BIRMINGHAM CITY UNIVERSITY

DOCTORAL THESIS

Leveraging Evaluation Time to Produce Simple Machine Learning Models with Genetic Programming

Supervisors:
Dr. R Muhammad Atif AZAD
Dr. Yevgeniya KOVALCHUK
Dr. Vivek P. INDRAMOHAN
Prof. Hanifa SHAH

Author:
Aliyu Sani SAMBO
15142485

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Data Analytics and Artificial Intelligence Group,
School of Computing and Digital Technology

January 25, 2022

Abstract

The quest for simple solutions is not new in machine learning (ML) and related methods such as genetic programming (GP). GP is a nature-inspired approach to the automatic programming of computers used to create solutions to a broad range of computational problems. However, the evolving solutions can grow unnecessarily complex, which presents considerable challenges. Typically, the control of complexity in GP means reducing the sizes of the evolved expressions – known as *bloat-control*. However, size is a function of solution representation, and hence it does not consistently capture complexity across diverse GP applications. Instead, this thesis proposes to estimate the complexity of the evolving solutions by their *evaluation time* – the computational time required to evaluate a GP evolved solution on the given task. After all, the evaluation time depends not only on the size of the evolved expressions but also on other aspects such as their *composition*, thus acting as a more nuanced measure of model complexity than the expression size alone. Also, GP evaluates all the solutions in a population identically to determine their relative performance, for example, with the same dataset. Therefore, evaluation time can consistently compare the relative complexity.

To discourage complexity using the proposed evaluation time, two approaches are used. The first approach explicitly penalises models with long evaluation times by customising well-tested techniques that traditionally control the size. The second uses a novel technique that implicitly discourages long evaluation times by incorporating a race condition in the GP process. The proposed methods yield accurate yet simple solutions; furthermore, the implicit method improves the runtime and training speed of GP.

Across a diverse suite of GP applications, the evaluation time methods proffer several qualitative advantages over the bloat-control methods. They effectively manage the functional complexity of regression models to enable them to predict unseen data (generalise) better than those produced by bloat-control. In two feature engineering applications, they decrease the number of features – principally responsible for model complexity – while bloat-control does not. In a robot control application, they evolve accurate and efficient routines – efficient routines use fewer time steps to complete their tasks; bloat-control could not detect the efficiency of the programs. In Boolean logic problems where size emerges as the major cause of complexity, these methods are not hindered and perform at least as well as bloat-control. Overall, the proposed system characterises and manages various forms of complexity; also, it is broadly applicable and, hence, suitable for an automatic programming system.

Acknowledgements

Reaching this point in my studies is underpinned by many enablers. I wish to start by acknowledging and pronouncing my gratitude to The Almighty Allah (SWA) for the grace that enabled this great wonder – our existence – and the many blessings I enjoy.

I wish to acknowledge and thank my supervisor *Dr Atif Azad*, who goes far beyond his required duties to support me in every way; as a result, I learn from him much more than research and academic skills.

Likewise, I wish to acknowledge and thank the rest of my supervisory team for their invaluable contributions: *Dr Yevgeniya Kovalchuk*, *Dr Vivek P. Indramohan*, and *Prof. Hanifa Shah*.

Further, I wish to acknowledge and appreciate the contributions of many enablers like *Prof. Mohamed Gaber*, *Prof. Andrew Aftelak* and other faculty members. Also, I acknowledge my fellow research students for their support and for making the journey pleasant and stimulating.

I also wish to acknowledge the superwomen in my life, starting with my mother *H. Lanti Sani Sambo*, then my wife and my sisters (kith and kin), whose contributions are often unsung. Likewise, I thank my dear brothers (kith and kin) for their invaluable and steadfast support.

Lastly and certainly not least, I wish to acknowledge my partner in the journey of life, *Hafsat Bashir Aliyu*, for her love, patience, support, and for making our journey joyful and fulfilling; also, for her labour in nurturing our vibrant brood.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Introduction of the Thesis 1

1.2 Aim and Objectives 7

1.3 Major Contributions of this Thesis 8

1.4 Published Works 11

1.5 Organisation of the Thesis 12

2 Background **16**

2.1 Introduction 16

2.2 Overview of Evolutionary Algorithms 17

2.2.1 The Basic Elements of Evolutionary Algorithms 17

2.2.2 The Process Flow of Evolutionary Algorithms (EAs) 19

2.2.3 Evolutionary Algorithms as Guided Random Search . . . 20

2.3 Genetic Programming (GP) 22

2.3.1 Representation of Solutions 23

2.3.2 Population Initialisation 23

Ramped-half-and-half Initialisation 25

Other Initialisation Methods 26

2.3.3 Breeding 28

Selection of Parents 28
Sexual Reproduction – Crossover 30
Mutations 31
2.3.4 Replacement 33
2.3.5 Fitness Evaluation 35
2.3.6 The Genetic Programming Algorithm 35
2.3.7 The Complexity Challenge in GP 37
Bloat in Genetic Programming 37
Applications of GP and their Complexity Challenges . . . 38
2.4 Conclusion 40

3 Complexity in Genetic Programming 41
3.1 Introduction 41
3.2 The Challenge of Defining Complexity in GP 42
3.2.1 Complexity in Various GP Applications 44
3.3 Existing Complexity Control Methods 46
3.3.1 Approaches Based on the Structure of Representations . . 47
Theories Explaining Bloat 48
Bloat Control Methods 50
3.3.2 Functional-Based Approaches 53
3.3.3 Time Can Indicate Complexity 58
3.4 Conclusion 59

4 Time as a Measure of Complexity 61
4.1 Introduction 61
4.1.1 Time Complexity 62
4.2 The Evaluation Time 64
4.3 Time Detects Size and Functional Complexity 65
4.3.1 Experimental Setup 66

4.3.2 Experimental Results 66
4.3.3 Challenges with Administering Evaluation Time 68
4.4 Measuring Evaluation Time Reliably 68
4.4.1 CPU Management Options 69
4.4.2 Experimental Setup 70
4.4.3 Experimental Results 71
4.5 Conclusion 72

5 Explicit Control of Evaluation Time 74
5.1 Introduction 74
5.2 The Explicit Time-Control Techniques 75
5.3 Time Control for Functionally Diverse Populations 78
5.3.1 The Fixed Length Initialisation Scheme (FLI) 78
5.4 Experimental Setup 80
5.4.1 Test Problems 80
5.4.2 Experimental Settings 82
5.5 Experimental Results 83
5.5.1 The Impact of the Fixed Length Initialisation 84
5.5.2 Comparing Time-Control and Size-Control 87
5.5.3 Discussion 92
5.6 Conclusion 93

6 Implicit Control of Evaluation Time 95
6.1 Introduction 95
6.2 Parallelism in Genetic Programming 98
6.3 Asynchronous Parallel Genetic Programming (APGP) 99
6.3.1 The APGP Algorithm 101
6.4 Validating the Race in APGP 102
6.4.1 Experimental Setup 103

6.4.2 Results 103

6.5 Proof of Concept: Experimental comparison of APGP with Other Methods 105

6.5.1 Test Problems 106

6.5.2 Experiment Settings 107

6.6 Experimental Results 108

6.6.1 APGP compared with GP and GP+BC 108

Accuracy and Complexity Control 108

Training Efficiency of the Methods 110

6.6.2 Comparison of APGP with Explicit Time Control 111

6.6.3 APGP in a Functionally Diverse Population 111

6.6.4 APGP with FLI vs. Time Control with FLI 113

6.7 Conclusion 114

7 Analysis of the APGP **117**

7.1 Introduction 117

7.2 Effect of Degree of Concurrency in APGP 118

7.2.1 Testing Concurrency in a Simplified Environment 119

Results: Higher Degree of Concurrency Increase the Opportunity for Simple Solutions to Finish Early . . 119

7.2.2 Testing Concurrency on Real Problems 121

Results: Higher concurrency still suits simpler solutions in APGP 122

7.3 A dynamic complexity control based on the accuracy of simple models 123

7.4 Balancing Accuracy with Simplicity Across a Wide Range of Concurrency 129

7.5 Varying the Lengths in the Fixed Length Initialisation 131

7.5.1 Result: Effect of varying the length in the FLI 131

7.6 Conclusion 132

8 Application of Evaluation Time Schemes in GP with Multiple Linear Regression 135

8.1 Introduction 135

8.2 Enhancing GP with Multiple Linear Regression 138

8.2.1 The Proposed Multiple Linear Regression Genetic Programming (MLR-GP) 142

MLR-GP Representation and Genetic Operators 142

Evaluating an MLR-GP Model 143

8.3 Validation of the MLR-GP 144

8.3.1 Accuracy Improvement in MLR-GP 144

8.3.2 Crossover Performance in MLR-GP 146

8.4 Experiments with Time-Control Methods 148

8.4.1 Methods for Controlling Complexity Using Evaluation Time 149

8.4.2 The Contending GP Methods 149

8.4.3 Test Problems 151

8.4.4 Configurations and Settings 152

8.5 Results and Discussion 155

8.5.1 Evaluation Time Characterises Complexity Beyond Size . 155

8.5.2 Complexity Control and Generalisation Ability of Models 158

8.5.3 Comparing Adjusted R^2 with the Evaluation Time Schemes 161

8.6 Time Control Schemes with Adjusted R^2 163

8.6.1 APGP with Adjusted R^2 164

8.6.2 Time Control (TC) with Adjusted R^2 165

8.6.3 TCAR2 vs. APGPAR2 166

8.6.4 Summary of Time Control Schemes with Adjusted R^2 . . . 167

8.7 Conclusion 168

9 Applications - Beyond Regression **170**

9.1 Introduction . . . 170

9.2 Robot Control Application - The Artificial Ant . . . 172

9.2.1 Experiment . . . 175

9.2.2 Size is Not Time in ANT . . . 175

9.2.3 Results . . . 178

9.3 Boolean Logic Application - Even-Parity . . . 182

9.3.1 Experiment . . . 182

9.3.2 Results . . . 183

9.4 Boolean Logic Application - Multiplexer . . . 186

9.4.1 Experiment . . . 187

9.4.2 Results . . . 187

9.4.3 Summary of Boolean Logic Results . . . 189

9.5 Classification with GP Aided by Machine Learning . . . 190

9.5.1 Experiment . . . 191

GPML Evaluation function . . . 191

The Classification Problems . . . 192

9.5.2 Benchmarking GPML with other Classification Algorithms 193

9.5.3 Time Detects More than Size in Classification with GPML 195

9.5.4 Results: Complexity Control . . . 195

9.5.5 Results: Accuracy and Generalisation . . . 198

9.6 Discussion . . . 200

9.7 Conclusion . . . 202

10 Conclusion and Future Work **203**

10.1 Conclusion . . . 203

10.2 Future Work . . . 208

Bibliography **211**

A Code Base **247**

B Publications **248**

List of Figures

2.1 Overview of the evolutionary algorithm process. 19
2.2 Types of evolutionary algorithms. 21
2.3 A tree representation of the mathematical equation $(x^2) + sin(x + 1)$. Not all functions or terminals necessarily appear on a single tree. 24
2.4 Crossover in genetic programming: copies of parents exchange segments to produce offspring. 31
2.5 Subtree mutation in genetic programming. 32

