

Hintze et al. proposed and demonstrated the evolutionary Buffet Method where Markov Brains [6] could be composed of heterogeneous computational substrates, allowing evolution to work out the most appropriate representation for a given problem [7]. Further, Hintze et al.'s Buffet Method demonstrated the success of hybrid solutions. Multi-representation SignalGP provides an unexplored, alternative approach to evolving multi-representation agents, bringing the Buffet Method into an event-driven context.

6.5.3 Major Transitions in SignalGP

In a major evolutionary transition in individuality, formerly distinct individuals unite to form a new, more complex lifeform, redefining what it means to be an individual. The evolution of eukaryotes, multi-cellular life, and eusocial insect colonies are all examples of transitions in individuality. Often the individuals that make up the higher-level entity are limited to local information, lacking direct access to the global state of the higher-level unit; lower-level units must rely on signaling and sensory information to coordinate their roles in the group [19, 24]. In a computational sense, a major transition in individuality is the evolution of a distributed system. Capturing these types of transitions in GP would give evolution a mechanism to incrementally form distributed systems from formerly individual programs.

In the previous section, we described how SignalGP could be extended to allow multi-representation programs where functions (modules) can be of any representation capable of receiving input and producing output. We can take this approach to multi-representation SignalGP one step further: any module within a SignalGP agent could be *another* (former) SignalGP agent. This approach is conceptually similar to Tangled Program Graph representation [12].

We can imagine a mutation operator that, when applied, induces transitions in individuality by injecting co-evolving SignalGP programs as self-contained, tagged modules into the program being mutated, allowing single individuals to be aggregates of lower-level individuals. Further, transitions in individuality can be applied *hierarchically*. Biological evolution has examples of such hierarchical transitions: eusocial insect colonies are composed of many multicellular individuals, each which are composed of many eukaryotic cells, which in turn are composed of organelles (many of which are thought to have been formally distinct individuals).

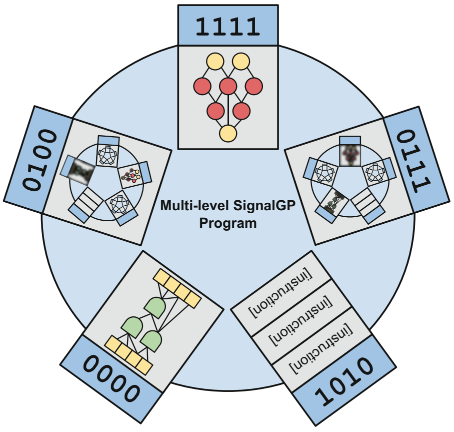


Fig. 6.6 Example of a multi-level SignalGP program. In this example, the agent is composed of five modules, including a neural network, a Markov Brain, a linear GP representation, and two multi-module programs at a lower level of organization

An individual SignalGP program may be composed of many SignalGP program modules, which may themselves be composed of many SignalGP programs, and so on (Fig. 6.6).

Implementing a mutation operator capable of inducing arbitrary numbers of hierarchical transitions in individuality requires us to answer the following questions: How should formerly individual programs interact when forced into an aggregate? And, how should an evolutionary algorithm handle evaluating both individuals and aggregates of individuals?

From the evolutionary algorithm's perspective, a multi-level SignalGP program is indistinguishable from a single-level. However, just as biological organisms composed of lower-level units of individuality require more energy to subsist, multi-level SignalGP programs require many more CPU cycles than single-level SignalGP programs. This is consistent with biology where major transitions disproportionately occur in energy-rich environments [19].

Extending SignalGP to support hierarchical transitions in individuality would provide a useful model for studying biological evolutionary transitions, allowing us to ask general questions about their dynamics. A transition in individuality mutation operator would also allow us to solve problems that might be best solved by a distributed system without knowing the optimal configuration of that distributed system a priori.

6.6 Conclusion

In this chapter, we explored the importance of inexactness when calling a tag-based name in GP. We show that allowing for inexactness when performing tag-based references is crucial for rapid adaptive evolution. Conversely, when some signals need to be ignored (such as in our distracting environment) it can be critical to prevent dissimilar tags from finding incorrect matches. As such, intermediate thresholds for tag similarity may be ideal for optimal evolution in a broad range of environments. The most appropriate similarity thresholds for a given problem will depend on the specifics of the problem and the representation used. For example, we would need to consider the ways tags are used in a particular problem, as well as how those tags are represented, mutated, and compared.

Interestingly, while exact naming is the most intuitive referencing mechanism for human programmers, evolution is far more successful when program references are allowed to be inexact. In fact, mutation-selection balance may prevent exact references from being stably maintained over evolutionary time. If tags are mutated such that we expect at least one of a program's tags (referring or referent) to be mutated per reproduction event, the relationships between referring and referent tags are unlikely to be stably maintained.

Both SignalGP and our proposed extensions to SignalGP are inspired by biological systems and processes. As we continue to develop SignalGP, our goal is to continue to push the boundary of GP and to use SignalGP as a tool to study the natural systems that inspired its development, such as the evolution of modularity, gene regulation, cell signaling, and major evolutionary transitions in individuality.

Acknowledgements We extend our thanks to the members of the Digital Evolution Laboratory at Michigan State University and the attendees of the 2018 Genetic Programming Theory and Practice Workshop for thoughtful discussions and feedback on this work. This research was supported by the National Science Foundation (NSF) through the BEACON center (Cooperative Agreement DBI-0939454), a Graduate Research Fellowship to AL (Grant No. DGE-1424871), and NSF Grant No. DEB-1655715 to CO. Michigan State University provided computational resources through the Institute for Cyber-Enabled Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or MSU.

References

1. Banzhaf, W.: Artificial Regulatory Networks and Genetic Programming. In: Genetic Programming Theory and Practice, pp. 43–61. Springer US, Boston, MA (2003)
2. Cassandras, C.G.: The event-driven paradigm for control, communication and optimization. Journal of Control and Decision **1**, 3–17 (2014)
3. Dolson, E., Lalejini, A., Ofria, C.: Exploring genetic programming systems with MAP-Elites. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-04735-1_1
4. Etzion, O., Niblett, P.: Event Processing in Action. ISBN: 9781935182214. Manning Publications (2010)
5. Heemels, W.P., Johansson, K.H., Tabuada, P.: An introduction to event-triggered and self-triggered control. Proceedings of the IEEE Conference on Decision and Control pp. 3270–3285 (2012)
6. Hintze, A., Edlund, J.A., Olson, R.S., Knoester, D.B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P., Sheneman, L., Goldsby, H., Bohm, C., Adami, C.: Markov Brains: A Technical Introduction. arXiv **1709.05601** (2017)

7. Hintze, A., Schossau, J., Bohm, C.: The evolutionary Buffet method. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI. Springer International Publishing, Cham (2018). http://10.1007/978-3-030-04735-1_2
8. Holland, J.: The effect of labels (tags) on social interactions. Santa Fe Inst., Santa Fe, NM, Working Paper pp. 93–10 (1993). URL http://samoa.santafe.edu/media/workingpapers/93-10-064.pdf
9. Holland, J.H.: Genetic Algorithms and Classifier Systems: Foundations and Future Directions. In: Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87), pp. 82–89 (1987)
10. Holland, J.H.: Concerning the emergence of tag-mediated lookahead in classifier systems. Physica D: Nonlinear Phenomena **42**, 188–201 (1990). DOI 10.1016/0167-2789(90)90073-X
11. Holland, J.H.: Studying complex adaptive systems. Journal of Systems Science and Complexity **19**, 1–8 (2006)
12. Kelly, S., Heywood, M.I.: Emergent Tangled Graph Representations for Atari Game Playing Agents. In: European Conference on Genetic Programming (EuroGP-2017), pp. 64–79 Springer (2017)
13. Lalejini, A.: amlalejini/GPTP-2018-Exploring-Evolvable- Specificity-with-SignalGP (2018). Available on GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP
14. Lalejini, A., Ofria, C.: Evolving Event-driven Programs with SignalGP. arXiv **1804.05445** (2018)
15. Lalejini, A., Wiser, M.J., Ofria, C.: Gene Duplications Drive the Evolution of Complex Traits and Regulation. Proceedings of the European Conference on Artificial Life (ALIFE-2017), 4–8 (2017)
16. Lopes, R.L., Costa, E.: Genetic programming with genetic regulatory networks. In: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation - GECCO '13, p. 965. ACM Press, New York, New York, USA (2013)
17. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv **1504.04909**, 1–15 (2015)
18. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2016). URL https://www.R-project.org/
19. Smith, J.M., Szathmary, E.: The major transitions in evolution. Oxford University Press (1997)
20. Spector, L., Harringtion, K., Martin, B., Helmuth, T.: What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In: R. Riolo, E. Vladislavleva, J.H. Moore (eds.) Genetic Programming Theory and Practice IX, Genetic and Evolutionary Computation, chap. 1, pp. 1–16. Springer New York, New York, NY (2011)
21. Spector, L., Harrington, K., Helmuth, T.: Tag-based modularity in tree-based genetic programming. GECCO '12: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation pp. 815–822 (2012)
22. Spector, L., Martin, B., Harrington, K., Helmuth, T.: Tag-based modules in genetic programming. GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation pp. 1419–1426 (2011)
23. Waskom, M., et al.: mwaskom/seaborn: v0.8.1 (september 2017) (2017). URL https://doi.org/10.5281/zenodo.883859
24. West, S.A., Fisher, R.M., Gardner, A., Kiers, E.T.: Major evolutionary transitions in individuality. Proceedings of the National Academy of Sciences **112**, 10,112–10,119 (2015)

Chapter 7
Lexicase Selection Beyond Genetic Programming

Blossom Metevier, Anil Kumar Saini, and Lee Spector

7.1 Introduction

Lexicase selection is a selection algorithm for evolutionary computation systems, used to determine which individuals will be permitted to contribute to future generations in the evolutionary process. Although it has been used for survivor selection [8], its primary use has been as a *parent* selection algorithm, selecting individuals to be provided as inputs to genetic operators. The genetic operators, such as mutation and crossover, use the selected parents as source material out of which to construct children.

Lexicase selection selects individuals by filtering a pool of individuals which, before filtering, typically contains the entire population. The filtering is accomplished in steps, each of which filters according to performance on single test case (input/output pair). The test cases are considered one at a time in random order. Lexicase selection has been tested most extensively in genetic programming systems, where it has been shown to outperform other selection methods in several contexts [2–5, 7, 9–11]. However, the effectiveness of lexicase selection in other settings has not been fully explored.

In this paper, we investigate the utility of lexicase selection in traditional genetic algorithms with linear, fixed-length genomes. We chose this framework in part

B. Metevier · A. K. Saini
College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA
e-mail: bmetevier@umass.edu; aks@cs.umass.edu

L. Spector (✉)
School of Cognitive Science, Hampshire College, Amherst, MA, USA

College of Information and Computer Sciences, University of Massachusetts,, Amherst, MA, USA
e-mail: lspector@hampshire.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_7

because the large literature of traditional genetic algorithms provides context for the interpretation of our results that is both broad and deep.

The problems to which we apply traditional genetic algorithms, using several parent selection methods, are randomly generated Boolean constraint satisfaction problems [1]. Although the problems are derived from Boolean satisfiability problems, the problem-solvers (in this case, the genetic algorithms) are not given access to the constraints themselves, or to the variables that they contain. Rather, the problem-solvers are given access only to a procedure that determines whether each constraint is satisfied by an assignment of truth values to all variables. Crucially, the problem-solvers are given no information about which constraints may depend on which others. This design is intended to allow the problems serve as abstractions of problems in many real-world domains, in which we can tell whether or not a candidate solution satisfies a constraint, but we have no further information about the nature of the constraint, or of the ways in which different constraints might be interdependent.

In this chapter, we present the results of experiments using the traditional genetic algorithm, with lexicase selection, to solve Boolean constraint satisfaction problems. We compare the performance of the algorithm using lexicase selection to the performance of the same algorithm run with more traditional selection algorithms, specifically fitness-proportionate selection and tournament selection (with several tournament sizes).

In the following sections we first describe the lexicase selection algorithm that is the focus of this investigation. We then describe the Boolean constraint satisfaction problems that we use for our experiments, and our experimental methods. We then present our results, and discuss their implications for future work.

7.2 Lexicase Selection

Lexicase selection is a method by which individuals can be selected from a population for use as the source material out of which genetic operators, such as mutation and crossover, construct offspring for the following generation. Lexicase selection is distinctive in that it allows selection to depend on multiple assessment criteria and all of their combinations, without requiring that these criteria be aggregated into overall "fitness" values. This is different from the selection methods used traditionally in genetic programming, which require the assignment of a single scalar value to each candidate solution in order to guide search.

In most of the prior work on lexicase selection, it has been used in genetic programming systems to select parent programs that are then subjected to variation to produce the next generation of programs. In this context, the assessment criteria are generally the errors of the program on different inputs, which are often referred to, in the genetic programming literature, as "fitness cases" or "test cases."

Algorithm 1: Lexicase selection

```
Result: Individual to be used as a parent
candidates := the entire population
cases := list of all test cases in a random order
while True do
    candidates := candidates performing best on the first case
    if only one candidate exists in candidates then
        return that candidate
    end
    if cases is empty then
        return a randomly selected candidate from candidates
    end
    delete the first case from cases
end
```

In lexicase selection, each time a parent is needed, a pool of individuals, which initially contains the entire population,[1] is winnowed in successive stages until a single individual remains and is selected. In the first stage, only the individuals that perform best over a randomly chosen test case are retained. If more than one individual remains, a second randomly chosen test case is used for the next stage of winnowing. This process repeats until only a single individual remains, or until the test cases have been exhausted, in which case a random individual from the remaining pool is selected. Pseudocode for the most commonly used form of lexicase selection is provided in Algorithm 1.

Sometimes, lexicase selection chooses individuals with performance that is good over only a small number of test cases. Many of these "specialists" would, under many selection methods that require aggregation of performance on all test cases into single scalar values, rarely be selected for reproduction and variation. This would often be the case even if one of the cases solved by the specialist was difficult for a majority of the population to solve. The reason for this is the assumption of uniformly distributed selection pressure, with all parts of a problem being equally hard."Specialists" are ignored even though they are better at certain subsets of the problem and may contain a partial solution to the task at hand. By allowing these "specialists" to contribute to the next generation, lexicase selection allows for offspring that may contain solutions to a particular subset of the problem. Comparisons of lexicase selection to other methods developed with similar motivations, such as "implicit fitness sharing" and "deterministic crowding," are presented elsewhere [5, 7].

Several variants of the lexicase selection method have also been developed and studied. For example, *epsilon lexicase selection*, a variant in which candidates that are not strictly "best" on the current case but which are "close enough" (within

[1]It is also possible to limit the initial pools in various ways. When the initial pool contains the entire population, which is the best-studied setting, we refer to the algorithm more specifically as "global pool" lexicase selection.

epsilon, for some definition of epsilon) has been developed and shown to be particularly effective on problems with floating-point errors. Other variants have been developed and explored in previous instances of the *Genetic Programming Theory and Practice* workshop [12]. For the present study, however, we used the simplest and most standard version of the method, as described above.

7.3 Problems

7.3.1 Boolean Constraint Satisfaction

The problems used for the experiments in this study are Boolean constraint satisfaction problems [1] that are randomly generated based on three parameters: a total number of variables v, a number of constraints c, and a number of clauses per constraint n.

Each clause is a disjunction of three literals, each of which is either a variable or a negated variable. Each constraint is a conjunction of clauses, and a problem is a conjunction of constraints. An assignment of truth values to the variables is a solution if all of the constraints evaluate to true in the context of the assignment.

These problems are similar in some respects to Boolean satisfiability problems expressed in 3-CNF (conjunctive normal form, with three literals per clause), with the clauses grouped to form constraints. However, unlike the case with standard satisfiability problems, such as those used in SAT-solver competitions [6], we do not allow our problem solvers to see the formulae themselves, or to have any information about which variables appear in which constraints. The problem-solvers can evaluate an assignment of truth values to the variables with respect to each constraint, determining whether or not each constraint is satisfied by the assignment, but this is the only information that the problem solver receives about the problem.

7.3.2 Random Problem Generation

We generate a problem by starting with a random assignment of truth values to all variables. This assignment will be a solution to the generated problem, but we will discard it after generating the problem, and it will be the task of the problem solver to re-discover the assignment, or to discover another assignment that also satisfies all of the constraints in the problem.

Once we have a random assignment of truth values to all variables, we generate the problem itself with a simple, iterated generate-and-test algorithm: We create a random set of constraints of the specified size (which may include duplicate clauses, possibly in different constraints), and we check to see if it evaluates to true with

Table 7.1 Problem parameters

Parameter	Value
Number of variables (v)	20, 30, 40
Number of constraints (c)	8, 12, 16, 32
Number of clauses per constraint (n)	20, 25, 30, 35, 40
Number of problems per combination of v, c, and n	15
Number of runs per method per problem	50
Total number of runs per method per combination of v, c, and n	750

Table 7.2 Genetic algorithm parameters

Parameter	Value
Population size	200
Number of generations	500
Mutation operator	Bit-flip
Probability of mutation	0.1
Crossover operator	One-point
Probability of crossover	0.9

the given assignment. If it does, then we use the constraints as a problem for our experiments; if it doesn't, then we randomly generate a new set of constraints, repeating the process until we find one that is satisfied by the assignment.

For each (v, c, n) triple, we generated 15 different problems. Each problem-solving method was run 50 times on each of these problems, resulting in 750 runs per problem-solving method for each combination of v, c, and n. Results were evaluated by averaging over all 750 runs for each parameter combination. The specific parameters used for generating problems, and for the numbers of runs conducted on each problem with each method, are shown in Table 7.1.

7.4 Experimental Methods

7.4.1 Genetic Algorithm

Our problem-solving methods were all instances of the same genetic algorithm, with identical parameters (shown in Table 7.2) except for the parent selection method.

Individuals in the population were truth assignments, with genomes consisting of genes for each variable, indicating whether that variable had a value of true or false in the specified assignment.

We used a generational genetic algorithm that began with a population of random individuals, and then entered a cycle in which, for each generation, all individuals were tested for errors, parents were selected for the production of children on the basis of those errors, and children were produced by varying the selected parents.

7.4.2 Variation

At the variation step, individuals selected to serve as parents were first (possibly) subjected to crossover and then (possibly) to mutation.

The standard one-point method was used as the crossover operator, allowing parent recombination at a randomly chosen crossover point. The crossover rate was 0.9, meaning that 90% of children were produced by crossover.

For mutation, a bit-flip mutation operator was used, allowing for a randomly chosen bit in a chromosome to be flipped. The mutation rate was 0.1, meaning that 10% of children were subject to mutation.

7.4.3 Parent Selection

We compared lexicase parent selection, tournament parent selection, and fitness proportionate parent selection. All selection methods performed selection with replacement; that is, the same individual might be selected to be a parent several times in the same generation.

For tournament selection and fitness proportionate selection, an individual's total error value was determined from the number of constraints it satisfied. If an individual satisfied all constraints, then its error was 0. Otherwise, its error value was the number of constraints that it did not satisfy.

For tournament selection with an integer-valued tournament size t, we first form a tournament set of t individuals, each of which is chosen with uniform probability (with possible duplication) from the entire population. We then return, as the selected parent, the individual in the tournament set that has the lowest total error.

Higher tournament sizes make tournament selection more selective, in the sense that individuals with high total error are less likely to be selected, while lower tournament sizes make it less selective. Because our preliminary experiments showed that less selective settings appeared to perform better, we wanted to consider methods even less selective than tournament selection with tournament size 2, which is normally considered to be the minimum, since with tournament size 1 tournament selection is equivalent to selecting individuals entirely randomly. For this purpose we adopted the convention that for a non-integer-valued tournament size t between 1 and 2 we would use tournament size 2 with probability $t - 1$, and select a parent entirely randomly otherwise. For example, with $t = 1.25$, 25% of the time we will choose 2 individuals randomly and return, as the selected parent, the one with the lower total error; the remaining 75% of the time we will return, as the selected parent, a parent chosen with uniform probability from the entire population.

We performed fitness-proportionate selection in the standard way: The probability of selection for an individual i that satisfies s_i constraints is s_i divided by sum of s_j for all individuals j across the population. In the degenerate case of no individuals satisfying any constraints, which would produce a denominator of zero, an individual is selected at random.

7.5 Results

7.5.1 Success Rates by Parent Selection Method

Our primary results are shown in Table 7.3, comparing success rates of the genetic algorithm when run with fitness proportionate parent selection, tournament parent selection (with tournament size 2), and lexicase parent selection.

Table 7.3 contains a row for every combination of number of variables (v) and number of constraints (c). All runs with a specified value of v and c are aggregated in the corresponding line, regardless of the number of clauses per constraint (n). Because we conducted 750 runs with each combination of v, c, and n for each parent selection method, and because we conducted experiments with 5 different values of n (see Table 7.1), each row in Table 7.3 reports data from $5 * 750 = 3750$ runs with each of the three parent selection methods listed in the table.

The numbers reported in Table 7.3 are success rates, defined as the proportion of the total runs that produced a successful solution (with error vector consisting only of zeros). Lexicase selection produces the highest success rate in every case, and the improvement provided by lexicase selection is statistically significant in most cases.

Table 7.3 Success rate for the genetic algorithm with fitness proportionate, tournament (size 2), and lexicase parent selection for each studied combination of v (number of variables) and c (number of constraints)

Number of variables (v)	Number of constraints (c)	Fitness proportionate	Tournament (size 2)	Lexicase
20	8	0.835	0.867	$\underline{0.992}$
20	12	0.940	0.954	$\underline{1.000}$
20	16	0.980	0.987	$\underline{1.000}$
20	32	0.999	1.000	1.000
30	8	0.415	0.475	$\underline{0.889}$
30	12	0.614	0.697	$\underline{0.995}$
30	16	0.815	0.869	$\underline{1.000}$
30	32	0.983	0.995	1.000
40	8	0.205	0.257	$\underline{0.689}$
40	12	0.224	0.310	$\underline{0.927}$
40	16	0.433	0.576	$\underline{0.993}$
40	32	0.861	0.944	$\underline{1.000}$

Underlines indicate statistically significant improvements, determined using a pairwise chi-square test with Holm correction and $p < 0.05$

The only cases in which the improvement is not significant are those in which all the selection algorithms approach a perfect success rate.

7.5.2 Success Rates by Tournament Size

Because the success rate of tournament selection is better or equal to fitness proportionate selection, many of our analyses in the remainder of this paper will compare lexicase selection only against tournament selection. Furthermore, because tournament selection is itself parameterized by the tournament size, we conducted additional experiments to compare performance across settings with different tournament sizes.

Table 7.4 shows the results of these runs. We again conducted 3750 runs for each combination of number of variables (v) and number of constraints (c), across the range of values for number of clauses per constraint (n) given in Table 7.1. We conducted runs for tournament sizes ranging from 1.25 to 8, with non-integer-valued tournament sizes handled as described in Sect. 7.4. The runs with tournament size 2 were independent of those conducted for the experiments documented in Table 7.3, so the numbers differ between the two tables, but not by much.

From Table 7.4 it appears that the most effective tournament size is around 1.5 or 2, and that larger tournament sizes perform poorly. None of the tested tournament sizes performs better than lexicase selection.

Table 7.4 Success rate for different tournament sizes

Number of variables (v)	Number of constraints (c)	Tournament size 1.25	Tournament size 1.5	Tournament size 2	Tournament size 4	Tournament size 8
20	8	0.850	**0.860**	0.856	0.818	0.777
20	12	0.948	0.955	**0.959**	0.952	0.934
20	16	0.982	0.987	0.988	**0.989**	0.979
20	32	**1.000**	**1.000**	0.999	**1.000**	0.999
30	8	0.443	**0.485**	0.471	0.428	0.367
30	12	0.644	0.702	**0.773**	0.712	0.618
30	16	0.850	**0.888**	0.879	0.846	0.766
30	32	0.993	**0.996**	**0.996**	0.990	0.974
40	8	0.226	**0.271**	0.137	0.120	0.105
40	12	0.254	**0.322**	0.293	0.245	0.213
40	16	0.510	**0.614**	0.503	0.423	0.335
40	32	0.938	**0.958**	0.901	0.794	0.680

Boldfaced numbers indicate the highest success rate in a particular row

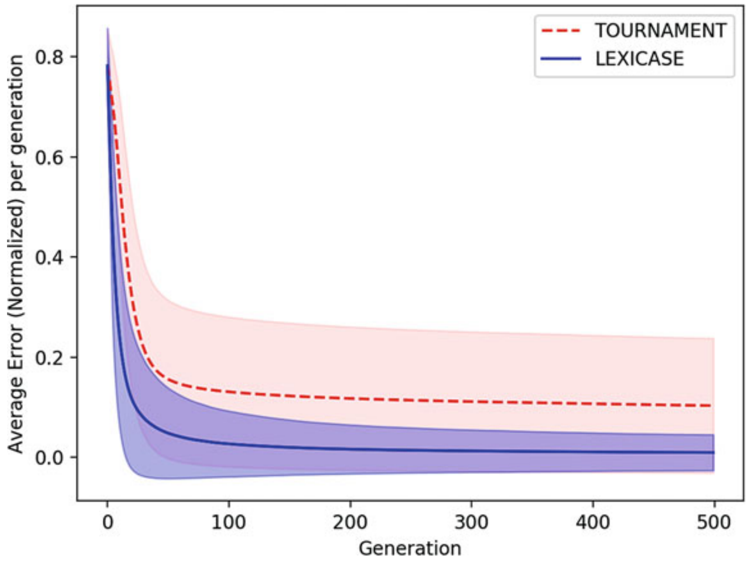


Fig. 7.1 Average error per generation for runs with lexicase selection and tournament selection with tournament size 2, with 50% confidence intervals

7.5.3 Errors over Evolutionary Time

Other features of the data produced by the runs described above, in Sect. 7.5.1, may be revealing, aside from the success rates described above.

In Fig. 7.1 we show the normalized average error across all runs. Here each error has been normalized by the total number of constraints (maximum error) used for that particular run. For a particular generation, the error has been averaged across the runs which were active up to that generation.

Here we see that lexicase selection not only produces lower errors, but also that lower errors are reached earlier in evolutionary time. We also see that both methods make most of their gains quite early in evolutionary time, with few improvements occurring after 100 generations.

7.5.4 Mean Least Error

Figure 7.2 shows the mean lest error (MLE) for each combination of number of variables (v) and number of constraints (c). MLE is defined as the average of the error values of the lowest-error individuals in each of the runs:

$$MLE = (1/N) \sum_i error(best\_ind_i),$$

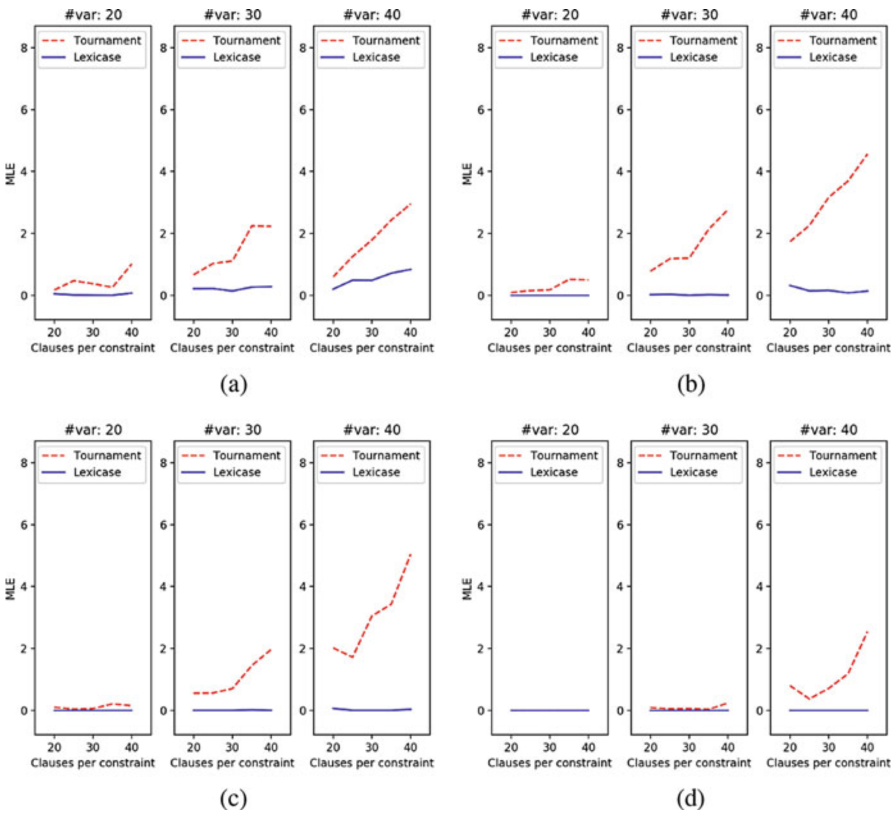


Fig. 7.2 Mean Least Error (MLE) for each combination of parameters. The number C denotes the number of constraints used for the corresponding plot. (**a**) $C = 8$, (**b**) $C = 12$, (**c**) $C = 16$, (**d**) $C = 32$

where $best\_ind_i$ is the individual having lowest total error in a given run i.

These plots show that tournament selection not only fails to solve problems in many cases, but also that the best errors achieved in the failing runs are often quite high. This effect is particularly pronounced for runs with large numbers of clauses per constraint.

7.5.5 *Success Generations*

Figure 7.3 shows the success generation—that is, the generation in which the genetic algorithm was able to find a zero-error solution—for each combination of number of variables (v) and number of constraints (c), averaged over the runs in which the genetic algorithm was able to find a solution.

Here we see that even when the genetic algorithm with tournament selection was able to find a solution, it generally required more generations to do so than did the genetic algorithm with lexicase selection.

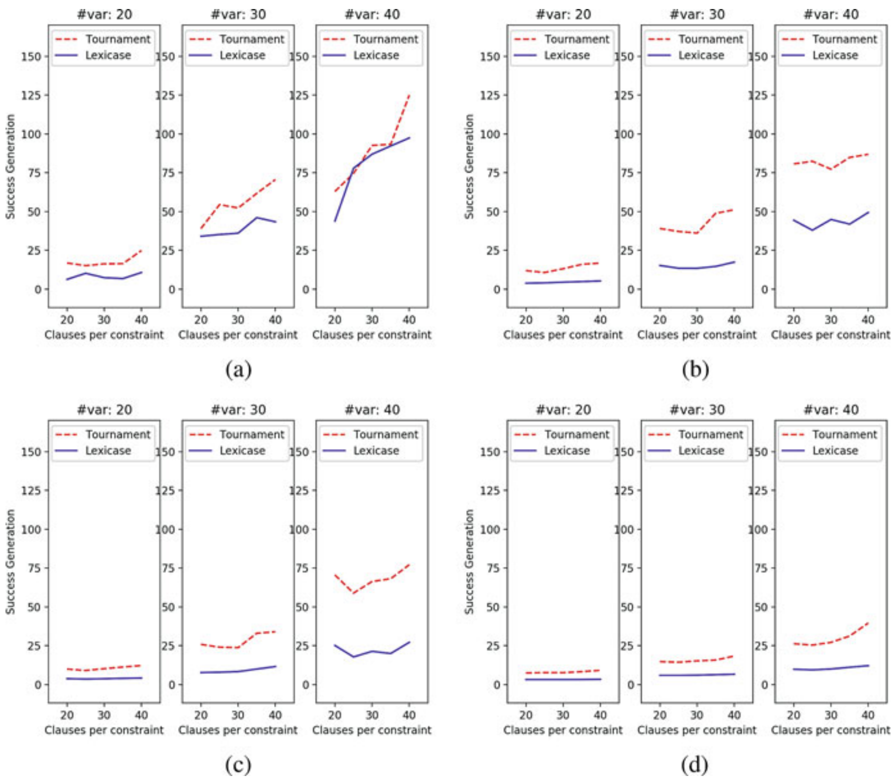


Fig. 7.3 Success Generation for each combination of parameters. The number C denotes the number of constraints used for the corresponding plot. (**a**) $C = 8$, (**b**) $C = 12$, (**c**) $C = 16$, (**d**) $C = 32$

7.5.6 Diversity over Evolutionary Time

Figure 7.4 shows the average number of unique chromosomes (individuals) in the population over evolutionary time, aggregated over all parameter combinations, and grouped by selection method and whether each method found a solution to the problem or not. We see from these plots that, with the exception of brief periods at the starts of runs, lexicase selection maintains relatively high diversity throughout the process of evolution, both in successful and in unsuccessful runs.