4.1 The line graphs show how time changes with input size for different Big O time complexities. 63
4.2 The line graphs illustrate the relationship between the evaluation time and features of a model. The graphs show the mean evaluation times of models made up of mathematical operators of different computational complexity. Individuals made up of COS/SIN operators have higher evaluation times than same-sized ones made up of less complex operators. Moreover, the size of models correlates with their evaluation time. 67
4.3 Result of improving the reliability of evaluation time measurements with CPU management options. Applying the option reduced the variability in the time measurements. 71

5.1 A tree representing an individual made up of binary functions and terminals. 79
5.2 Test fitness scores of both size-control and time-control (using DS and DT) with and without FLI. The red and green lines represent size-control and time-control, respectively; the bold and thin lines indicate FLI and VLI, respectively 85
5.3 Test fitness scores of both size-control and time-control (using OpEq and TP) with and without FLI. The red and green lines represent size-control and time-control, respectively; the bold and thin lines indicate FLI and VLI, respectively 86
5.4 Result of significance test for the difference FLI makes. The FLI test fitness accuracy improved 11 out of 20 for size-control and 16 out of 20 for time-control. 87
5.5 Comparison of size-control and time-control with Death By Size (DS). 88
5.6 Comparison of size-control and time-control with Double Tournament (DT). 89
5.7 Comparison of size-control and time-control with Operator Equalisation (OpEq). 90
5.8 Comparison of size-control and time-control with The Tarpeian (Tp). 91
5.9 The result of the significance test for the difference between size-control and time-control. Time-control methods have significantly higher accuracy scores (on both training and test) in 17 out of 20 test cases. Whereas DS and DT on time-control produced simpler models than on size-control in 9 out of 10 test cases, TP and OpEq on time-control produced more complex models than size-control in 8 out of 10 test cases. 92

6.1 The race enables the tasks with smaller dataset sizes to complete earlier than those with larger datasets. Note, x-axis shows the dataset sizes in thousands. Y-axis shows the number of completed evaluations; thus, the distribution of the first 100 completions is at the top. 104
6.2 The race enables tasks with smaller evaluation times to complete earlier than those with larger evaluation times. 104
6.3 Result of significance test in the differences between the APGP populations and those of GP and GP+BC. The APGP models are more accurate (on both training and test data) than those of GP and GP+BC in all tests. Also, APGP models are simpler than GP. However, APGP models showed a greater complexity than GP+BC that is associated with higher accuracy. 109
6.4 Comparison of APGP and the time-control methods (DS, DT, OpEq and Tp). APGP consistently produced significantly more accurate models (on both training and test data) than all the time-control methods. The time-control methods tend to produce simple models at the expense of accuracy. 112
6.5 Result of test for significance of the impact of FLI on APGP. FLI significantly improved the accuracy scores (on both training and test data) of APGP on 5 out of 6 problems. Also, FLI reduced size (on 4 out of 6 problems) and evaluation times (on 5 out of 6 problems). 113

6.6 Result of significance test for differences in the final populations of APGP with FLI and the time-control methods with FLI. APGP-FLI produced models with higher accuracy (on both training and test data) in all cases. The models produced by time-control methods with FLI were simpler (smaller size or evaluation time) than APGP in 34 out of 48 tests. 114

7.1 Illustrated is the effect of degree of concurrency on the rate that simple evaluation tasks (with small dataset sizes) complete their concurrent evaluations. Higher degrees of concurrency (threads) allow the simpler tasks to complete early more frequently. 120

7.2 Effect of the Degree of Concurrency of APGP on Real Problems. The charts in each row show the distribution of size in the population at the 1,000th evaluation that different degrees of concurrency produced. 122

7.3 APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100th and bottom for 0 to 1,000th), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes. 125

7.4 APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100th and bottom for 0 to 1,000th), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes. 127

7.5 APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100^{th} and bottom for 0 to $1{,}000^{th}$), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes. 128

7.6 The development in size and training fitness (by generations) in the population of APGP with 5, 100, 250 and 500 parallel threads are on the plots. The higher degree of concurrency (threads) increases the opportunities for the smaller sized individuals to lower the average population sizes; yet, as preferred, their average training fitness is generally not reduced. 130

7.7 Result of varying the fixed length in FLI. The variation did not significantly change the sizes of individuals in the final populations. However, their test-fitness accuracy showed some differences. 132

8.1 The MLR-GP process flow. 139

8.2 An MLR-GP representation of a model. The nodes labeled *tag* are placeholders that are not evaluated; and the subtrees F1 - F4 are treated as features. 142

8.3 The training fitness values by generation of MLR-GP, GP, and LS-GP are compared. MLR-GP has improved training to the point of overfitting the data. 145

8.4 The 3D histograms show fitness (accuracy) improvement after crossover by generation. MLR-GP (column 3) shows fewer negative improvements (deterioration) than GP and LS-GP. 148

8.5 The training and test fitness values of MLR-GP and standard GP are compared. For all the test problems, MLR-GP has improved training to the point of overfitting the data. 154

8.6 Correlations between evaluation time and size and between evaluation time and the number of features for Problem 1 - 5. 156

8.7 Correlations between evaluation time and size and between evaluation time and the number of features for Problem 6 - 10. 157

8.8 Result of the significance test for the difference between the MLR-GP with APGP and MLR-GP with bloat-control (BC) and without (STD). 159

8.9 Result of the significance test for the difference between MLR-GP with Time-control (TC) and MLR-GP with bloat-control (BC) and without (STD). 160

8.10 Comparison of the average number of features in the final populations of AR2, TC and APGP. 162

8.11 Result of the test for significance of the difference in the final populations of AR2 against APGP and TC. 163

8.12 Result of significance test for difference between the final populations of APGPAR2 (APGP with Adjusted R2 as fitness) and those of the other Methods. 165

8.13 Result of significance test for difference between the final populations of TCAR2 (TC with Adjusted R2 as fitness) and those of the other Methods. 166

8.14 Result of significance test for difference between the final populations of APGPAR2 and those of TCAR2. 167

9.1 Santa Fe food trail for the Artificial Ant problem (ANT). The black cells represent the food items, and the grey cells are gaps in the trail. The artificial ant begins in the top left corner facing eastward. 173

9.2 An example of ANT solutions. 174

9.3 The correlation in ANT between evaluation time and size in the final populations of the contending methods. 177
9.4 The figure shows the average number of moves used by the ANT solutions for the different methods. The time schemes (APGP and TC) effectively reduce the average number of moves required by their solutions; therefore, their solutions are more efficient than those of the others. 179
9.5 The average fitness scores of the ANT solutions produced by the methods by generation. 179
9.6 The average size (length of expressions) of the ANT solutions produced by the methods by generation. 180
9.7 The average evaluation times of the ANT solutions produced by the methods by generation. 180
9.8 ANT application result of the test for significance in the difference between Time-control (TC) and standard GP (GP), and between TC and GP with bloat-control (BC). 181
9.9 ANT application result of the test for significance in the difference between APGP and standard GP (GP), and between APGP and GP with bloat-control (BC). 181
9.10 Parity bits of binary strings. 182
9.11 Correlations between size and evaluation time in Even Parity (Parity10). 183
9.12 The charts show the evolution of fitness in the parity problems by the contending methods. 184
9.13 Even Parity: significance test result for the difference between TC vs GP and TC vs BC. 185
9.14 Even Parity: significance test result for the difference between APGP vs GP and APGP vs BC. 185

9.15 The Multiplexer with 3 address and 8 data lines. 186
9.16 Multiplexer: The mean fitness scores by generation of the contending methods. 188
9.17 Correlations between size and evaluation time in Multiplexer applications. 188
9.18 Multiplexer: significance test result for the difference between Time-control (TC) vs standard GP and GP with bloat-control. . . . 189
9.19 Multiplexer: significance test result for the difference between APGP vs standard GP and GP with bloat-control. 189
9.20 Comparing GP and GPML with other machine learning classification algorithms. 194
9.21 The figure shows the correlation of evaluation times with size (column 1) and the number of features (column 2) in the classification with GPML application. Evaluation time shows a stronger correlation with the number of features than with size, except for the SPAM problem. 196
9.22 Result of statistical tests of the difference of the complexity control in the GPML application between APGP and GPML (GPML without complexity control) and between APGP AND BC (GPML with bloat-control). The difference in the final populations of the methods were tested. 197
9.23 Result of statistical tests of the difference of the complexity control in the GPML application between TC and GPML (GPML without complexity control) and between TC AND BC (GPML with bloat-control). The difference in the final populations of the methods were tested. 198
9.24 The mean training and test fitness (accuracy) by generation of the methods in GPML. 199

List of Tables

5.1 Test problems for Explicit Time Control experiment. 82
5.2 Experimental settings for Explicit Time Control. 83
5.3 Composition of the populations produced by Time-control and Size-control are compared. 91

6.1 Test problems for Implicit Time Control Experiment. 106
6.2 Experimental settings for Implicit Time Control. 107
6.3 The relative training speeds of the methods as determined by the number of evaluations needed to meet a set training accuracy target. Columns 2 - 4 show the average number of evaluations the methods used. Column 5 - 6 show the percentage differences of APGP relative to GP and GP+BC, respectively. APGP uses significantly fewer evaluations than the others. 111

8.1 The training accuracy gain by MLR-GP over MLR, LS-GP and GP; both the mean training fitness scores and the percentage improvement by MLR-GP are detailed. 145
8.2 Comparison of percentage of crossover with negative improvements (a deterioration). MLR-GP is having fewer negative improvements on all test problems. 147
8.3 Test problems for the MLR-GP experiment. 152
8.4 Experimental settings for the MLR-GP experiment. 153

9.1 Experimental settings for the Ant problem. 176

9.2 Experimental settings for the Parity problems. 183
9.3 Datasets for the classification experiments. 193
9.4 Peak average test fitness scores of GPML and the compared methods on classification problems. 200

Dedicated to my late father,
Ambassador *Muhammadu Sani Sambo*, a
man of vision, integrity and kindness,
with whom I enjoyed a wonderful
relationship.

Chapter 1

Introduction

1.1 Introduction of the Thesis

Genetic Programming (GP) is an evolutionary algorithm (developed based on ideas from Darwinian evolution) for automatically evolving computer programs. As computer programs can represent solutions to any computational problem, GP, at least in theory, can be used to produce solutions to any problem automatically. Correspondingly, its applications concern a wide variety of computational problems such as automatic programming, machine learning, design, and general problem-solving [1]. The user of the system must merely specify the ingredients of the possible solution (such as system variables and useful functions that can help model the solution to the underlying computational problem) and then the system automatically searches for the best possible solution.

GP uses a flexible structure to represent potential solutions; this flexible structure is designed to enable an effective search for increasingly better solutions. However, although the flexibility makes the search potent and versatile, the emerging solutions grow increasingly and often unnecessarily complex to present considerable challenges.

This thesis addresses the challenges of managing the complexity of evolving solutions in a variety of contexts. Since GP is a versatile tool that evolves solutions to problems in several domains, the motivations for avoiding it are diverse and therefore, the thesis argues the notion of complexity in GP must be understood in a nuanced fashion.

Due to its versatility, the motivations for managing the complexity of GP evolved solutions vary. For example, when applying GP to model data arising from a phenomenon under study, a well-known reason for avoiding complexity is to avoid *overfitting* [2]. Meaning, data models must not over-learn the observed data to the extent that they cannot explain other data arising out of the same phenomenon. Literature in Machine Learning argues that a common indicator of overfitting is model complexity: a functionally complex model has a greater ability to mimic the quirks of the noise (such as due to measurement errors) in the data used to *train* the model than a simpler one. Such a model inevitably fails in predicting system behaviour on similar data that was not used to train the model. Instead, the goal (in machine learning based data modelling) is to infer models that accurately predict on unseen data [2], that is, to avoid overfitting the training data.

Another reason to avoid needless complexity is to attain solutions with minimal computational effort and runtimes. For example, constraints on computing resources on *Internet of Things* (IoT) devices restrict acceptable models to shorter evaluation times even if this compromises the model's accuracy [3]. Also, short runtimes are, at least, desirable if not essential for computer programs, for example, interrupt service routines in operating system kernels [4], evolvable neural networks [5] and circuit specifications [6] represent but only a few instances that require efficiency. The short runtimes improve efficiencies such as computing resource utilisation, power consumption, response time, amongst other objectives.

A well known and studied concern of the excessive complexity in GP is that it can render GP's search process ineffective; in fact, if left uncontrolled, it can exhaust the computer system resources to physically halt a GP process [1].

Therefore, complexity in the GP context is multifaceted, and a universal and formal definition of complexity – that forms the basis of a quantifiable measure for detecting and controlling the complexity – is yet to exist.

Current solutions for managing complexity are limited. For example, a form of complexity that is easily measured and popularly used is the size of the expression that represents a solution. Discouraging this notion of complexity is by far the most popular and most studied approach to managing the complexity of GP solutions – known as *bloat-control*. However, this approach ignores the underlying functional and computational complexity of GP solutions; moreover, the small expressions (e.g., $sin(x)$) can be computationally and functionally more complex than larger-sized expressions (e.g., $5x + 4x + 3x + 2 + 1$). Accordingly, previous studies show that discouraging the size of a GP solution (bloat-control) does not always overcome the overfitting problem in data modelling (poor performance on unseen data) [7, 8], which is associated with the model's functional complexity. Likewise, in other GP applications, attaining solutions with small expressions does not always address the motivation behind managing complexity. For example, an evolved program with a small-sized representation (e.g., nested loops) may require much more computational resources to execute than a larger-sized one that does not use these loops.

Despite these limitations, bloat-control remains by far the most popular complexity control in GP. Although there has been some parallel work on complexity control, it is restricted to specialised domains (for example, approaches for data modelling may not cover the diversity of GP applications such as robot control); and even so, exacts demands on the evolutionary dynamics that are not always easy to fulfil.

Therefore, this thesis presents a broadly-applicable indicator of the complexity of GP solutions and methods of controlling the complexity. Instead of using the sizes of the expressions representing GP solutions, this thesis proposes *evaluation time* – the computational time required to evaluate a GP solution (e.g., a data model on training data) – as a notion of its complexity. This notion is founded on the observation that contending GP solutions within an application can have different evaluation times based on their computational composition. For instance, a data model can take a long time to evaluate because it comprises of computationally-expensive building blocks (functions), is structurally large enough to take a long time to be evaluated, or both; hence the model is computationally complex. Therefore, curbing the evaluation time can potentially discourage both the structural and functional complexity. Moreover, this notion can reflect complexity in other GP applications as well. For example, in automatic programming, the evaluation times (execution times) of evolved programs reflects their efficiency (a form of complex behaviour; simple programs require minimal computational effort).

To manage the complexity of GP solutions through their evaluation times, this thesis demonstrates two main approaches.

The first approach controls complexity *explicitly* by leveraging the machinery of the well-precedented bloat-control techniques, but to constrain the evaluation times instead of the sizes of GP solutions. Since these bloat-control techniques are well studied, this study merely observes the benefits that may accrue by replacing the objective of minimising sizes with evaluation times. This approach is a relatively minor adjustment to a well-accepted practice and therefore is easy to adopt widely by GP practice.

The second approach controls complexity *implicitly* by simply allowing the computationally simpler solutions to thrive and guide the remaining evolutionary search by virtue of completing their evaluations before their more expensive counterparts. Since GP simulates natural evolution, it searches for increasingly better solutions by maintaining a population of candidate solutions at all times instead of merely improving a single solution. As new solutions emerge from the existing population, the current population indicates the direction of the evolutionary search at the time; the present members of the population simulate sexual reproduction to produce offspring solutions. Since the future generations of solutions depend upon the genetic constitution of the current population, the current population influences what kind of solution will eventually emerge at the end of the simulation. The second approach, therefore, induces a competition, a *race* (as is called in parallel computing), whereby multiple candidates evaluate in parallel, and those that finish evaluating first and get into the breeding population can steer the remaining search by reproducing earlier, while their expensive counterparts remain busy evaluating. This race gives the simpler models an evolutionary advantage, not because of any explicit penalty inflicted upon their complex counterparts but simply owing to their computational efficiency. Note, however, that evolutionary search is guided primarily by the notion of *fitness* or quality of the evolving solutions; that is, in a given population, the fitter solutions reproduce more often and thus direct the evolutionary search. Consequently, if a qualitatively better but more complex solution emerges later, it will influence the remaining evolutionary search. Therefore, in this system, a complex interplay of computational efficiency and quality ensues without explicit and subjective penalties on the solutions based on their defined measure of complexity.

This thesis compares the impact of evaluation time control (both its flavours) with several effective bloat-control techniques on several applications. The objective of this study is two-fold: a) to see as to whether the proposed schemes can produce good quality solutions in a domain-specific way and b) as to whether the idea of controlling complexity via evaluation times is broadly applicable.

Positive results on both the counts reported in this thesis confirm that evaluation time indeed effectively supports the genetic search to produce good quality solutions across disparate applications. Therefore, evaluation time control is a versatile approach to complexity control that can characterise complexity in different scenarios to proffer several qualitative advantages over the popular bloat-control techniques. For example, in machine learning applications (regression and classification) where the principal objective is to avoid overfitting, evaluation times help distil simpler models that generalise (avoid overfitting) better by differentiating between the relative complexity of transcendental functions and simpler arithmetic operations. In robot control programs the proposed measures produce solutions that are both more accurate and efficient; in fact, the presence of looping constructs in this application – that decrease expression size but increase computational complexity – made evaluation times based control a whole lot more useful than bloat-control. While evolving Boolean logic circuits where the computational differences in the circuit components were not as pronounced as perhaps in the earlier applications, the proposed schemes were at least no worse than the standard bloat-control methods. This result, though apparently unfavourable, actually demonstrates the versatility of time-based control to mimic bloat-control when all else fails.

The efficacy of the evaluation time-based control is still clearer in another interesting application where this thesis hybridises GP with the well known multiple linear regression (MLR) to generate powerful regression models. While MLR works very well if the features (or system variables) are well identified, it relies on the human user to identify these features. GP, in contrast, can manufacture effective features but can still benefit from the ability of MLR to optimally combine these features into a powerful regression model. While this makes for an effective hybrid system (MLR+GP), this system runs the risk of generating very complex models with a very high number of complex features. Complexity, in this case, is twofold: the number of features and the complexity of the constitution of the GP evolved features themselves. The results indicate that time-based complexity control covers both these aspects of complexity much more effectively than bloat-control.

Furthermore, the thesis analyses the proposed techniques by empirically verifying as to whether the results are indeed induced by internal evolutionary dynamics influenced by time control. For example, it empirically verifies whether that if under time control, the computationally efficient solutions outpace their expensive counterparts to enrol into the population. Also, for the implicit time control, this study verifies the impact of the number of parallel threads on the race: it checks as to whether increasing the number of parallel threads leads to a more equitable race. Also, this study investigates a new population initialisation scheme to kick start evolution in GP. Although the scheme is motivated by the evolutionary dynamics of the time control introduced here, the results indicate that the new population initialisation also improves the performance of a standard implementation of GP with bloat-control.

This thesis highlights the inadequacy of the popularly used structural notions of complexity in Genetic Programming. The success of the proposed alternative – evaluation times – and the methods derived from it across a broad set of GP applications attest to the ability of the evaluation time to characterise complexity in a nuanced way and the breadth of their applicability.

1.2 Aim and Objectives

This thesis aims to highlight the inadequacy of the popularly used structural based notions of complexity in Genetic Programming (GP) and, to introduce and analyse an alternative – based on the evaluation times of evolving solutions

– that helps produce a more effective complexity control across a broad set of GP applications.

Essentially, the thesis addresses the question: Can the knowledge that simple tasks take a shorter time to perform than complex ones be used to effectively produce simple and efficient computational models?

To achieve the aim of this thesis, the following objectives are pursued:

- *Objective 1*: Review the current notions of complexity in genetic programming and the techniques for controlling them.
- *Objective 2*: Propose an alternative measure of complexity that characterises complexity effectively enough to be useful and broadly applicable across GP applications.
- *Objective 3*: Propose alternative methods to control complexity in GP based on the complexity measured as proposed in this thesis.
- *Objective 4*: Compare the proposed system for producing simple GP solutions against the state-of-the-art GP methods on a diverse set of applications.
- *Objective 5*: Analyse the behaviour of the proposed methods from Objective 3 to verify and optimise them.

1.3 Major Contributions of this Thesis

- *A time-based measure of complexity.* This thesis presents and demonstrates that a *time-based* measure of complexity – *evaluation time* – detects many notions of complexity across applications and, therefore, can be used to manage complexity broadly and effectively. This contribution corresponds to Objective 2.

- *Strategies for addressing the challenges of the practical use of the time-based complexity measure.* This thesis demonstrates how to overcome challenges with the practicality of using this time-based complexity measure; these include avoiding inconsistencies due to the work of the operating system when scheduling and managing processor power. As evaluation time is central to the proposals of this thesis, the contribution here is crucial to achieving the objectives.

- *Methods for controlling the complexity of GP solutions explicitly.* This thesis demonstrates how to curb the complexity of GP solutions by explicitly discouraging solutions with high evaluation times. The methods customise some of the best existing bloat-control techniques traditionally used to manage the sizes of the GP solutions. Accordingly, this contribution is in fulfilment of objectives 3 and 4.