7.6 Discussion

The results presented above show that, for the Boolean constraint satisfaction problems studied here, the traditional genetic algorithm performs better with lexicase

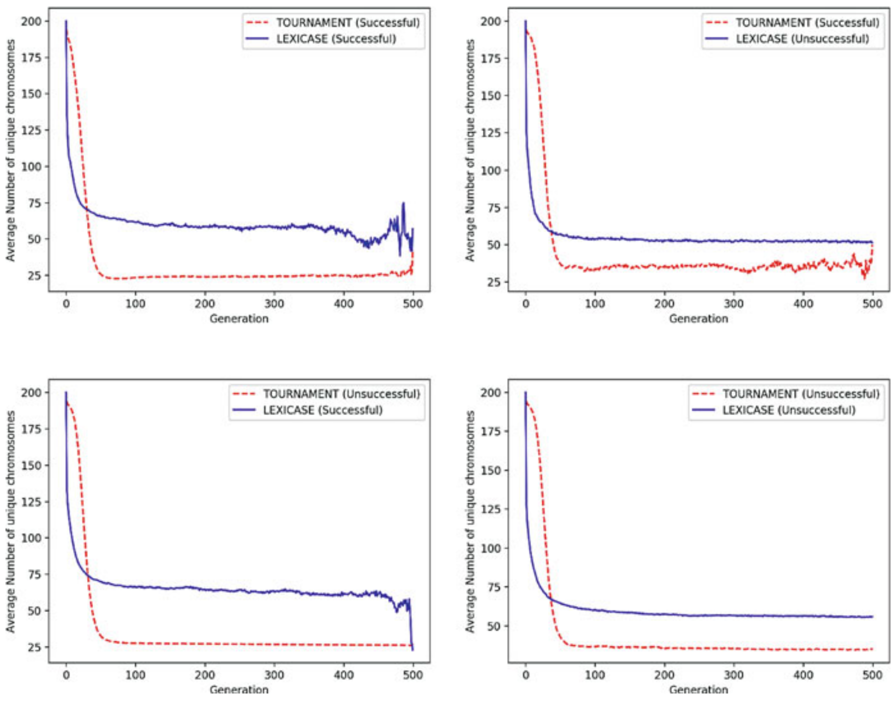


Fig. 7.4 Average number of unique chromosomes (individuals) in the population, over evolutionary time, under different conditions

parent selection than with tournament parent selection or fitness proportionate parent selection. In these experiments, lexicase selection found solutions more frequently and in fewer generations than did the other parent selection methods.

In addition, more diverse populations were maintained under lexicase selection, although the results here do not say anything definitive about the causal relations that may hold between diversity, success rate, and the number of generations required to find solutions.

With respect to the central question of this study, about whether lexicase selection has utility outside of genetic programming, these results suggest a positive answer: Lexicase selection does appear likely to have broader applicability than has been demonstrated previously.

The problems studied in this investigation were artificial, but they were designed to have features that resemble those many real-world problems. More specifically, the problems studied here were designed to be resemble real-world problems in which the goal is to satisfy many constraints simultaneously, but in which both the constraints themselves, and their interdependencies, are opaque.

For the problems studied here, the problem-solver is given access to a procedure that indicates whether or not each constraint is satisfied by a candidate solution, but it has no other information about the nature of the constraints or about shared components or structure among multiple constraints. To the extent that a real-world problem fits this characterization, the results here suggest that lexicase parent selection could help to solve it.

Nonetheless, lexicase parent selection is probably not appropriate for all problems. For example, it seems unlikely that it would work well on problems that involve only a single constraint. In these cases, only the single individual in the population with the best performance on that constraint (or other individuals with the same performance) could be selected to serve as a parent. It seems reasonable to assume that this would undermine population diversity, making it more difficult to find solutions. Problems with more than a single constraint, but not many more, may be a poor match to lexicase parent selection for the same reason.

Previous work has also shown that lexicase parent selection sometimes performs poorly on problems with floating-point error values. The epsilon lexicase selection method appears to address this problem quite well [9], and it seems reasonable to assume that it would also work well on floating-point versions of the constraint satisfaction problems presented here.

One exciting avenue for future work would be an investigation of whether lexicase selection may have even broader applicability, perhaps extending beyond evolutionary computation altogether. Other machine learning methods might also be able to take advantage of the core idea of lexicase selection, that whenever we must make a decision based on the quality of candidate solutions, instead of aggregating multiple measures of quality into single, scalar values, we may instead consider them one at a time, in random order.

The ways in which this core idea of lexicase selection can be fleshed out will differ from one machine learning method to another, and we cannot yet provide definitive guidance on how it should be done for any specific methods outside of evolutionary computation. The results presented here, however, lead us to speculate that some such efforts will be rewarded with improvements in the problem-solving capabilities of the machine learning methods to which they are applied.

Acknowledgements We thank other members of the Hampshire College Institute for Computational Intelligence, along with other participants in the Genetic Programming Theory and Practice workshop, for helpful feedback and stimulating discussions.

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Craenen, B.G.W., Eiben, A.E., van Hemert, J.I.: Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Transactions on Evolutionary Computation **7**(5), 424–444 (2003). https://doi.org/10.1109/TEVC.2003.816584

2. Helmuth, T., McPhee, N.F., Spector, L.: Effects of Lexicase and Tournament Selection on Diversity Recovery and Maintenance. In: Companion Proceedings of the 2016 Conference on Genetic and Evolutionary Computation, pp. 983–990. ACM (2016). http://dl.acm.org/citation.cfm?id=2931657
3. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: Proceedings of the 2016 Conference on Genetic and Evolutionary Computation, pp. 717–724. ACM (2016). http://dl.acm.org/citation.cfm?id=2908851
4. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: Genetic Programming Theory and Practice XIII, pp. 151–167. Springer (2016)
5. Helmuth, T., Spector, L., Matheson, J.: Solving Uncompromising Problems with Lexicase Selection. IEEE Transactions on Evolutionary Computation **19**, 630–643 (2014). https://doi.org/10.1109/TEVC.2014.2362729
6. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international sat solver competitions. AI Magazine **33**, 89–92 (2012)
7. La Cava, W., Helmuth, T., Spector, L., Moore, J.H.: A probabilistic and multi-objective analysis of lexicase selection and -lexicase selection. Evolutionary Computation (2018). https://doi.org/10.1162/evco_a_00224. In press.
8. La Cava, W., Moore, J.: A general feature engineering wrapper for machine learning using epsilon-lexicase survival. In: M. Castelli, J. McDermott, L. Sekanina (eds.) EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming, *LNCS*, vol. 10196, pp. 80–95. Springer Verlag, Amsterdam (2017). https://doi.org/10.1007/978-3-319-55696-3_6
9. La Cava, W., Spector, L., Danai, K.: Epsilon-Lexicase Selection for Regression. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, pp. 741–748. ACM, New York, NY, USA (2016). http://doi.acm.org/10.1145/2908812.2908898
10. McPhee, N.F., Casale, M.M., Finzel, M., Helmuth, T., Spector, L.: Visualizing Genetic Programming Ancestries. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, pp. 1419–1426. ACM (2016). http://dl.acm.org/citation.cfm?id=2931741
11. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, pp. 401–408. ACM (2012)
12. Spector, L., La Cava, W., Shanabrook, S., Helmuth, T., Pantridge, E.: Relaxations of lexicase parent selection. In: W. Banzhaf, R.S. Olson, W. Tozier, R. Riolo (eds.) Genetic Programming Theory and Practice XV, pp. 105–120. Springer International Publishing, Cham (2018)

Chapter 8
Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems

Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanc

8.1 Introduction

Artificial neural networks (ANNs) were first proposed 75 years ago [26] Yet, ANNs still have poorer general learning capabilities than relatively simple organisms. Organisms can learn to perform well on many tasks and can generalise from few examples. Most ANNs models encode learned knowledge solely in the form of connection strengths (i.e. weights). Biological brains do not learn merely by the adjustment of weights, they undergo topological changes during learning. Indeed, restricting learning to weight adjustment leads to "catastrophic forgetting" (CF) in which ANNs trained to perform well on one problem, forget how to solve the original problem when they are re-trained on a new problem [9, 25, 34]. Although the original inspiration for ANNs came from knowledge about the brain, very few ANN models use evolution and development, both of which are fundamental to the construction of the brain [29]. In principle, developmental neural approaches could alleviate catastrophic forgetting in at least two ways. Firstly, new networks could form in response to learning. Secondly, by growing numerous connections between pairs of neurons. In this way the influence of individual weighted connection could be lessened.

Developmental neural networks have not widely been explored in the literature and there remains a need for concerted effort to explore a greater variety of effective models. In this paper, we propose a new conceptually simple neural model. We

J. F. Miller
University of York, Heslington, York, UK
e-mail: julian.miller@york.ac.uk

D. G. Wilson (✉) · S. Cussat-Blanc
University of Toulouse, IRIT - CNRS - UMR5505, Toulouse, France
e-mail: dennis.wilson@irit.fr; sylvain.cussat-blanc@irit.fr

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_8

suggest that at least two neural programs are required to construct neural networks. One to represent the neuron soma and the other the dendrite. The role of the soma program is to allow neurons to move, change, die or replicate. For the dendrite, the program needs to be able to grow and change dendrites, cause them to die and also to replicate. Since developmental programs build networks that change over time it is necessary to define new problem classes that are suitable to evaluate such approaches. We argue that trying to solve multiple computational problems (potentially even of different types) is an appropriate class of problems.

In this chapter, we show that the pair of evolved programs can build a network from which multiple conventional ANNs can be extracted each of which can solve a different classification problem. As far as we can tell, this is the first work that attempts to evolve developmental neural networks that can solve multiple problems, indeed it appears to be the first attempt to solve standard classification problems using a developmental approach. We investigate many parameters and algorithmic variants and assess experimentally which aspects are most associated with good performance. Although we have concentrated in this paper on classification problems, our approach is quite general and it could be applied to a much wider variety of problems.

8.2 Related Work

A number of authors have investigated ways of incorporating development to help construct ANNs [24] and [42]. Researchers have investigated a variety of genotype representations at different levels of abstraction. Cangelosi et al. defined genotypes which were a mixture of variables, parameters, and rules (e.g. cell type, axon length and cell division instructions) [4]. The task was to control a simple artificial organism. Rust et al constructed a genotype consisting of developmental parameters (encoded in binary) that controlled the times at which dendrites could branch and how the growing tips would interact with patterns of attractants placed in an environment [38]. Balaam investigated controlling simulated agents using a two-dimensional area with chemical gradients in which neurons were either sensors, affectors, or processing neurons according to location [2]. The neurons were defined as standard CTRNNS. The genotype was effectively divided into seven chromosomes each of which read the concentrations of the two chemicals and the cell potential. Each chromosome provided respectively the neuron bias, time constant, energy, growth increment, growth direction, distance to grow and new connection weight.

Gruau used a more abstract approach, called cellular encoding in which ANNs were developed using graph grammars [12, 13]. He evaluated this approach on hexapod robot locomotion and pole-balancing. Kodjabachian and Meyer used a “geometry-orientated” variant of cellular encoding to develop recurrent neural networks to control the behaviour of simulated insects [23].

Jacobi presented a low-level approach in which cells used artificial genetic regulatory networks (GRNs). The GRN produced and consumed simulated proteins that defined various cell actions (protein diffusion movement, differentiation, division, threshold). After a cellular network had developed it was interpreted as a neural network [18]. Eggenberger also used an evolved GRN [6]. A neural network phenotype was obtained by comparing simulated chemicals in pairs of neurons to determine if the neurons are connected and whether the connection is excitatory or inhibitory. Weights of connections were initially randomly assigned and Hebbian learning used to adjust them subsequently. Astor and Adami also encoded a form of GRN together with an artificial chemistry (AC), in which cells were predefined to exist in a hexagonal grid. Genes encoded conditions involving concentrations of simulated chemicals which determine the level of activation of cellular actions (e.g. grow axon or dendrite, increase or decrease weight, produce chemical) [1]. They evaluated the approach on a simple artificial organism.

Federici used a simple recursive neural network as a developmental cellular program [7]. In his model, cells could change type, replicate, release chemicals or die. The type and metabolic concentrations of simulated chemicals in a cell were used to specify the internal dynamics and synaptic properties of its corresponding neuron. The position of the cell within the organism is used to produce the topological properties of neuron: its connections to inputs, outputs and other neurons. From the cellular phenotype, Federici interpreted a network of spiking neurons to control a Khepera robot.

Some researchers have studied the potential of Lindenmeyer systems for developing artificial neural networks. Kitano used a kind of L-system in which he evolved matrix re-writing rules to develop an adjacency matrix defining a neural network [22]. Boers and Kuiper adapted L-systems to develop artificial feed-forward neural networks [3]. They found that this method produced more modular neural networks that performed better than networks with a predefined structure. They showed that their method could produce ANNs for solving problems such as the XOR function. Hornby and Pollack evolved L-systems to construct complex robot morphologies and neural controllers [15, 16].

Downing adopted a higher-level approach which avoided axonal and dendritic growth, while maintaining key aspects of cell signaling, competition and cooperation of neural topologies [5]. He applied this technique to the control of a multi-limbed starfish-like animat.

Khan and Miller created a complex developmental neural network model that evolved seven programs each representing various aspects of biological neurons [19]. These were divided into two categories. Three of the CGP encoded chromosomes were responsible for 'electrical' processing of the 'potentials'. These were the dendrite, soma and axo-synapse chromosomes. One chromosome was devoted to updating the weights of dendrites and axo-synapses. The remaining three chromosomes were developmental responsible for updating the neural variables for the soma (health and weight), dendrites (health, weight and length) and axo-synapse (health, length). The evolved developmental programs were responsible for the death and replication of neural components. The model was used in various applications: intelligent agent behaviour (wumpus world), checkers playing, and maze navigation [20, 21].

Stanley introduced the idea of using evolutionary algorithms to build neural networks constructively (called NEAT). The network is initialised as a simple structure, with no hidden neurons consisting of a feed-forward network of input and output neurons. An evolutionary algorithm controls the gradual complexification of the network by adding a neuron along an existing connection, or by adding a new connection between previously unconnected neurons [39]. However, using random processes to produce more complex networks is potentially very slow. It also lacks biological plausibility since natural evolution does not operate on aspects of the brain directly. Later Stanley introduced an interesting extension to the NEAT approach called HyperNEAT [41] which uses an evolved generative encoding called a Compositional Pattern Producing Network (CPPN) [40]. The CPPN takes coordinates of pairs of neurons and outputs a number which is interpreted as the weight of that connection. The advantage this brings is that ANNs can be evolved with complex patterns where collections of neurons have similar behaviour depending on their spatial location. It also means that one evolved function (the CPPN) can determine the strengths of connections of many neurons. It is a form of non-temporal development, where geometrical relationships are translated into weights.

Developmental Symbolic Encoding (DSE) [43] combines concepts from two earlier developmental encodings, Gruau's cellular encoding and L-systems. Like HyperNEAT it can specify connectivity of neurons via evolved geometric patterns. It was shown to outperform HyperNEAT on a shape recognition problem defined over small pixel arrays. It could also produce partly general solutions to a series of even-parity problems of various sizes. Huizinga et al. added an additional output to the CPP program in HyperNEAT that controlled whether or not a connection between a pair of neurons was expressed or not [17]. They showed that the new approach produced more modular solutions and superior performance to HyperNEAT on three specially devised modular problems.

Evolvable-substrate HyperNEAT (ES-HyperNEAT) implicitly defined the positions of the neurons [35], however it proved to be computationally expensive. Iterated ES-HyperNEAT proposed a more efficient way to discover suitable positioning of neurons [37]. This idea was taken further leading to Adaptive HyperNEAT which demonstrated that not only could patterns of weights be evolved but also patterns of local neural learning rules [36]. Like [17] in Adaptive HyperNEAT Risi et al. increased the number of outputs from the CPPN program to encode learning rate and other neural parameters.

8.3 The Neuron Model

Our aim is to construct a *minimal* developmental model. Minimal means that if we take a snapshot of the neural network at a particular time we would see a conventional graphs of neurons, weighted connections and a standard activation functions. However, to make a *developmental* neural network we require a mechanism whereby the ANN can change over time (possibly even during training). In addition, we take a cellular view of development, in which an entire network is developed from a few cells (possibly a single cell). The network itself grows from the interaction of neurons acting in parallel (but sequentially simulated).

To construct such a developmental model of an artificial neural network we need neural programs that not only apply a weighted sum of inputs to an activation function to determine the output from the neuron, but a program that can adjust weights, create or delete connections, and create or delete neurons. Following [21] we have used the concept of health to make this possible.

The model is illustrated in Fig. 8.1. The neural programs are represented using Cartesian Genetic Programming (CGP) (see Sect. 8.4). The programs are actually sets of mathematical equations that read variables associated with neurons and dendrites to output updates of those variables. This approach was inspired by some aspects of a developmental method for evolving graphs and circuits proposed by Miller and Thomson [32]. It was also influenced by some of the ideas described in [21]. In the proposed model, weights are determined from a program that is a function of neuron position, together with the health, weight and length of dendrites. It is neuro-centric and temporal in nature. Thus the neural networks can change over time.

The inputs to the soma program are as follows: the health, bias and position of the neuron and the average health, length and weight of all dendrites connected to the neuron and problem type.

The problem type is a constant (in range $[-1,\ 1]$) which indicates whether a neuron is not an output or in the case of an output neuron what computational problem the output neuron belongs to. Let P_t denote the computational problem. Define $P_t = 0$ to denote a non-output neuron, and P_t =1,2 or N_p to respectively denote output neurons belonging to different computational problems. Where, N_p denotes the number of computational problems. We define the problem type input to be given by $-1 + 2P_t/N_p$. For example, if the neuron is not an output neuron the problem type input is -1.0. If it is an output neuron belonging to the last problem its value is 1.0. For all other computational problems its value is a value greater than -1.0 and less than 1.0. The thinking behind the problem type input is that since output neurons are dedicated to a particular computational problem, they should be given information that relates to this, so that the identical neural programs can behave differently according to the computational problem they are associated with.

Bias refers to an input to the neuron activation function which is added to the weighted sum of inputs (i.e. it is unweighted). The soma program updates its own health, bias and position based on these inputs. These are indicated by primed symbols in Fig. 8.1). The user can decide between three different ways of using the program outputs to update the neural variables. Which is most effective is a research question. The update method is decided by a user defined parameter called Incr_{opt} (see Sect. 8.3.4) which defines how neuron variables are adjusted by the evolved programs (using user-defined incremental constants or otherwise).

Every dendrite belonging to each neuron is controlled by an evolved dendrite program. The inputs to this program are the health, weight and position of the

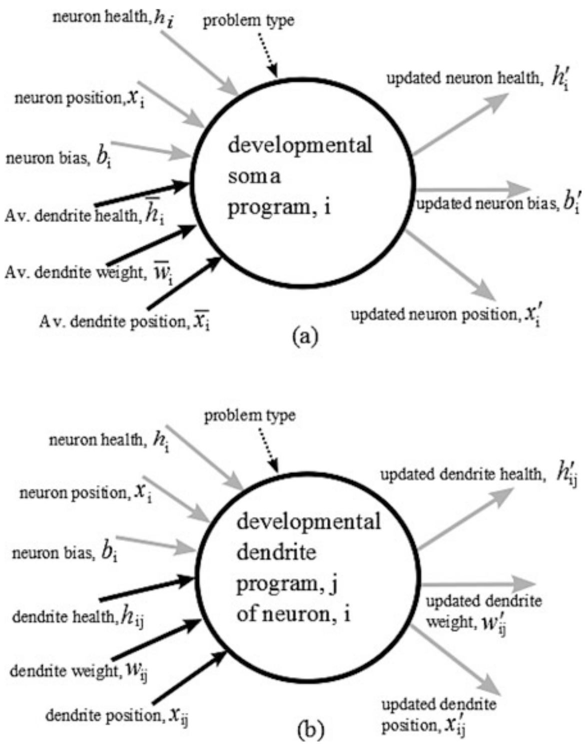

neuron health, h_i
problem type
neuron position, x_i
neuron bias, b_i
Av. dendrite health, $\bar{h}_i$
Av. dendrite weight, $\bar{w}_i$
Av. dendrite position, $\bar{x}_i$
developmental soma program, i
updated neuron health, h'_i
updated neuron bias, b'_i
updated neuron position, x'_i
(a)
problem type
neuron health, h_i
neuron position, x_i
neuron bias, b_i
dendrite health, h_{ij}
dendrite weight, w_{ij}
dendrite position, x_{ij}
developmental dendrite program, j of neuron, i
updated dendrite health, h'_{ij}
updated dendrite weight, w'_{ij}
updated dendrite position, x'_{ij}
(b)

Fig. 8.1 The model of a developmental neuron. Each neuron has a position, health and bias and a variable number of dendrites. Each dendrite has a position, health and weight. The behaviour of a neuron soma is governed by a single evolved program. In addition each dendrite is governed by another single evolved program. The soma program decides the values of new soma variables position, health and bias based on previous values, the average over all dendrites belonging to the neuron of dendrite health, position and weight and an external input called *problem type*. The latter is a floating point value that indicates the neuron type. The dendrite program updates dendrite health, position and weight based on previous values, the parent neuron's health, position and bias and problem type. When the evolved programs are executed, neurons can change, die replicate and grow more dendrites and their dendrites can also change or die

dendrite and also the health, bias and position of the parent neuron. In addition as mentioned earlier, dendrite programs can also receive the problem type of the parent neuron The the evolved dendrite program decides how the health, weight and position of the dendrite are to be updated.

In the model, all the neuron and dendrite parameters (weights, bias, health, position and problem type) are defined by numbers in the range $[-1, -1]$.

A fictitious developmental example is shown in Fig. 8.2. The initial state of the brain is represented in (a). Initially there is one non-output neuron with a single dendrite. The curved nature of the dendrites is purely for visualisation. In reality the dendrites are horizontal lines emanating from the centre of neurons and of various lengths. When extracting ANNs the dendrites are assumed to connect to their nearest

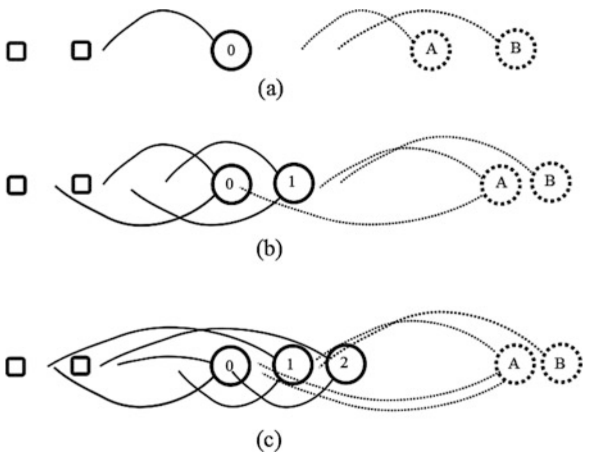


Fig. 8.2 Example showing a developing brain. The squares on the left represent the inputs. The solid circles indicate non-output neurons. Non-output neurons have solid dendrites. The dotted circles represent output neurons. Output neuron's dendrites are also dotted. In this example we assume that only output neurons are allowed to move. The neurons, inputs and dendrites are all bound to the interval [−1,1]. Dendrites connect to nearest neurons or inputs on the *left* of their position (*snapping*). (**a**) shows the initial state of the brain. (**b**) shows the brain after one developmental step and (**c**) shows it after two developmental steps

neuron on the left (referred to as 'snapping'). Output neurons are only allowed to connect to non-output neurons or the first input (by default, if their dendrites lie on the left of the leftmost non-output neuron). Thus the ANN that can be extracted from the initial brain, has three neurons. The non-output neuron is connected to the second input and both output neurons are connected via their single dendrite to the non-output neuron.

Figure 8.2b shows the brain after a single developmental step. In this step, the soma program and dendrite programs are executed in each neuron. The non-output neuron (labeled 0) has replicated to produce non-output neuron (labeled 1) it has also grown a new dendrite. Its dendrites connect to both inputs. The newly created non-output neuron is identical to its parent except that its position is a user-defined amount, MN_{inc}, to the right of the parent and its health is set to 1 (an assumption of the model). Both its dendrites connect to the second input. It is assumed that the soma programs running in the two output neurons A and B have resulted in both output neurons having moved to the right. Their dendrites have also grown in length. Neuron A's first dendrite is now connected to neuron one. In addition, neuron A has high health so that it has grown a new dendrite. Every time a new dendrite grows it is given a weight and health equal to 1.0. Also its new dendrite is given a position equal to half the parent neuron's position. These are assumptions of the model. This its new dendrite is connected to neuron zero.Neuron B's only dendrite is connected to neuron one.

Figure 8.2c shows the brain after a two developmental steps. The dendrites of neuron zero have changed little and it is still connected in the same way as the previous step. Neuron one's dendrites have both changed. The first one has become longer but remains connected to the first input. The second dendrite has become shorter but it still snaps to the second input. Neuron one has also replicated as a result of its health being above the replication threshold. It gets dendrites identical to its parent, its position is again incremented to the right of its parent and its health is set to 1.0. Its first dendrite connects to input one and its second dendrite to neuron zero. Output neuron A has gained a dendrite, due to its health being above the dendrite birth threshold. The new dendrite stretches to a position equal to half of its parent neuron. So it connects to neuron zero. The other two dendrites remain the same and they connects to neuron one and zero respectively. Finally, output neuron B's only dendrite has extended a little but still snaps to neuron one. Note, that at this stage neuron two is not connected to this is redundant. It will be stripped out of the ANN that is extracted from the brain.

8.3.1 Model Parameters

The model necessarily has a large number of user-defined parameters these are shown in Table 8.1.

The total number of neurons allowed in the network is bounded between a user-defined lower (upper)bound NN_{min} (NN_{max}). The number of dendrites each neuron

Table 8.1 Neural model constants and their meanings

Symbol	Meaning
$NN_{min}(NN_{max})$	Min. (Max.) allowed number of neurons
N_{init}	Initial number of non-output neurons
$DN_{min}(DN_{max})$	Min. (Max.) number of dendrites per neuron
ND_{init}	Initial number of dendrites per neuron
$NH_{death}(NH_{birth})$	Neuron health thresholds for death (birth)
$DH_{death}(DH_{birth})$	Dendrite health thresholds for death (birth)
δ_{sh}	Soma health increment (pre, while)
δ_{sp}	Soma position increment (pre, while)
δ_{sb}	Soma bias increment (pre, while)
δ_{dh}	Dendrite health increment (pre, while)
δ_{dp}	Dendrite position increment (pre, while)
δ_{dw}	Dendrite weight increment (pre, while)
NDS_{pre}	Number of developmental steps before epoch
NDS_{whi}	Number of 'while' developmental steps during epoch
N_{ep}	Number of learning epochs
MN_{inc}	Move neuron increment if collision
I_u	Max. program input position
O_l	Min. program output position
α	Sigmoid/Hyperbolic tangent exponent constant

can have is bounded by user-defined lower (upper) bounds denoted by DN_{min} (DN_{max}). These parameters ensure that the number of neurons and connections per neuron remain in well-defined bounds, so that a network can not eliminate itself or grow too large. The initial number of neurons is defined by N_{init} and the initial number of dendrites per neuron is given by ND_{init}.

If the health of a neuron falls below (exceeds) a user-defined threshold, NH_{death} (NH_{birth}) the neuron will be deleted (replicated). Likewise, dendrites are subject to user defined health thresholds, DH_{death} (DH_{birth}) which determine whether the dendrite will be deleted or a new one will be created. Actually, to determine dendrite birth the parent *neuron* health is compared with DH_{birth} rather than dendrite health. This choice was made to prevent the potential very rapid growth of dendrite numbers.

When the soma or dendrite programs are run the outputs are used to decide how to adjust the neural and dendrite variables. The amount of the adjustments are decided by the six user-defined δ parameters.

The number of developmental steps in the two developmental phases (‘pre’ learning and ‘while’ learning) are defined by the parameters, NDS_{pre} and NDS_{whi}. The number of learning epochs is defined by N_{ep}. Note that the pre-learning phase of development, ‘pre’, can have different incremental constants (i.e. δs) to the learning epoch phase, ‘while’.

In some cases, neurons will collide with other neurons (by occupying a small interval around an existing neuron) and the neuron has to be moved by a certain increment until no more collisions take place. This increment is given by MN_{inc}.

The places where external inputs are provided is predetermined uniformly within the region between -1 and I_u. The parameter I_u defines the upper bound of their position. Also output neurons are initially uniformly distributed between the parameter O_l and 1. However, depending on a user-defined option the output neurons as with other neurons can move according to the neuron program. All neurons are marked as to whether they provide an external output or not. In the initial network there are N_{init} non-output neurons and N_o output neurons, where N_o denotes the number of outputs required by the computational problem being solved.

Finally, the neural activation function (hyperbolic tangent) and the sigmoid function (which is used in nonlinear incremental adjustment of neural variables) have a slope constant given by α.

8.3.2 Developing the Brain and Evaluating the Fitness

An overview of the algorithm used for training and developing the ANNs is given in Overview 3.

The brain is always initialised with at least as many neurons as the maximum number of outputs over all computational problems. Note, all problem outputs are represented by a unique neuron dedicated to the particular output. However, the

Overview 3 Overview of fitness algorithm

```
 1: function FITNESS
 2:     Initialise brain
 3:     Load ‘pre’ development parameters
 4:     Update brain $NDS_{pre}$ times by running soma and dendrite programs
 5:     Load ‘while’ developmental parameters
 6:     repeat
 7:         Update brain $NDS_{whi}$ times by running soma and dendrite programs
 8:         Extract ANN for each benchmark problem
 9:         Apply training inputs and calculate accuracy for each problem
10:         Fitness is the normalised average accuracy over problems
11:         If fitness reduces terminate learning loop and return previous fitness
12:     until $N_{ep}$ epochs complete
13:     return fitness
14: end function
```

maximum and initial number of non-output neurons can be chosen by the user. Non-output neurons can grow change or give birth to new dendrites. Output neurons can change but not die or replicate as the number of output neurons is fixed by the choice of computational problems. The detailed algorithm for training and developing the ANN is given in Algorithm 1.

8.3.3 Updating the Brain

Updating the brain is the process of running the soma and dendrite programs once in all neurons and dendrites (i.e. it is a single developmental step). Doing this will cause the brain to change and after all changes have been carried out a new updated brain will be produced. This replaces the previous brain. Overview Algorithm 4 gives a high-level overview of the update brain process.

Overview 4 Update brain overview

```
 1: function UPDATEBRAIN
 2:     Run soma program in *non-output* neurons to update soma
 3:     Ensure neuron does not collide with neuron in updated brain
 4:     Run dendrite program in all *non-output* neurons
 5:     If neuron survives add it to updated brain
 6:     If neuron replicates ensure new neuron does not collide
 7:     Add new neuron to updated brain
 8:     Run soma program in *output* neurons to update soma
 9:     Ensure neuron does not collide
10:     Run dendrite program in all *output* neurons
11:     If neuron survives add it to updated brain
12:     Replace old brain with updated brain
13: end function
```

Section 8.13 presents a more detailed version of how the brain is updated at each developmental step (see Algorithm 2) and gives details of the neuron collision avoidance algorithm.

8.3.4 Running and Updating the Soma

The UPDATEBRAIN program calls the RUNSOMA program to determine how the soma changes in each developmental step. As we saw in Fig. 8.1a the seven soma program inputs are: neuron health, position and bias, the averaged position, weight and health of the neuron's dendrites and the problem type. Once the evolved CGP soma program is run the soma outputs are returned to the brain update program. These steps are shown in Overview 2.

Overview 2 Running the soma: algorithm overview

```
1: function RUNSOMA
2:     Calculate average dendrite health, position and weight
3:     Gather soma program inputs
4:     Run soma program
5:     Return updated soma heath, bias and position
6: end function
```

The detailed version of the RUNSOMA function can be found in Sect. 8.13. The RUNSOMA function uses the soma program outputs to adjust the health, position and bias of the soma according to three user-chosen options defined by a variable Incr_{opt}.This is carried out by the UPDATENEURON overview Algorithm 3.

Overview 3 Update neuron algorithm overview

```
1: function UPDATENEURON
2:     Assign original neuron variables to parent variables
3:     Assign outputs of soma program to health, position and bias
4:     Depending on $\text{Incr}_{opt}$ get increments
5:     If soma program outputs $> 0$ ($\leq 0$) then incr(decr.) parent variables
6:     Assign parent variables to neuron
7:     Bound health, position and bias
8: end function
```

8.3.5 Updating the Dendrites and Building the New Neuron

This section is concerned with running the evolved dendrite programs. In every dendrite, the inputs to the dendrite program have to be gathered. The dendrite program is executed and the outputs are used to update the dendrite. This is carried out by a function called RunDendrite. Note, in RunAllDendrites we build the completely updated neuron from the updated soma and dendrite variables. The simplified algorithm for doing this is shown in overview Algorithm 4. The more detailed version is available in Sect. 8.13.

Overview 4 An overview of the RunAllDendrites algorithm which runs all dendrite programs and uses all updated variables to build a new neuron

```
1:  function RunAllDendrites
2:      Write updated soma variables to new neuron
3:      if Old soma health > $DH_{birth}$ then
4:          Generate a dendrite for new neuron
5:      end if
6:      for all Dendrites do
7:          Gather dendrite program inputs
8:          Run dendrite program to get updated dendrite variables
9:          Run dendrite to get updated dendrite
10:         if Updated dendrite is alive then
11:             Add updated dendrite to new neuron
12:             if Maximum number of dendrites reached then
13:                 Stop processing any more dendrites
14:             end if
15:         end if
16:     end for
17:     if All dendrites have died then
18:         Give new neuron the first dendrite of the old neuron
19:     end if
20: end function
```

Overview Algorithm 4 (in line 9) uses the updated dendrite variables obtained from running the evolved dendrite program to adjust the dendrite variables (according to the incrementation option chosen). This function is shown in the overview Algorithm 5. The more detailed version is available in Sect. 8.13.

The RunDendrite function begins by assigning the dendrite's health, position and weight to the parent dendrite variables. It writes the dendrite program outputs to the internal variables health, weight and position. It respectively carries out the increments or decrements of the parent dendrite variables according whether the corresponding dendrite program outputs are greater than or less than or equal to zero. After this it bounds those variables. Finally, it updates the dendrites health, weight and position provided the adjusted health is above the dendrite death threshold.