- *An effective population initialisation method for effective complexity control.* This thesis introduces a population initialisation scheme – Fixed length Initialisation (FLI) – that promotes functional diversity in the population. While results indicate that evaluation time-based control can curb size growth, it works truer to its spirit if it can differentiate between computational complexity derived out of functional diversity (or semantics). To this end, FLI ensures the production of a functionally diverse initial population. Surprisingly, however, the results indicate that FLI improves the outcomes for GP regardless of which kind of complexity control is at play. Therefore, FLI is another contribution to the general practice of GP. FLI is one of the methods that help achieve Objective 3.

- *A Novel genetic programming method that implicitly controls complexity.* This thesis proposes the Asynchronous Parallel Genetic Programming (APGP), which is a GP version that uses a *race* condition in the evolutionary process to manage the complexity of GP solutions implicitly. This novel approach provides advantages, such as managing complexity naturally without subjective penalties, improving training speed and accuracy. Furthermore, the APGP challenges the common but unnatural practice in GP of producing offspring in a synchronised manner. This contribution is a fulfilment of objectives 3 and 4.

- *A proof of concept across a variety of popular GP.* As proof of concept and further investigation, this thesis presents the application of the proposed system to a variety of popular GP applications. While this demonstrates the system's utility and broad applicability, it also shows the various forms of complexity that the system can unearth and its qualitative effect on the solutions. For example, it produces models that generalise well by detecting functional complexity through computational complexity, size, and the number of features in regressions models; and the methods yielded efficient and accurate robot control routines. As this contribution expands the meaning of complexity in GP and compares the proposed methods with existing ones, it addresses objectives 2 and 4.
- *Expanding the discussion on what complexity is in artificial intelligence.* This thesis adds to the dialogue on complexity in artificial intelligence by proposing a measure of complexity and demonstrating how it can unearth several notions of complexity across several GP applications (e.g., size of the representation of a model, number of features in a model, functional complexity, and efficiency). This contribution partially addresses objectives 1 and 2.

1.4 Published Works

The following are publications that were derived from this thesis:

1. Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek P. Indramohan, Hanifa Shah "Evolving Simple and Accurate Symbolic Regression Models via Asynchronous Parallel Computing". In: *Applied Soft Computing* 104 (2021), p. 107198. ISSN: 1568-4946. URL: https://doi.org/10.1016/j.asoc.2021.107198

2. Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek P. Indramohan, Hanifa Shah "Time control or size control? reducing complexity and improving the accuracy of genetic programming models", In: *European Conference on Genetic Programming*, Springer, 2020, pp. 195–210. URL: https://doi.org/10.1007/978-3-030-44094-7_13

3. Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek P. Indramohan, Hanifa Shah "Leveraging asynchronous parallel computing to produce simple genetic programming computational models", In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, Association for Computing Machinery, NY, USA, 2020, p521–528. URL: https://doi.org/10.1145/3341105.3373921

4. Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek P. Indramohan, Hanifa Shah "Feature Engineering for Enhanced Performance of Genetic Programming Models", In: *GECCO '20 Companion, Genetic and Evolutionary Computation Conference Companion*, July 2020. URL: https://doi.org/10.1145/3377929.3390078.

5. Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek P. Indramohan, Hanifa Shah "Improving the Generalisation of Genetic Programming Models with Evaluation Time and Asynchronous Parallel

Computing", In: *GECCO '21 Companion, Genetic and Evolutionary Computation Conference Companion*, July 2021. URL: `https://doi.org/10.1145/3449726.3459583`

1.5 Organisation of the Thesis

The rest of the thesis is organised as follows:

- *Chapter 2 - Background.* Chapter 2 provides an overview of genetic algorithms (EAs) (a family GP belongs to) and explains how they work. The chapter then builds on the introduction of EAs to explain the workings of GP; next, it introduces how and why the challenge of complexity may arise. The chapter starts to build an appreciation of the challenges of managing complexity in GP, including showing that it is a multifaceted issue driven by many motives.
- *Chapter 3 - Complexity in Genetic Programming.* Chapter 3 starts by showing why producing a formal definition of complexity in GP is challenging; it shows that in the absence of this definition, the practice is to resort to using proxies that indicate complexity. Then, the chapter reviews the current notions of complexity (the proxies) in GP and the techniques used to manage the complexity. Finally, the chapter introduces the *evaluation time* as an alternative that can better fill the gap in addressing the complexity management challenge of GP.
- *Chapter 4 - Time as a measure of complexity.* Chapter 4 presents an examination of the *evaluation time* as a measure of complexity in GP. Accordingly, the chapter theoretically and empirically validates the inference of this thesis that the evaluation time can reflect more than the sizes of GP models. In addition, the chapter highlights the expected behaviour of the evaluation time, such as the insight that in a functionally diverse population,

it distinguishes functional complexity; and, in a functionally converged population, it distinguishes structural complexity (size). Furthermore, the chapter addresses the crucial issue of measuring time reliably – the entire system relies on estimates of complexity using evaluation time. Therefore, this chapter proposes and tests strategies that overcome this challenge.

- *Chapter 5 – Explicit Control of Evaluation Time.* This chapter produces the first results on complexity control with evaluation times. Specifically, it demonstrates how to control the evaluation time to control the complexity of GP models by largely using the machinery of the pre-existing bloat-control methods that explicitly penalise large sizes; the implementations only replace size with evaluation time. Note that a new proposal is far more palatable and easier to adopt widely if it requires minimal changes to the existing practice. Hence, the proposed methods in this chapter merely prescribe the replacement of the notion of expression size with evaluation time in what are otherwise well-accepted as techniques for bloat-control. Furthermore, this chapter presents a new population initialisation scheme that promotes functional diversity in the population – Fixed Length Initialisation (FLI); the motivation for the initialisation scheme is to create an environment for the evaluation time to detect functional differences between the models better.

- *Chapter 6 – Implicit Control of Evaluation Time.* This chapter explores a system that, instead of using explicit penalties, incorporates a race condition that allows simple solutions (with short evaluation times) to gain an evolutionary advantage – a metaphor from natural evolution. The fast evaluating (simple) candidate solutions gain an evolutionary advantage when they finish evaluating and get into the breeding population earlier than their slower (more complex) counterparts. These simple solutions in the breeding population can breed (similarly simple and potentially more accurate offspring) while the slower counterparts are busy evaluating. This chapter explores whether this relatively gentler and implicit advantage makes a noticeable difference and if the method encourages simplicity in the GP populations.

- *Chapter 7 – Analysis of the APGP.* Chapter 7 analyses the implicit time control method (APGP) from Chapter 6 and investigates hyperparameters for the population initialisation scheme that improves functional diversity in the population (FLI). The analysis verifies whether computationally efficient solutions outpace their expensive counterparts to enrol into the population. In addition, it verifies as to whether the results produced by the time control method are indeed induced by internal evolutionary dynamics influenced by the evaluation time control method, as theorised. Furthermore, the analysis studies the effect of the degree of concurrency of the parallelisation on the APGP performance.

- *Chapter 8 – Application of Evaluation Time Schemes in GP with Multiple Linear Regression.* This chapter further demonstrates that the evaluation time is more than size using this thesis's novel hybridisation of GP and multiple linear regression (MLR-GP) as a platform. The MLR-GP evolves solutions that incorporate several notions of complexity and, therefore, enables a detailed study of how time characterises complexity and how well it controls the complexity of the MLR-GP solutions in this environment where complexity has a broader meaning than in regression. This chapter shows that in this application, where complexity reflects the number of features and the complexity of the constituents of the GP evolved features themselves, the time-based complexity control covers both these aspects of complexity much more effectively than bloat-control.

- *Chapter 9 – Beyond regression.* This chapter demonstrates the versatility of the evaluation time schemes by testing them on various diverse GP applications. This chapter compares the time schemes with bloat-control on popular benchmarks problems to demonstrate their ability to unearth various notions of complexity. The benchmark problems are from robot control, Boolean logic applications, and classification using hybridisation of GP with multiple linear regression. Therefore, this chapter confirms the ability of the proposed methods to provide a nuanced way to detect various notions of complexity and offer qualitative improvements in a broad range of applications.

- *Chapter 10 – Conclusions and Future Work.* This final chapter provides a conclusion of the thesis and presents the direction of future work.

Chapter 2

Background

2.1 Introduction

This chapter presents the background of this study. The central algorithmic paradigm for this thesis – *genetic programming* – is one of the several algorithms inspired by natural evolution (Darwinian Evolution). Therefore, this chapter introduces the main concepts at work that these algorithms have borrowed from Darwinian evolution. Then, it presents the details of genetic programming (GP) and highlights the challenge this thesis addresses – the challenge of managing the complexity of GP solutions.

Complexity in machine learning (ML), and GP in particular, is multifaceted. Across the many diverse applications of GP, complexity means many things; likewise, within an application of GP, several reasons may drive the need to manage it. Because of the importance of this challenge, overcoming it has many precedents. Yet, the challenge still demands effective solutions that address the needs.

The rest of this chapter is organised as follows: Section 2.2 introduces evolutionary algorithms; Section 2.3 explains how GP works and introduces the challenge of managing complexity in GP; and, Section 2.4 concludes the chapter.

2.2 Overview of Evolutionary Algorithms

Evolutionary algorithms take inspiration from natural evolution and have a common root – the idea that living organisms adapt over time to suit their environment. Charles Darwin (1809-1882) popularised the theories that explain the mechanism that drives this adaptation [9]. Moreover, this adaptation mechanism relies on the presence and interplay of several factors and principles.

In nature, the evolutionary process involves a population of living organisms that have the potential to multiply. Furthermore, this multiplication is associated with variation of the genetic makeup (genotype) of individuals in the population through sexual reproduction, which involves the mixing of genetic materials of individuals to produce offspring, and through mutation, which makes random changes in genotype. The variation in genotype may result in changes in the physical characteristics (phenotype) of the individual. Over an extended period, new generations of the organism replace the old ones that die off. As the population of individuals moves from one generation to another, new traits may manifest, which can help them survive their environment better or otherwise. However, only individuals with favourable adaptations survive in their environment long enough to pass down those traits to their offspring. Thus, heritable traits that help organisms survive and reproduce in a given environment become more common in the population over time. This natural filtering out and preference for the relatively fitter individuals is termed *natural selection* and is a central theme in Darwinian evolutionary theory.

2.2.1 The Basic Elements of Evolutionary Algorithms

To harness the power of natural evolution, which optimises a population of organisms to fit its environment better, evolutionary algorithms identify and mimic the core ideas and their relationships. Although natural evolution is a highly complex system, evolutionary algorithms benefit from borrowing basic concepts; some versions may borrow more than others. The basic concepts include the following:

- Natural evolution maintains a population of living organisms. Accordingly, evolutionary algorithms typically start by creating a population of candidate solutions for a given problem.
- In natural evolution, the potential for the species to multiply is essential. In addition, the multiplication of the organism involves the variation of genetic makeup (with a possible resultant change in characteristics) through sexual reproduction and mutation. Therefore, evolutionary algorithms make new solutions from the existing ones using *artificial genetic operators* that mimic reproduction and mutation.
- In natural evolution, organisms die off to make way for new generations that increasingly *fit* the environment they inhabit. Correspondingly, evolutionary algorithms incorporate methods that replace individuals of a population with better performing offspring.
- In natural evolution, the struggle for existence is within a specific environment that determines the advantageous characteristics or *fitness* of an individual organism. Therefore, an evolutionary algorithm employs a *fitness function*, which evaluates and scores every candidate solution to quantify how well it solves the problem. The exact definition of the fitness function varies from problem to problem, much like the variation in the environment in natural evolution. The fitness function computes the fitness scores, which the evolutionary algorithm uses to determine the performance of an individual and, in turn, its survival and the opportunity to reproduce.

Experimental observations show that when algorithms apply these concepts to a problem (a computational problem), the population of candidate solutions improves with time. The following section describes how these borrowed concepts work together to form evolutionary algorithms (henceforth abbreviated as EA or EAs).

2.2.2 The Process Flow of Evolutionary Algorithms (EAs)

Natural evolution is a complex process, and EAs only mimic them coarsely. Moreover, several variations of EAs exist, and more are emerging. Therefore, this section describes a simplified version of a common approach to evolutionary computing to illustrate how they work.

As illustrated in FIGURE 2.1, the EA starts by producing a population of sample (random) solutions and evaluating them. Typically, the number of individuals that make up a population is predetermined.

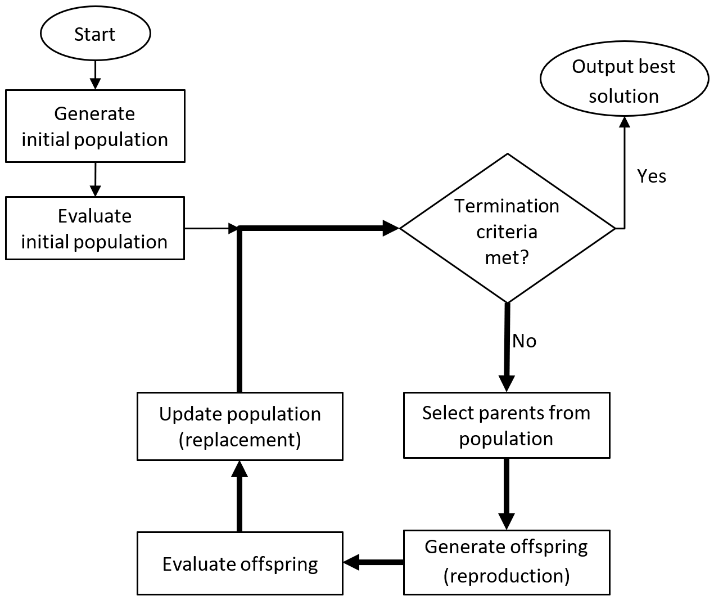

Start
Generate initial population
Evaluate initial population
Termination criteria met?
Yes
Output best solution
No
Select parents from population
Generate offspring (reproduction)
Evaluate offspring
Update population (replacement)

FIGURE 2.1: Overview of the evolutionary algorithm process.

Then, breeding that iteratively produces several generations of evolving individuals begins. Each iteration or a cycle includes a probabilistic selection of good parents from the existing population to reproduce. Copies of the selected parents are recombined (sexual reproduction) and occasionally mutated to produce offspring.

While the population sizes in natural evolution vary, in contrast, EAs typically maintain a fixed population size (although some attempts to vary the population sizes have been made, e.g., [10], they are not commonly used). Therefore, in this example, a cycle generates only as many offspring as there are individuals in the existing population. Next, all offspring are evaluated and assigned fitness scores.

A cycle (representing a generation) ends when the current breeding population is replaced by the new one. Several methods for replacing the current population are available. The new generation may replace the older one entirely, or a selection of the best from a combined pool of both parents and offspring becomes the new generation. As such, there is no consensus in the community as to which approach is universally the best.

The breeding cycle continues for several generations until a termination condition is satisfied; for example, if the evolution produces an individual that meets a set target fitness or completes the maximum number of allowed generations.

2.2.3 Evolutionary Algorithms as Guided Random Search

Considering evolutionary algorithms as search algorithms provides additional insight into how they work. Evolutionary algorithms explore the possibilities of a search space of solutions using a guided random search [11–13]. While a random search relies on luck to find a solution to a problem that has a vast search space, a guided random algorithm has a means of knowing whether every step it takes is an improvement or not. Hence, although the immediate steps (change in the solution) are somewhat randomised (e.g. genetic material is randomly recombined between the selected parents), the algorithm probabilistically selects better performing parents to produce offspring; therefore, the algorithm is guided during the search.

Moreover, instead of working with a single solution (e.g., in Simulated Annealing [14] or Gradient Descent [15]) the EAs work with a population of solutions; this allows them to simultaneously sample multiple areas in the search space (especially at the start of the evolution when the population is randomly generated). As a result, the search in EAs is not restricted to the neighbourhood of a single solution; instead, it is *parallelised*. Therefore, it increases the feasibility of finding innovative solutions within a reasonable time in vast search spaces. Also, since artificial evolution in the EA is driven by sexual recombination, offspring leverage the learning of different individuals, which differs from independent agents exploring the search space.

Furthermore, the parallelisation in EAs can more rapidly explore (randomly) different areas of the search space than simple random search to find at least an acceptable solution, if not an optimal one.

Essentially, by coarsely mimicking natural evolution, evolutionary algorithms embody a system where solutions are evolved by probabilistically sampling from the promising regions of the search space. This provides an acceptable trade-off between a highly exploratory but expensive random search and highly exploitative but potentially limited local search.

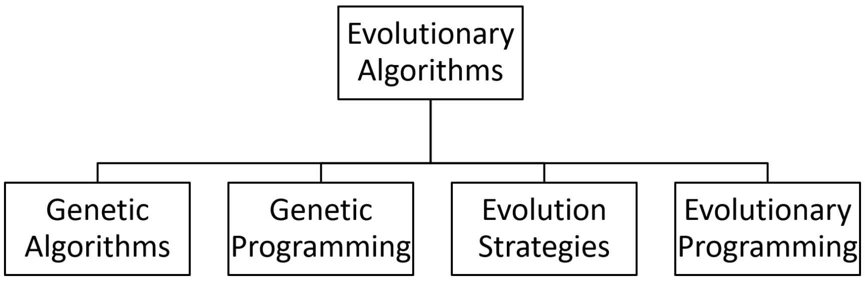


FIGURE 2.2: Types of evolutionary algorithms.

The categorisation of EAs shown in FIGURE 2.2 is popular in the EA literature [16]; they include Genetic Algorithms (GA), Evolution strategies (ES), Evolutionary Programming (EP), and Genetic Programming (GP). GA, ES, and EP all appeared at around the same time (the 1960s), but they differ in terms of how they implement ideas from natural evolution. However, broadly they all optimise parameters of some system under consideration.

GP is a relatively newer branch of this field and is also different in its target applications. GP emerged in the mid-1980s [17] and became popular in the 1990s [18]. Although its application is broad-based, it is generally considered an automatic-programming EA. To borrow an analogy from the modelling applications, while the other EAs optimise only the parameters of the system under consideration, GP evolves the model underlying the system itself. The next sections describe GP in detail.

2.3 Genetic Programming (GP)

Genetic programming (GP) is a specialised evolutionary algorithm initially designed for the automatic generation of computer programs [17, 18]. Today, GP is used to generate solutions to several other types of problems. An important feature that makes it versatile and distinguishes it from many other EAs is the flexible structure GP uses to represent solutions. Instead of assuming a fixed form, the structure of GP solutions is allowed to change as needed. For example, while typical Genetic Algorithms (GA) often optimise a given set of parameters (thus a set structure), GP can explore vast possibilities to produce unexpected solutions without the benefit of prior knowledge of what an appropriate form is for the solutions [19]. This GP's ability to find and optimise appropriate structures for its solutions makes it a powerful tool with broad applications. However, the flexible representation used in GP comes with challenges. The following sections describe how GP works and some of its challenges.

2.3.1 Representation of Solutions

The traditional and most popular representation of GP individuals is a tree-style structure [20]; other variations are also common [21–26]. It is easy to extend or contract the tree structure and, therefore, it is suitable for variable-length encoding.

To construct a GP tree, the user must first specify two sets, that is, *functions* and *terminals*. The function set contains all the possible choices that can appear in the *internal* nodes of a tree, while the terminals set contains all the possible choices that can appear in the *leaf nodes* of a tree.

For example, consider a GP tree in FIGURE 2.3 that represents the mathematical equation $(x^2) + sin(x + 1)$. The figure also shows a possible choice of the functions and terminal sets from which the exemplified tree can be instantiated. The internal nodes are sampled from within the functions set, and require arguments that can be specified via either subtrees extending beneath them or leaf nodes. Similarly, the leaf nodes are sampled from the terminals set but do not require arguments (e.g., variables and constants are terminals); therefore, the leaf nodes can not have subtrees extending beneath them. In the example, the instance of ADD function at the top of the tree is the *root node*; MUL, SIN, and the other instance of ADD are internal nodes (functions); and nodes X and 1 are leafs (terminals).

The representation of solutions in GP allows the flexibility of growing and shrinking. While this flexibility is essential in how GP works, it also poses challenges; Later Section 2.3.7 details these challenges.

2.3.2 Population Initialisation

The makeup of the initial populations of evolutionary algorithms (EAs) affects how effectively and quickly they find fit (good) solutions [27, 28]. The more

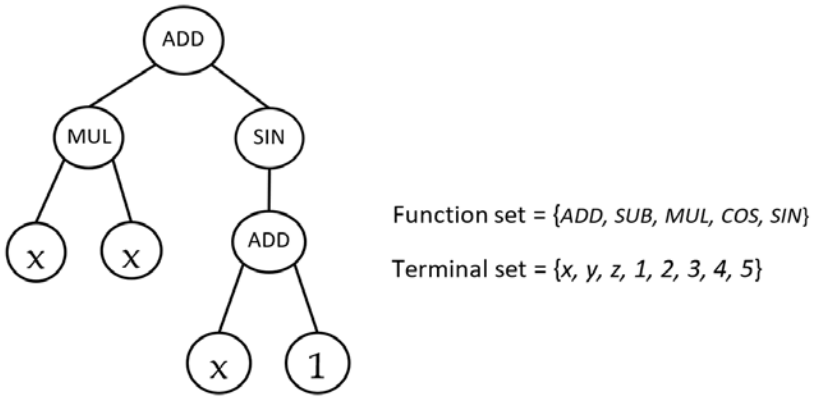

ADD
MUL
SIN
x
x
ADD
x
1
Function set = {ADD, SUB, MUL, COS, SIN}
Terminal set = {x, y, z, 1, 2, 3, 4, 5}

FIGURE 2.3: A tree representation of the mathematical equation $(x^2) + sin(x + 1)$. Not all functions or terminals necessarily appear on a single tree.

diverse the initial population is, the better EAs can explore the different possibilities within the search space. The search space is the underlying theoretical space of candidate solutions the EA can sample from; typically, the quality of solutions varies across different subspaces within the broader search space. Due to the parallelised exploration of the different areas of the search space (as mentioned in Section 2.2.3), the chances of EAs of finding optimal solutions (global optima) increase. Otherwise, the EA may be searching within a limited area of the search space at a time, which means that the search may only find the best within that area (local optima) or may take a long time to move to other regions with better solutions.