We saw in the fitness function that we extract conventional ANNs from the evolved brain. The way this is accomplished is as follows.

Since we share inputs across problems we set the number of inputs to be the maximum number of inputs that occur in the computational problem suite. If any problem has less inputs the extra inputs are set to zero.

The next phase is to go through all dendrites of the neurons to determine which inputs or neurons they connect to. To generate a valid neural network we assume

Overview 5 Change dendrites according to the evolved dendrite program

```
 1: function RUNDENDRITE
 2:     Assign original dendrite variables to parent variables
 3:     Assign outputs of dendrite program to health, position and weight
 4:     Depending on $\mathrm{Incr}_{opt}$ get increments
 5:     If dendrite program outputs $> 0$ ($\leq 0$) then incr(decr.) parent variables
 6:     Assign parent variables to neuron
 7:     Bound health, position and weight
 8:     if (health $> DH_{death}$) then
 9:         Update dendrite variables
10:         Dendrite is alive
11:     else
12:         Dendrite is dead
13:     end if
14:     Return updated dendrite variables and whether dendrite is alive
15: end function
```

that dendrites are automatically connected to the nearest neuron or input on the left. We refer to this as "snapping". The dendrites of non-output neurons are allowed to connect to either inputs or other non-output neurons on their left. However, output neurons are only allowed to connect to non-output neurons on their left. It is not desirable for the dendrites of output neurons to be connected directly to inputs, however, when output neurons are allowed to move, they may only have inputs on their left. In this case the output neuron dendrite neuron will be connected to the first external input to the ANN network (by default).

The detailed version of the ANN extraction process is given in Sect. 8.13.

8.4 Cartesian GP

The two neural programs are represented and evolved using a form of Genetic Programming (GP) known as Cartesian Genetic Programming (CGP). CGP [28, 31] is a form of GP in which computational structures are represented as directed, often acyclic graphs indexed by their Cartesian coordinates. Each node may take its inputs from any previous node or program input (although recurrent graphs can also be implemented see [45]). The program outputs are taken from the output of any internal node or program input. In practice, many of the nodes described by the CGP chromosome are not involved in the chain of connections from program input to program output. Thus, they do not contribute to the final operation of the encoded program, these inactive, or "junk", nodes have been shown to greatly aid the evolutionary search [30, 46, 47]. The representational feature of inactive genes in CGP is also closely related to the fact that it does not suffer from bloat [27].

In general, the nodes described by CGP chromosomes are arranged in a rectangular $r \times c$ grid of nodes, where r and c respectively denote the user-defined number of rows and columns. In CGP, nodes in the same column are not allowed to be connected together. CGP also has a connectivity parameter l called "levels-back" which determines whether a node in a particular column can connect to a node in a previous column. For instance if $l = 1$ all nodes in a column can only connect to nodes in the previous column. Note that levels-back only restricts the connectivity of nodes; it does not restrict whether nodes can be connected to program inputs (terminals). However, it is quite common to adopt a linear CGP configuration in which $r = 1$ and $l = c$. This was done in our investigations here. CGP chromosomes can describe multiple input multiple output (MIMO) programs with a range of node functions and arities. For a detailed description of CGP, including its current developments and applications, see [28]. Both the soma and dendrite program have

Table 8.2 Node function gene values, mnemonic and function definition

Value	Mnemonic	Definition
0	abs	$\lvert z_0 \rvert$
1	sqrt	$\sqrt{\lvert z_0 \rvert}$
2	sqr	${z_0}^2$
3	cube	${z_0}^3$
4	exp	$(2exp(z_0 + 1) - e^2 - 1)/(e^2 - 1)$
5	sin	$\sin(z_0)$
6	cos	$\cos(z_0)$
7	tanh	$\tanh(z_0)$
7	inv	$-z_0$
9	step	**if** $z_0 < 0.0$ **then** 0 **else** 1.0
10	hyp	$\sqrt{({z_0}^2 + {z_1}^2)/2}$
11	add	$(z_0 + z_1)/2$
12	sub	$(z_0 - z_1)/2$
13	mult	$z_0 z_1$
14	max	**if** $z_0 >= z_1$ **then** z_0 **else** z_1
15	min	**if** $z_0 <= z_1$ **then** z_0 **else** z_1
16	and	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** 1.0 **else** -1.0
17	or	**if** $(z_0 > 0.0$ **or** $z_1 > 0.0)$ **then** 1.0 **else** -1.0
18	rmux	**if** $z_2 > 0.0$ **then** z_0 **else** z_1
19	imult	$-z_0 z_1$
20	xor	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** -1.0 **else if** $(z_0 < 0.0$ **and** $z_1 < 0.0)$ **then** -1.0 **else** 1.0
21	istep	**if** $z_0 < 1.0$ **then** 0 **else** -1.0
22	tand	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** 1.0 **else if** $(z_0 < 0.0$ **and** $z_1 < 0.0)$ **then** -1.0 **else** 0.0
23	tor	**if** $(z_0 > 0.0$ **or** $z_1 > 0.0)$ **then** 1.0 **else if** $(z_0 < 0.0$ **or** $z_1 < 0.0)$ **then** -1.0 **else** 0.0

7 inputs and 3 outputs. (see Fig. 8.1). The function set chosen for this study are defined over the real-valued interval [−1.0, 1.0]. Each primitive function takes up to three inputs, denoted z_0, z_1 and z_2. The functions are defined in Table 8.2.

8.5 Benchmark Problems

In this study, we evolve neural programs that build ANNs for solving three standard classification problems. The problems are cancer, diabetes and glass. The definitions of these problems are available in the well-known UCI repository of machine learning problems.[1] These three problems were chosen because they are well-studied and also have similar numbers of inputs and a small number of classes. Cancer has nine real attributes and two Boolean classes. Diabetes has eight real attributes and two Boolean classes. Glass has nine real attributes and six Boolean classes. The specific datasets chosen were cancer1.dt, diabetes1.dt and glass1.dt which are described in the PROBEN suite of problems.[2] Since, for each benchmark problem we extract an ANN the order of presentation of the benchmark problems is unimportant.

8.6 Experiments and Results

The long-term aim of this research is to explore effective ways to *develop* ANNs. The work presented here is a just a beginning and there are many aspects that need to be investigated in the future (see Sect. 8.12). The specific research questions we have focused on are:

- Are multiple learning epochs more effective than a single epoch?
- Should neurons be allowed to move?
- Should evolved program outputs update neural variables directly or should they determine user-defined increments in those variables (linear or non-linear)?

To answer these questions a series of experiments were carried out to investigate the impact of various aspects of the neural model on classification accuracy. Twenty evolutionary runs of 20,000 generations of a 1+5-ES were used. Genotype lengths for soma and dendrite programs were chosen to be 800 nodes. Goldman mutation [10, 11] was used which carries out random point mutation until an active gene is changed. For these experiments a subset of allowed node functions were chosen as they appeared to give better performance. These were: step, add, sub, mult, xor, istep. The remaining experimental parameters are shown in Table 8.3.

[1]https://archive.ics.uci.edu/ml/datasets.html.

[2]https://publikationen.bibliothek.kit.edu.

Table 8.3 Table of neural model parameters.

Parameter	Value	Value
$NN_{min}(NN_{max})$	0 (20)	
N_{init}	5	
$DN_{min}(DN_{max})$	1 (40)	
ND_{init}	5	
NDS_{pre}	8	
NDS_{whi}	3	
NDS_{aft}	0	
N_{ep}	1	
MN_{inc}	0.03	
I_u	-0.6	
O_l	0.8	
α	1.5	
Development parameters	'Pre'	'While'
$NH_{death}(NH_{birth})$	-0.6 (0.308)	-0.58 (0.8)
$DH_{death}(DH_{birth})$	-0.404772 (-0.2012)	-0.38 (0.85)
δ_{sh}	0.1	0.01
δ_{sp}	0.1	0.01
δ_{sb}	0.07	0.0402
δ_{dh}	0.1	0.01
δ_{dp}	0.2032	0.01
δ_{dw}	0.1	0.02029

Four types of experiments were carried out to investigate the utility of neuron movement. Acronyms describe these experiments. AMA means all neuron movement was allowed (both non-output and output neurons). OMA means only the movement of output neurons is allowed. NOMA means only the movement of non-output neurons is allowed and finally, AMD means all movement of neurons is disallowed. In addition, we examined three ways of incrementing or decrementing neural variables. In the first the outputs of evolved programs determines directly the new values of neural variables (position, health, bias, weight), that is to say there is no incremental adjustment of neural variables. In the second, the variables are incremented or decremented in user-defined amounts (the deltas in Table 8.1). In the third, the adjustments to the neural variables are nonlinear (they are adjusted using a sigmoid function). It should be noted that the scenario AMD does not imply that all neurons remain in the fixed positions that they were initially given. The collision avoidance mechanism and the birth of new neurons means that neurons will be assigned different positions during development. However, the neuron positions can not be adjusted by incrementing neuron position.

Table 8.4 Training and testing accuracy for various neuron movement scenarios

Acc.	AMA train (test)	OMA train (test)	NOMA train (test)	AMD train (test)
Mean	0.7093 (0.6959)	0.7456 (0.7206)	0.6929 (0.6803)	0.6920 (0.6821)
Median	0.7066 (0.7020)	0.7481 (0.7329)	0.6886 (0.6954)	0.6929 (0.6950)
Maximum	0.7598 (0.7539)	0.7854 (0.7740)	0.7617 (0.7643)	0.7363 (0.7245)
Minimum	0.6627 (0.6275)	0.7022 (0.6498)	0.6254 (0.6028)	0.6575 (0.6089)

All neurons allowed to move (AMA), only output neurons are allowed to move (OMA), only non-output neurons are allowed to move (NOMA) and no neurons are allowed to move (AMD)

Table 8.5 Training and testing accuracy on individual problems when only output neurons are allowed to move

Acc.	Cancer train (test)	Diabetes train (test)	Glass train (test)
Mean	0.9397 (0.9534)	0.7094 (0.6622)	0.5879 (0.5462)
Median	0.9471 (0.9598)	0.7031 (0.6510)	0.5888 (0.5849)
Maximum	0.9657 (0.9942)	0.7526 (0.7500)	0.6636 (0.6415)
Minimum	0.8771 (0.8391)	0.6693 (0.6094)	0.4766 (0.3774)

Table 8.6 Comparison of test accuracies on three classification problems

Acc.	Cancer ML (OMA)	Diabetes ML (OMA)	Glass ML (OMA)
Mean	0.935 (0.9534)	0.743 (0.6622)	0.610 (0.5462)
Maximum	0.974 (0.9942)	0.790 (0.7500)	0.785 (0.6415)
Minimum	0.655 (0.8391)	0.582 (0.6094)	0.319 (0.3774)

OMA compared with huge suite of classification methods as described in [8]

8.7 Tables of Results

The mean, median, maximum and minimum accuracies achieved over 20 evolutionary runs when all neurons are allowed to move are shown in Table 8.4. We can see that the best values of mean, median, maximum and minimum are all obtained when only output neurons are allowed to move. The mean, median, maximum and minimum are shown for each individual problem (cancer, diabetes and glass) in Table 8.5.

Table 8.6 shows how the results for OMA compare with the performance of 179 classifiers (covering 17 families) [8].[3] The figures are given just to show that the results for the developmental ANNs are respectable and are especially encouraging considering that the evolved developmental programs build classifiers for three different classification problems simultaneously.

[3]The paper gives a link to the detailed performance of the 179 classifiers which contain the figures given in the table.

8.8 Comparisons and Statistical Significance

The results for the four experimental scenarios are presented graphically in Figs. 8.3 and 8.4. Maximum outliers are shown as asterisks and minimum outliers as filled squares. The ends of the whiskers are set at 1.5*IQR above the third quartile and at 1.5*IQR below the first quartile, where IQR is the inter quartile range (Q3–Q1). Clearly, the figures show that allowing only the output neurons to move (OMA) produces the best results both on the training data set and the test data set. Also, in this scenario there is a high level of generalisation as the results on the unseen data set are close to the training results.

The Wilcoxon Ranked-Sum test (WRS) was used to assess the statistical difference between pairs of experiments. In this test, the null hypothesis is that the results (best accuracy) over the multiple runs for the two different experimental conditions are drawn from the same distribution and have the same median. If there is a statistically significant difference between the two then null hypothesis is false

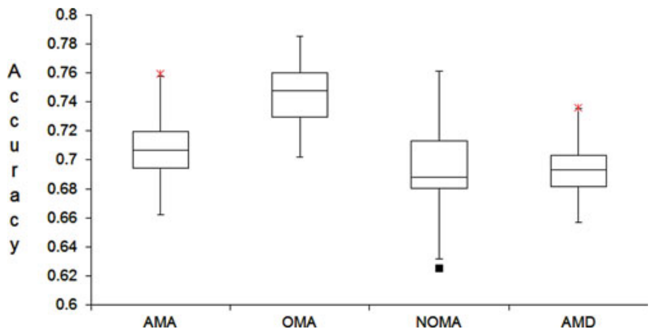


Fig. 8.3 Results for four experiments which allow or disallow neurons to move. The four neuron movement scenarios are: all neurons allowed to move (AMA), only output neurons are allowed to move (OMA), only non-output neurons are allowed to move (NOMA) and no neurons are allowed to move (AMD). The figure shows classification accuracy on training set

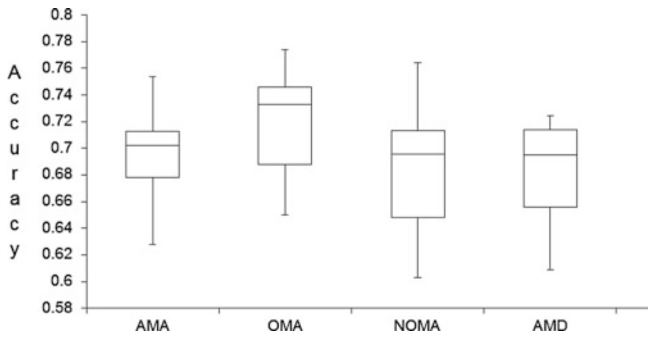


Fig. 8.4 Results on test set for four experiments which allow or disallow neurons to move

Table 8.7 Statistical comparison of *training* results from experiments (Wilcoxon Rank-Sum two-tailed)

Question	Expt. A	Expt. B	W	W critical	P-value	Significant
Output movement v. no. movement?	OMA	AMD	0	21	$p < 0.001$	Very
Output movement v. non-output movement?	OMA	NOMA	1	21	$p < 0.001$	Very
Output movement v. all movement?	OMA	AMA	33	37	$0.005 < p < 0.01$	Yes
Linear v. nonlinear incr.	OMA	OMA-non-lin	19	37	$p < 0.001$	Very
Non-linear v. no increment?	OMA-non-lin	OMA no incr.	54	69	$0.05 < p < 0.1$	Weakly

with a degree of certainty which depends on the smallness of a calculated statistic called a p-value. However, in the WRS before interpreting the p-value one needs to calculate another statistic called Wilcoxon's W value. This value needs to be compared with calculated values which depend on the number of samples in each experiment. Results are statistically significant when the calculated W-value is less than or equal to certain critical values for W[48]. The critical values depend on the sample sizes and the p-value. We used a publicly available Excel spreadsheet for doing these calculations.[4] The critical W-values can be calculated in two ways: one-tailed or two-tailed. The two-tailed test is appropriate here as we are interested in whether one experiment is better than another (and vice versa).

For example, in Table 8.7 the calculated W-value is 0 and the critical W-value for the paired sample sizes of 20 (number of runs) with p-value less than 0.001 is 21 (assuming a two-tailed test).[5] The p-value gives a measure of the certainty with which the null hypothesis can be accepted. Thus the lower the value the more likely that the two samples come from different distributions (i.e. are statistically different). Thus in this case, the probability that the null hypothesis can be rejected is 0.999.

The results of these tests are shown in Tables 8.7 and 8.8. Comparing OMA with AMD shows that there is an large advantage to allowing output neurons to move. Indeed, allowing only output neurons to move is statistically significantly better than either only allowing non-outputs neurons to move (NOMA) or allowing all neurons to move (AMA). This is true both for testing and training. However, as expected the differences are even more significant in training than testing. Also using a linear increment is statistically significantly better than using a nonlinear increment. Nonlinear increment is only marginally better than no increment (i.e. $Incr_{opt}$=0).

[4]http://www.biostathandbook.com/wilcoxonsignedrank.html.

[5]http://www.real-statistics.com/statistics-tables/wilcoxon-signed-ranks-table/.

Table 8.8 Statistical comparison of *test* results from experiments (Wilcoxon Rank-Sum two-tailed)

Question	Expt. A	Expt. B	W	W critical critical	P-value	Significant
Output movement v. no movement?	OMA	AMD	35	37	$0.005 < p < 0.01$	Yes
Output movement v. non-output movement?	OMA	NOMA	44	52	$0.02 < p < 0.05$	Yes
Output movement v. all movement?	OMA	AMA	53	60	$0.05 < p < 0.1$	Yes

It is important to understand how the experimental parameters shown in Table 8.3 were discovered. They were found by carrying out many experiments in the *OMA scenario*. It turned out that when small changes in parameters had a clear improvement on the quality of the first evolutionary run they significantly improved the average performance over all twenty runs.[6] This was fortuitous in that one could investigate the effect of changing parameters quickly by observing the performance of the first evolutionary run. However, it is possible that the parameters found were particular to the OMA scenario and that a similar process of tuning in the other scenarios would probably improve the results in those scenarios. Ideally, one would tune the parameters in each scenario (OMA, AMA, etc.) and compare results for the best parameter set for each scenario. This would be computationally prohibitive.

8.9 Evolved Developmental Programs

The average number of *active* nodes in the soma and dendrite programs for the OMA experiments was respectively, 56.75 and 55.0. Thus the programs are relatively simple. It is also possible that the graphs can be logically reduced to even simpler forms. The graphs of the active nodes in the CGP graphs for the best evolutionary run in the OMA scenario are shown in Figs. 8.5 and 8.6. The red input connections between nodes indicate the first input in the subtraction operation. This is the only node operation where node input order is important.

8.10 Developed ANNs for Each Classification Problem

The ANNs for the evolutionary run with only output neuron movement allowed were extracted (using Algorithm 9) and can be seen in Figs. 8.7, 8.8 and 8.9. The plots ignore connections with weight equal to zero. The average training accuracy

[6]Indeed, sometimes adjustments to a parameter in the fourth decimal place had a significant effect.

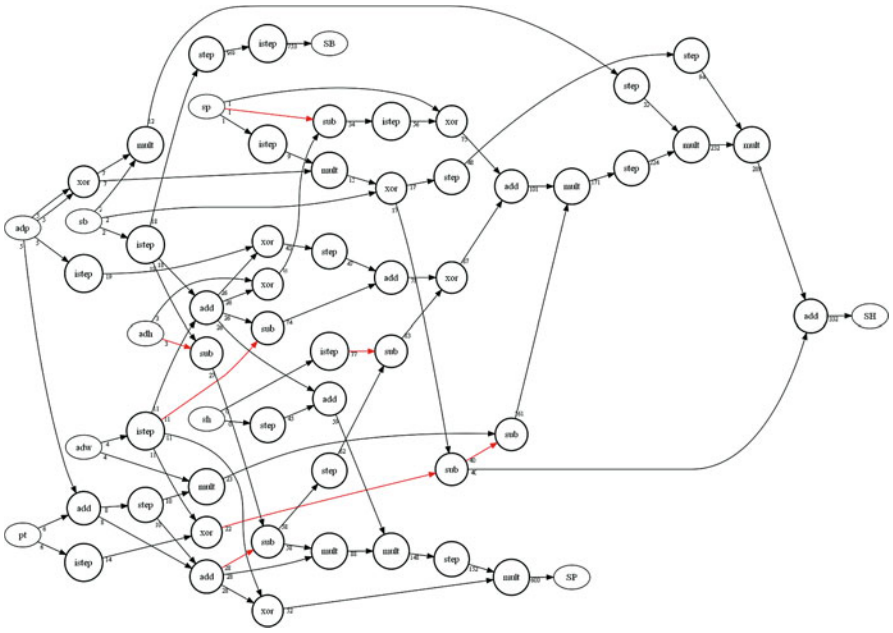


Fig. 8.5 Best evolved soma program when only output neurons can move. The input nodes are: soma heath (sh), soma bias (sb), soma position (sp), average dendrite health (adh), average dendrite weight (adw), average dendrite position (adp) and problem type (pt). The output nodes are: soma health (SH), soma bias (SB) and soma position (SP)

of these three networks is 0.78538 (best) and the average test accuracy is 0.7459. In the figures, each neuron is labeled with the problems it belongs to (cancer, diabetes, glass) and it is also labeled with the neuron ID (in blue). Using these labels makes it easy to identify which neurons are shared between problems. Note that output neurons are not allowed to be shared.

The ANN which classifies the cancer data has 6 neurons with IDs: 9, 10, 12, 13, 19, 20. Neurons 9 and 13 are shared with the glass ANN, Neurons 10 and 12 are shared over all three problems. Class 0 of the cancer dataset is provided by a simple function. The first attribute in the dataset is multiplied by a single weight and then is the only input to a biased tanh neuron.

The ANN classifier for the diabetes dataset has 5 neurons with IDs: 10, 12, 15, 21, 27. Neurons with IDs 10 and 12 are shared across all problems. Neuron 15 is shared with glass ANN.

The ANN classifier for the glass dataset has 15 neurons with IDs: 9, 10, 11, 12, 13, 14, 15, 16, 18, 22, 23, 24, 25, 26, 28. There are 4 neurons that are shared.

An interesting feature is that pairs of neurons often have multiple connections. This is equivalent to a single connection where the weighted value is the sum of the individual connections weights. This phenomenon was also observed in CGP encoded and evolved ANNs [44].

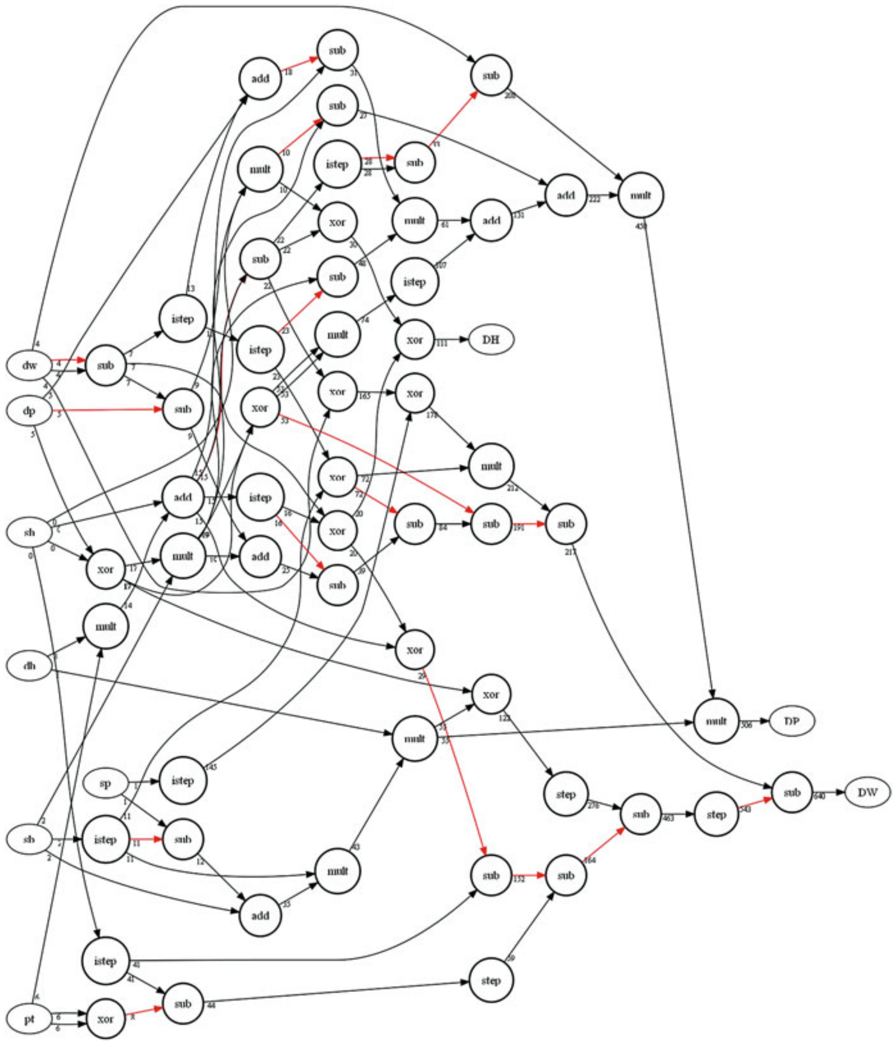


Fig. 8.6 Best evolved dendrite program when only output neurons can move. The input nodes are: soma heath (sh), soma bias (sb), soma position (sp), dendrite health (dh), dendrite weight (dw), dendrite position (dp) and problem type (pt). The output nodes are: dendrite health (DH), dendrite weight (DW) and dendrite position (DP)

8.11 Evolving Neural Learning Programs

The fitness function (see overview Algorithm 3) included the possibility of learning epochs. In this section we present and discuss results when a number of learning epochs have been chosen. The task for evolution is then to construct two neural programs that develop ANNs that improve with each learning epoch. The aim

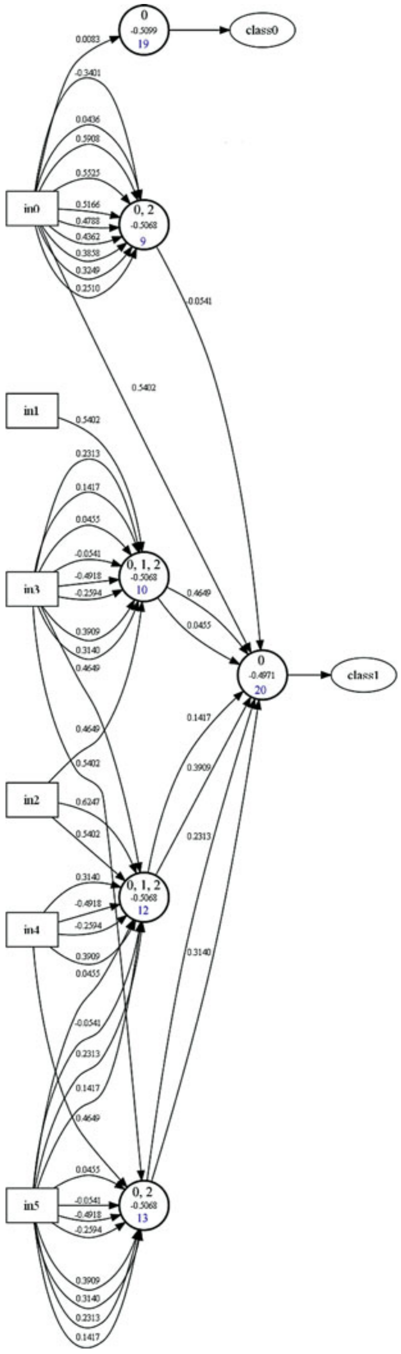


Fig. 8.7 Developed ANN for cancer dataset. This dataset has nine attributes and two outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. If any attributes are not present it means they are unused. The training accuracy is 0.9571 and the test accuracy is 0.9828

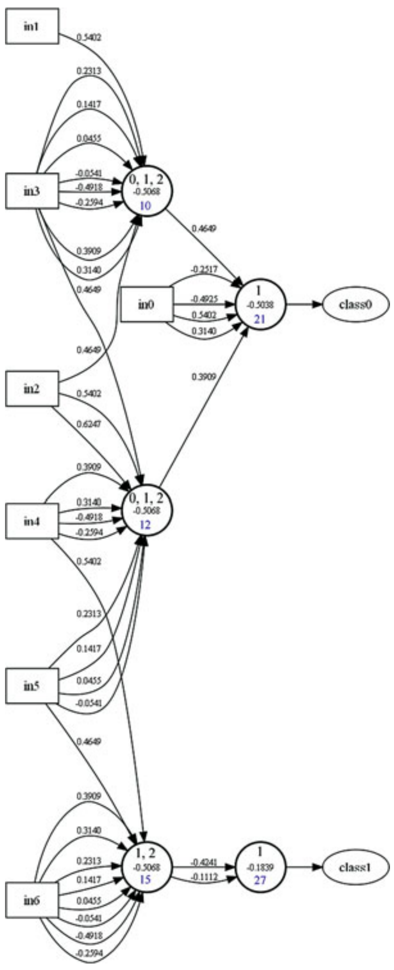


Fig. 8.8 Developed ANN for diabetes dataset.This dataset has eight attributes and two outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. Attributes not present are unused. The training accuracy is 0.7448 and the test accuracy is 0.6510

is to find a general learning algorithm in which the ANNs change and improve with each learning epoch beyond the limited number of epochs used in training. The experimental parameters required to investigate were changed from those used previously when there were no learning epochs. For the experiments here we retained most of the previous parameters but altered the number of learning epochs and the delta parameters in the 'while' loop. The new parameters are shown in Table 8.9.

Informal experiments were undertaken to investigate suitable neural parameters when using multiple learning epochs. The best results appeared to be obtained when only the dendrite length was incrementally adjusted. As before, results are quite sensitive to exact values for these parameters. It is interesting and surprising to observe that adjustment of weights only in while phase does not produce as good

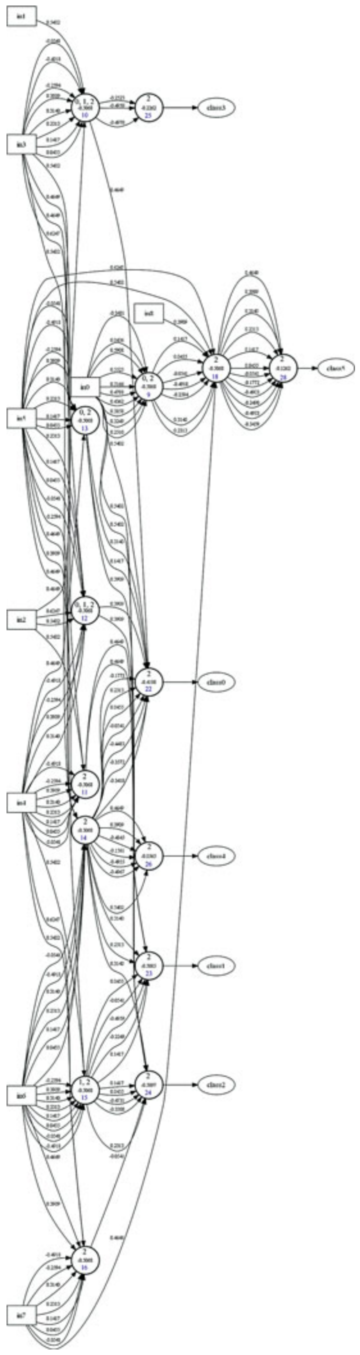


Fig. 8.9 Developed ANN for glass dataset. This dataset has nine attributes and six outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. Attributes not present are unused. The training accuracy is 0.6542 and the test accuracy is 0.6038

Table 8.9 Changed neural model parameters when using multiple learning epochs

Parameter	Value
N_{ep}	10
δ_{sh}	0.0
δ_{sp}	0.0
δ_{sb}	0.0
δ_{dh}	0.0
δ_{dp}	0.00106
δ_{dw}	0.0

Table 8.10 Training and testing accuracy for ten learning epochs versus no learning epochs when only output neurons are allowed to move

Acc.	Learning epochs train (test)	No learning epochs train (test)
Mean	0.7226 (0.6482)	0.7456 (0.7206)
Median	0.7298 (0.6767)	0.7481 (0.7329)
Maximum	0.7609 (0.7864)	0.7854 (0.7740)
Minimum	0.6613 (0.3919)	0.7022 (0.6498)

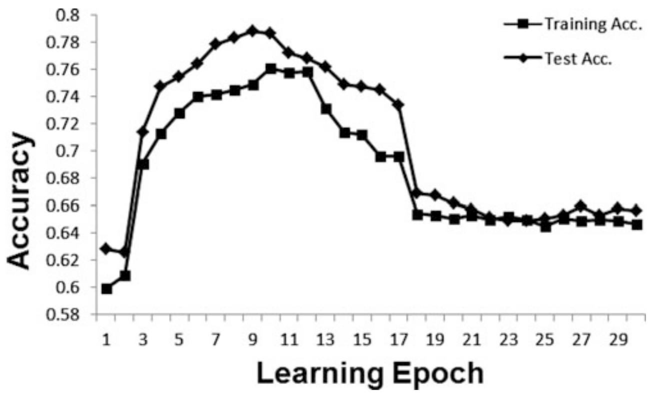


Fig. 8.10 Variation of classification accuracy for training and testing with learning epoch when only output movement is allowed

results as adjusting dendrite length. Twenty evolutionary runs were carried out using these parameters and the results are shown in Table 8.10.

In Table 8.10 we compare the results with multiple learning epochs with no learning epochs. Using no learning epochs gives better results. However, the results with multiple learning epochs is reasonable despite the fact that the task is much more difficult, effectively one is to trying evolve a *learning algorithm*. It is possible that further experimentation with developmental parameters could produce better results with multiple epochs.

In Fig. 8.10 we examine how the accuracy of the classifications varies with learning epochs. We set the maximum number of epochs to 30 now to see if learning continues beyond the upper limit used during evolution (10). We can see that both the test and training classification accuracy increases with each epoch up to 10 epochs and there is a gradual decline in accuracy after this point. However, at 18 epochs the accuracy stabilises to an accuracy of 0.65. Interestingly, the test accuracy is always better than the training accuracy. We obtained several evolved solutions in which training accuracy increased at each epoch until the imposed maximum number of epochs, however, as yet none of these were able to improve beyond the limit.

8.12 Further Work

It remains unclear why better results can not at present be obtained when evolving developmental programs with multiple epochs. Neither is it clear why programs can be evolved that continuously improve the developed ANNs over a number of epochs (i.e. 10) yet do not improve subsequently. It is worth contrasting the model discussed in this chapter with previous work on Self-Modifying CGP [14]. In SMCGP phenotypes can be iterated to produce a sequence of programs or phenotypes. The fitness was *accumulated* over all the correct test cases summed over all the iterations. In the problems studied (i.e. even-n parity, producing π) there was also a notion of perfection. For instance in the parity case perfection meant that at each iteration it produced the next parity case (with more inputs) perfectly. If at the next iteration, the appropriate parity function was not produced, then the iteration stopped. In the work discussed here, the fitness is not cumulative. At each epoch, the fitness is the average accuracy of the classifiers over the three classification problems. If the fitness reduces at the next epoch, then the epoch loop is terminated. However, in principle, we could sum the accuracies at each epoch and if an accuracy at a particular epoch is reduced, terminate the epoch loop. Summing the accuracies would give reward to developmental programs that produced the best *history* of developmental changes.

At present, the developmental programs do not receive a reward signal during multiple epochs. This means that the task for evolution is to continuously improve developed ANNs without being supplied with a reward signal. However, one would expect that as the fitness increases at each epoch the number of changes that need to be made to the developed ANNs should decrease. This suggests that supplying the fitness at the previous epoch to the developmental programs might be useful. In fact this option has already been implemented but as yet evidence is inconclusive that this produces improved results.

While learning over multiple epochs, we have assumed that the developmental parameters should be fixed (i.e. they are chosen before the development loop—see line 5 of Overview Algorithm 3). However, it is not clear that this should be so. One could argue that during early learning topological changes in the brain network are more important and weight changes more important in later phases of learning. This suggests that at each step of the learning loop one could load developmental parameters, this would allow control of each epoch of learning.

The neural variables that are given as inputs to the CGP developmental programs are an assumption of the model. For the soma these are: health, bias, position, problem type and average dendrite health, position and weight. For the dendrite they are: dendrite health, weight, position, problem type and soma health, bias and position. Further experimental work needs to be undertaken to determine whether they all are useful. The program written already has the inclusion of any of these variables as an option.

There are also many assumptions made in quite small aspects of the whole model. When new neurons or dendrites are born what should the initial values of the neural variables be? What are the best upper bounds on the number of neurons and dendrites? Currently, dendrite replication is decided by comparing the parent *neuron* health with DH_{birth} rather than comparing dendrite health with this threshold. If dendrite health was compared with a threshold it could rapidly lead to very large numbers of dendrites. Many choices have been made that need to be investigated in more detail.

There are also very many parameters in the model and experiment has shown that results can be very sensitive to these. Thus further experimentation is required to identify good choices for these parameters.

A fundamental issue is how to handle inputs. In the classification problems the number of inputs is given by the problem with the most attributes, problems with less are given the value zero for those inputs. This could be awkward if the problems have hugely varying numbers of inputs. Is there another way of handling this? Perhaps one could borrow more ideas from SMCGP and make all input connections access inputs using pointer to a circular register of inputs. Every time a neuron connected to an input, a global pointer to the register of inputs would be incremented.

So far, we have examined the utility of the developmental model on three classification problems. However, the aim of the work is to produce general problem solving on many different kinds of computational problems. Clearly, a favourable direction to go is to expand the list of problems and problem types. How much neuron sharing would take place across problems of different types (e.g. classification and real-time control)? Would different kinds of problems cause whole new sub-networks to grow?

Currently the neurons exist in a one-dimensional space however it would be relatively straightforward to extend it to two or even three spatial dimensions. This remains for future work.

Eventually, the aim is to create developmental networks of spiking neurons. This would allow a study of activity dependent development [33] which is an extremely important aspect of biological brains.

8.13 Conclusions

We have presented a conceptually simple model of a developmental neuron in which neural networks develop over time. Conventional ANNs can be extracted from these networks. We have shown that an evolved pair of programs can produce networks that can solve multiple classification problems reasonably well. Multiple-problem solving is a new domain for investigating more general developmental neural models.

Appendix: Detailed Algorithms

Developing the Brain and Evaluating the Fitness

The detailed algorithm for developing the brain and assessing its fitness is shown in Algorithm 1 There are two stages to development. The first (which we refer to as 'pre') occurs prior to a learning epoch loop (lines 3–6). While the second phase (referred to as 'while') occurs inside a learning epoch loop (lines 9–12).

Lines 13–22 are concerned with calculating fitness. For each computational problem an ANN is extracted from the underlying brain. This is carried by a function $ExtractANN(problem, OutputAddress)$ which is detailed in Algorithm 9. This function extracts a feedforward ANN corresponding to each computational problem (this is stored in a phenotype which we do not detail here). The array $OutputAddress$ stores the addresses of the output neurons associated with the computational problem. It is used together with the phenotype to extract the network of neurons that are required for the computational problem. Then the input data is supplied and the outputs of the ANN calculated. The class of a data instance is determined by the largest output. The learning loop (lines 8–29) develops the brain and exits if the fitness value (in this case classification accuracy) reduces (lines 23–27 in Algorithm 1). One can think of the 'pre' development phase as growing a neural network prior to training. The 'while' phase is a period of development within the learning phase. N_{ep} denotes the user-defined number of learning epochs. N_p represents the number of problems in the suite of problems being solved. $N_{ex}(p)$ denotes the number of examples for each problem. A is the accuracy of prediction for a single training instance. F is the fitness over all examples. TF is the accumulated fitness over all problems. Fitness is normalised (lines 20 and 22).

Updating the Brain

Algorithm 2 shows the update brain process. This algorithm is run at each developmental step. It runs the soma and dendrite programs for each neuron and from the previously existing brain creates a new version ($NewBrain$) which eventually overwrites the previous brain at the last step (lines 52–53).

Algorithm 2 starts by analyzing the brain to determine the addresses and numbers of non-output and output neurons (lines 3–11). Then the non-output neurons are processed. The evolved soma program is executed and it returns a neuron with

Algorithm 1 Develop network and evaluate fitness

```
 1: function FITNESS
 2:     Initialise brain
 3:     Use ‘pre’ parameters
 4:     for $s = 0$ to $s < NDS_{pre}$ do                # develop prior to learning
 5:         UpdateBrain
 6:     end for
 7:     $TF_{prev} = 0$
 8:     for $e = 0$ to $e < N_{ep}$ do                   # learning loop
 9:         Use ‘while’ parameters                       # learning phase
10:         for $s = 0$ to $s < NDS_{whi}$ do
11:             UpdateBrain
12:         end for
13:         $TF = 0$                                     # initialise total fit
14:         for $p = 0$ to $p < N_p$ do
15:             ExtractANN(p, OutputAddress)             # Get ANN for problem p
16:             $F = 0$                                  # initialise fit
17:             for $t = 0$ to $t < N_{ex}(p)$ do
18:                 $F = F + Acc$                        # sum acc. over instances
19:             end for
20:             $TF = TF + F/N_{ex}(p)$                  # sum normalised acc. over problems
21:         end for
22:         $TF = TF/N_p$                                # normalise total fitness
23:         if $TF < TF_{prev}$ then                     # has fitness reduced?
24:             $TF = TF_{prev}$                         # return previous fitness
25:             Break                                    # terminate learning loop
26:         else
27:             $TF_{prev} = TF$                         # update previous fitness
28:         end if
29:     end for
30:     return TF
31: end function
```

updated values for the neuron position, health and bias. These are stored in the variable $UpdatedNeurVars$.

If the user-defined option to disallow non-output neuron movement is chosen then the updated neuron position is reset to that before the soma program is run (lines 16–17). Next the evolved dendrite programs are executed in all dendrites. The algorithmic details are given in Algorithm 6 (See Sect. 8.3.5).

The neuron health is compared with the user-defined neuron death threshold NH_{death} and if the health exceeds the threshold the neuron survives (see lines 22–28). At this stage it is possible that the neuron has been given a position that is identical to one of the neurons in the developing brain ($NewBrain$) so one needs a mechanism for preventing this. This is accomplished by Algorithm 3 (Lines 19 and 46). It checks whether a collision has occurred and if so an increment MN_{inc} is added to the position and then it is bound to the interval $[-1, 1]$. In line 23 the updated neuron is written into $NewBrain$. A check is made in line 25 to see if the allowed number of neurons has been reached, if so the non output neuron update loop (lines 12–38) is exited and the output neuron section starts (lines 39–51). If the limit on numbers of neurons has not been reached, the updated neuron may replicate depending on whether its health is above the user-defined threshold, NH_{health} (line 29). The position of the new born neuron is immediately incremented by MN_{inc} so that it does not collide with its parent (line 30). However, its position needs to be checked also to see if it collides with any other neuron, in which case its position is incremented again until a position is found that causes no collision. This is done in the function IFCOLLISION.

In CREATENEWNEURON (see line 32) the bias, the incremented position and dendrites of the parent neuron are copied into the child neuron. However, the new neuron is given a health of 1.0 (the maximum value). The algorithm examines the non-output neurons (lines 39–51) and again is terminated if the allowed number of neurons is exceeded. The steps are similar to those carried out with non-output neurons, except that output neurons can not either die or replicate as their number is fixed by the number of outputs required by the computational problem being solved.

The details of the neuron collision avoidance mechanism is shown in Algorithm 3.

Running the Soma

The UPDATEBRAIN program calls the RUNSOMA program (Algorithm 4) to determine how the soma changes in each developmental step. The seven soma program inputs comprising the neuron health, position and bias, the averaged position, weight and health of the neuron's dendrites and the problem type are supplied to the CGP encoded soma program (line 12). The array $ProblemTypeInputs$ stores $NumProblems$+1 constants equally spaced between -1 and 1. These are used to allow output neurons to know what computational problem they belong to.

The soma program has three outputs relating to the position, health and bias of the neuron. These are used to update the neuron (line 13).

Changing the Neuron Variables

The UPDATENEURON Algorithm 5 updates the neuron properties of health, position and bias according to three user-chosen options defined by a variable Incr_{opt}. If this is zero, then the soma program outputs determine directly the updated values of the soma's health, position and bias. If Incr_{opt} is one or two, the updated values of the soma are changed from the parent neuron's values in an incremental way. This is either a linear or nonlinear increment or decrement depending on whether the soma program's outputs are greater than or less than or equal to zero (lines 8–16). The magnitudes of the increments is defined by the user-defined constants: δ_{sh}, δ_{sp}, δ_{sb} and sigmoid slope parameter, α (see Table 8.1).

The increment methods described in Algorithm 5 change neural variables, so action needs to be taken to force the variables to strictly lie in the interval $[-1, 1]$.

Algorithm 2 Update brain

```
 1: function UpdateBrain
 2:     NewNumNeurons = 0
 3:     for $i = 0$ to $i <$ NumNeurons do            # get number and addresses of neurons
 4:         if (Brain[i].out = 0) then
 5:             NonOutputNeuronAddress[NumNonOutputNeurons] = i
 6:             increment NumNonOutputNeurons
 7:         else
 8:             OutputNeuronAddress[NumOutputNeurons] = i
 9:             increment NumOutputNeurons
10:         end if
11:     end for
12:     for $i = 0$ to $i <$ NumNonOutputNeurons do   # process non-output neurons
13:         NeuronAddress = NonOutputNeuronAddress[i]
14:         Neuron = Brain[NeuronAddress]
15:         UpdatedNeurVars = RunSoma(Neuron)        # get new position, health and bias
16:         if (DisallowNonOutputsToMove) then
17:             UpdatedNeurVars.x = Neuron.x
18:         else
19:             UpdatedNeurVars.x = IfCollision(NewNumNeurons,NewBrain,UpdatedNeurVars.x)
20:         end if
21:         UpdatedNeuron = RunAllDendrites(Neuron, UpdatedNeurVars)
22:         if (UpdatedNeuron.health $> NH_{death}$) then # if neuron survives
23:             NewBrain[NewNumNeurons] = UpdatedNeuron
24:             Increment NewNumNeurons
25:             if (NewNumNeurons = $NN_{max}$-NumOutputNeurons) then
26:                 Break                            # exit non-output neuron loop
27:             end if
28:         end if
29:         if (UpdatedNeuron.health $> NH_{health}$) then# neuron replicates
30:             UpdatedNeuron.x = UpdatedNeuron.x+$MN_{inc}$
31:             UpdatedNeuron.x = IfCollision(NewNumNeurons, NewBrain, UpdatedNeuron.x)
32:             NewBrain[NewNumNeurons] = CreateNewNeuron(UpdatedNeuron)
33:             Increment NewNumNeurons
34:             if (NewNumNeurons = $NN_{max}$ - NumOutputNeurons) then
35:                 Break                            # exit non-output neuron loop
36:             end if
37:         end if
38:     end for
39:     for $i = 0$ to $i <$ NumOutputNeurons do      # process output neurons
40:         NeuronAddress = OutputNeuronAddress[i]
41:         Neuron = Brain[NeuronAddress]
42:         UpdatedNeurVars = RunSoma(Neuron)        # get new position, health and bias
43:         if (DisallowOutputsToMove) then
44:             UpdatedNeurVars.x = Neuron.x
45:         else
46:             UpdatedNeurVars.x = IfCollision(NewNumNeurons,NewBrain,UpdatedNeurVars.x)
47:         end if
48:         UpdatedNeuron = RunAllDendrites(UpdatedNeuron)
49:         NewBrain[NewNumNeurons] = UpdatedNeuron
50:         Increment NewNumNeurons
51:     end for
52:     NumNeurons = NewNumNeurons
53:     Brain = NewBrain
54: end function
```

Algorithm 3 Move neuron if it collides with another

```
function IfCollision(NumNeurons, Brain, NeuronPosition)
    NewPosition = NeuronPosition
    collision = 1
    while collision do
        collision = 0
        for $i = 0$ to $j <$ NumNeurons do
            if (| NeuronPosition - Brain[i].x | $<$ 1.e-8) then
                collision = 1
            end if
            if collision then
                break
            end if
        end for
        if collision then
            NewPosition = NewPosition+$MN_{inc}$
        end if
    end while
    if collision then
        NewPosition = Bound(NewPosition)
    end if
    return NewPosition
end function
```

Algorithm 4 RunSoma(Neuron)

```
function RunSoma(Neuron)
    AvDendritePosition = GetAvDendritePosition(Neuron)
    AvDendriteWeight = GetAvDendriteWeight(Neuron)
    AvDendriteHealth = GetAvDendriteHealth(Neuron)
    SomaProgramInputs[0] = Neuron.health
    SomaProgramInputs[1] = Neuron.x
    SomaProgramInputs[2] = Neuron.bias
    SomaProgramInputs[3] = AvDendritePosition
    SomaProgramInputs[4] = AvDendriteWeight
    SomaProgramInputs[5] = AvDendriteHealth
    SomaProgramInputs[6] = ProblemTypeInputs[WhichProblem]
    SomaProgramOutputs = SomaProgram(SomaProgramInputs)
    UpdatedNeuron = UpdateNeuron(Neuron, SomaProgramOutputs)
    return UpdatedNeuron.x, UpdatedNeuron.health, UpdatedNeuron.bias
end function
```

We call this 'bounding' (lines 34–36).This is accomplished using a hyperbolic tangent function.

Running All Dendrite Programs and Building a New Neuron

Algorithm 6 takes an existing neuron and creates a new neuron using the updated soma variables, position, health and bias which are stored in $UpdateNeurVars$ (from Algorithm 4) and the updated dendrites which result from running the dendrite

Algorithm 5 Neuron update function

```
 1: function UPDATENEURON(Neuron, SomaProgramOutputs)
 2:     ParentHealth = Neuron.health
 3:     ParentPosition = Neuron.x
 4:     ParentBias = Neuron.bias
 5:     health = SomaProgramOutputs[0]
 6:     position = SomaProgramOutputs[1]
 7:     bias = SomaProgramOutputs[2]
 8:     if ($\text{Incr}_{opt} = 1$) then                    # calculate increment
 9:         HealthIncrement $= \delta_{sh}$
10:         PositionIncrement $= \delta_{sp}$
11:         BiasIncrement $= \delta_{sb}$
12:     else if ($\text{Incr}_{opt} = 2$) then
13:         HealthIncrement $= \delta_{sh}$*sigmoid(health, $\alpha$)
14:         PositionIncrement $= \delta_{sp}$*sigmoid(position, $\alpha$)
15:         BiasIncrement $= \delta_{sb}$*sigmoid(bias, $\alpha$)
16:     end if
17:     if ($\text{Incr}_{opt} > 0$) then                    # apply increment
18:         if (health $> 0.0$) then
19:             health = ParentHealth + HealthIncrement
20:         else
21:             health = ParentHealth - HealthIncrement
22:         end if
23:         if (position $> 0.0$) then
24:             position = ParentPosition + PositionIncrement
25:         else
26:             health = ParentPosition - PositionIncrement
27:         end if
28:         if (bias $> 0.0$) then
29:             bias = ParentBias + BiasIncrement
30:         else
31:             bias = ParentBias - BiasIncrement
32:         end if
33:     end if
34:     health = Bound(health)
35:     position = Bound(position)
36:     bias = Bound(bias)
37:     return health, position and bias
38: end function
```

program in all the dendrites. Initially (line 3–5), the updated soma variables are written into the updated neuron. The number of dendrites in the updated neuron is set to zero. In lines 8–11, the health of the non-updated *neuron* is examined and if it is above the *dendrite health threshold* for birth, a new dendrite is generated and the updated neuron gains a dendrite. If so, the neuron gains a dendrite created by a function $GenerateDendrite()$. This assigns a weight, health and position to the new dendrite. The weight and health is set to one and the position set to half the parent neuron position. These choices appeared to give good results.

Lines 12–33 are concerned with processing the dendrite program in all the dendrites of the non-updated neuron and updating the dendrites. If the updated

Algorithm 6 Run the evolved dendrite program in all dendrites

```
 1: function RunAllDendrites(Neuron, DendriteProgram, NewSomaPosition, NewSoma-
    Health, NewSomaBias)
 2:     WhichProblem = Neuron.isout
 3:     OutNeuron.x = NewSomaPosition
 4:     OutNeuron.health = NewSomaHealth
 5:     OutNeuron.bias = NewSomaBias
 6:     OutNeuron.isout = WhichProblem
 7:     OutNeuron.NumDendrites = 0
 8:     if (Neuron.health > $DH_{birth}$ ) then
 9:         OutNeuron.dendrites[NumDendrites] = GenerateDendrite()
10:         Increment OutNeuron.NumDendrites
11:     end if
12:     for $i = 0$ to $i <$ OutNeuron.NumDendrites do
13:         DendriteProgramInputs[0] = Neuron.health
14:         DendriteProgramInputs[1] = Neuron.x
15:         DendriteProgramInputs[2] = Neuron.bias
16:         DendriteProgramInputs[3] = Neuron.dendrites[i].health
17:         DendriteProgramInputs[4] = Neuron.dendrites[i].weight
18:         DendriteProgramInputs[5] = Neuron.dendrites[i].position
19:         DendriteProgramInputs[6] = ProblemTypeInputs[WhichProblem]
20:         DendriteProgramOutputs = DendriteProgram(DendriteProgramInputs)
21:         UpdatedDendrite = RunDendrite(Neuron, DendriteProgramOutputs)
22:         if (UpdatedDendrite.isAlive) then
23:             OutNeuron.dendrites[NumDendrites] = UpdatedDendrite
24:             increment OutNeuron.NumDendrites
25:             if (OutNeuron.NumDendrites > MaxNumDendrites) then
26:                 break
27:             end if
28:         end if
29:     end for
30:     if (OutNeuron.NumDendrites = 0) then            # if all dendrites die
31:         OutNeuron.dendrites[0] = Neuron.dendrites[0]
32:         OutNeuron.NumDendrites = 1
33:     end if
34:     return OutNeuron
35: end function
```

dendrite has a health above its death threshold then it survives and gets written into the updated neuron (lines 22–28). Updated dendrites do not get written into the updated neuron if it already has the maximum allowed number of dendrites (line 25–27). In lines 30–33 a check is made as to whether the updated neuron has no dendrites. If this is so, it is given one of the dendrites of the non-updated neuron. Finally, the updated neuron is returned to the calling function.

Algorithm 6 calls the function RunDendrite (line 21). This function is detailed in Algorithm 7. It changes the dendrites of a neuron according to the evolved dendrite program. It begins by assigning the dendrites health, position and weight to the parent dendrite variables. It writes the dendrite program outputs to the internal variables health, weight and position. Then in lines 8–16 it defines the

Algorithm 7 Change dendrites according to the evolved dendrite program

```
 1: function RunDendrite(Neuron, WhichDendrite, DendriteProgramOutputs)
 2:     ParentHealth = Neuron.dendrites[WhichDendrite].health
 3:     ParentPosition = Neuron.dendrites[WhichDendrite].x
 4:     ParentWeight = Neuron.dendrites[WhichDendrite].weight
 5:     health = DendriteProgramOutputs[0]
 6:     weight = DendriteProgramOutputs[1]
 7:     position = DendriteProgramOutputs[2]
 8:     if (Incr$_{opt}$ $= 1$) then
 9:         HealthIncrement $= \delta_{dh}$
10:         WeightIncrement $= \delta_{dw}$
11:         PositionIncrement $= \delta_{dp}$
12:     else if (Incr$_{opt}$ $= 2$) then
13:         HealthIncrement $= \delta_{dh}$*sigmoid(health, $\alpha$)
14:         WeightIncrement $= \delta_{dw}$*sigmoid(weight, $\alpha$)
15:         PositionIncrement $= \delta_{dp}$*sigmoid(position, $\alpha$)
16:     end if
17:     if (Incr$_{opt}$ $> 0$) then
18:         if (health $> 0.0$) then
19:             health = ParentHealth + HealthIncrement
20:         else
21:             health = ParentHealth - HealthIncrement
22:         end if
23:         if (position $> 0.0$) then
24:             position = ParentPosition + PositionIncrement
25:         else
26:             health = ParentPosition - PositionIncrement
27:         end if
28:         if (weight $> 0.0$) then
29:             weight = ParentWeight + BiasIncrement
30:         else
31:             weight = ParentWeight - BiasIncrement
32:         end if
33:     end if
34:     health = Bound(health)
35:     position = Bound(position)
36:     weight = Bound(weight)
37:     if (health $> DH_{death}$) then
38:         UpdatedDendrite.weight = weight
39:         UpdatedDendrite.health = health
40:         UpdatedDendrite.x = position
41:         UpdatedDendriteisAlive = 1
42:     else
43:         UpdatedDendriteisAlive = 0
44:     end if
45:     return UpdatedDendrite and UpdatedDendriteisAlive
46: end function
```

possible increments in health, weight and position that will be used to increment or decrement the parent variables according to the user defined incremental options (linear or non-linear). In lines 17–33 it respectively carries out the increments or

Algorithm 8 Create new neuron from parent neuron

```
 1: function CREATENEWNEURON(ParentNeuron)
 2:     ChildNeuron.NumDendrites = ParentNeuron.NumDendrites
 3:     ChildNeuron.isout = 0
 4:     ChildNeuron.health = 1
 5:     ChildNeuron.bias = ParentNeuron.bias
 6:     ChildNeuron.x = ParentNeuron.x
 7:     for $i = 0$ to $i <$ ChildNeuron.NumDendrites do
 8:         ChildNeuron.dendrites[i] = ParentNeuron.dendrites[i]
 9:     end for
10: end function
```

decrements of the parent dendrite variables according whether the corresponding dendrite program outputs are greater than or less than or equal to zero. After this it bounds those variables. Finally, in lines 37–44 it updates the dendrites health, weight and position provided the adjusted health is above the dendrite death threshold (in other words it survives). Note that if $Incr_{opt} = 0$ then there is no incremental adjustment and the health, weight and position of the dendrites are just bounded (lines 34–36).

Algorithm 2 uses a function CREATENEWNEURON to create a new neuron if the neuron health is above a threshold. This function is described in Algorithm 8. It makes the new born neuron the same as the parent (note, its position will be adjusted by the collision avoidance algorithm) except that it is given a health of one. Experiments suggested that this gave better results.

Extracting Conventional ANNs from the Evolved Brain

In Algorithm 1, a conventional feed-forward ANN is extracted from the underlying collection of neurons (line 15). The algorithm for doing this is shown in Algorithm 9. Firstly, this algorithm determines the number of inputs to the ANN (line 5). Since inputs are shared across problems the number of inputs is set to be the maximum number of inputs that occur in the computational problem suite. If an individual problem has less inputs than this maximum, the extra inputs are set to 0.0. The brain array is sorted by position. The algorithm then examines all neurons (line 7) and calculates the number of non-output neurons and output neurons and stores the neuron data in arrays $NonOutputNeurons$ and $OutputNeurons$. It also calculates their addresses in the brain array.

The next phase is to go through all dendrites of the non-output neurons to determine which inputs or neurons they connect to (lines 19–33). The evolved neuron programs generate dendrites with end positions anywhere in the interval [−1, 1]. The end positions are converted to lengths (line 25). In this step the dendrite position is linearly mapped into the interval [0, 1]. To generate a valid neural network we assume that dendrites are automatically connected to the nearest

Algorithm 9 The extraction of neural networks from the underlying brain

```
function EXTRACTANN(problem, OutputAddress)
    NumNonOutputNeurons = 0
    NumOutputNeurons = 0
    OutputCount=0
    $N_i = max(N_i, p)$
    sort(Brain, 0, NumNeurons-1)                # sort neurons by position
    for $i = 0$ to $i <$ NumNeurons do
        Address = $i + N_i$
        if (Brain[i].isout $>$ 0) then            # non-output neuron
            NonOutputNeur[NumNonOutputNeur] = Brain[i]
            NonOutputNeuronAddress[NumNonOutputNeur]= Address
            Increment NumNonOutputNeur
        else                                    # output neuron
            OutputNeurons[NumOutputNeurons]= Brain[i]
            OutputNeuronAddress[NumOutputNeurons]= Address
            Increment NumOutputNeurons
        end if
    end for
    for $i = 0$ to $i <$ NumNonOutputNeur do      # do non-output neurons
        Phenotype[i].isout = 0
        Phenotype[i].bias = NonOutputNeur[i].bias
        Phenotype[i].address = NonOutputNeuronAddress[i]
        NeuronPosition = NonOutputNeur[i].x
        for $j = 0$ to $j <$ NonOutputNeur[i].NumDendrites do
            Convert DendritePosition to DendriteLength
            DendPos = NeuronPosition - DendriteLength
            DendPos = Bound(DendPos)
            AddressClosest = GetClosest(NumNonOutputNeur, NonOutputNeur, 0, DendPos)
            Phenotype[i].ConnectionAddresses[j] = AddressClosest
            Phenotype[i].weights[j] = NonOutputNeur[i].weight[j]
        end for
        Phenotype[i].NumConnectionAddress = NonOutputNeur[i].NumDendrites
    end for
    for $i = 0$ to $i <$ NumOutputNeurons do      # do output neurons
        $i1 = i+$NumOutputNeurons
        Phenotype[i1].isout = OutputNeurons[i].isout
        Phenotype[i1].bias = OutputNeurons[i].bias
        Phenotype[i1].address = OutputNeuronAddress[i]
        NeuronPosition = OutputNeurons[i].x
        for $j = 0$ to $j <$ OutputNeurons[i].NumDendrites do
            Convert DendritePosition to DendriteLength
            DendPos = NeuronPosition - DendriteLength
            DendPos = Bound(DendPos)
            AddressClosest = GetClosest(NumNonOutputNeur, NonOutputNeur, 1, DendPos)
            Phenotype[i1].ConnectionAddresses[j] = AddressClosest
            Phenotype[i1].weights[j] = OutputNeuron[i].weight[j]
        end for
        Phenotype[i1].NumConnectionAddress = OutputNeurons[i].NumDendrites
        if (OutputNeurons[i].isout == problem+1) then
            OutputAddress[OutputCount] = OutputNeuronAddress[i]
            Increment OutputCount
        end if
    end for
end function
```

Algorithm 10 Find which input or neuron a dendrite is closest to

```
function GetClosest(NumNonOutNeur, NonOutNeur, IsOut, DendPos)
    AddressOfClosest = 0
    min = 3.0
    if (IsOut = 0) then                              # only non-out neurons connect to inputs
        for ($i$ = 0 to $i$ < MaxNumInputs) do
            distance = DendPos - InputLocations[i]
            if distance > 0 then
                if (distance < min) then
                    min = distance
                    AddressOfClosest = i
                end if
            end if
        end for
    end if
    for $j$ = 0 to $j$ <NumNonOutputNeur do
        distance = DendPos - NonOutNeur[j].x
        if distance > 0 then                         # feed-forward connections
            if (distance < min) then
                min = distance
                AddressOfClosest = j + MaxNumInputs
            end if
        end if
    end for
    return AddressOfClosest
end function
```

neuron or input *on the left*. We refer to this as "snapping" (lines 28 and 44). The dendrites of non-output neurons are allowed to connect to either inputs or other non-output neurons on their left. However, output neurons are only allowed to connect to non-output neurons on their left. Algorithm 10 returns the address of the neuron or input that the dendrite snaps to. The dendrites of output neurons are not allowed to connect directly to inputs (see Line 4 of the GetClosest function), however, when neurons are allowed to move, there can occur a situation where an output neuron is positioned so that it is the first neuron on the right of the outputs. In that situation it can only connect to inputs. If this situation occurs then the initialisation of the variable $AddressOfClosest$ to zero in the GetClosest function (line 2) means that all the dendrites of the output neuron will be connected to the first external input to the ANN network. Thus a valid network will still be extracted albeit with a rather useless output neuron. It is expected that evolution will avoid using programs that allow this to happen.

Algorithm 9 stores the information required to extract the ANN in an array called $Phenotype$. It contains the connection addresses of all neurons and their weights (lines 29–30 and 45–46). Finally it stores the addresses of the output neurons ($OutputAddress$) corresponding to the computational problem whose ANN is being extracted (lines 49–52). These define the outputs of the extracted ANNs when they are supplied with inputs (i.e. in the fitness function when the Accuracy is assessed (see Algorithm 1). The *Phenotype* is stored in the same format as Cartesian Genetic Programming (see Sect. 8.4) and decoded in a similar way to genotypes.