Therefore, the standard practices of creating an initial population in EAs aim to start with a diverse population of individuals that samples from many regions of the search space. However, maintaining a large population size also requires a large computational effort because each population member must also be evaluated. Therefore, selecting the right population size involves a trade-off between adequate sampling and low processing requirements.

Ramped-half-and-half Initialisation

Although an obvious method of producing a diverse population would be to randomly generate individuals, the most popular initialisation method in GP is one of the earliest – the so called *Ramped-half-and-half (RHH)* was proposed by the early work of Koza [19] and it remains popular to this day. The RHH leverages two other methods to produce equal percentages of the population – the *FULL* and the *GROW* methods.

The FULL method builds a fully formed tree of a specified depth. Therefore, while creating a tree, the FULL method only uses functions (specified in the functions-set) from the root node until the tree grows to a specified maximum depth level. At the last level, it only uses terminal nodes. As the choice of internal nodes (functions) is typically random, two trees of the same depth may produce trees with different makeup and even different structure; the choice of internal nodes may still influence the breadth of the tree. The choice of different functions will differentiate the makeup, whereas choice of functions of different arity will make the structure also look different. However, the one commonality across all trees is that all the branches extend to a common maximum-depth.

The GROW method can produce trees that are not necessarily fully formed because the branches can terminate at various depths. To create a tree, GROW randomly selects a function as a root node. Then, unlike the FULL method, it randomly picks from either the functions or the terminals to extend the tree – subject to a pre-specified maximum depth. If a branch reaches a depth that is one less than the maximum depth, an appropriate terminal is randomly selected to terminate the tree growth. Furthermore, if GROW chooses a terminal type for a node position, the branch stops growing, sometimes before reaching the maximum depth. Consequently, the GROW method produces trees with diverse structures (often unbalanced) and generally smaller-sized than those produced by FULL.

Note, the GROW method produces a range of differently sized and structured trees. However, the probability of producing a fully formed tree with the GROW method is small (because at each level functions and terminals are randomly selected from); therefore, the RHH methods uses both of the methods to produce a population that has equal shares of fully formed and irregularly shaped trees.

The Ramped-half-and-half (RHH) method systematically combines the FULL and the GROW methods to produce a diverse population. RHH specifies a range of maximum depths and allocates equal quotas of individuals in the population to each of the maximum depth values. Then for each maximum depth, half the quota of individuals is generated via the FULL method and the other half is generated via GROW method. The RHH produces a diverse population and remains the popular population initialisation technique in GP.

Other Initialisation Methods

Other reasons have motivated the development of a variety of population initialisation methods. For example, one approach [29] seeded the population with data models with perfect training accuracy scores to get GP to evolve smaller-sized versions of the models while maintaining their accuracy. For GP to evolve parsimonious solutions from a population of these perfectly trained models, it uses a Pareto fitness function, where a combination of model accuracy and size determines fitness. The method used simple deterministic algorithms to attain well-trained models; virtually, the models memorise the training data. For example, on a classification problem, they created individuals by combining *if-then-else* clauses; the collection of clauses are created by either finding a combination of input variables for each training case or by building a decision tree that uses interval tests on the input variables to get the desired output. For the experiments, they used a population of 500 such individuals with other logical operators, constants, and arithmetic operators. However, the models that memorise large datasets can be unwieldy and often take many generations to reduce. Therefore, they also used subsets of the data (a popular approach to reducing expense in GP, for example, see [30]) to create the clauses for building the individuals in some of the experiments. In one experiment, they used the first half of the data, and in another, they used a combination of the first ten negative training cases and the first ten positive cases. The results show that this GP method can reduce the sizes of the models while maintaining accuracy.

Another example that aims to improve the quality of solutions and reduce run-times incorporates domain knowledge in the form of existing high-quality solutions [31]. However, while this not-random initialisation improved the average solution quality and run-time, it deteriorated the quality of the best solutions found over many runs.

Similarly, to improve performance and increase diversity, the work in [32] initialises the population with competent candidate solutions to the problem, thus avoiding random individuals that tend to perform poorly. However, none of the two versions of this method resulted in significant differences in problem-solving performance or population diversity across the benchmark problems.

To speed up population initialisation, closely control the sizes of the trees, and manage the distribution of functions in the population, the work in [33] proposed two methods. The first method, which is an adaptation of GROW, controls the appearance of functions (specified in the functions-set) in a tree using user-defined probabilities. Also, it keeps the average tree sizes to a specified size; but, it does not control the variance in tree sizes. The second method enables the user to also control the variance of the tree sizes in the initial population. The proposed initialisation methods show low computational complexity and are competitive with the GROW method.

Yet, the RHH has remained the method of choice [34–46]; also, popular open source implementations of GP offer RHH [47–49]. In fact, the method has been so popular that other GP methods like Grammatical Evolution have devised their own versions of RHH for non-traditional GP systems [46, 50].

2.3.3 Breeding

To produce a new generation of solutions, GP selects parents from the population of solutions to breed offspring. Since the selection strategy impacts the quality of offspring and the makeup of the newly generated population, different selection techniques exist; the next section details some popular techniques. However, let us first complete the overview of the breeding process.

After selecting the parents, GP creates a new generation of solutions using three methods [19]. The first method mimics sexual reproduction (also known as crossover) by recombining copies of selected parents from the current population to breed offspring. The second method *mutates* existing individuals. The third method termed *elitism* selects some small fraction of fit individuals from the breeding generation and passes them onto the next generation unaltered.

Selection of Parents

Breeding involves selecting parents from the existing populations that will produce offspring for the next generation. Studies have shown that the choice of selection mechanism impacts the performance of GP, such as its convergence rate [51–53]. While selecting fit parents for breeding may produce fit offspring, the selection methods are designed to avoid over-selecting the same parent even if it is the best individual to maintain diversity in the population. Otherwise, an individual with an outstanding fitness score will be selected repeatedly and take over the breeding population, and the resulting generation would comprise entirely of the genetic material coming from the same parent. Such a situation is typically undesirable because the search for solutions will thus focus on a single region of the search space and stagnate; if the resulting quality of the solutions is undesirable such stagnation is called *premature convergence* [54–56].

Therefore, maintaining a balance that rewards performance yet encourages diversity in the population is a crucial consideration in the design of methods for selecting parents. For example, tournament selection – a popular selection schemed in EAs [57, 58] – randomly selects several individuals from within the population and then in a manner of a tournament chooses the fittest of the selection as the winner that will breed (together with the winner of another tournament, if the genetic operator is crossover). The tournament size is normally much smaller than the population size (typically 2 - 7 [57], while population sizes are 100+). The random selection of tournament contestants ensures that the fittest (and hence the same) members of the population are not necessarily selected each time; however, crucially, *relatively fitter* solutions are selected from the population. Thus, tournament selection provides a trade-off between diversity and performance. The method is easy to implement and very popular, hence the method of choice in this thesis. Other popular selection methods that attempt the same although to varying extents include Fitness Proportionate Selection [59] whereby the chances of selecting a parent are proportional to the magnitude of its fitness, and Ranking Selection [55] whereby the individuals are ranked according to their fitness before conducting Fitness Proportional Selection. Since these methods are independent of the type of the solution representation, they are often usable across evolutionary algorithms.

Sexual Reproduction – Crossover

To mimic sexual reproduction, copies of two parents (selected individuals) exchange segments of genetic material to generate two offspring. Accordingly, the offspring differs from the parents while likely containing inherited components that were previously useful. As a result, sexual reproduction – also known as *crossover* – is the most popular method of stimulating change in the population of Evolutionary Algorithms.

Despite attracting considerable debate, the so-called *building block hypothesis* has often been invoked to support the utility of crossover in GP in that crossover can combine fit subunits (building blocks) of parents that are partial solutions to eventually produce fitter solutions than those in the earlier generations [60, 61]. The other side of the debate countered that a crossover operation is simply a form of mutation and no building blocks are involved [62, 63]. Subsequent studies show that the building block hypothesis has merit even though the exchange of randomly selected subtrees is often destructive to fitness. They demonstrate that when GP discovers primary partial solutions, they spread in the population and become constituents of fitter ones [64–67]. The crossover operator is the dominant method of making changes to a population during evolution in GP.

The crossover techniques are often specific to the type of representation. For the tree representation in GP, subtree crossover is popular [19, 68]. The basic subtree crossover randomly selects a crossover point in each of the parents and swaps the subtrees beneath the points; see FIGURE 2.4 for the illustration of this process.

Crossover operators are customised to suit a purpose. For example, the one-point crossover first chooses the crossover point in one parent before determining a corresponding point on the second parent [69]. Thus, offspring maintain structural similarity with their parents after the crossover. Others choose the

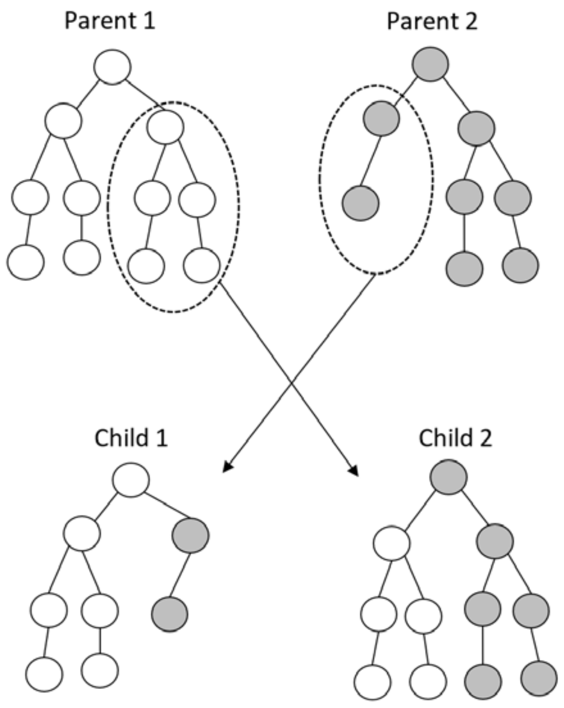


FIGURE 2.4: Crossover in genetic programming: copies of parents exchange segments to produce offspring.

crossover points according to the size of the exchanged subtrees [70] or the semantics of the exchanged subtrees [71] to manage the sizes of the offspring and improve their performance.However, the standard subtree crossover remains the dominant choice in GP [47, 68, 72–75].