References

1. Astor, J.C., Adami, C.: A development model for the evolution of artificial neural networks. Artificial Life **6**, 189–218 (2000)
2. Balaam, A.: Developmental neural networks for agents. In: Advances in Artificial Life, Proceedings of the 7th European Conference on Artificial Life (ECAL 2003), pp. 154–163. Springer (2003)
3. Boers, E.J.W., Kuiper, H.: Biological metaphors and the design of modular neural networks. Master's thesis, Dept. of Comp. Sci. and Dept. of Exp. and Theor. Psych., Leiden University (1992)
4. Cangelosi, A., Nolfi, S., Parisi, D.: Cell division and migration in a 'genotype' for neural networks. Network-Computation in Neural Systems **5**, 497–515 (1994)
5. Downing, K.L.: Supplementing evolutionary developmental systems with abstract models of neurogenesis. In: Proc. Conf. on Genetic and evolutionary Comp., pp. 990–996 (2007)
6. Eggenberger, P.: Creation of neural networks based on developmental and evolutionary principles. In: W. Gerstner, A. Germond, M. Hasler, J.D. Nicoud (eds.) Artificial Neural Networks — ICANN'97, pp. 337–342 (1997)
7. Federici, D.: A regenerating spiking neural network. Neural Networks **18**(5–6), 746–754 (2005)
8. Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems? J. Mach. Learn. Res. **15**(1), 3133–3181 (2014)
9. French, R.M.: Catastrophic Forgetting in Connectionist Networks: Causes, Consequences and Solutions. Trends in Cognitive Sciences **3**(4), 128–135 (1999)
10. Goldman, B.W., Punch, W.F.: Reducing wasted evaluations in cartesian genetic programming. In: Genetic Programming: 16th European Conference, EuroGP 2013, Vienna, Austria, April 3–5, 2013. Proceedings, pp. 61–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
11. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programmings evolutionary mechanisms. Evolutionary Computation, IEEE Transactions on **19**, 359–373 (2015)
12. Gruau, F.: Automatic definition of modular neural networks. Adaptive Behaviour **3**, 151–183 (1994)
13. Gruau, F., Whitley, D., Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. In: Proc. Conf. on Genetic Programming, pp. 81–89 (1996)
14. Harding, S., Miller, J.F., Banzhaf, W.: Developments in cartesian genetic programming: Self-modifying CGP. Genetic Programming and Evolvable Machines **11**(3–4), 397–439 (2010)
15. Hornby, G., Lipson, H., Pollack, J.B.: Generative representations for the automated design of modular physical robots. IEEE Trans. on Robotics and Automation **19**, 703–719 (2003)
16. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life **8(3)** (2002)
17. Huizinga, J., Clune, J., Mouret, J.B.: Evolving neural networks that are both modular and regular: HyperNEAT plus the connection cost technique. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 697–704 (2014)
18. Jakobi, N.: Harnessing Morphogenesis, COGS Research Paper 423. Tech. rep., University of Sussex (1995)
19. Khan, G.M.: Evolution of Artificial Neural Development - In Search of Learning Genes, *Studies in Computational Intelligence*, vol. 725. Springer (2018)

20. Khan, G.M., Miller, J.F.: In search of intelligence: evolving a developmental neuron capable of learning. Connect. Sci. **26**(4), 297–333 (2014)
21. Khan, G.M., Miller, J.F., Halliday, D.M.: Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture. Evol. Computation **19**(3), 469–523 (2011)
22. Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex Systems **4**, 461–476 (1990)
23. Kodjabachian, J., Meyer, J.A.: Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. IEEE Transactions on Neural Networks **9**, 796–812 (1998)
24. Kumar, S., Bentley, P. (eds.): On Growth, Form and Computers. Academic Press (2003)
25. McCloskey, M., Cohen, N.: Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. The Psychology of Learning and Motivation **24**, 109–165 (1989)
26. McCulloch, Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics **5**, 115–133 (1943)
27. Miller, J.F.: What bloat? cartesian genetic programming on boolean problems. In: Proc. Conf. Genetic and Evolutionary Computation, Late breaking papers, pp. 295–302 (2001)
28. Miller, J.F. (ed.): Cartesian Genetic Programming. Springer (2011)
29. Miller, J.F., Khan, G.M.: Where is the Brain inside the Brain? Memetic Computing **3**(3), 217–228 (2011)
30. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in Cartesian Genetic Programming. IEEE Trans. on Evolutionary Computation **10**(2), 167–174 (2006)
31. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. European Conf. on Genetic Programming, *LNCS*, vol. 10802, pp. 121–132 (2000)
32. Miller, J.F., Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: Proc. Int. Conf. on Evolvable Systems, *LNCS*, vol. 2606, pp. 93–104 (2003)
33. Ooyen, A.V. (ed.): Modeling Neural Development. MIT Press (2003)
34. Ratcliff, R.: Connectionist Models of Recognition and Memory: Constraints Imposed by Learning and Forgetting Functions. Psychological Review **97**, 205–308 (1990)
35. Risi, S., Lehman, J., Stanley, K.O.: Evolving the placement and density of neurons in the HyperNEAT substrate. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 563–570 (2010)
36. Risi, S., Stanley, K.O.: Indirectly encoding neural plasticity as a pattern of local rules. In: From Animals to Animats 11: Conf. on Simulation of Adaptive Behavior (2010)
37. Risi, S., Stanley, K.O.: Enhancing ES-HyperNEAT to evolve more complex regular neural networks. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 1539–1546 (2011)
38. Rust, A., Adams, R., Bolouri, H.: Evolutionary neural topiary: Growing and sculpting artificial neurons to order. In: Proc. Conf. on the Simulation and synthesis of Living Systems, pp. 146–150 (2000)
39. Stanley, K., Miikkulainen, R.: Efficient evolution of neural network topologies. In: Proc. Congress on Evolutionary Computation, vol. 2, pp. 1757–1762 (2002)
40. Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines **8**, 131–162 (2007)
41. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artificial Life **15**, 185–212 (2009)
42. Stanley, K.O., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life **9(2)**, 93–130 (2003)
43. Suchorzewski, M., Clune, J.: A novel generative encoding for evolving modular, regular and scalable networks. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 1523–1530 (2011)
44. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming encoded artificial neural networks: A comparison using three benchmarks. In: Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO), pp. 1005–1012 (2013)
45. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming. In: Proc. Parallel Problem Solving from Nature, pp. 476–486 (2014)

46. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Proc. Int. Conf. on Evolvable Systems, *LNCS*, vol. 1801, pp. 252–263. Springer Verlag (2000)
47. Yu, T., Miller, J.F.: Neutrality and the Evolvability of Boolean function landscape. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 2038, pp. 204–217 (2001)
48. Zar, J.H.: Biostatistical Analysis, 2nd edn. Prentice Hall (1984)

Chapter 9
The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming

Michael O'Neill and David Fagan

9.1 Introduction

The origin of what we now call the field of Genetic Programming lies in the problem of automatic programming [1–8]. Yet in our field this is a hibernating Elephant. Somewhere along the way we got distracted, in particular by success in model induction, system identification, or as we call it symbolic regression. However, in recent years we have also seen the emergence of arguably superior methods for symbolic regression (e.g., [9–11]), which if nothing else, should serve as a nudge to the community to turn its attention fully back towards automatic programming.

Taking a step back, looking at the bigger picture, there is much hype these days around Artificial Intelligence, which is largely driven by the significant real-World success demonstrated by neural networks and so-called deep learning. Gaining in confidence in being able to tackle problems traditionally thought to be beyond the ability of current technology, recently the machine learning community has started to turn its head towards what is arguably the holy grail problem of automatically creating code, historically referred to as automatic programming, and more recently terms like program synthesis and machine programming have been coined. Solving Automatic Programming promises the ability of a machine capable of programming itself, perhaps a necessary step in machines being capable of reaching true intelligence.

As the wider field of Machine Learning turns its head toward the holy grail of automatic programming, we need to awake the Elephant. Building on the open issues in our field [12], O'Neill and Nicolau [13] propose that our focus could

M. O'Neill (✉) · D. Fagan
Natural Computing Research & Applications Group, School of Business, University College Dublin, Dublin, Ireland
e-mail: m.oneill@ucd.ie; david.fagan@ucd.ie

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*, Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_9

be directed towards "*What are the sufficient set of features in natural, genetic, evolutionary and developmental systems, which can translate into the most effective computational approaches for program synthesis?*". More generally, we propose that we should now focus efforts towards the problem of Automatic Programming and find the best combination of *search, optimisation and machine learning methods* which might be used to tackle this problem.

The remainder of this chapter continues with an overview of the journey of genetic programming and automatic programming with a perspective from our own groups efforts in this space in Sect. 9.2, followed by an examination of a very successful collaboration between our group and Bell Labs in the domain of software-defined communication networks (Sect. 9.3), before providing some Concluding Remarks in Sect. 9.4 on what this all means for the future of Genetic Programming and its application to Automatic Programming.

9.2 A Journey with Genetic Programming and Automatic Programming

It is incorrect to state that Genetic Programming has not been applied to automatic programming, and there are many examples including perhaps its earliest incarnations towards machine code [14, 15]. However, arguably the majority of effort in our field has been directed towards symbolic regression. For example, it was noted that 36% of papers at GECCO between 2009 and 2012 were devoted to symbolic regression, and taken in combination with applications to classification the figure rose to 57% with only 3.8% on what might be considered more traditional programming problems [16]. This popularity and focus on symbolic regression is perhaps unsurprising due to the flexibility of the approach and its potential for impact. However, if we treat symbolic regression as the application (or problem domain), there are now alternative methods, some inspired by genetic programming, which outperform genetic programming approaches.

Building on the Open Issues paper [12], at the GECCO 2013 tutorial on the same topic we also asked "*whatever happened to evolving algorithms*" and noted "*algorithm induction*" as an outstanding open issue for our community [17]. Of course we are not alone in recognising the opportunity presented by the hibernating elephant. For some recent examples, in 2015, Helmuth and Spector [18] proposed a suite of 29 benchmark programming problems and have provided a GitHub resource, which includes the problem data sets [19], and Krawiec et al. [20] have proposed the use of formal specifications in combination with Genetic Programming to improve generalisation.

The research of our group at University College Dublin has followed an interesting path rooted in method development, and predominantly in grammar-based approaches to genetic programming [21], in particular Grammatical Evolution [22–26], but coupled to persistent application of these methods to a diverse set of problem domains.

As a group we are grounded in Natural Computing methods, in particular genetic and evolutionary computation, and have sought to distil the salient computational properties and mechanisms of nature into in-silico algorithms which exhibit superior performance and robustness in search, optimisation and machine intelligence. At the core of our research agenda, our target problem has always been automatic programming. We can see evidence of this in the title of the PhD thesis of one of the authors of this study “Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution” [27]. Our earlier work in automatic programming includes tackling the generation of Caching [28] and Sorting algorithms [29] in addition to multi-line C code [30]. We have undertaken many applications in Finance using grammars to constrain the model form, including generating trading rules [31, 32], corporate failure prediction [33], bond rating [34], and evolving trade execution strategies for Algorithmic Trading [35]. More recently we have been tackling program synthesis examining Helmuth and Spectors 26 benchmark programming suite for this work [36–39], and through search-based software engineering in the form of genetic improvement [45, 46].

We have examined a much wider set of problem domains through the years, including engineering applications such as truss optimisation and design [40, 43], pylon design [42], and aircraft design [44], to platform games [41] and business analytics [47–49]. The following section outlines our progress in the domain of software-defined networks in which we have enjoyed much success over the past 7 years, achieving beyond human-competitive performance.

9.3 A Journey in Software-Defined Communications Networks

In 2011 we began a collaboration with Bell Labs in the domain of Software-defined Communications Networks. The “*getting to know each other*” phase of the collaboration with Bell Labs involved approaching the problem of improving network coverage using genetic programming. Bell Labs are pioneers in femtocell technology, which was being used to augment macro cells, filling in the coverage gaps, and managing the configuration of femtocells became the manner in which we addressed this and a number of publications followed [50–53]. Figure 9.1 illustrates a multi-tier heterogeneous network comprised of macro (MC) and small cells (SC). Small cells are used to offload user equipment (e.g., smart phones and tablets) from congested macro cells.

Today, coverage is no longer the primary concern of network operators. In the era of smart phones and the proliferation of connected devices represented by the concept of the *internet of everything*, providing and efficiently managing capacity is the new challenge. In our second phase of collaboration with Bell Labs our focus shifted with this change in objective [54–56]. In Fig. 9.2 we can see how heterogeneous networks can be configured by controlling the power (a) and bias

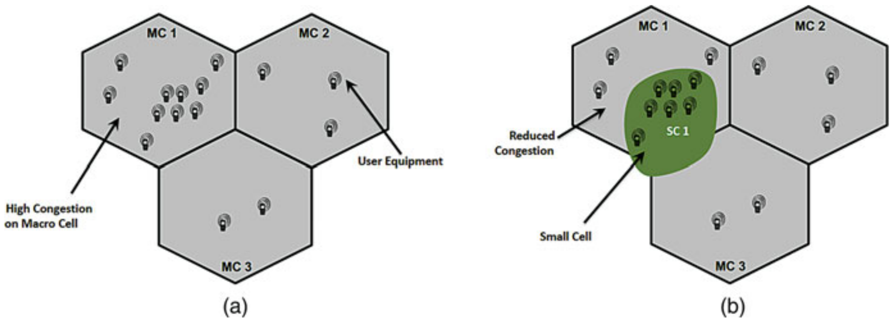


Fig. 9.1 Multi-tier heterogeneous networks containing (**a**) Macro and (**b**) Small cells

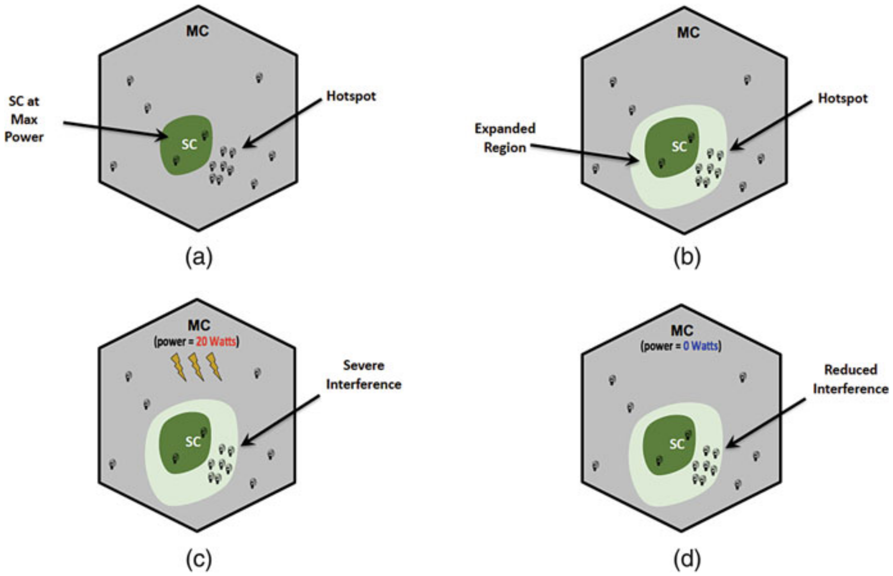


Fig. 9.2 Heterogeneous networks can be configured, such that Small Cells can control their power and bias (see (**a**) and (**b**)). Using bias creates an artificially expanded region around the small cell (**b**), however, user equipment in this **cell edge** region experience severe interference from the macro cell (**c**). Macro cells can be configured using Almost Blank Subframes, where they are silent (except for control signals) allowing small cells to communicate with user equipment with significantly reduced interference (**d**)

(b) of small cells, and managing interference through the adoption of almost blank subframes (ABS) by macro cells. When user equipment attaches to a small cell as a result of being in the artificially expanded region of a small cell as a result of bias, it suffers from severe interference from the more powerful macro cell. Managing power and bias in combination with the adoption of appropriate ABS patterns allows us to increase data transmission rates in the network resulting in improved quality of service for users.

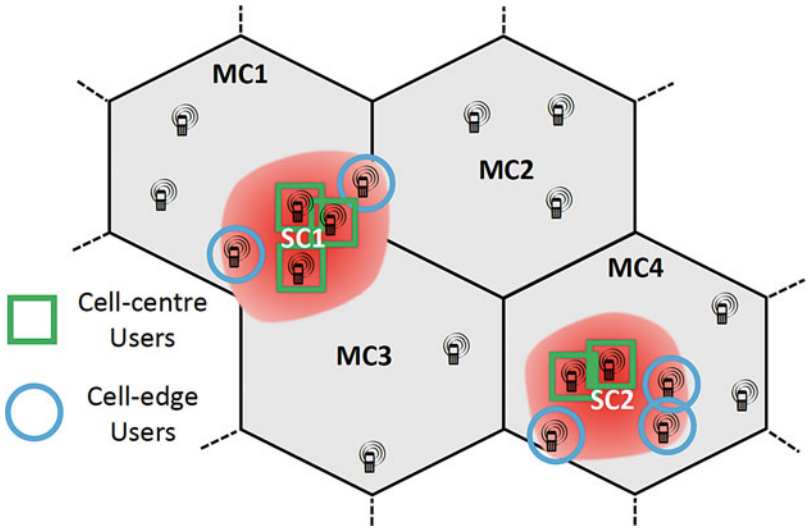


Fig. 9.3 From an optimisation perspective we explore two facets of the heterogeneous network. (1) We wish to reduce interference experienced by the cell-edge users, and (2) manage fairness of service to all users. As such we explore a number of approaches to optimise the network configuration and user equipment scheduling

Figure 9.3 illustrates the bi-level optimisation problem we tackle to improve fairness of service for users and to minimise interference, through the search for improved scheduling strategies and optimised network configurations.

Our experiments involve a real-World network simulation of downtown Dublin city centre in Ireland (see Fig. 9.4), which includes 21 macro cells and varying numbers of small cells and items of user equipment in low, medium and high-density scenarios. Network performance is calculated using the **Sum Log Rate** measure, which calculates the sum of the log of the average downlink rate for each user across all forty subframes of the simulation time.

$$Performance = \sum_{u \in U} \log(R_u^{avg}) \tag{9.1}$$

Fitness is the difference between the optimised network from the original network state (in terms of sum log rates).

In the subsections that follow we outline the main elements of the work to date. Firstly we will examine the scheduling of users within a fixed network configuration before examining how the augmentation of the network configuration can improve upon the networks performance with a fixed industry standard scheduler. Finally, we look at combining both approaches and adapt the network configuration whilst also using evolved solutions to achieve large gains in performance.

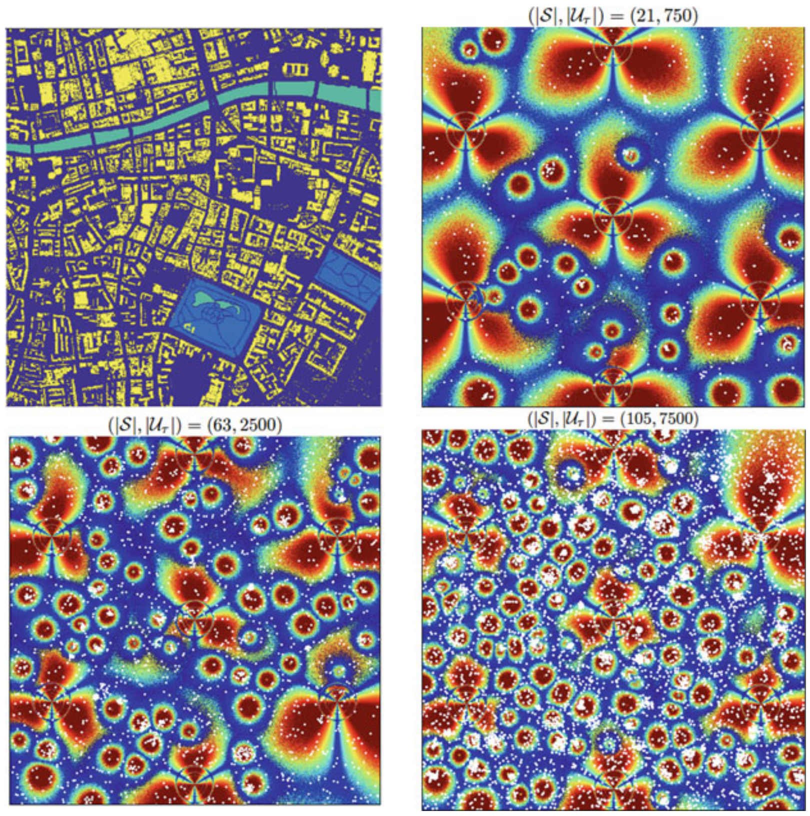
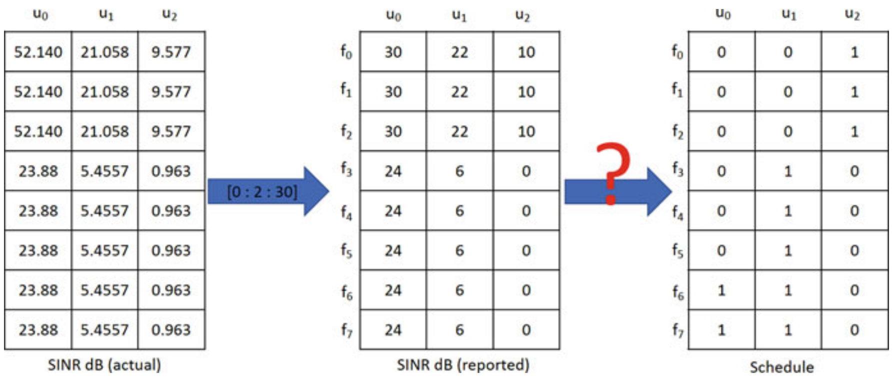

$(|\mathcal{S}|, |\mathcal{U}_\tau|) = (21, 750)$
$(|\mathcal{S}|, |\mathcal{U}_\tau|) = (63, 2500)$
$(|\mathcal{S}|, |\mathcal{U}_\tau|) = (105, 7500)$

Fig. 9.4 The simulation environment of downtown Dublin city centre with the river Liffey flowing across the top of the simulated area. The top right, and bottom row figures illustrate different levels of densification (low, medium and high numbers of small cells and user equipment) on the network

9.3.1 Network Scheduling

Heterogeneous networks have multiple layers that exist on differing change time-scales. Whilst the network configuration utilises a time-scale of minutes between possible configuration changes, the schedulers utilise a micro second scale. What is required from an effective scheduler is schedule all connected UEs for a frame (40 ms in duration), that consists of 40 subframes. The industry standard is to schedule across 8 subframes and then repeat this five times. Figure 9.5 outlines the required process. The UEs report there SINR to the hosting cell. These values are then quantised and normalised between a range. These values are then used by a scheduler to output a schedule for that frame. This all has to be done within 40 ms so that the network can effectively transmit to the UEs.

During the course of this project several approaches to this have been undertaken as outlined briefly below:



u_0	u_1	u_2
52.140	21.058	9.577
52.140	21.058	9.577
52.140	21.058	9.577
23.88	5.4557	0.963
23.88	5.4557	0.963
23.88	5.4557	0.963
23.88	5.4557	0.963
23.88	5.4557	0.963

SINR dB (actual)

	u_0	u_1	u_2
f_0	30	22	10
f_1	30	22	10
f_2	30	22	10
f_3	24	6	0
f_4	24	6	0
f_5	24	6	0
f_6	24	6	0
f_7	24	6	0

SINR dB (reported)

	u_0	u_1	u_2
f_0	0	0	1
f_1	0	0	1
f_2	0	0	1
f_3	0	1	0
f_4	0	1	0
f_5	0	1	0
f_6	1	1	0
f_7	1	1	0

Schedule

Fig. 9.5 Each subframe corresponds to 1 ms. A full frame is comprised of 40 subframes, with the first 8 subframes repeated as a block five times. The actual SINR values (dB) are reported to the cell by both being clipped to a range and quantized, and as such are a coarse-grained representation of the actual network state. The reported SINR values are then used to generate a user schedule

- **GA Approach**—The first approach taken was to employ a GA to schedule the users as the representation of the problem was suited to a GA approach. What was found was that the GA could produce a very high quality schedule for each frame, that outperformed the state of the art. The only caveat being that it was not possible to meet the hard 40 ms time constraint.
- **GP Approach**—To try and address the time constraint problem of the GA approach, we used the GA to produce example solutions and then used GP to try and evolve a model that would functionally approximate to what the GA was evolving. This approach was successful in that it still outperformed the industry standard scheduler but it still lacked the outright performance of the GA approach.
- **NN Approach**—Another model induction approach to try and utilise the GAs power was to employ a hybrid neural network. A large dataset of 500k cases was generated and the GA was used to evolved high performing schedules for each case, 25k of which were then used as a training set on a deep NN, with the remained used for test. This proved to be very successful approach resulting in performance very close to the GA. The only caveat was that training time was long, but once trained the networks proved very robust.
- **Advanced GP Approach**—The decision was made to then try and evolve a scheduler using GP that was not influenced by what the GA was outputting. This fresh outlook at the problem used summary statistics on the reported UEs SINR values. These new features where then used by GP in a regression manner to output a function that would schedule each user. This approach proved very successful and also had the added ability to adapt the desired characteristics of the scheduler for certain scenarios e.g. cell edge users priority, fairness, boost high performing users more. Also during this work ensemble approaches where examined, such as random forests, and the transition to partial assigned

bandwidth was introduced. With partially assigned bandwidth a user was not just scheduled in a binary manner as before, but was now assigned a percentage of its possible bandwidth within a subframe. These new models that GP evolved proved to be great starting points in the development of robust models that had great general performance. Domain experts, armed with the evolved solutions, could take the simplified GP models and break down what the model was doing and produce very robust models.

- **Adaptive Tuning**—The advanced GP approach proved to be the most expressive and tunable approach for scheduling. The pinnacle of this tuning was achieved when we introduced the ability for users to tune what sort of profile they wished to evolve for. The addition of a Σ parameter to allowed for the user to define what percentile of the UEs would take priority during evolution. Essentially what is now possible is for us to evolve a scheduler that would focus on the bottom fifth percentile of users and bias our gains to focusing on improving the lowest performing users.

While initially the direct-encoding GA approach had shown to produce impressive performance, the time constraints that it fails to meet have led to the discovery of this whole body of research grounded in the model induction approach of GP. We have uncovered the ability to evolve, and tune the evolution of schedulers, that are beyond human-competitive, and also can guide domain experts to refine and tune these models into very robust models with good general performance.

9.3.2 Network Configuration

The configuration of the network as stated above can be augmented by changing the levels or power and the bias of the various cells. The ABS patterns can also be changed to allow for better performance of UEs. But how these elements should be augmented over time is the question we set out to understand. Several approaches to this will now be outlined:

- **GA Load Balancing with Power and Bias**—The first foray into the network configuration was to try and balance the load on each network component. A GA was used to set the power level for each device, as well as the bias level, that would assign UEs to small cells even though they are better suited to connecting to the macro cell. This approach was simple in its goal to remove congestion and thus improve the overall network performance. While the approach was successful we were not tuning for pure performance of the UEs and especially the users in areas or high noise.
- **GA Maximise Fairness with Power, Bias, and ABS ratios**—Fairness became the overriding goal of the project as time went on. Adopting a Robin Hood approach of stealing from the rich and giving to the poor became the goal. In this approach the GA was used to not only set the power and bias levels as before but

we now were tuning the ABS ratios for the macro cells to allow for the network to give every UE usable performance when connected to the network.

- **GP Maximise Fairness with Power, Bias, and ABS ratios**—The GA approach experienced the same issues as in scheduling in terms of not being able to operate in a timely manner. Similarly we took the model induction approach, like in scheduling, to use GP and summary statistical features, constructed from the reported signal qualities of users attached to cells. This was then given to GP to evolve a function that would set the power, bias, and ABS ration for each small cell in the network. The performance of the GP model was acceptable and gains were seen in fairness.
- **Hill Climbing Approach**—Several search mechanisms were tried in further studies to establish the validity of the GA/GP performance. While random search and others fell short of the mark it was discovered that adopting a hill climbing approach to setting power, bias and ABS ratios proved to be not only very fast but also produced performance on par with and in many cases exceeding the GP approach. This was only in a static network situation. If any mobility of small cells exists within the network the GA/GP approaches remain the best performers.

With the exploration of the network configuration completed it was decided that if the network could be shown to be static that the hill climber produced the best results in a timely manner, but that if the network was subject to fluctuations in the numbers of available small cells then the GP/GA approaches should be considered. Now that we had stable approaches to both levels of the problem could we combine them?

9.3.3 Combining Network Configuration and Scheduling

Could the power of both approaches now be unlocked. By having a very effective scheduler in the GP solutions that the network operators could deploy based on certain criteria (boost cell edge user performance, boost bottom 25% of users etc;) and the ability to evolve the network configuration to minimize SINR within the network we investigated if the two approaches could exist synergistically in tandem or if we would be faced with both approaches competing against each other. To this end the best approaches to each problem level were deployed on the Dublin simulation. Starting from a baseline configuration for the network, the hill climber was allowed to adjust the networks power, bias, and ABS ratios. This was happening on a minute time scale. While the network was being adapted the GP based scheduler was deployed. With both approaches operating in tandem we saw performance increase for the lower performing users improve in the range of 600%. The adoption of the evolved network configuration provided the scheduler with the ability to vastly increase its performance than had been observed previously.

The main bulk of work to this point had focused on augmenting the network configuration and scheduling processes to deliver better performance to cell edge users. This while of great benefit to infrastructure providers, provides for little in terms of a recurring revenue stream for cellular network operators. All performance measures had been displayed in industry standard style plots as above, where every user is just a point on the graph. We wanted to focus in, and study the real world impact of our approaches to the average user of a network. To this end we investigated how single users experience was enhanced by the application of our enhanced network management solutions. It was seen that we could offer minimum level of performance guarantees to nearly 30% of all connected users before we would see any noticeable drop in quality to the majority of users [57].

9.3.4 Summary

In summary we have observed that GP alone is not necessarily the ideal approach to the heterogeneous communications network problem, however, it and more broadly evolutionary computation is an essential component of the toolkit. Through the best combination of tools available to us we have successfully achieved network performance gains significantly beyond the control algorithms deployed on networks and the state of the art [55, 56]. Not only that but the generated solutions are transparent, being amenable to human understanding and modification [54].

In terms of implications for the field of Genetic Programming, the take home message we would like to convey in this chapter is that the field of GP set out to tackle the challenging problem of Automatic Programming. As a community of researchers we have however become distracted by the vast number of interesting real-World problems which our flexible method can tackle, as a consequence of how many of these problems can be cast in the form of symbolic regression. Automatic Programming requires more than the ability to generate single line symbolic expressions.

Has the field of GP not achieved scalable automatic programming to date as a result of this distraction, or due to the limitations of the method itself? Arguably a fresh approach to automatic programming is required which considers a wider set of methods drawn from, for example, machine learning and software engineering.

9.4 Discussion and Concluding Remarks

We are not alone in recognising the opportunity presented by the hibernating elephant in our field, and in recent years in particular, Helmuth and Spector [18] have made a significant contribution to direct our focus by introducing the set of 29 benchmark programming problems, which our own group have directed attention.

Genetic Programming as a method is not perfect, and does not have all the necessary ingredients to tackle Automatic Programming. Despite overlooking the entire field of genetic programming, a recent position paper [58] provides an interesting perspective on what they call machine-based programming (i.e., automatic programming), stating that there are three pillars required for an automatic programming system, namely intention, invention and adaptation. The first of these, intention, is overlooked (or at the very least simplified) by our community and involves understanding the users intent. In genetic programming this equates to encoding a fitness function. Invention is captured by the generation of code from the specification of intent, and adaptation refers to targeting and optimising the generated code towards different platforms in addition to its maintenance. It is clear to see how advances in fields such as Natural Language Processing and Brain Computer Interfaces may bring advances in automating intention. Even if we restrict our consideration to the program synthesis (invention) component of automatic programming, Genetic Programming has a number of open issues which need to be addressed many of which have already been captured [12] and include scalability and modularity, generalisation, complexity and usability and semantics. Indeed, there are many opportunities which might already exist in the literature including logic-based approaches including the use of formal specifications, deduction, induction and learning [1], the field of Inductive Logic Programming [59], in addition to approaches from the wider field of software engineering. It is notable that Krawiec et al have recently proposed the adoption of formal specifications as an approach to address generalisation [20], and a book on Behavioral Program Synthesis with Genetic Programming [60].

From our own journey in understanding the utility of genetic programming in the domain of software defined networks, it is clear that variants of Genetic Programming used in combination with other methods results in performance beyond human competitiveness [54, 55]. And this is not the first real-World problem domain in which we have observed this. As a community who set out to tackle automatic programming, we need to sit up and pay attention. Arguably we need to re-orient our field, to re-define ourselves as the field of Automatic Programming, and as such embrace any and all methods to drive success in this domain.

We hope that this chapter acts as a clarion call to our community, to truly awaken the hibernating elephant in our field, and to broader our horizons to embrace any and all methods to address the holy grail of computer science, automatic programming.

Acknowledgements We would like to acknowledge and thank the significant contributions of members, past and present, of the UCD Natural Computing Research & Applications Group to the body of work, which this chapter represents. This research is based upon works supported by Science Foundation Ireland under grants 08/IN.1/I1868, 13/IA/1850 and 13/RC/2094.

References

1. Bierman A.W., Guiho G., Kodratoff Y. (Ed's) (1984). Automatic Program Construction Techniques. Macmillan Publishing Company.
2. Rich C., Waters R. (1988). Automatic Programming: Myths and prospects. IEEE Computer 21(8):40–51.
3. Koza J.R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, Detroit, MI, pp.768–774. Morgan Kaufmann.
4. Koza J.R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press
5. Koza J.R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press
6. Koza J.R., Andre D., Bennet III Forrest H., Keane M. (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann
7. Koza J.R., Keane M., Streeter M.J., Mydlowec W., Yu J., Lanza G. (2003). Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers.
8. Samuel A.L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3):210–229.
9. Austel V., et al. (2017). Globally Optimal Symbolic Regression. In Interpretable ML Symposium, 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.
10. McConaghy T.(2011). FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. Genetic Programming Theory and Practice IX, Edited by R. Riolo, E. Vladislavleva, and J. Moore. Springer.
11. Moraglio A., Krawiec K., Johnson C.G. (2012). Geometric Semantic Genetic Programming. In LNCS 7491 Proceedings of the International Conference on Parallel Problem Solving from Nature PPSN 2012, pp.21–31. Springer.
12. O'Neill M., Vanneschi L., Gustafson S., Banzhaf W. (2010). Open Issues in Genetic Programming. Genetic Programming and Evolvable Machines, 11(4):339–363
13. O'Neill M., Nicolau M. (2017). Distilling the salient features of natural systems: Commentary on "On the mapping of genotype to phenotype in evolutionary algorithms" by Whigham, Dick and Maclaurin. Genetic Programming and Evolvable Machines, 18(3):379–383
14. Friedberg R. (1958). A Learning Machine: Part 1. IBM Journal of Research and Development. 2(1):2–13.
15. Friedberg R., Dunham B., North J. (1959). A Learning Machine: Part 2. IBM Journal of Research and Development pp282–287.
16. White, D.R., McDermott, J., Castelli, M. et al. (2013). Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines, 14(1):3–29.
17. O'Neill M., Vanneschi L., Gustafson S., Banzhaf W. (2013). Tutorial on Open Issues in Genetic Programming. In the GECCO 2013 Companion Proceedings, Amsterdam, The Netherlands. ACM Press.
18. Helmuth T., and Spector L. (2015). General Program Synthesis Benchmark Suite. In GECCO '15: Proceedings of the 17th annual conference on Genetic and Evolutionary Computation. July 2015. ACM
19. Helmuth T., and Spector L. https://thelmuth.github.io/GECCO_2015_Benchmarks_Materials/
20. Krawiec K., Bladek I., Swan J. (2017). Counterexample-driven Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference, pp.953–960, Berlin, Germany, 2017. ACM
21. McKay R.I., Nguyen X.H., Whigham P.A., Shan Y., O'Neill M. (2010). Grammar-based Genetic Programming - A Survey. Genetic Programming and Evolvable Machines, 11(4):365–396

22. O'Neill M., Ryan C. (2003). Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers.
23. Brabazon A., O'Neill M. (2006). Biologically Inspired Algorithms for Financial Modelling. Springer.
24. Dempsey I., Brabazon A., O'Neill M. (2009). Foundations in Grammatical Evolution for Dynamic Environments. Springer.
25. Brabazon A., McGarraghy S., O'Neill M. (2015). Natural Computing Algorithms. Springer.
26. Ryan C., O'Neill M., Collins J.J. (Ed's) (2018). Handbook of Grammatical Evolution. Springer
27. O'Neill M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. PhD Thesis. University of Limerick, Ireland.
28. O'Neill M., Ryan C. (1999). Automatic Generation of Caching Algorithms. In Proceedings of EUROGEN 1999, Short Course on Evolutionary Algorithms in Engineering and Computer Science, University of Jyvaskyla, Jyvaskyla, Finland.
29. O'Neill M., Nicolau M., Agapitos A. (2014). Experiments in Program Synthesis with Grammatical Evolution: a focus on Integer Sorting. In Proceedings of IEEE Congress on Evolutionary Computation, Beijing. IEEE Press.
30. O'Neill M., Ryan C. (1999). Evolving Multi-Line Compilable C Code. In LNCS 1598 Proceedings of the Second European Workshop on Genetic Programming EuroGP 1999, pp.83–92, Goteborg, Sweden. Springer.
31. O'Neill M., Brabazon A., Ryan C., Collins J.J. (2001). Developing a Market Timing System using Grammatical Evolution Genetic and Evolutionary Computation Conference GECCO 2001, pp.1375–1381, San Francisco, CA, USA. Morgan Kaufmann.
32. Brabazon A., O'Neill M. (2004). Evolving Technical Trading Rules for Spot Foreign-Exchange Markets Using Grammatical Evolution. Computational Management Science, 1(3–4):311–328
33. Brabazon A., O'Neill M. (2004). Diagnosing Corporate Stability using Grammatical Evolution. International Journal of Applied Mathematics and Computer Science, 14(3):363–374
34. Brabazon A., O'Neill M. (2004). Bond Issuer Credit Rating with Grammatical Evolution Applications of Evolutionary Computing. In LNCS 3005 Proceedings of EvoIASP 2004, pp. 270–279, Coimbra, Portugal. Springer.
35. Cui W., Brabazon A., O'Neill M. (2011). Adaptive Trade Execution: A Grammatical Evolution Approach. International Journal of Financial Markets and Derivatives, 2(1–2):4–31
36. S. Forstenlechner, D. Fagan, M. Nicolau and M. O'Neill (2018). "Towards Understanding and Refining the General Program Synthesis Benchmark Suite with Genetic Programming," 2018 IEEE Congress on Evolutionary Computation (CEC), Rio de Janeiro, 2018, pp. 1–6.
37. Forstenlechner S., Fagan D., Nicolau M., O'Neill M. (2018). Towards Effective Semantic Operators for Program Synthesis in Genetic Programming, In Proceedings of ACM GECCO 2018, Kyoto, Japan. pp. 1119–1126, ACM Press.
38. Forstenlechner S., Fagan D., Nicolau M., O'Neill M. (2017). A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In Proceedings of EuroGP 2017 European Conference on Genetic Programming, Amsterdam. Springer.
39. Forstenlechner S., Fagan D., Nicolau M., O'Neill M. (2017). Semantics-based Crossover for Program Synthesis in Genetic Programming. In Proceedings of EA2017, Paris, France. Springer.
40. Fenton M., McNally C., Byrne J., Hemberg E., McDermott J., O'Neill M. (2016). Discrete Planar Truss Optimization by Node Position Variation using Grammatical Evolution. IEEE Transactions on Evolutionary Computation, 20(4):577–589
41. Nicolau M., Perez-Liebana D., O'Neill M., Brabazon A. (2016). Evolutionary Behaviour Tree Approaches for Navigating Platform Games. IEEE Transactions on Computational Intelligence and AI in Games, 9(3):227–238
42. Byrne J., Fenton M., Hemberg E., McDermott J., O'Neill M. (2014). Optimising Complex Pylon Structures with Grammatical Evolution. Information Sciences, 316:582–597
43. Fenton M., McNally C., Byrne J., Hemberg E., McDermott J., O'Neill M. (2014). Automatic innovative truss design using grammatical evolution. Automation in Construction, 39:59–69

44. Byrne J., Cardiff P., Brabazon A., O'Neill M. (2014). Evolving Parametric Aircraft Models for Design Exploration and Optimisation. Neurocomputing, 142:39–47.
45. Cody-Kenny B., Fenton M., Ronayne A., Considine E., McGuire T, O'Neill M. (2017). A Search for Improved Performance in Regular Expressions, In Proceedings of the GECCO 2017 Conference Companion, Berlin, Germany. ACM
46. Cody-Kenny B., Manganielloa U., Farrelly J., Ronanye A., Considine E., McGuire T., O'Neill M. (2018). Investigating the Evolvability of Web Page Load Time, In Proceedings of the European Conference on the Applications of Evolutionary Computation, Parma, Italy. Springer.
47. Borlikova G., Phillips M., Smith L., O'Neill M. (2016). Evolving classification models for prediction of patient recruitment in multi centre clinical trials using grammatical evolution, In Proceedings of EvoAPP 2016, pp.46–57, Porto, Portugal. Springer.
48. Borlikova G., Phillips M., Smith L., O'Neill M. (2016). Alternative fitness functions in the development of models for prediction of patient recruitment in multicentre clinical trials, In Proceedings of OR 2016, Hamburg, Germany. Springer.
49. Borlikova G., O'Neill M., Phillips M., Smith L. (2017). Development of a multi-model system to accommodate unknown misclassification costs in prediction of patient recruitment in multicentre clinical trials, In Proceedings of GECCO 2017, Berlin, Germany. ACM.
50. Hemberg E., Ho L., O'Neill M., Claussen H. (2013). A comparison of grammatical genetic programming grammars for controlling femtocell network coverage. Genetic Programming and Evolvable Machines 14(1):65–93
51. Hemberg E., Ho L., O'Neill M., Claussen H. (2012). Representing Communication and Learning in Femtocell Pilot Power Control Algorithms. In Rick Riolo and Ekaterina Vladislavleva and Marylyn D. Ritchie and Jason H. Moore editors, Genetic Programming Theory and Practice X, chapter 15, pages 223–238. Springer, Ann Arbor, USA, 2012
52. Hemberg E., Ho L., O'Neill M., Claussen H. (2012). Comparing the robustness of grammatical genetic programming solutions for femtocell algorithms. In GECCO Companion '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion, pages 1525–1526, Philadelphia, Pennsylvania, USA, 2012. ACM.
53. Hemberg E., Ho L., O'Neill M., Claussen H. (2011). A symbolic regression approach to manage femtocell coverage using grammatical genetic programming. In 3rd symbolic regression and modeling workshop for GECCO 2011, pages 639–646, Dublin, Ireland, 2011. ACM.
54. Fenton M., Lynch D., Fagan D., Kucera S., Claussen H., O'Neill M. (2018). Towards Automation & Augmentation of the Design of Schedulers for Cellular Communications Networks. Evolutionary Computation (online first)
55. Fenton M., Lynch D., Kucera S., Claussen H., O'Neill M. (2017). Multilayer Optimization of Heterogeneous Networks Using Grammatical Genetic Programming. IEEE Transactions on Cybernetics, 47(9):2938–2950
56. Fagan D., Fenton M., Lynch D., Kucera S., Claussen H., O'Neill M. (2017). Deep Learning through Evolution: A Hybrid Approach to Scheduling in a Dynamic Environment International Joint Conference on Neural Networks, In Proceedings of IJCNN 2017, Anchorage, Alaska. IEEE Press
57. Lynch D., Fagan D., Kucera S., Claussen H., O'Neill M. (2018). Managing Quality of Service through Intelligent Scheduling in Heterogeneous Wireless Communications Networks. In Proceedings of the 2018 IEEE Congress on Evolutionary Computation, Rio de Janeiro, Brazil. IEEE Press.
58. Gottschlich J., et al (2018). The Three Pillars of Machine-Based Programming. arXiv preprint arXiv:1803.07244. https://arxiv.org/pdf/1803.07244.pdf
59. Muggleton, S.H. (1991). Inductive logic programming. New Generation Computing. 8 (4): 295–318
60. Krawiec K. (2016). Behavioral Program Synthesis with Genetic Programming. Springer.

Chapter 10
Untapped Potential of Genetic Programming: Transfer Learning and Outlier Removal

Leonardo Trujillo, Luis Muñoz, Uriel López, and Daniel E. Hernández

10.1 Introduction

The landscape of machine learning (ML) and pattern recognition has changed markedly over the last decade. The mainstream of ML focused on statistical and probabilistic approaches in the 1990s and early 2000s, while neural networks (NNets), fuzzy systems and evolutionary computation based methods were often considered to be a different discipline, referred to as soft computing, computational intelligence and bio-inspired computation [6]. This distinction, however, was at the approach and paradigm level, not at the application level, since the type of problems being studied and solved were often the same in both communities. However, thanks to substantial developments in computing hardware that has increased the access to massively parallel architectures, NNets in particular have become the tool of choice in ML in the form of Deep Learning (DL) [11].

Evolutionary-based approaches to ML, on the other hand, have not followed suit, at least not at the same level. For many years genetic programming (GP) in particular has been seen as a mirror opposite to NNets, in the sense that the type of models produced by each method were fundamentally different even though they were designed to solve the same type of tasks, mainly supervised learning. On the one hand, GP produces symbolic expressions that could be more interpretable and amenable to human improvement, producing white or grey box models. On the other hand, NNets are black-box models that could not be inspected in any practical way. However, NNets were more efficiently trained and easier to setup. The choice of one

L. Trujillo (✉) · L. Muñoz · U. López · D. E. Hernández
Instituto Tecnológico de Tijuana, Tijuana, B.C., México
e-mail: leonardo.trujillo@tectijuana.edu.mx

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_10

approach over another was a matter of preference or specific requirements, since the type of performance achieved by both methods was more or less comparable.[1]

However, the gap in efficiency has changed dramatically as ML has moved on to tackle much larger problems in the age of Big Data [23]. While both methods perform some form of search/optimization to generate a model, there are important differences between both paradigms in this respect. While NNets rely on traditional mathematical programming methods, namely variants of gradient descent. GP uses a more complex syntactic based search that is far more inefficient, given the poor locality of the search space and the almost impossibility to define a search gradient. The gap in efficiency between both methods has gotten wider because NNets learning algorithms can easily exploit powerful parallel architectures such as GPUs [11], whereas the migration of GP to these architectures has been comparatively less successful [3]. Moreover, thanks to these technological developments, the new NNet models, namely DL architectures, have produced results that far exceed those of other ML algorithms, tackling problems that are not even considered in most GP research [11].

One possible road forward is to work on improving the implementations of GP-based systems on new parallel architectures [3], and to also simplify the use of GP-based algorithms, following the model of popular machine learning libraries such as Scikitlearn, TensorFlow or PyLearn. Another approach is to use a more constrained approach to GP-based learning, such as exploiting the hierarchical architecture of some models of the human brain [4], which are very similar to DL architectures.

This chapter, conversely, takes a look at more traditional GP-based systems and argues that there may still be significant untapped potential in (close to) the canonical GP paradigm for the most common GP application domain, symbolic regression. The goal is to present preliminary results that highlight two aspects of GP that have not received sufficient attention from the GP community. The first is transfer learning, where solutions evolved for one problem are re-used on another. This approach has become popular in DL [11, 22], but has not been explored in GP. The second is the ability for a GP system to be robust in the presence of outliers in the training data. For the first point, a GP system that evolves feature transformations called M3GP [19] is studied, showing that transfer learning is in fact possible in this domain, the first such result to the authors knowledge. For the second point, this work shows evidence that GP can exploit the inherent structure in semantic space to easily detect outliers within a set of training samples (fitness cases). These results are presented to show that the evolutionary GP paradigm may still have untapped potential that could be leveraged to develop future GP-based ML systems.

The remainder of this work proceeds as follows. Section 10.2 presents how transfer learning can be carried out in symbolic regression problems. Afterward, Sect. 10.3 describes how GP can detect outliers in the target variables using two real-

[1]Experimental evidence more or less confirmed the No-Free-Lunch theorem in many domains where, on average, many algorithms tended to perform similarly.

world datasets contaminated with extreme numbers of outliers. Finally, conclusions and future research directions are discussed in Sect. 10.4.

10.2 Transfer Learning

While Deep NNets have proven to be extremely powerful learning tools, they also suffer from some important shortcomings. Training these massive architectures requires a lot of computational power and incur long processing times [11]. Despite the increased access to massively parallel hardware, much research has been devoted to improving computational efficiency and reducing wait times. One approach is to use transfer learning [22], which can be explained quite easily. Basically, transfer learning entails the reuse of a previously trained model on a new problem. In other words, training a model with data from Problem A, lets call this problem the Donor, and then reusing the model on Problem B, lets call this the Recipient. The actual process is slightly more complicated, most transfer learning approaches re-use a large part of the original model but may replace the deepest levels of the NNet and then re-train (weight optimization with gradient descent) the network using training data from the Recipient. However, this second training phase is comparatively much shorter, making the re-use quite efficient.

The logic of why transfer learning works is straightforward. It is assumed that the Donor and the Recipient are similar in some sense; i.e. that they require the same type of low-level feature extraction or feature construction processes. For instance, the Donor may be a person recognition problem while the Recipient may be a car recognition problem. Since both tasks would require the detection and extraction of similar low-level features, then it is reasonable to assume that these shallow levels could be reused. Such an approach greatly simplifies the design of a Deep NNet, and can reduce the training time substantially.

While transfer learning has become popular in DL, there are no works, to the authors knowledge, that study this approach in the context of GP. It may be that this is the case, simply because there are no GP systems that produce models that are comparable to Deep NNets. For the scale of models usually sought by a GP search [27], it may not seem like a good idea to try to transplant solutions from one problem to another. It may seem reasonable to assume that if a model is small enough, then it will probably be specialized to the problem for which it was derived. And large models in GP are normally assumed to be bloated, not really the type of model one would want to reuse.

In this section, we present the first study of transfer learning in GP. Our intent is to provide a proof-of-concept experiment, which shows that it is possible to use this approach with a GP-based system for symbolic regression . In particular, we use a recently proposed variant of GP called M3GP [19] to derive linear in parameter models that are then fitted with Multiple Linear Regression (MLR). Two real world problems are used, and two sets of results are presented, using each problem as a Donor and a Recipient. The next subsections present the M3GP system, the test problems, experiments and a discussion of the results.

10.2.1 Case Study

M3GP evolves a transformation $k : \mathbb{R}^p \rightarrow \mathbb{R}^d$ with $p, d \in \mathbb{N}$ using a special tree representation, in essence mapping the p input features of the problem to a new feature space of size d [19]. M3GP uses a multi-tree representation, where each individual is composed by d standard GP-trees, each one defining a new feature dimension. Each one of the new features is constructed by a linear or non-linear (depending on the function set) combination of the original problem features. The method includes specialized search operators, that operate at both the subtree level and at the feature level, in conjunction with a greedy pruning method to help keep the number of new feature dimensions as low as possible. Originally, M3GP is a wrapper-approach to ML problems, where genetic operators that function at syntax level are used to evolve a population of transformations, and an additional learning process is used to fit the final model. For instance, M3GP was originally proposed to tackle multi-class classification problems using a Mahalanobis distance classifier applied to the transformed feature set, where fitness is defined by the training accuracy of the classifier [19].

Recently, M3GP was extended to tackle symbolic regression problems [20]. In essence, the new feature set generated by an M3GP individual is used to construct a linear in parameter model that is fitted using MLR. Results clearly show that the evolved feature set is more efficient at describing the target variable, with the evolved transformations increasing the mutual information with respect to the output. Moreover, the method compares favorably with several GP and non-GP methods, including GSGP [18], FFX [16] and MARS [8], producing very accurate models that are also far more parsimonious than those produced by the compared algorithms.

In this sense, M3GP can be considered to be a memetic algorithm [2], where both an explorative metaheuristic (GP) and a local optimizer (MLR) are combined. The general idea to test transfer learning for M3GP is to use an evolved feature transformation that was generated for the Donor problem, and to apply it on the Recipient problem. No further evolution will be carried out on the Recipient, but model parameters are re-fitted using MLR. One important issue to consider is to determine a proper ordering of the original problem features; i.e. how will the Recipient features will be given as input to the transformation evolved on the Donor. To do so, the original features are first transformed using Principal Component Analysis (PCA), and the feature set is truncated so both problems have the same number of features. Brute force is then used to test all possible feature combinations in the Recipient. For each combination, model parameters are re-fitted using a training partition from the Recipient and tested on unseen data. The results show that a large number of possible combinations can lead to relatively accurate models that improve upon naive MLR computed on the original features. Furthermore, the transferred transformations reach similar performance levels as running M3GP on the original problem features. However, as will be shown, the possibility of transferring solutions from one problem to another is not symmetric, an interesting result that is also discussed.

10.2.2 *Experiments and Results*

The proposed experimental strategy to test the viability of transfer learning with M3GP is carried out as follows. First, two real world problems are chosen, which have previously been solved with M3GP [19]. The problems are summarized in Table 10.1. The first is called the Tower problem, an industrial problem where the goal is to model gas chromatography measurements of the composition of a distillation tower, where the target variable is propylene concentration at the top of the tower. The input variables are related to temperature, flow and pressure, and the goal is to obtain a function that directly relates the inputs to the target [29]. The second is the Energy Efficiency Heating problem, a real-world problem related to the optimal prediction of the energy performance of residential buildings. The problem is to predict the heating load of a building based on eight descriptive features, such as the glazing area, height and orientation [28].

Both input datasets are transformed using PCA and only the top 6 new feature dimensions are selected. The number of features is set to 6 to keep the brute force search as small as possible without loosing too much information. Afterward, 10 independent runs of M3GP are performed over 10 random partitions of each problem, using a 70–30 data split for training and testing, and the setup specified in [19]. The results of these initial runs are summarized in Table 10.2, showing the minimum, maximum and median RMSE. The table also summarizes the size of the evolved models, showing the median size of the ten models (number of nodes), medium depth and medium number of new feature dimensions generated.

From the 10 runs we obtain 10 transformations, the best solution found in each run. Each of these solutions will be transferred to the Recipient problem. Moreover, since each problem has been reduced to 6 feature dimensions, we will test every possible combination to match the features of the Recipient with the inputs of the evolved transformations. This corresponds to 720 possible combinations for each model. A total of 7200 possible transformations when considering all the models.

Table 10.1 Symbolic regression real-world problems

No	Real-world problem	Attributes	Samples
1	Energy heating [28]	8	768
2	Tower [29]	25	4999

Table 10.2 Performance of M3GP on the two test problems, based on 10 independent runs

Problem/measure	Min. RMSE	Med. RMSE	Max. RMSE	Size	Depth	Dimensions
Energy heating	0.422	0.433	0.452	300.5	13.5	32
Tower	36.65	37.98	38.00	572.5	14	32

For each combination we transform the data from the Recipient, the new features are used to build a linear model. The Recipient dataset is randomly split into a training and testing sets (70–30), the former is used to fit the model parameters with MLR. In what follows a single training/testing split is reported, but multiple runs confirmed that the performance of the M3GP transformations showed low variance (as reported in [19]).

10.2.2.1 Donor-Tower/Recipient-Energy

We start by using the Tower problem as the Donor and the Energy problem as the Recipient. For a comparative baseline to evaluate the transferred solutions from the Donor to the Recipient, we will use two measures. The testing performance of MLR using the original problem features and the performance of the full M3GP learning on the reduced feature set after PCA, as reported by the median performance in Table 10.2.

For the first comparison the performance of the transferred solutions, all 7200 possibilities, is summarized in the first row of Table 10.3, which shows: the minimum, median and maximum RMSE in the first three columns; the RMSE of MLR on all the original features in the fourth column; and the number of combinations that achieves a lower RMSE than MLR (using all 10 features) in the fifth column, referred to as improvements.

The results are very interesting. As can be seen, a substantial majority (95%) of the possible models performed better than the naive MLR applied to the original problem data. In fact the best transferred model is one order of magnitude better than MLR. These results are summarized graphically in Fig. 10.1, which shows a frequency histogram relative to the testing error of how many combinations achieve a certain level of performance. The figure also shows a vertical mark to indicate the performance of MLR.

However, the transferred solutions did not reach the same performance when compared with solutions found by applying M3GP directly on the Energy problem. Nonetheless, the performance gap between the best transferred solution and the best performance on the Energy problem, reported in Table 10.2, is smaller than the difference between the best transferred model and the standard MLR model.

Table 10.3 Analysis of transfer learning results, each column focuses on a different Donor → Recipient configuration

Problem/measure	Min. RMSE	Med. RMSE	Max. RMSE	MLR	Improvements
Tower → Energy	0.818	2.11	5.35	2.87	6847
Energy → Tower	47.94	58.00	181.1	28.96	0

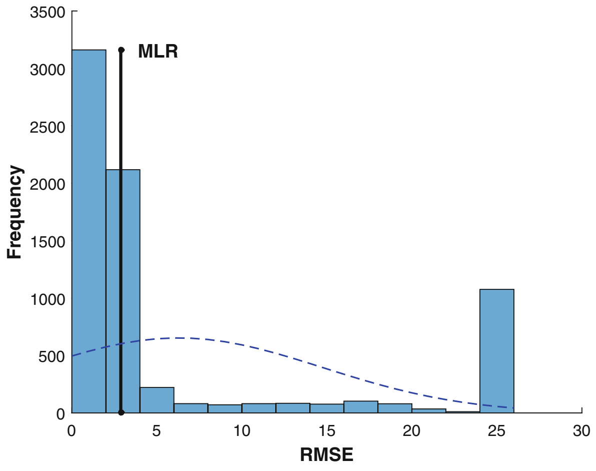


Fig. 10.1 Donor-Tower/Recipient-Energy

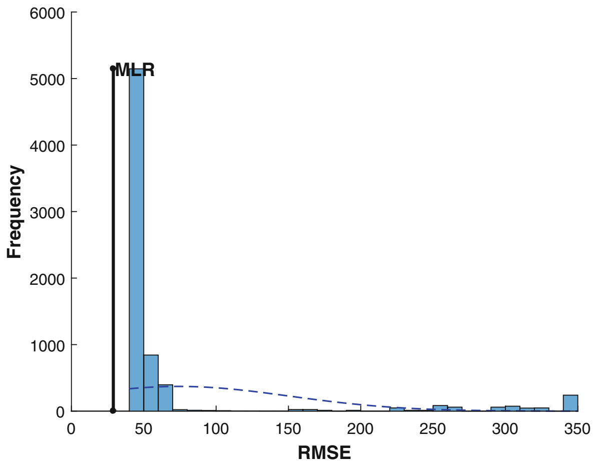


Fig. 10.2 Donor-Tower/Recipient-Tower

10.2.2.2 Donor-Energy/Recipient-Tower

The same experiments and results are reported when the roles are reversed. Energy is used as the Donor problem and Tower is the Recipient; the results are summarized in the second row of Table 10.3 and in Fig. 10.2. However in this case the results are completely different. None of the transferred models achieved adequate performance, all performed worse than using MLR on the original problem features. This result is informative for two reasons. First, it shows that transfer learning will not necessarily be possible in all problem pairs. It seems that when defining a Donor/Recipient pair, some problems will be amenable to transfer learning while others will not. Moreover, the relationship between problems, in terms of the usefulness of transferred solutions, does not seem to be symmetric; i.e. if one problem can act as Donor for another, this is not necessarily true the other way around.

10.2.3 Discussion

This section presents a preliminary experimental evaluation of transfer learning in GP. The chosen task is probably the most studied application domain of GP, symbolic regression. The lack of work on this subject tacitly suggests that it has been widely assumed that GP learning will produce highly specialized models, that are only applicable to a specific problem instance. These results suggest the contrary, with several questions immediately coming to the forefront.

- Can transfer learning with GP be possible in other domains, such as classification or automatic program synthesis?
- What problems offer good potential to act as Donors and which problems are good Recipients?
- Is transfer learning related in any way with problem difficulty in GP [15]?
- Is it possible to define a process or system that uses transfer learning to optimize the creation of solutions for new problems, such as the one suggested in [24]?

Besides these questions, probably the most exciting outcome of these results is the possibility of continuous open ended evolution. These results clearly show that at least some solutions (the best solution) from a population evolved for one problem, are relevant for another problem. Hence, it may be that other individuals might also be relevant. If this is the case, and it seems intuitive to believe they would be, then using the population from one problem to kick-start the evolutionary process in another process seems entirely feasible.

Finally, it is worth studying how transfer learning relates to problems with dynamic fitness landscapes [9, 21]. It is possible to appreciate a continuum of problem types where both dynamic fitness problems and transfer learning problems may reside. First, we have problems with static fitness landscapes, such as standard supervised learning. Second, dynamic fitness landscapes where changes are monotonic for all individuals. Third, dynamic problems where changes are not monotonic but follow a smooth dynamic. Fourth, dynamic problems where the fitness landscapes varies in a highly irregular manner. Finally, transfer learning problems, where the fitness landscapes changes in abrupt and discontinuous manner. While this is only a vague high-level outline, it may provide the basis for future experimental work on this topic.

10.3 Detecting Outliers

The presence of outliers in a training set can severely skew a regression system. Even a single outlier can eliminate the possibility of deriving the true underlying model that best describes a dataset, for linear [12] and symbolic regression [14]. In the case of linear regression there is a large amount of literature dealing with the issue of robust regression methods that can handle a certain percentage of outliers in a dataset. The breakdown point of a regression algorithm is the percentage of outliers in a training set above which the algorithm fails. For instance, ordinary least squares regression has a breakdown point of 0%, while robust methods have a breakdown point as high as 50% [12]. Above this point only sampling techniques are useful, such as Random Sample Consensus (RANSAC) [5] that samples the training set to find a sufficiently clean sample to build a useful model. RANSAC can be used in extreme cases of dataset contamination, but the computational cost increases exponentially with the percentage of outliers in the dataset.

In the case of GP this topic has received far less attention. In previous work [14], we presented several relevant results. First, the work showed that both robust objective functions, such as Least Median Squares (LMS) and Least Trimmed Squares (LTS), are applicable to GP, and that empirically their breakdown point appears to be 50% for symbolic regression. Second, the work also evaluated fitness-case sampling techniques for GP, such as interleaved sampling [10] and Lexicase selection [25]. Results showed that these approaches are not useful in dealing with outliers. The best results were obtained using RANSAC and LMS with GP for data contamination above 50%. The approach was called RANSAC-GP, and it achieved almost equal test set prediction than directly learning on a clean training set. The main drawback is the high computational cost, since GP had to be executed on each sample taken by RANSAC. Moreover, an assumption of RANSAC-GP is that the GP search will be able to find an accurate model on a clean subset of training examples, but this assumption might not hold for some real-world problems.

In [17] GP and Geometric Semantic Genetic Programming (GSGP) are compared to determine which method was more sensitive to noisy data. The training sets are corrupted with Gaussian noise, up to a maximum of 20% of the training instances, concluding that GSGP is more robust when the contamination is above 10%. However, the issue of outliers is not considered. Another example is [26], in this case focusing on classification problems with GP-based multiple feature construction when the data set is incomplete, data samples which can be considered to be outliers. The proposed method performs well, even when there is up to 20% of missing data, but extreme cases where most of the data is missing (above 50%) are not considered. In [13] the authors build ensembles of GP models evolved using a multiobjective approach, where both accuracy and program size are minimized. The proposed approach is based on the same general assumption of many techniques intended to be robust to outliers, that model performance will be worse on outlier points that inliers. The ensembles are built from hundreds of independent GP runs, making it a relatively costly approach. Experiments are carried out using a single real-world test case, where the number of outliers in not known in advance, but results suggest that the number is not higher than 5%. However that work does not present an automatic approach for outlier detection or removal, the proposed method requires human interpretation of the results to determine which samples could be labeled as outliers.

10.3.1 Case Study

In this work we reveal an interesting property of a randomly generated GP population, an ability to automatically detect which samples in a training set are outliers and which are not. In particular, we will focus on outliers in the output variable, what are referred to as vertical outliers. As stated above, many robust regression methods assume a model will better fit inlier data points than outliers. Since this is assumed to be true for a fitted model, it is intuitively clear why robust methods can only deal with less than 50% of outliers.

In this work we want to evaluate this assumption for random GP trees, by attempting to answer the following question. Is the accuracy of a random GP tree different for inlier samples than for outliers? If this question is answered in the affirmative, then it may be possible to automatically detect outlier data. To answer this question empirically the following approach is taken. First, we start with a real-world dataset with multiple input features, a more difficult scenario than the one considered in [14]. Second, we add fitness cases that are vertical outliers relative to the distribution of target values in the original dataset. Third, we generate a large random population of GP trees using standard Ramped Half-and-Half initialization. Fourth, using each GP tree we rank the fitness cases based on their residual error. Finally, we take the $p\%$ of fitness cases with the lowest residual error and compute the percentage of inliers in this set. If our assumption is correct, then the question will be answered in the affirmative. Such a result might lead to automatic robust symbolic regression with GP.

10.3.2 Experiment and Results

To perform this study, we use the Energy Efficiency Cooling and Heating problems defined in [28]. These problems deal with the prediction of the cooling and heating load, respectively, in residential buildings [1]. The datasets are contaminated as in [14], using the inverse of the Hampel identifier. To generate an outlier sample, we perform the following steps. The input features $\mathbf{x}_l$ of an outlier sample are generated by randomly sampling the input values already present in the dataset for each feature. In this way, we are assured that the input values are consistent with the original dataset. To generate the outlier output y_l we follow the approach of [14] based on the Hampel identifier. First, a reference y_r value is chosen randomly from all the inlier values in the original dataset. Then $y_l = y_r \pm M \times MAD$, where the MAD is computed over all the inlier target outputs in the original dataset, M takes a random value in the range [10, 100], and the sign of the sum is equiprobable at random.

A clean dataset is contaminated by adding outlier points such that the percentage of contamination (proportion of outliers) can be controlled. For instance, if the size of a given dataset is N, and we want a 50% contamination level, then we generate

N outliers and add them to the dataset. The considered datasets were contaminated by different amounts of outliers, 10%, 50% and 90% contamination.

The experiment proceeds as follows. We generate 20,000 random GP trees using Ramped Half-and-Half initialization, with a maximum depth of 7 levels, with the function set $F = \{+, -, \times, \div, sin, cos\}$ where $\div$ is the protected division, and considering all the input features as terminals. For each individual k_i in the sample we compute its residual error to each fitness case and its fitness (RMSE); we do this for each problem and for each level of contamination (percentage of outliers). Using the residuals we order the fitness cases in the dataset and we extract the $\rho\%$ percentile of fitness cases with the lowest residual errors, setting $\rho\%$ to the known percentage of inliers in the dataset (the compliment of the contamination level). From this subset we compute the percentage of samples that correspond to inliers. If our assumption is correct, then the best individuals (with lowest error) should mostly return fitness cases that are inliers. The GP system used in this study is the tree-based algorithm defined in the Distributed Evolutionary Algorithms in Python library (DEAP) [7].

Figures 10.3 and 10.4 present a summary of the results as scatter plots where each point represents a single GP individual. Each tree is plotted based on its fitness (horizontal axis) and the percentage of inliers detected within the top $\rho\%$ percentile of fitness cases. The top row of plots shows the scatter plots for all 20,000 individuals, but to get a better visualization of the point cloud the bottom row of plots only shows the best 15,000 individuals based on fitness. The plot also shows a horizontal line at 50%, which is the breakdown point of most robust regression methods.