Mutations

Mutation in natural populations represents random copying errors when genetic material is transferred from the parents to offspring. As such, mutation is largely destructive to the developing embryo. However, studies in genetics and particularly in unicellular organisms show that mutations allow organisms to occasionally discover new traits that can be advantageous. Essentially, mutations can inject new genetic material and hence genetic diversity. However, the rate of mutation is typically small.

Correspondingly, mutation in GP allows introducing new (random) genetic material into an individual and, in turn, the population [19, 76]. Although mutation is typically detrimental to the fitness of GP individuals, it plays a vital role in increasing the diversity of genetic material. Thus, it can prevent stagnation during the search. The common practice is to allow mutation to occur with a small probability (e.g., 0.1). If a very high mutation rate is applied, the evolution will tend towards random search.

In tree-based GP, subtree mutation is a popular choice. The process creates a random subtree, identifies a subtree of the individual that will be excised, and swaps in a randomly generated subtree (of up to a pre-specified depth) to replace the excised subtree. FIGURE 2.5 shows an illustration of subtree mutation. Further, both offspring and existing individuals in the population may be mutated.

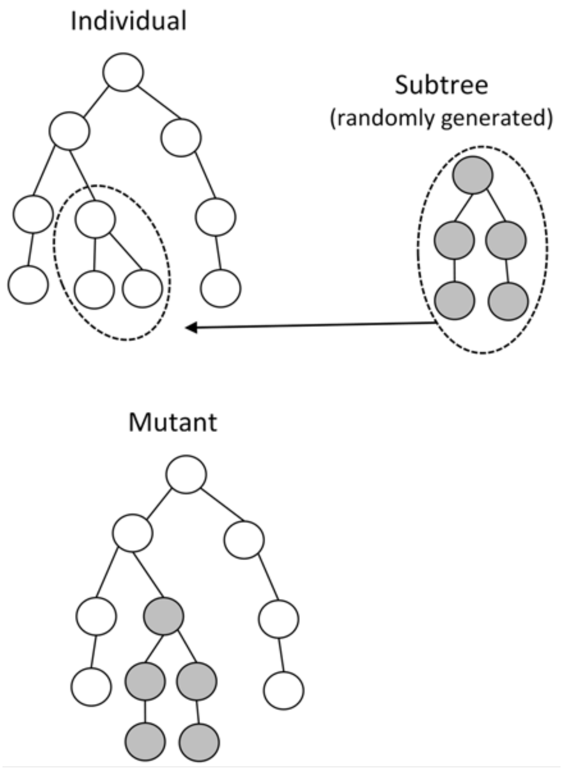


FIGURE 2.5: Subtree mutation in genetic programming.

Examples of alternative mutation techniques include: Point Mutation [19], which changes a single node; Semantically Driven Mutation [76], which detects possible behavioural changes caused by syntactic modification and applies appropriate changes as mutations; and Permutation Mutation [19], which modifies the argument order of a node (nodes and subtrees).

During both the crossover and mutation, a larger subtree can replace a small one or vice versa. Therefore, an offspring may be larger or smaller than its parent; likewise, a mutant may be larger sized than the original. Section 2.3.7 discusses the challenges associated with this potential for GP solutions to grow.

2.3.4 Replacement

Replacement in GP refers to the process used to update the breeding population with offspring to move the evolution from one generation to the next. As GP typically maintains a fixed number of individuals in the breeding population, the GP process involves removing some (or all) of the individuals of the breeding population to make room for the better performing offspring. The challenges during replacement are preserving better performing individuals and maintaining diversity in the population. As discussed in Section 2.3.2, a diverse population enables GP to explore different areas of the space of possible solutions productively.

There are two popular approaches to replacement in GP. The first approach is *generational* replacement, which swaps out the breeding population with a new one. In this approach, the breeding population first produces a new offspring generation. After this, the generational GP has two options: it can replace the parent population with either the entire offspring population or a selection from the combined pool of parents and offspring [77].

In contrast, the other method – *steady-state* replacement – does not maintain a distinct separation between the parent and offspring populations. Instead, after each breeding operation, the offspring is considered to replace an existing member of the breeding (parent) population. Whether or not the replacement actually occurs depends upon the exact strategy; for example, some replacement strategies require that the offspring must be better than the worst member of the parent population, while others may select a member of the parent population based on other criteria and replace it with the offspring just created. As a result, the population steadily undergoes changes as opposed to generational replacement, where changes occur at fixed generational intervals.

While replacing individuals, it is important to avoid losing diversity in the population as well as the high performing solutions. For example, in steady-state replacement replacing a random individual may kill off the best individuals, while the relatively unfit individuals in the population may survive and reproduce [57]. Alternatively, replacing only the worst individual may make it harder to inject new genetic material into the population.

To ensure diversity in the breeding population and avoid premature convergence (when the search stops progressing because the population is homogenous), [78] proposed a replacement strategy that checks the similarity of individuals in the current population and replaces duplicates.

Other methods use probabilistic approaches to select individuals to replace. For example, instead of replacing a random individual, a popular strategy replaces the worst of a randomly selected set of individuals – known as inverse tournament selection [79].

The random selection and inverse tournament are popular because they are easy to implement and do not add significant processing overheads. Also, the inverse tournament has the advantage of retaining the best solutions; furthermore, it represents a compromise between maintaining diversity and accuracy because it randomly picks contenders and then replaces the worst of them.

2.3.5 Fitness Evaluation

The fitness function takes a candidate solution to a problem as input and outputs a *fitness* value that reflects how well the solution solves the problem, that is, fits the set objectives. Defining a suitable fitness function is crucial because it determines the direction of the evolution; in natural evolution, the population evolves to better suit its specific environment. Accordingly, the fitness function must be specific to the problem.

For example, when GP is applied to evolve solutions for a classification problem, a fitness function must compute the classification accuracy of the evolving models. Therefore, the fitness function evaluates the model with the training data. Then, the fraction of correctly classified instances serves as its fitness score.

Although fitness functions are highly problem-specific, some general considerations exist. For example, developers ensure the fitness functions avoid unnecessary computation effort because a typical GP run will use the fitness function to do multitudes of evaluations.

Also, while typically only one objective (accuracy) determines fitness, the fitness function can also be multi-objective [80].

2.3.6 The Genetic Programming Algorithm

To outline how the elements of GP work together, this section describes a conventional GP version that uses the generational replacement strategy in this section; **Algorithm 1** provides the pseudocode for it.

The process starts by setting the size of the population to maintain and the number of generations. Next, GP produces an initial population with random individuals; all the individuals are then evaluated and assigned fitness scores. After this initialisation, the evolution begins.

The next stage is to proceed with reproduction to move the evolution from generation to generation cyclically. The cycle that produces offspring from the existing population involves selecting parents and applying crossover and mutation operations on their copies based on set probabilities to produce offspring. In this example, GP creates as many offspring as the individuals in the breeding population for a generation. A generational cycle ends when a population of offspring replaces the population of parents. The GP run ends after producing the maximum number of generational cycles that are allowed.

Algorithm 1: A Genetic Programming Algorithm

```
/* Initialise                                                  */
Gen ← set number of generations;
PS ← set size of population;
/* Produce and evaluate the initial population                 */
population ← generate initial population of size PS;
Evaluate(population);
/* Generate and evaluate generations of offspring              */
$g = 0$;
while $g < Gen$ do
    offspringPop ← {} ;
    $i \leftarrow 0$ ;
    while $i < PS$ do
        Select offspring individuals from population ;
        Generate offspring from selected parent ;
        Evaluate fitness of offspring ;
        offspringPop[i] ← offspring ;
        $i \leftarrow i + 1$ ;
    end
    population ← population ∪ offspringPop ;
    $g \leftarrow g + 1$ ;
end
```

From the description of GP provided thus far, it is apparent that many configuration options are available. The experiments in this thesis use a common practice in GP of using popular and well-tested settings (where possible) to make the experiments relatable and reproducible.

2.3.7 The Complexity Challenge in GP

The complexity of GP solutions can affect both their quality and the effectiveness of the GP evolutionary process that searches for the solutions. Moreover, complexity is a multifaced concept in GP that can mean several things within and across GP applications.

As typical GP runs involve maintaining hundreds to thousands of individuals across generations with resultant evaluations, a computationally expensive and time-consuming evaluation function can make attaining solutions difficult or impractical. Therefore, the fitness functions have to be as efficient as possible. This challenge becomes amplified when individuals grow in size during the evolution to require more computing resources to evaluate.

Bloat in Genetic Programming

Since the inception of GP, studies showed that the evolving solutions tend to grow without a corresponding improvement in fitness scores [81–87]. Hence, the excessive and non-beneficial growth of the representation of a GP solution is termed *bloat*. In many GP applications, bloat can make the GP runs resource-intensive (CPU processing and memory utilisation) to the point of slowing down or halting the search for solutions. As a result, bloat has received much attention, which has yielded several theories explaining it and many techniques to manage it; Section 3.3.1 in Chapter 3 details bloat further.

Bloat-control is the most studied and most used complexity control approach in GP. Using a form of bloat-control has become a standard practice in GP because of its ease of implementation and the benefit of making GP runs manageable. However, bloat-control offers a limited view of complexity because while it controls the size of the expression (e.g., tree) representing a model, it disregards its functional and computational complexity. Moreover, complexity means different things across the various application domains of GP; the next section exemplifies that.

Applications of GP and their Complexity Challenges

GP is being used to produce solutions to many practical problems in a wide variety of domains [88, 89]; a detailed account of practical and innovative results is available in [90]. The need to manage the complexity of GP solutions across these applications is a reoccurring theme. Moreover, the notion of complexity and the reason for containing complexity can differ from one application to another. To illustrate, this section introduces a few categories of GP applications.

Machine Learning: The most popular GP application is symbolic regression (SR), which is a machine learning application [91, 92]; GP is also used for many other forms of machine learning [93–97]. SR evolves mathematical equations that explain the relationships in data. While a popular method like linear regression (LR) assumes that the relationship between the features of data and their output is linear and, therefore, the structure of the model is predefined, symbolic regression (SR) makes no such assumptions about the structure and instead evolves both the model structure and the parameters to fit the data. Accordingly, GP uses mathematical functions, variables, and constants as genetic material to evolve suitable mathematical equations of varying structures. The primary objective of machine learning is to attain models that generalise well, i.e., they reflect the phenomenon that produces the data and therefore extrapolate to unseen data well. Because a failure to generalise is associated with excessive *functional* complexity, there is a fundamental need to manage the complexity of data models [7, 98]. As the subsequent chapters show, even with bloat-control, GP is not immune to such effects of complexity; in fact, the complexity of GP solutions must be tamed at many different levels.

Automatic programming: GP is applied to produce computer programmes automatically [89, 99]. Hence, GP is used to independently discover programmes that perform a task without being explicitly guided. For example, GP is used to produce a programme to control a robot to perform a specific task [100]. Additionally, GP can build software [101] and optimise existing ones [23, 102]. Due to its stochastic process, GP may produce unnecessarily complex programs or those that contain dead code (do not affect functionality). Despite bloat-control, parsimonious programs can still contain expensive computational constructs like nested loops or other time consuming operations that are otherwise syntactically succinct. Therefore, efficiency should be an important consideration when assessing the quality of automatic programming solutions.