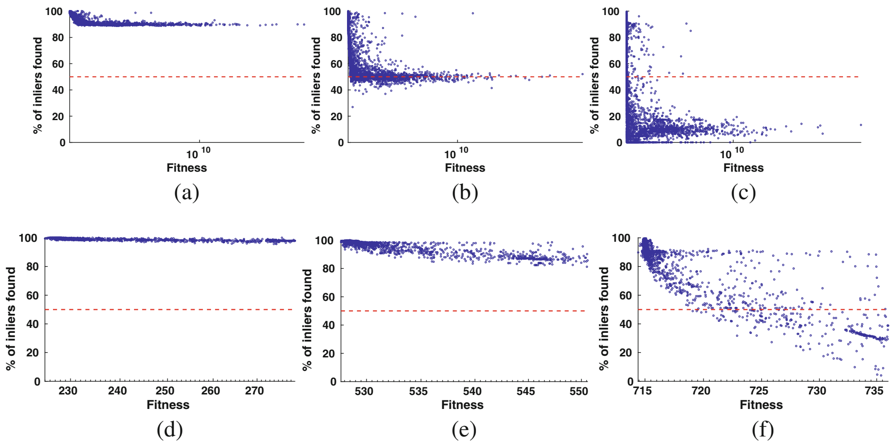


Fig. 10.3 Analysis of random GP individuals showing scatter plots where each individual is plotted based on their fitness and percentage of inliers found. Results for the Energy Cooling problem, with the top row showing the full 20K individuals and the bottom row showing the 15K individuals with the best fitness. (**a**) 10% of contamination, (**b**) 50% of contamination, (**c**) 90% of contamination, (**d**) 10% of contamination, (**e**) 50% of contamination, (**f**) 90% of contamination

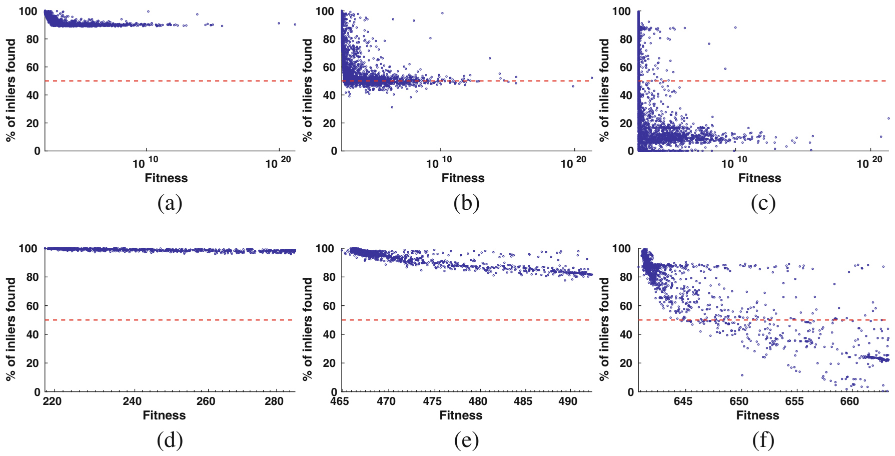


Fig. 10.4 Analysis of random GP individuals showing scatter plots where each individual is plotted based on their fitness and percentage of inliers found. Results for the Energy Heating problem, with the top row showing the full 20K individuals and the bottom row showing the 15K individuals with the best fitness. (**a**) 10% of contamination, (**b**) 50% of contamination, (**c**) 90% of contamination, (**d**) 10% of contamination, (**e**) 50% of contamination, (**f**) 90% of contamination

For low contamination levels it is easier to detect which samples are outliers, particularly for the most fit individuals. As the percentage of outliers increases, then it becomes progressively more difficult to detect which samples are outliers. However, even when the contamination level reaches 90% there is still a significant number of individuals that detect a large percentage of inliers. What is particularly important is that the number of inliers found is larger then 50% for some of these individuals, since in such a scenario a robust regression method could solve the regression task using the $\rho\%$ percentile of training samples with the lowest associated residual error.

10.3.3 Discussion

This section presented results that suggest that random GP individuals can detect which fitness cases are inliers and which are outliers, even when the number of outliers reaches as high as 90%. This results may seem counterintuitive, the GP trees were randomly generated, why would they better fit some fitness cases over another? The answer may be related to how random GP trees sample semantic space, the space of all possible program outputs for a given the inputs specified in the training set. While it is usually assumed that we are performing a random sampling of the solution space, there is substantial experimental evidence that suggests that the sampling is actually quite biased to some regions of the semantic space. This bias can be assumed to be due to the nature of the program representation and the manner in which it interacts with a problem's input features. Hence, it seems easy for GP trees to detect which fitness cases do not follow this implicit sampling bias. Such a result may lead to a powerful outlier detection method, which we are currently developing.

10.4 Conclusions and Future Outlook

This paper explores to aspects of GP that have received little or no attention. The first is transfer learning, where the results showed that it is indeed possible to transfer solutions from one problem to another. The transferred solutions performed better than a naive linear regression approach, and some reached a performance that was comparable to that of a full GP evolutionary process. Moreover, the possibility of transferring solutions from one problem to another does not seem to be symmetric, and interesting result that will be the focus of future research. Furthermore, is it possible to find an universal donor, or a solutions structure that can be used in multiple problems. The second aspect was a study of how GP trees respond to the presence of outliers in the training set. Results clearly show that a large proportion of randomly generated individuals have a different response to inliers and outliers. This difference can be used to rank the fitness cases and attempt to differentiate between both types, a possible road towards robust symbolic regression with GP. We feel that both results are noteworthy, and could lead to extracting promising untapped potential from the GP paradigm.

Acknowledgements This research was funded by CONACYT (Mexico) Fronteras de la Ciencia 2015-2 Project No. FC-2015-2/944 and TecNM project no. 6826-18-P, and first and third authors were respectively supported by CONACYT graduate scholarship No. 302526 and No. 573397.

References

1. Castelli, M., Trujillo, L., Vanneschi, L., Popovi, A.: Prediction of energy performance of residential buildings: A genetic programming approach. Energy and Buildings **102**, 67–74 (2015)
2. Chen, X., Ong, Y.S., Lim, M.H., Tan, K.C.: A multi-facet survey on memetic computation. IEEE Transactions on Evolutionary Computation **15**(5), 591–607 (2011)
3. Chitty, D.M.: Faster GPU based genetic programming using A two dimensional stack. CoRR **abs/1601.00221** (2016)
4. Dozal, L., Olague, G., Clemente, E., Hernández, D.E.: Brain programming for the evolution of an artificial dorsal stream. Cognitive Computation **6**(3), 528–557 (2014)
5. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM **24**(6), 381–395 (1981)
6. Floreano, D., Mattiussi, C.: Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies. MIT Press (2008)

7. Fortin, F.A., et al.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**, 2171–2175 (2012)
8. Friedman, J.H.: Multivariate adaptive regression splines. Ann. Statist. **19**(1), 1–67 (1991)
9. Galván-López, E., Vazquez-Mendoza, L., Schoenauer, M., Trujillo, L.: On the Use of Dynamic GP Fitness Cases in Static and Dynamic Optimisation Problems. In: EA 2017- International Conference on Artificial Evolution, pp. 1–14. Paris, France (2017)
10. Gonçalves, I., Silva, S.: Balancing learning and overfitting in genetic programming with interleaved sampling of training data. In: K. Krawiec, et al. (eds.) Genetic Programming, *LNCS*, vol. 7831, pp. 73–84. Springer Berlin Heidelberg (2013)
11. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
12. Hubert, M., Rousseeuw, P.J., Van Aelst, S.: High-breakdown robust multivariate methods. Statist. Sci. **23** (2008)
13. Kotanchek, M., et al.: Pursuing the Pareto Paradigm: Tournaments, Algorithm Variations and Ordinal Optimization, pp. 167–185. Springer US (2007)
14. López, U., Trujillo, L., Martinez, Y., Legrand, P., Naredo, E., Silva, S.: RANSAC-GP: Dealing with Outliers in Symbolic Regression with Genetic Programming, pp. 114–130. Springer International Publishing, Cham (2017)
15. Martínez, Y., Trujillo, L., Legrand, P., Galván-López, E.: Prediction of expected performance for a genetic programming classifier. Genetic Programming and Evolvable Machines **17**(4), 409–449 (2016)
16. McConaghy, T.: Genetic Programming Theory and Practice IX, chap. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology, pp. 235–260. Springer New York, New York, NY (2011)
17. Miranda, L.F., Oliveira, L.O.V.B., Martins, J.F.B.S., Pappa, G.L.: How noisy data affects geometric semantic genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, pp. 985–992. ACM, New York, NY, USA (2017)
18. Moraglio, A., Krawiec, K., Johnson, C.G.: Parallel Problem Solving from Nature - PPSN XII: 12th International Conference, Taormina, Italy, September 1–5, 2012, Proceedings, Part I, chap. Geometric Semantic Genetic Programming, pp. 21–31. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
19. Muñoz, L., Silva, S., Trujillo, L.: M3GP: multiclass classification with GP. In: P. Machado, et al. (eds.) 18th European Conference on Genetic Programming, *LNCS*, vol. 9025, pp. 78–91. Springer, Copenhagen (2015)
20. Muñoz, L., Trujillo, L., Silva, S., Vanneschi, L.: Evolving multidimensional transformations for symbolic regression with m3gp. Memetic Computing (2018). https://doi.org/10.1007/s12293-018-0274-5
21. Nguyen, T.T., Yang, S., Branke, J.: Evolutionary dynamic optimization: A survey of the state of the art. Swarm and Evolutionary Computation **6**, 1–24 (2012)
22. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Trans. on Knowl. and Data Eng. **22**(10), 1345–1359 (2010)
23. Qiu, J., Wu, Q., Ding, G., Xu, Y., Feng, S.: A survey of machine learning for big data processing. EURASIP Journal on Advances in Signal Processing **2016** (1), 67 (2016)
24. Roberts, S.C., Howard, D., Koza, J.R.: Evolving modules in genetic programming by subtree encapsulation. In: Proceedings of the 4th European Conference on Genetic Programming, EuroGP '01, pp. 160–175. Springer-Verlag, Berlin, Heidelberg (2001)
25. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion, GECCO Companion '12, pp. 401–408. ACM (2012)
26. Tran, C.T., Zhang, M., Andreae, P., Xue, B.: Genetic programming based feature construction for classification with incomplete data. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, pp. 1033–1040. ACM, New York, NY, USA (2017)
27. Trujillo, L., Muñoz, L., Galván-López, E., Silva, S.: Neat genetic programming. Inf. Sci. **333**, 21–43 (2016)

28. Tsanas, A., Xifara, A.: Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. Energy and buildings **49**, 560–567 (2012)
29. Vladislavleva, E.J., Smits, G.F., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Transactions on Evolutionary Computation **13**(2), 333–349 (2009)

Chapter 11
Program Search for Machine Learning Pipelines Leveraging Symbolic Planning and Reinforcement Learning

Fangkai Yang, Steven Gustafson, Alexander Elkholy, Daoming Lyu, and Bo Liu

11.1 Introduction

Automatic programming is one of the original goals of Artificial Intelligence and has begun to see a resurgence of research activity. The same technology trends (the availability of low-cost computing and data storage) that are enabling this resurgence are also driving industries to seek improvements in their operations by leveraging more data and analytics to optimize the process of decision making. However, as the development of such data and analytics solutions remain difficult, and the human resources needed are in short supply, the development of automated and semi-automated capabilities that assist in development is of particular interest. The benefits of such assistant technology extend far beyond industrial applications to areas of basic science. In our research, we aim to contribute to these efforts to advance the field of automated programming, and in particular, we aim to explore capabilities that help users develop knowledge about machine learning pipelines. That is, we not only want to assist in performing the highly repetitive task of trying various techniques and parameters that represent data and analytic solutions, but also our goal is to provide the user with the new, concrete knowledge about what are good selections for their particular problem instances, and give them confidence and efficiency to build production solutions that will stand up to validation and verification efforts in their industries.

F. Yang · S. Gustafson (✉) · A. Elkholy
Maana, Inc., Bellevue, WA, USA
e-mail: fyang@maana.io; sgustafson@maana.io; aelkholy@maana.io

D. Lyu · B. Liu
Auburn University, Auburn, AL, USA
e-mail: daoming.lyu@auburn.edu; boliu@auburn.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_11

The task of automated machine learning (AutoML) has gained more interest recently due to the availability of public dataset repositories and open source machine learning code bases. AutoML differs from the many previous attempts to automatically select parameters of individual machine learning methods in that it attempts to optimize the entire machine learning pipeline, which can consist of very independent steps like featurization that encodes data into a numeric form, or feature selection that attempts to pick the best subset of features to train a model from data. Most prior work would make various assumptions about the data having been already processed and featurized and instead focus on making it easier to build a specific classifier like a random forest, a neural network, or a regression model. AutoML attempts to learn a *program*, or *machine learning pipeline*, that can accomplish many more functions in the data science, or machine learning, process. In real-world, and basic science applications, the ability for the automated programming approach to address the entire pipeline of data processing to model building is critical, otherwise, the process will still require significant resources to build customer data processing and featurization pipelines for new problem instances and new problems.

Three recent examples of AutoML include Auto-WEKA [19], Auto-SKLEARN [3] and TPOT[14]. Whereas the former two approaches, Auto-WEKA and Auto-SKLEARN, combine Bayesian optimization with the machine learning libraries, WEKA and scikit-learn, to learn a probability graph from model building over many datasets, the TPOT algorithm leverages the DEAP genetic programming library with scikit-learn to learn a specific pipeline over an individual dataset. Due to our use case of assisting users by providing them with concrete knowledge about the machine learning pipelines, the TPOT algorithm more closely meets our objective. That is because machine learning pipelines are significantly different from previous applications of genetic programming, and with our added goal of delivering knowledge to the user, we are also investigating alternative representations and algorithms similar to the early days of genetic programming when basic algorithm and representation studies included analysis of other approaches like local search [15].

Highlighted in the recent AutoML work are some additional challenges that we hope to address. For example, AutoML can take a significant amount of resources to build, requiring either massive computation infrastructure or a lot of time cost, despite the availability of low-cost computing resources. Therefore finding approaches that are computationally efficient is still a key goal to not only make such techniques available to a wider audience but to prevent the scientific process of hypothesis to insight to a new hypothesis from being disrupted while long-running computing jobs finish. Secondly, the No Free Lunch Theorem for machine learning algorithms [20] would suggest to us that it also applies to the AutoML space, and as such, except providing the user with a candidate solution, it would be highly valuable to transfer knowledge to them about the individual interactions within the solution space via a strong knowledge representation. So finding a good knowledge representation for AutoML is desirable, as it would allow users to query and reason over the results directly so that the knowledge can be acquired more efficient, as opposed to simply applying data mining approaches to the log files. Lastly, because AutoML pipelines have a lot of dependencies between the steps, assigning rewards, or learning, back to individual pipelines or stages is another interesting and challenging problem. In TPOT, reinforcement learning is handled via the natural selection metaphor of allowing better-performing solutions to be used to generate future solutions. In both Auto-WEKA and Auto-SKLEARN, Bayes learning is used as a representation of rewards from dataset features through to pipeline elements.

In this paper, we solve the problem of searching for machine learning pipelines using PEORL framework [21], an integration of symbolic planning and hierarchical reinforcement learning. In the context of ML tasks, generating machine learning pipeline is treated as a symbolic planning problem [2]. We provide a formulation of dynamic domains that consist of actions such as *preprocessing*, *featurizing*, *cross validation*, *training* and *prediction*, using action language $\mathscr{BC}$ [8], a logic-based formalism that describes dynamic transition domain. Such formulation can be used to solve planning problem by a translation to answer set programming[9] and solving by an answer set solver such as CLINGO [4]. The pipeline is sent to execution by mapping to options [1] that consists of primitive actions in a Markov Decision Process [17] (MDP) space. The primitive actions in MDP space are ML pipeline components instantiated with random hyper-parameters, in order to learn the quality of the actions in the pipeline. The learning process is value iteration through reinforcement learning algorithm R-learning [11, 18], where cross-validation accuracy of the pipeline is used as rewards. After the quality of the current pipeline is measured, an improved ML pipeline is generated thereafter using the learned values, and the interaction with learning continues, until no better pipeline can be found. This step is called *model profiling*. After that, a more systematic parameter sweeping is performed, i.e., *model searching*. This allows us to describe the pipeline steps in an intuitive representation and explore the program space more systematically and efficiently with the help of reinforcement learning.

The paper is organized as follows. We start with a brief survey of related work in Sect. 11.2. After that, our method is presented in Sect. 11.4 and we will demonstrate its usage in Sect. 11.5. The conclusion of our paper with final summary is made in Sect. 11.6.

11.2 Related Work

Automated machine learning via automated programming touches several areas of prior art. In our work, we are mainly concerned with finding program structures and their parameters where the program primitives are steps in a machine learning pipeline. This is unlike other automated programming where the primitives may be the primitives in a programming language like Java or come from a domain specific language like string functions used to learn programs that manipulate data in a spreadsheet [6]. In machine learning pipelines, it is common for one to consider it as a task of moving a set of training data through various stages of data processing, creating features from which to build a model from, selecting features that are informative, sampling data for training models and enable various forms of testing and parameter tuning. Using a combination of knowledge about the data and knowledge about the machine learning techniques, practitioners typically go through a fair amount of trial-and-error to identify good machine learning pipelines. Our motivation is to provide assistance in this challenge to help the user identify good pipelines and the elements in them so that the user can be allowed to more quickly focus on other issues like model robustness and parameter tuning.

AutoML, or automated machine learning, also tries to learn a machine learning pipeline, but is primarily focused on learning an overall model that given features of the dataset will select with high probability various model steps and parameters. Two recent advances in AutoML are Auto-WEKA [19] and Auto-SKLEARN [3]. The Auto-WEKA approached the problem of automated machine learning by formalizing the problem as a *Combined Algorithm Selection and Hyperparameter optimization* problem (or CASH problem), whereby both the machine learning algorithm and its hyperparameters are selected for in a Bayesian optimization scheme.

Auto-SKLEARN extends Auto-WEKA approach with a meta-learning capability, ensemble model selection, and use of the popular Python machine learning library scikit-learn. While these approaches show impressive results, they are particularly suited for black-box style optimization where the users do not need to necessarily understand the underlying model, parameters or computational complexity, but rather focus on the performance of the model based on its predictions. Auto-SKLEARN builds a repository of top performing ML pipelines and their parameters offline on OpenML datasets, and when new data comes, by comparison with the original dataset on L1 scores, it chooses a few top performing ML pipelines and uses SMAC [7] for further offline tuning. In our work, where we want to assist a model developer, or data scientists, who may very much need to validate the model for various attributes due to possible industry regulations or sensitivities, we would like an approach that is more iterative and transparent to allow users to learn at the same time while providing guidance on what methods or parameters are useful.

With the TPOT approach[14], the authors more closely match the type of objectives we have as they search for good machine learning pipelines using the genetic programming algorithm and Python package DEAP. They compare their approach to several variations, including random search, over machine learning methods available in the scikit-learning Python library and find that while TPOT can find competent pipelines, the search time can be significant and the search heuristic does not seem to be more efficient than random search. The search algorithm with TPOT is different from AutoML which uses the SMAC algorithm and the representation of pipelines using genetic programming expression trees allows for a more expression, less constrained solution space. So while TPOT offers a compelling representation and intuitive search heuristic, the lack of pipeline constraints and the complexities within the fitness landscape may be responsible for the excessive run times and lack of competitiveness with a random search. In our approach, we want to explore if similarly richer representations of pipelines as programs, or plans, can be easily coupled with another intuitive search heuristic but achieve a more efficient search.

11.3 Background

11.3.1 Symbolic Planning

Symbolic planning concerns on formulating a dynamic transition system T where states denote the properties of the world (using a set of logical atoms) and the edges are actions, and reasoning over them to solve *planning problem*: given one initial state I and goal state D, find a path in T that starts from I and ends at G, and all actions on the edges of the path form a *plan*. Several languages have been developed to formulating the transition system such as PDDL [13] or action languages [5] that relates to logic programming under answer set semantics (answer set programming) [9]. In this paper, we will be using $\mathscr{BC}$ [8] that belongs to the latter family.

An *action description D* in the language $\mathscr{BC}$ includes two kinds of symbols, *fluent constants* that represent the properties of the world, denoted as $\sigma_F(D)$, and *action constants*, denoted as $\sigma_A(D)$. A fluent atom is an expression of the form $f = v$, where f is a fluent constant and v is an element of its domain. For boolean domain, denote $f = \mathbf{t}$ as f and $f = \mathbf{f}$ as $\sim f$. An action description is a finite set of *causal laws* that describe how fluent atoms are related with each other in a single time step, or how their values are changed from one step to another, possibly by executing actions. For instance,

$$A \textbf{ if } A_1, \ldots, A_m$$

is a *static law* that states at a time step, if $A_1, \ldots, A_m$ holds then A is true. Another static law

$$\textbf{default } f = v$$

states that by default, the value of f equals v at any time step.

$$a \textbf{ causes } A_0 \textbf{ if } A_1, \ldots, A_m$$

is a *dynamic law*, stating that at any time step, if $A_1, \ldots, A_m$ holds, by executing action a, A_0 holds in the next step.

$$\textbf{nonexecutable } a \textbf{ if } A_1, \ldots, A_m$$

states that at any step, if $A_1, \ldots, A_m$ holds, action a is not executable. Finally, the dynamic law

$$\textbf{inertial } f$$

states that by default, the value of fluent f does not change from one step to another, formalizing the *commonsense law of inertia* that addresses the frame problem.

Automated planning with an action description in $\mathscr{BC}$ can be achieved by an answer set solver such as CLINGO,[1] and an output answer set encodes a sequence of actions that solves the planning problem.

11.3.2 Reinforcement Learning

A Markov Decision Process (MDP) [17] is defined as the tuple $(\mathscr{S}, \mathscr{A}, P^a_{ss'}, r, \gamma)$, where $\mathscr{S}$ and $\mathscr{A}$ are the sets of symbols denoting states and actions, the transition kernel $P^a_{ss'}$ specifies the probability of transition from the state $s \in \mathscr{S}$ to the next state $s' \in \mathscr{S}$ by taking an action $a \in \mathscr{A}$, $r(s, a) : \mathscr{S} \times \mathscr{A} \mapsto \mathbb{R}$ is a reward function bounded by $r_{\max}$, and $0 \leq \gamma < 1$ is a discount factor. A solution to an MDP is a policy $\pi : \mathscr{S} \mapsto \mathscr{A}$ that maps a state to an action. RL concerns on learning a near-optimal policy by executing actions and observing the state transitions and rewards, and it can be applied even when the underlying MDP is not explicitly given, a.k.a, model-free policy learning.

To evaluate a policy π, there are two types of performance measures: the expected discounted sum of reward for infinite horizon problems and the expected un-discounted sum of reward for finite horizon problems. In this paper we adopt the latter metric defined as $J^\pi_{\text{avg}}(s) = \mathbb{E}[\sum_{t=0}^{T} r_t | s_0 = s]$. We define the *gain reward* $\rho^\pi(s)$ reaped by policy π from s as

$$\rho^\pi(s) = \lim_{T\to\infty} \frac{J^\pi_{\text{avg}}(s)}{T} = \lim_{T\to\infty} \frac{1}{T}\mathbb{E}[\sum_{t=0}^{T} r_t].$$

R-learning [11, 18] is designed for the average reward case which is a model-free value iteration algorithm that can be used to find the optimal policy for average reward criteria. At the t-th iteration (s_t, a_t, r_t, s_{t+1}), update:

$$\begin{aligned} R_{t+1}(s_t, a_t) &\overset{\alpha_t}{\longleftarrow} r_t - \rho_t(s_t) + \max_a R_t(s_{t+1}, a), \\ \rho_{t+1}(s_t) &\overset{\beta_t}{\longleftarrow} r_t + \max_a R_t(s_{t+1}, a) - \max_a R_t(s_t, a) \end{aligned} \tag{11.1}$$

where α_t, β_t are the learning rates, and $a_{t+1} \overset{\alpha}{\longleftarrow} b$ denotes the update law as $a_{t+1} = (1 - \alpha)a_t + \alpha b$.

Compared with regular reinforcement learning, hierarchical reinforcement learning (HRL) [1] specifies on real-time-efficient decision-making problems over a

[1] http://potassco.sourceforge.net/.

series of tasks. A MDP can be considered as a flat decision-making system where the decision is made at each time step. On the contrary, humans make decisions by incorporating temporal abstractions. An option is temporally extended course of action consisting of three components: a policy $\pi : \mathscr{S} \times \mathscr{A} \mapsto [0, 1]$, a termination condition $\beta : \mathscr{S} \mapsto [0, 1]$, and an initiation set $\mathscr{I} \subseteq \mathscr{S}$. An option (I, π, β) is available in state s_t iff $s_t \in I$. After the option is taken, a course of actions is selected according to π until the option is terminated stochastically according to the termination condition β. With the introduction of options, the decision-making has a hierarchical structure with two levels, where the upper level is called the option level (also termed as task level) and the lower level is called the (primitive) action level. The Markovian property exists among different options at the option level.

11.4 Methodology

In this section we describe our approach of integrating ML pipeline search and parameter tuning into an unified framework. Intuitively, generating machine learning pipelines and tuning parameters are operations that are different levels of abstraction. Given a dataset and a ML task (i.e., classification or regression), the data scientist begins by empirically choosing a pipeline, a sequence of operations such as preprocessing, feature computation, feature selection, model selection, cross validation, training and prediction. After the high-level pipeline is chosen, different sets of parameters for preprocessing, featurizers and models will be empirically tried, evaluated and selected based on certain metric. It is possible that the original pipeline is modified due to the poor performance observed during parameter tuning, leading to a different pipeline that produces better results on the dataset. The loop of searching for ML pipelines, tuning parameters, observing results, retuning parameters, and refining the ML pipelines continues until the data scientist cannot find better solutions. The data scientist finally comes up with the pipeline and respective parameters that have the best performance on the dataset.

Our approach of automating the process above is the integration of symbolic planning with reinforcement learning. The high-level pipeline generation is treated as a symbolic planning problem studied by the classical AI planning community. The low-level parameter sweeping is treated as a reinforcement learning problem. Symbolic planning and reinforcement learning are integrated using the recent result of PEORL (Planning—Execution—Observation—Reinforcement Learning) framework [21].

11.4.1 Pipeline Generation

We use action language $\mathscr{BC}$ to represent dynamic domain of ML operations. We first introduce three types of objects:

- Preprocessors, including
  - matrix decompositions (`truncatedSVD`,`pca`,`kernelPCA`,`fastICA`),
  - kernel approximation (`rbfsampler`,`nystroem`),
  - feature selection (`selectkbest`,`selectpercentile`),
  - scaling (`minmaxscaler`,`robustscaler`,`absscaler`), and
  - no preprocessing,
- Featurizers: including two standard featurizers for text classification, i.e., `Count-Vectorizor` and `TfidfVectorizer`.
- Classifier: including logistic regression, Gaussian naive Bayes, linear SVM, random forest, multinomial naive Bayes and stochastic gradient descent.

We treat each operation in the pipeline as an action, with describing their causal laws accordingly.

- Facts about compatibility with sparse vectors.

$$\begin{array}{l} acceptsparse(randomforest). \quad acceptsparse(linearsvc). \\ acceptsparse(logistic). \quad acceptsparse(sgd). \\ acceptsparse(truncatedsvd). \quad acceptsparse(kernelpca). \\ acceptsparse(noop). \quad acceptsparse(fastica). \\ acceptsparse(nystroem). \quad acceptsparse(rbfsampler). \end{array}$$

- Importing dataset. If the dataset D has type T (i.e., training or testing), then import data Y has the effect of having text and labels from data D of type T.

$$import(D)\ \textbf{causes}\ hasdata(D, text, T), hasdata(D, label, T)\ \textbf{if}\ datatype(Y, T)$$

- Tokenize dataset. If we have text from the data, then it can be tokenized, leading to the results of having tokens:

$$tokenize(text, D)\ \textbf{causes}\ hasdata(D, token, T)\ \textbf{if}\ hasdata(D, text, T)$$

- Select featurizer:

$$selectfeaturizer(F)\ \textbf{causes}\ featurizerselected(F)\ \textbf{if}\ featurizer(P)$$

- Select preprocessor:

$$selectpreprocessor(P)\ \textbf{causes}\ proprocessorselected(P)\ \textbf{if}\ preprocessor(P)$$

- Crossvalidate. If we have token and label from the data, we can cross validate the pipeline by choosing a featurizer, preprocessor and a classifier, and the effect is the model being validated. If one of preprocessor and classifier does not accept sparse vector, it needs to be transformed into dense vector.

$crossvalidate(C, F, P, dense, token, label)$ **causes**
$\quad modelvalidated(C, F, P, dense, token, label)$
$\quad\quad$ **if** $classifier(C), featurizerselected(F), preprocessorselected(P),$
$\quad\quad\quad hasdata(D, token, train), hasdata(D, label, train), \sim acceptsparse(P).$

$crossvalidate(C, F, P, dense, token, label)$ **causes**
$\quad modelvalidated(C, F, P, dense, token, label)$
$\quad\quad$ **if** $classifier(C), featurizerselected(F), preprocessorselected(P),$
$\quad\quad\quad hasdata(D, token, train), hasdata(D, label, train), \sim acceptsparse(C).$

$crossvalidate(C, F, P, sparse, token, label)$ **causes**
$\quad modelvalidated(C, F, P, sparse, token, label)$
$\quad\quad$ **if** $classifier(C), featurizerselected(F), preprocessorselected(P),$
$\quad\quad\quad hasdata(D, token, train), hasdata(D, label, train), acceptsparse(P),$
$\quad\quad\quad acceptsparse(C).$

- Featurize. If we have token from the data, and the model that uses the featurizer has been validated, then featurizing tokens has the effect of generating raw features.

$featurize(F, token, D)$ **causes** $hasdata(D, feature, T)$
$\quad$ **if** $hasdata(D, token, T), modelvalidated(C, F, P, S, token, label).$

- Preprocess. If we have raw feature computed from the data, and the model that uses the preprocessor has been validated, then preprocessing the features leads to the feature been processed.

$preprocess(P, feature, S, D)$ **causes** $processed(D, feature, S, T)$
$\quad$ **if** $hasdata(D, feature, T), modelvalidated(C, F, P, S, token, label).$

- Train. If we have feature preprocessed, and the model that uses the classifier has been validated, training the classifier leads to the model been trained.

$train(C, F, P, token, label)$ **causes** $modeltrained(D, text, label)$
$\quad$ **if** $modelvalidated(C, F, P, S, token, label), processed(D, feature, train).$

- Predict. If model has been trained using classifier, featurizer and preprocessors, and test data has been featurized and preprocessed, then predicting on test data set gives us labels.

$predict(C, F, P, label)$ **causes** $hasdata(D, label, test)$
$\quad$ **if** $modelvalidated(C, F, P, S, token, label), processed(D, feature, test).$

Besides causal laws described above, all fluents are declared inertial, and concurrent execution of actions are prohibited. Given a training data set as initial condition

$$datatype(data, train), datatype(datatest, test)$$

and a goal *hasdata*(*datatest*, *label*, *test*), a plan can be generated by translating the action description above to ASP and run answer set solver CLINGO

1 : *import*(*data*) 2 : *tokenize*(*text*, *data*)
3 : *selectfeaturizer*(*countervectorizer*) 4 : *selectpreprocessor*(*kernelPCA*)
5 : *crossvalidate*(*randomforest*, *countervectorizer*, *kernelPCA*,*sparse*,*tokens*,*label*)
6 : *featurize*(*countervectorizer*, *token*, *data*)
7 : *preprocess*(*kernelPCA*, *sparse*,*feature*, *data*)
8 : *train*(*randomforest*, *countervectorizer*, *kernelPCA*, *token*, *label*)
9 : *import*(*datatest*) 10 : *tokenize*(*text*, *datatest*)
11 : *featurize*(*countervectorizer*, *token*, *datatest*)
12 : *preprocess*(*kernelPCA*, *sparse*,*feature*, *datatest*)
13 : *predict*(*randomforest*, *countervectorizer*, *kernelPCA*, *label*)

Currently we only pick up one feature preprocessors, following the same paradigm with Auto-SKLEARN [14]. This leads to total of 132 pipelines. But the action description can be formulated such that more than one preprocessor can be applied sequentially.

Automated planning provides the flexibility of generating ML pipelines given different initial and goal state. For instance, given goal *modeltrained*(*data*, *text*, *label*) only the first six steps of the plan above will be generated. Given an initial state such as as *hasdata*(*data*, *token*, *train*), the pipeline skip the first two steps and start cross-validation. Finally, the user can also change the set of preprocessors, featurizers and classifiers to only explore a part of all possible pipelines. This paradigm is one of the differentiation of our method in comparison to Auto-SKLEARN that specifies fixed ML pipeline structure.

11.4.2 Parameter Tuning and Pipeline Evaluation

In our framework, we map symbolic actions to options, in the sense of hierarchical reinforcement learning. In particular:

- *selectfeaturizer* is mapped to $\widetilde{selectfeaturize}$ which in practice just checks the availability of the featurizer by printing out a checking status.
- *selectpreprocessor* is mapped to $\widetilde{selectpreprocessor}$ which in practice just checks the availability of the preprocessor by printing out a checking status.

- *crossvalidate*$(M, F, P, S, token, label)$ into an option of three sequential steps in MDP space if $S = sparse$[2]

$$\widetilde{featurize}(\tilde{F}, token), \widetilde{preprocess}(\tilde{P}, token, label), \widetilde{validate}(\tilde{M}, \tilde{F}, \tilde{P}, token, label) \tag{11.2}$$

  or four sequential steps if $S = dense$

$$\begin{gathered}\widetilde{featurize}(\tilde{F}, token), \widetilde{todense}(\tilde{F}, token) \\ \widetilde{preprocess}(\tilde{P}, token, label), \widetilde{validate}(\tilde{M}, \tilde{F}, \tilde{P}, token, label)\end{gathered} \tag{11.3}$$

  where $\tilde{M}$, $\tilde{F}$, $\tilde{P}$ are instantiation of the classifier M, featurizer F and preprocessor P with a set of parameters that can be selected by any parameter sweeping algorithm such as grid search, random search or Bayesian optimization. *todense* transforms the raw features into dense vector, if specified by the symbolic action. After each MDP action is performed, perform value iteration by following Eq. (11.1), where the final cross-validation score is the reward.

After the option terminates, update the value for the option by following Eq. (11.1) where the reward value of performing *crossvalidate*$(M, F, P, token, label)$ comes from the reward the option achieved. Following the methodology proposed in [21], ρ values are the learned score for the action performed at state following the policy. Since ρ values are learned at every state and every action in the MDP space, eventually this method allows us to learn the score not only for the pipeline, but also for individual classifier, featurizer, preprocessors and their combinations, providing valuable insight to the data scientists to build ML models that work well with the dataset.

11.4.3 AutoML Framework

The complete meta-learning algorithm is described in Algorithm 11. It includes two major steps:

1. **Model Profiling**. In this step, symbolic planner generates a plan (line 4) with the goal *modeltrained*(*data*), execute each action (lines 9–27). The algorithm performs R-learning for *selectfeaturizer* and *selectpreprocessor* (lines 10,11). When it it executes cross-validation, it performs episode iteration by randomly sampling hyper parameters on classifier, featurizer and preprocessors, perform cross validation and R-learning from the rewards (lines 13–20). It updates the values for the symbolic action (line 16). This model profiling process goes on for a given pipeline for a number of episodes, before switching to a different symbolic plan such that the plan quality is better than the current one (line 24).

[2]Tildes are used to distinguish actions in MDP space from actions in symbolic planning space.

In practice, the symbolic planner will either generate a pipeline that hasn't been tried before (all pipeline starts with a good initial score so that the exploration would be encouraged at the beginning), or pick up any one that was profiled before with the score higher then the current one. This process continues until the best ML pipeline converges to an optimal Π_o (line 5). After that, it goes to Step 2.

2. **Model Searching**. After Π_o is obtained, perform parameter sweeping on Π_o. In this stage all computational resources can be used to search for a wide space of parameters for classifier C, featurizer F and preprocessor P specified by Π_o, and return the best one when the search finishes.

The meta-learning loop to generate ML parameters by integrating symbolic planning and reinforcement learning is detailed in Algorithm 11. D denotes the action description formulated for ML pipeline domain, as is described in Sect. 11.4.1.

Algorithm 11 Meta-learning loop

```
Require: $(I, G, D, \mathbb{F}_A)$ where $G = (modeltrained(data), \emptyset)$, and an exploration probability $\epsilon$. A set of symbolic actions where R-learning will be performed for: $L = \{selectfeaturizer(F), selectpreprocessor(P), crossvalidate(C, F, P, label, data)\}$.
1: $P_0 \Leftarrow \emptyset, \Pi \Leftarrow \emptyset$
2: while True do
3:     $\Pi_o \Leftarrow \Pi$
4:     take $\epsilon$ probability to solve planning problem and obtain a ML pipeline $\Pi \Leftarrow$ CLINGO.$solve(I, G, D \cup P_t)$
5:     if $\Pi = \Pi_o$ then
6:         perform parameter sweeping on $\Pi_o$ and obtain the optimal $C, F, P$.
7:         return $(C, F, P)$
8:     end if
9:     for action $\langle s, a, s' \rangle \in \Pi$ do
10:        if $a \in [selectfeaturizer(F), selectpreprocessor(P)]$ then
11:            update $R(a, s)$ and $\rho_t^a(s)$ for action $a$
12:        end if
13:        if $a \in [crossvalidate(C, F, P, D, label, data)]$ then
14:            for $i < episode$ do
15:                instantiate $C, F, P$ by sampling their hyper-parameters.
16:                assemble pipeline using $\tilde{C}, \tilde{F}, \tilde{P}$ and $D$.
17:                perform cross validation using $\tilde{C}, \tilde{F}, \tilde{P}$
18:                update $R$ and $\rho$ by R-learning value iteration until the option terminates,where $reward \propto cvscore$.
19:                update $R(a, s)$ and $\rho_t^a(s)$ for action $a$
20:                $i \leftarrow i + 1$
21:            end for
22:        else
23:            execute $a$
24:        end if
25:    end for
26:    $quality(\Pi) \leftarrow \rho^a(s)$, where $a \in L \subset \Pi$
27:    update planning goal $G \Leftarrow (A, quality > quality(\Pi))$.
28:    update facts $P_t \Leftarrow \{\rho(a) = z : \rho_t^a(s) = z\}$, where $a = crossvalidate(C, F, P, label, data)$
29: end while
```

11.5 Empirical Evaluation and Discussion

To better the approach, we perform several experiments using a well known dataset, the IMDB movie review for sentiment analysis [10]. This dataset contains 10,000 movie reviews, with 5000 labeled as "positive" and 5000 labeled as "negative." Next, we motivate the empirical evaluation, summarize the known results of classification using this data, and then conclude with a discussion of AutoML using our approach on this data.

11.5.1 Dataset and Problem Instance

In prior work, a variety of datasets are used to test pipeline learning, where most datasets are already featurized into numerical values that are ready for machine learning classifiers. Some datasets have been heavily preprocessed to remove missing values, unnecessary features, and even newly created features from the raw data. These critical steps in the machine learning pipeline can not be assumed to have been completed in real-world settings, particularly in the industrial applications. Therefore, we used a common industrial problem where it is very common to work directly with raw data: document classification.

The task of document classification is that, given a set of documents of pieces of text, classify each into a given category. One example of this is sentiment analysis. Given a set of texts (e.g. movie reviews, product reviews), label if the text is positive or negative. [10] collected a large dataset of movie reviews labeled with their sentiment. This dataset (commonly called the IMDB datset) has since been used as a general benchmark dataset for NLP and machine learning tasks in general. It contains 25,000 records of movie reviews which express very strong sentiment, along with a 'positive' or 'negative' label for each, and 25,000 reviews which are unlabeled. The authors were able to achieve an accuracy of 88.89% with their model.

Their approach included a variety of machine learning tasks. Their base model was a linear support vector machine (linear SVC) with multiple features. While they achieved their highest score by learning a bag of words model and learning a word vector model, they also tested multiple other methods of generating features for comparison. These included latent Dirichlet allocation and latent semantic analysis. In our experiment, we sampled 300 labeled movie reviews, with 150 reviews in each category, to run meta-learning. We use meta-learning to find the best pipeline, and then test the results of using this pipeline with hyper-parameter search on two larger benchmark datasets.

In addition, two other common techniques used in document classification are dimensionality reduction and feature selection. Very often large sparse matrices are used built-in training a document classifier. This can often be inefficient and potentially over-fit the data. By reducing the size of the overall matrix with techniques such as principal component analysis (PCA), whereby the data is reduced to a smaller set of linearly uncorrelated variables, or select k best, where a statistical test is applied and a representative set of uncorrelated features are chosen, the trained model can generalize much better to unseen data.

11.5.2 Hypotheses

We believe that our approach for AutoML will help to address a few key areas for learning machine learning pipelines: provide a computationally efficient approach that helps users develop a feasible solution, capture and convey knowledge about the machine learning pipeline options for the particular problem instance, and provide an alternative mechanism for reinforcement learning over pipelines. Besides helping the AutoML space, we hope that our approach will also be useful for the more general automatic programming problem for similar reasons.

1. To test our hypotheses that the approach is viable, we will first validate that the approach can indeed find competitive solutions in a reasonable computationally efficient framework by comparing accuracy with run-time.
2. To test that knowledge from AutoML can be easily extracted, we will examine the usefulness of the rewards conveyed to pipelines and pipeline steps.
3. To test that our reinforcement learning approach, which leverages symbolic planning, we will look at the convergence efficiency of learning.

Additionally, we can look at the results to see if we are able to "rediscover" common knowledge about machine learning. Specifically about which techniques tend to work well together and on different problems. For example, in the realm of document classification, data scientists frequently use combinations of classifiers suited for text (e.g. SVM and logistic regression) over others.

11.5.3 Pipeline Profiling

In total we evaluated 80 pipeline profiling results which are shown in Figs. 11.1 and 11.2. These are curves of the ρ values plotted from R-learning over the complete pipeline, i.e., the gain reward of following the complete policy. Symbolic planner picked up pipeline by the metric on its plan quality, i.e., the sum of ρ values for applying featurizer, preprocessor and classifier. Some pipelines where dropped very early in profiling stages because of the low cross-validation score it receives (such as the ones in bottom left of Fig. 11.1), and for this reason the score of the whole pipeline remains flat. Some pipelines start with good results but after further attempts, the performance degrades and were

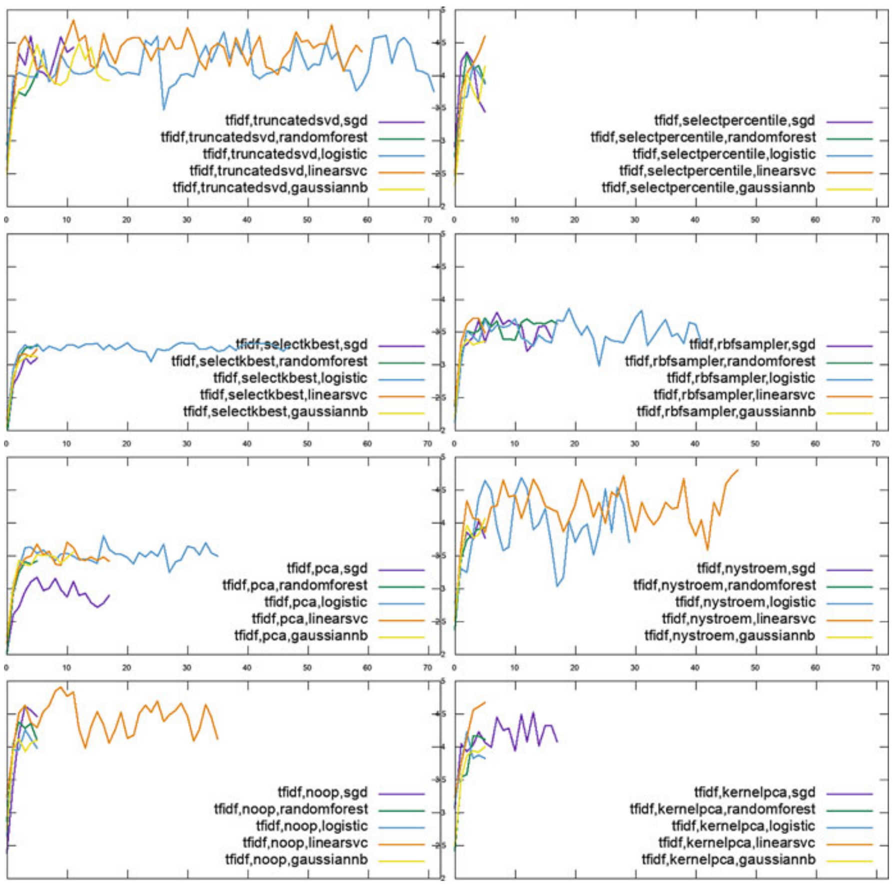


Fig. 11.1 Profiling of half the pipelines, grouped by featurizer and preprocessor, showing the ρ values (Y-Axis) against the total number of episodes (X-Axis) given to each pipeline, indicating how promising each pipeline was determined to be during learning. Figure 11.2 contains the other half of the results

dropped, such as TFIDF+Nystroem+SGD classifier. Finally, the good ones are constantly picked up for further profiling and leads to the final best scored pipelines, such as TFIDF+Nystroem+linear SVC and bag of words+fast ICA+SGD classifier.

The figures on profiling progress suggests that by using cross-validation score as a reward, reinforcement learning can gradually get rid of less promising pipelines considering the gain reward of each step of the pipeline, selectively pick up more promising pipelines for more profiling and dramatically narrow down the scope of interested pipeline. Planning-guided reinforcement learning has shown to improve data efficiency and leads to rapid policy search.

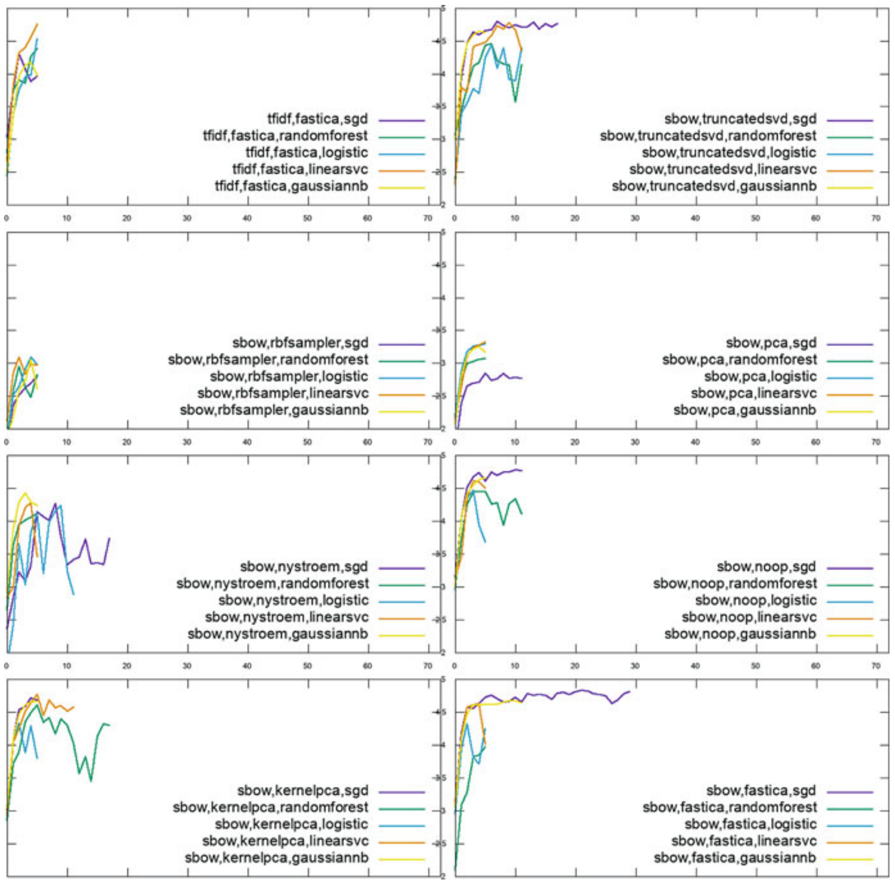


Fig. 11.2 Profiling of half the pipelines, grouped by featurizer and preprocessor, showing the ρ values (Y-Axis) against the total number of episodes (X-Axis) given to each pipeline, indicating how promising each pipeline was determined to be during learning. Figure 11.1 contains the other half of the results

11.5.4 Optimal Pipeline Generation

After running our algorithm on a selection of 300 documents dataset from the IMDB dataset, we obtain the optimal pipeline to be: Bag of words, fast ICA and stochastic gradient descent (SGD), with the following parameters:

- Bag of words: ngram_range = (1,2), lowercase = False
- Fast ICA: n_components = 3
- SGD Classifier: loss=log, penalty=l2.

The second best pipeline uses TF-IDF, Nystroem preprocessor and linear SVC classifier. In particular, their hyper parameters are:

- TFIDF: lowercase=True, max_df=0.93620, min_df=0.014796, ngram_range=(1, 1)
- Nystroem preprocessor: gamma=0.2, random_state=1
- Linear SVC: C=0.87664, class_weight=None, dual=True, fit_intercept=True, intercept_scaling=1, loss='squared_hinge', max_iter=1000, multi_class='ovr', penalty='l2', random_state=None, tol=1e-05, verbose=0.

This pipeline is shown to achieve fivefold cross validation mean accuracy to be 0.9. The third best pipeline that achieves cross validation score of 0.896 is hashing vectorizor, truncateSVD preprocessor and stochastic gradient descent. Their hyper-parameters are

- Bag of words: ngram_range = (1, 3),lowercase=False
- Truncated SVD: n_component = 2, iter = 8
- SGD Classifier: learning_rate='optimal', loss='log', penalty='l1'.

The third best pipeline uses bag of words, Nystroem preprocessor and linear SVC. The output of meta-learning pipeline in general agrees with the result investigated earlier [12, 16].

To evaluate the generality of the highly scored pipeline for larger dataset, we further use polarity dataset 2.0, [3] a smaller dataset used by [12, 16] containing 2000 movie reviews. It returns a model with cross validation accuracy of 0.84. The hyper parameters are

- Bag of words: ngram_range = (1,3), lowercase = True
- Fast ICA: n_components = 3
- SGD classifier: loss = modified_huber, penalty=elasticnet.

Finally, we tried this pipeline using the full IMDB dataset [10] with a cross-validation score of 0.88, with the hyper parameters as follows:

- Bag of words: ngram_range = (1,1), lowercase = False
- Fast ICA: n_components = 3
- SGD Classifier: loss=log, penalty=None.

The scores achieved by automated pipeline search is close to the accuracy achieved by the original authors on the same dataset [10].

11.5.5 Pipeline Ranking

We tested our algorithm in movie review data set and ranked over five classifiers (logistic regression, random forest, linear SVC, SGD classifier, Gaussian NB), two featurizers (bag of words and TFIDF), and preprocessors (truncated SVD, PCA, rbf

[3] http://www.cs.cornell.edu/people/pabo/movie-review-data/.

Table 11.1 Top five ranked pipelines by mean accuracy

Pipeline	Mean accuracy	ρ
Bag of words/fastICA/SGD classifier	0.88	4.83
Bag of words/noop/SGD classifier	0.87	4.77
Bag of words/truncatedSVD/SGD classifier	0.87	4.80
Bag of words/kernelPCA/SGD classifier	0.86	4.66
TFIDF/fastICA/linear SVC	0.86	4.76

Table 11.2 Bottom five ranked pipelines by mean accuracy

Pipeline	Mean accuracy	ρ
Bag of words/rbf sampler/logistic regression	0.79	3.10
Bag of words/rbf sampler/random forest	0.52	3.00
Bag of words/pca/SGD classifier	0.51	2.84
Bag of words/rbf sampler/GaussianNB	0.51	3.06
Bag of words/rbf sampler/SGD classifier	0.51	2.8

sampler, Nystroem, select k best, select percentile, fast ICA, kernel PCA, and no preprocessor [noop]) (Tables 11.1 and 11.2).

It should be noted that the score obtained from ML pipeline profiling is closely related to the profiling episode and the parameter sweeping strategy. Longer profiling episodes and more intelligently chosen parameter space may find better parameters, boost the score and reflect the true performance of the overall pipeline. The user has the flexibility of changing those configuration based on their own use cases and constraints on time and computational resources. The results also show that the highest accuracy does not necessarily show the highest learning rate (ρ). This is due to us taking the mean accuracy. While a single run of the of a pipeline may produce a higher score, the overall learning rate of this pipeline with different parameters may be lower.

On this dataset, the classifiers commonly used for text rose to the top. The meta-learning algorithm tended to favor both logistic regression and the linear svc classifier over the non-linear random forest. These pipelines, along with the sparse features used, tend to be favored more by data scientists in document classification. The pipeline search managed to hit upon commonly used techniques, whereby kernal approximation techniques (e.g. Nystroem) are used with linear svc.

However, it should be noted that each classifier was able to find a pipeline with $>80\%$ accuracy. Given the features and the classifier, finding the hyper-parameters and the preprocessing step were just ways of tweaking overall performance.

The ranking for the combination of feature and preprocessors can also be learned from the reinforcement learning process, indicating how promising these two steps are to lead to an ideal final result (Figs. 11.3 and 11.4).

The ρ values of the feature/preprocessor combinations are more important relative to each other than in the absolute value. This is because: (1) the algorithm

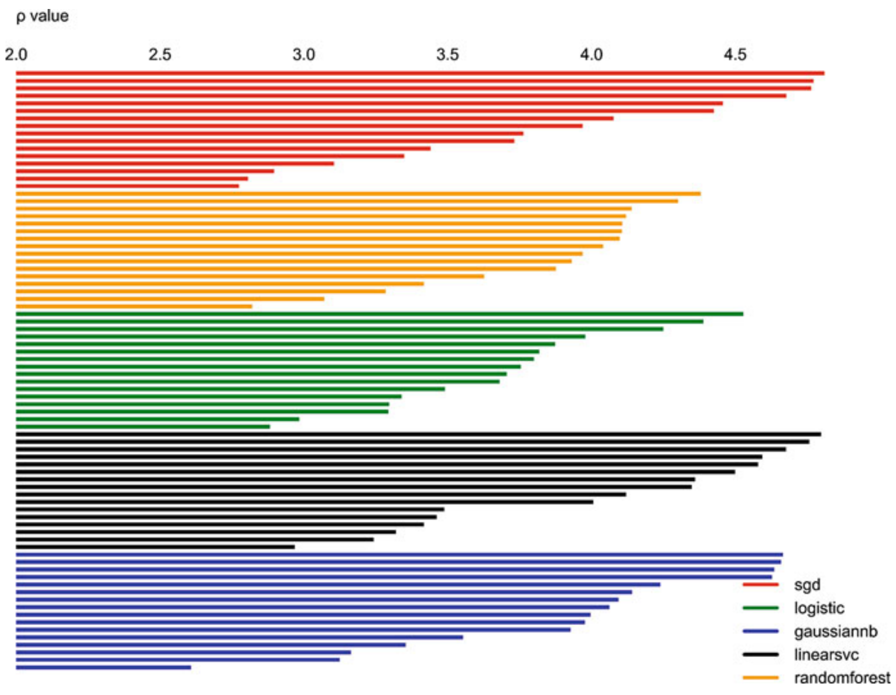


Fig. 11.3 Learning rate for each pipeline trained by classifier (higher is better)

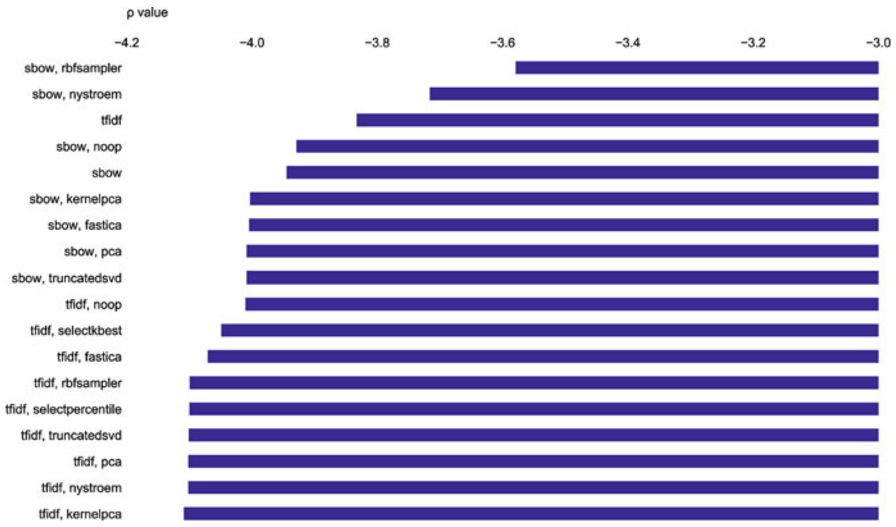


Fig. 11.4 Pipeline ranking results for featurizer and preprocessors review data set (higher is better)

will select the pipelines with higher ρ scores to pursue, as long as they rank relatively higher and, (2) a combination can ranked pretty low but when combined with a particular model, the overall pipeline is ranked high.

11.5.6 *Extracted Knowledge About Problem*

After searching the space of machine learning pipelines, we can begin to ask useful questions for the dataset. Questions such as, what are the good ways to featurize it? Or, given an existing featurization of the data, what are the best ways to preprocess it? What are the best classifiers to use? To demonstrate this idea, we collect the learned ρ values for the pipelines starting from the initial featurization selection, the featurization selection plus the selected preprocessor, and finally the entire pipeline. We keep only the two options with the greatest ρ value, and place them in the decision tree, in Fig. 11.5.

One of the benefits of AutoML is not just parameter tuning or automation of repetitive trial and error exploration of data and analytic selections, but AutoML

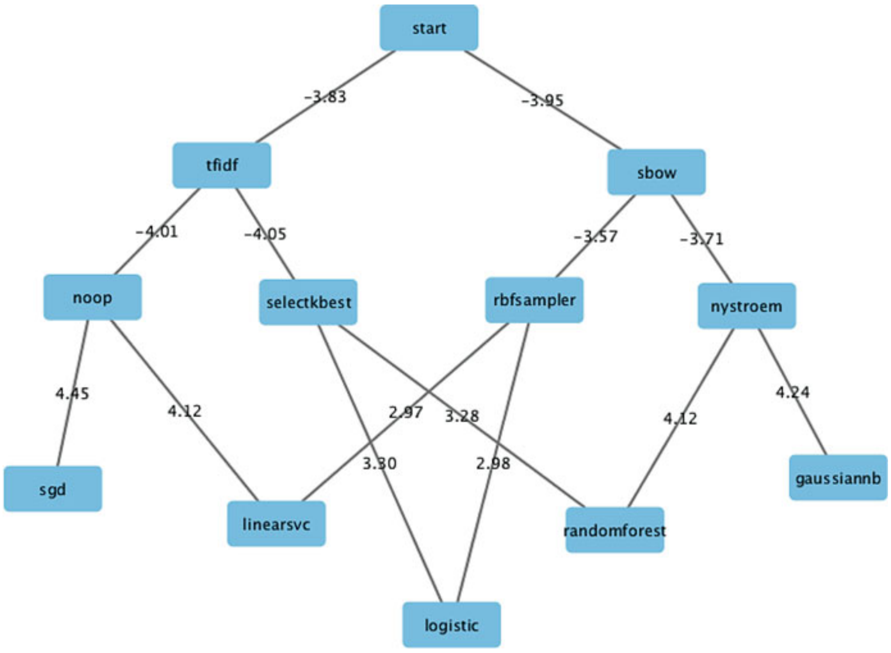


Fig. 11.5 Learned knowledge about machine learning pipelines specific to the task of document classification. Each node can be thought of as a made decision, and each directed edge with a ρ value as a choice, where the higher ρ value suggests a better option. Note that the edges and ρ values are dependent on the prior selections in the pipeline

allows a significant amount of knowledge to be gathered about the problem instance and what methods might be more suitable, regardless of whether the optimal parameter values were found. As can be seen in Fig. 11.5, the user has two options for featurizing their data, either using the TFIDF or bag of words approach. But given the selection of either one, they have two distinct choices for a preprocessing technique. Finally, there are some choices of classifiers. If the deployment environment has certain technical requirements, like scale or availability of performant libraries, this type of knowledge can be helpful to the data scientist. Additionally, being able to generate knowledge about the machine learning pipelines also helps us better understand the problem domain.

11.6 Conclusion and Future Work

We explore the use of PEORL framework for an AutoML task of learning machine learning pipelines. Our approach uses a symbolic planner to generate the solution space of possible pipeline steps, and reinforcement learning to inform the selection of the next best program to explore based on a crossvalidated, parameter-tuning step that is specific to the selected pipeline elements. The combination of the symbolic planner with reinforcement learning provides some nice properties of convergence while at the same time allowing the user to get a coarse grain idea of what a good machine learning pipeline would look like. To demonstrate this idea, we leverage benchmark dataset for sentiment analysis, where the machine learning problem is to learn to classify movie reviews into positive or negative. We show that the approach is feasible and provides several additional benefits that allow the user to reason and gather new knowledge about the pipelines for their specific problems. Our future work is to improve the computational efficiency of PEORL, testing on larger and more diverse datasets, and gather real-world user experience feedback to shape the next areas of feature research and development.

This preliminary work provides a potential in the future to perform more informative inference on the data obtained during profiling ML pipeline on the dataset that the data scientist needs to tackle. Such inference may be called “data science for data science”, which will provide a quick and valuable suggestion for data scientist of building fine-tuned models for complex dataset, such as providing ranking results on pipeline and their individual pieces, sensitivity ranking and many other suggestions. The best model it learned can also be a baseline for data scientist for further fine tuning as well.

References

1. Barto, A., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discrete Event Systems Journal **13**, 41–77 (2003)
2. Cimatti, A., Pistore, M., Traverso, P.: Automated planning. In: F. van Harmelen, V. Lifschitz, B. Porter (eds.) Handbook of Knowledge Representation. Elsevier (2008)
3. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems, pp. 2962–2970 (2015)
4. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187–188**, 52–89 (2012)
5. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence (ETAI) **6** (1998)
6. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, pp. 507–523. Springer (2011)
8. Lee, J., Lifschitz, V., Yang, F.: Action Language $\mathcal{BC}$: A Preliminary Report. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 983–989 (2013)
9. Lifschitz, V.: What is answer set programming? In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1594–1597. MIT Press (2008)
10. Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C.: Learning word vectors for sentiment analysis. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 142–150. Association for Computational Linguistics, Portland, Oregon, USA (2011)
11. Mahadevan, S.: Average reward reinforcement learning: Foundations, algorithms, and empirical results. Machine Learning **22**, 159–195 (1996)
12. Martineau, J., Finin, T.: Delta TFIDF: An Improved Feature Space for Sentiment Analysis. In: Proceedings of the Third AAAI Internatonal Conference on Weblogs and Social Media, pp. 258–261. AAAI Press, San Jose, CA (2009)
13. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL-the planning domain definition language. Tech. Rep. CVC-TR-98–003, Yale Center for Computational Vision and Control (1998)
14. Olson, R.S., Bartley, N., Urbanowicz, R.J., Moore, J.H.: Evaluation of a tree-based pipeline optimization tool for automating data science. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, pp. 485–492. ACM, New York, NY, USA (2016)
15. O'Reilly, U.M., Oppacher, F.: Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In: Y. Davidor, H.P. Schwefel, R. Männer (eds.) Parallel Problem Solving from Nature — PPSN III, pp. 397–406. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
16. Pang, B., Lee, L.: A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In: Proceedings of the 42nd annual meeting on Association for Computational Linguistics, p. 271. Association for Computational Linguistics (2004)
17. Puterman, M.L.: Markov Decision Processes. Wiley Interscience, New York, USA (1994)
18. Schwartz, A.: A reinforcement learning method for maximizing undiscounted rewards. In: Proceedings of the Tenth International Conference on International Conference on Machine Learning, ICML'93, pp. 298–305. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
19. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: Proc. of KDD-2013, pp. 847–855 (2013)

20. Wolpert, D.H.: The lack of a priori distinctions between learning algorithms. Neural Computation **8**, 1341–1390 (1996)
21. Yang, F., Lyu, D., Liu, B., Gustafson, S.: Peorl: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pp. 4860–4866. International Joint Conferences on Artificial Intelligence Organization (2018)

Index

A
ABS ratio, 186
Acknowledgments, x
Action, atomic, 47
Action, non-atomic, 47
Affenzeller, Michael, 85
Agent, 37
ALE, 41
ANN, 137
Artificial intelligence, 17, 179, 209
Atari learning environment, 41
Automatic programming, 179, 209
AutoML, 210
Auto-WEKA, 210

B
Bee, 69
Bell Labs, 181
Big data, 194
Bohm, Clifford, 17
Brain, 145
Burlacu, Bogdan, 85

C
Centroid, 66
CGP, 39, 141, 149
Classification problem, 156
Classification, symbolic, 59
Cluster, 66
Clustering, 85, 91
Constraint satisfaction, 126
Cussat-Blanc, Sylvain, 137

D
Dedication, v
Deep learning, 37, 59, 72, 193
Dimensionality reduction, 85
Distribution, 86
Dolson, Emily, 1
Donor, 195

E
Elkholy, Alexander, 209
Energy efficiency problem, 197
Events, 104

F
Fagan, David, 179
Foraging, 26

G
Genetic network programming, 39
Genetic regulatory network, 139
GNP, 39
GRN, 139
Gustafson, Steven, 209

H
Hernández, Daniel E., 193
Heywood, Malcolm I., 37
Hintze, Arend, 17

I
IMDB, 224
Inverted pendulum, 25

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1

K
Kammerer, Lukas, 85
Kelly, Stephen, 37
KNIME, 60
Kommenda, Michael, 85
Korns, Michael, 59
Kronberger, Gabriel, 85
Kumar Saini, Anil, 123

L
Lalejini, Alexander, 1, 103
LDA, 59, 62
Lexicase selection, 7, 123
LGP, 4
Liu, Bo, 209
Loan scoring, 76
Loop, 5
López, Uriel, 193
Lyu, Daoming, 209

M
Machine learning, 194, 210
MAP elite, 1, 111
Markov brain, 19
May, Tim, 59
Maze, 28
Metevier, Blossom, 123
Miller, Julian F., 137
Muñoz, Luis, 193

N
NEAT, 17, 39, 140
Network configuration, 186
Neural networks, 137
Neuron model, 140

O
Ofria, Charles, 1, 103
O'Neill, Michael, 179
Outliers, 193, 200

P
PADO, 39
PCA, 196, 222
Preface, ix

R
RANSAC, 201
Recipient, 195
Regression, symbolic, 24, 85, 188, 195
Reinforcement learning, 37, 211, 214

S
Schossau, Jory, 17
Scope, 4
Self-modifying CGP, 163
Signal GP, 103
Similarity, genotypic, 91
Similarity, phenotypic, 90
SKLEARN, 210
SMCGP, 163
Smith, Robert J., 37
Sørensen-Dice index, 91
Spector, Lee, 123
Support vector machine, 221
Swarm optimization, 69
Symbolic planning, 213
Symbolic regression, 24, 86, 179, 188, 195

T
Tags, 103
Tangled program graph, 38
Tournament, 6
Tower problem, 197
TPG, 38
TPOT, 211
Transfer learning, 193, 195
Trujillo, Leonardo, 193
t-SNE, 91
Typing, 69
Typing, strong, 72

V
VizDoom, 51

W
White box, 59, 79
Wilson, Dennis G., 137
Winkler, Stephan M., 85

X
XOR, 24

Y
Yang, Fangkai, 209