Design: The use of GP to solve design problems is another category of GP applications. For example, GP is used to design electrical circuits and neural network architectures. Designing simple digital circuits is easy using conventional methods but difficult and time-consuming for complex circuits. Therefore, GP has provided a means of automating the design of challenging circuits like asynchronous electronic circuits [103]. Furthermore, the manual configuration of Convolutional Neural Network architectures requires expert knowledge and a lot of trial and error. Hence, developers have adopted Neuroevolution, which is an application of GP that automates the design of such neural networks [104, 105]. Much like automatic programming, attaining efficient design solutions is an important objective. Therefore, the challenge is to find a means to enable GP to detect such efficiency and manage it when evolving solutions.

These example GP applications show that the challenge of managing the complexity of GP solutions is an important and reoccurring one. Also, they show that complexity in GP is a broad concept. Therefore, bloat-control, which merely contains the size of the representation of the solutions and disregards their functional complexity, cannot adequately address the various complexity-related challenges of GP applications presented in this section. After all, small-sized programmes, models, and designs can be more functionally complex than larger ones. Chapter 3 discusses how limited bloat-control is in detail.

2.4 Conclusion

This chapter introduced GP, explained how it works, and exemplified its applications to indicate how versatile it is as a tool, to provide the background for this thesis. This introduction enabled the discussion about the need to manage the complexity of GP solutions to begin. Furthermore, this chapter highlighted complexity as a reoccurring challenge across GP applications. Additionally, it showed that complexity has a broad meaning in GP. Consequently, to effectively manage complexity, one must either customise the methods per application or develop a broadly applicable one. This thesis targets the latter.

The next chapter reviews the current approaches to managing complexity in GP, then shows the need for a new approach to complexity control in GP, and justifies the proposal of this thesis.

Chapter 3

Complexity in Genetic Programming

3.1 Introduction

As genetic programming (GP) is a versatile tool used in several domains, many reasons drive the need to control the complexity of the solutions it produces. Therefore, complexity in GP means several things – it is a multifaceted notion.

However, to control the complexity of GP solutions, a quantifiable measure of complexity and techniques for manipulating it are needed. Yet, efforts to produce a concise characterisation of the complexity in GP to achieve a measure of complexity that is authentic continue; a universally accepted formal definition of complexity in GP does not exist. Therefore, the common practice is to use proxies as indicators of the complexity of the GP solutions; yet, finding an appropriate proxy with a broad application is challenging, and existing ones are often limited in their effectiveness at addressing the various motivations for managing the complexity.

This chapter examines the challenges with defining the notion of complexity in GP to set the stage for the ensuing discussions. Following that, it reviews the current approaches to managing the complexity of GP solutions. Then, it presents the measure of complexity that this thesis proposes.

The rest of this chapter is organised as follows: Section 3.2 highlights the challenge of defining complexity in GP ; Section 3.3 reviews the existing complexity control methods, and introduces and justifies the use of evaluation time as a measure of complexity; and, Section 3.4 concludes the chapter.

3.2 The Challenge of Defining Complexity in GP

The term *complex* exists in several fields of research, such as in the study of *complex systems* in natural and social sciences [106]. Hence, numerous attempts to define a complex system are available. Although a set of core features are widely associated with complex systems in the field, the search for a universally agreed formal definition is ongoing [107]. This section illustrates with examples that defining complexity in GP is challenging.

Many natural systems are regarded as complex because they show intricate interactions between their constituent components to exhibit elaborate behaviour of the whole. The interactions in such systems are non-linear and cannot be studied using the *reductionism* paradigm. Reductionism assumes that understanding a system is possible through understanding its isolated components. For example, reductionism is successful when applied to the kinetic theory of gases, where considering the motion of individual molecules enables us to accurately predict the global properties of the gas system (such as pressure, volume and temperature) [108].

However, the presence of interactions within complex systems means that if we consider the constituent in isolation, we can not sufficiently understand the whole system. For example, studying the behaviour of an isolated ant is inadequate to provide an understanding of the ant colony [109]. Likewise, genes do not operate as independent units but may up-regulate and down-regulate each other [110, 111]; thus, the alteration of a single gene may significantly affect the entire network. Also, some genes have pleiotropic effects, which is when one gene affects several phenotypic traits simultaneously [112]. Therefore, the effect of changes at the constituent component level on the whole system is not easily predictable in a complex system, nor is the complexity easily quantifiable; a study or assessment of the complexity of such systems must consider the non-linear interactions.

An explanation of the non-linearity of interactions within complex systems is their *non-homogenous* and *non-additive* attributes [113, 114]. For a simple example, consider function f that is regarded homogeneous if it exhibits the following property:

$$f(\alpha x) = \alpha f(x), \tag{3.1}$$

where α is a scalar and x is the argument. The output of a homogenous system only depends on the magnitude of the input into the system. However, the output of a non-homogenous system depends on both its input and its current state. In other words, the input of a non-homogenous system is not directly proportional to its output [114, 115].

The additive property means that a part of the system is completely independent of other parts. A function f is additive if:

$$f(x + y) = f(x) + f(y), \tag{3.2}$$

where x and y are arguments. For example, consider a system that produces *Output1, Output2, and Output3* given *Input1, Input2, and Input3*, respectively. In an additive system, if $Input3 = Input1 + Input2$ then $Output3 = Output1 + Output2$. In non-additive systems, the relationship given in Equation 3.2 does not hold because components of the system are not independent of each other. As a result, in a nonadditive system, $Output3 \neq Output1 + Output2$.

The non-linearity attribute of a complex system means that its output is not a simple (or even parameterised) sum of its inputs. Therefore, the number of components of a complex system alone does not reflect its complexity; instead, the complexity of such systems must reflect the intricate relations, amongst other things. Thus, defining complexity in a non-linear system is a non-trivial task.

3.2.1 Complexity in Various GP Applications

Studies of the evolution of complexity indicate that at some point all complex systems increase complexity [116]. The fitness improvement in complex systems relies on the intricate interactions between their subunits and is not a summation of the fitness of the subunits [117]. Therefore, evolutionary pressure simultaneously acts on the system's subunits as well as the interactions between them [118]. The constituents of the solution and their relationships result in a behaviour (function) that determines its fitness.

Genetic programming (GP) evolves solutions by combining basic primitives to build and optimise intricate structures. In most of the popular GP applications, these structures are non-linear [119–122]. For example, in symbolic regression (SR), GP combines mathematical operators (e.g., add, sin, cos, subtract), constants, and variables to produce mathematical equations that serve as data models. During the GP process, the SR solutions (models) evolve to fit their purpose. The models become increasingly complex by adding components or changing the relationship between existing ones (add new or intensify existing relationships);

The inherent and gradual complexification of solutions during the evolutionary process of GP is the source of its utility. However, the complexity of the GP solutions may become excessive to impact their quality. For example, data models should be functionally only complex enough to explain the phenomenon that generated the given data and therefore must predict well (generalise) on new data [2].

If the data models are functionally over-complex, they may go beyond reflecting the general phenomenon that produced the data by also learning the noise in the data; therefore, overly complex models predict very well on training data but poorly on unseen data (they overfit).

The challenge is to determine a measure of complexity, which enables the control of the functional complexity of GP models during their evolution. Like the complex systems described earlier, the constituents of GP SR models often have intricate and non-linear relationships. A single change in a model may dramatically change the functional complexity of the model; in contrast, several changes together may not change anything at all. Yet, the most popular approach to managing the complexity of GP solutions is to prefer those with small-sized representations (trees). Although the small-sized expressions that represent the models tend to (loosely) be less complex than much larger-sized ones, size neither fully encapsulates the functional complexity of the constituents nor their often non-linear relationships.

Furthermore, in GP based automatic programming and design, another motive for containing complexity is to produce efficient solutions – as discussed in Section 2.3.7; efficiency thus is another view on complexity. Inefficient solutions take longer and more resources to execute; this is sometimes a crucial criterion when choosing appropriate solutions, and most times a desired one. In such cases, the size of the representations of the programmes and designs does not necessarily indicate their efficiency; after all, small solutions can still incorporate inefficient components. For example, an electronic circuit may comprise of a small number of components but the design may still yield unacceptable latency. Similarly, a small program with multiple nested loops may take longer to complete than a syntactically larger program.

In addition to application-specific motivations, a common motive for managing complexity is to overcome computational constraints. In many instances across GP applications, if the complexity of GP solutions is unconstrained, the computing resource utilisation for evaluating many individuals over many generations may be excessively high to a level that slows GP's search or brings it to a halt. Furthermore, devices such as *Internet of Things* (IoT) may constrain the execution time of an acceptable model (evolved by GP) even if this compromises the model's accuracy [3].

From these examples of motives for managing complexity, it is apparent that the notion of complexity in GP is contextual and diverse. Furthermore, the discussion highlights a need to consider context and motivations when assessing any (existing or emerging) complexity control. Later, this chapter (Section 3.3.3) introduces the *evaluation time* of an evolving model as a measure of its complexity. Although not perfect, it is still a more nuanced approach to estimating complexity in GP than size as it can reflect many forms of complexity to address the various motives for controlling the complexity of GP solutions.

However, before that, the following section reviews some of the popular approaches to controlling the complexity of GP solutions.

3.3 Existing Complexity Control Methods

This section reviews the most widely used measures of complexity in GP; most of this literature concerns the classic applications of machine learning, that is, regression and classification – the most popular applications of GP. The popularity of these applications in GP literature ensures an adequate depth of the review. However, later in Chapter 9 examines the complexity of GP solutions in other domains as well.

3.3.1 Approaches Based on the Structure of Representations

The term structure here refers to the representation of the GP solutions – such as the tree structure – and not the relationships between its building blocks. As introduced in Section 2.3.7, managing this structure is the most popular approach to managing the complexity of GP solutions. However, this approach often assumes that simplifying the structure of the representation of a GP solution will automatically simplify its functional behaviour. While a complex structure (e.g., a large tree) may constitute many components and exhibit a more complex function than a small structure, this is not always the case. For example, a large structure may contain building blocks that do not affect its functionality or have components with weak or no relationships; thus, a complex structure may still be functionally simple. Accordingly, studies show that this approach to managing complexity may not address the motivations for managing complexity; for example, discouraging the sizes of data models does not always lead to models that extrapolate well to unseen data [7, 8].

However, the popularity of structural indicators of complexity at least partially owes to the ease of their use (e.g., counting the number of nodes and layers in a GP evolved expression), and therefore easy to monitor and control [123]. Furthermore, since the inception of GP, it was observed that the structure of GP solutions has a propensity to grow unproductively – a phenomenon termed *bloat*. Therefore, controlling bloat has been the most popular approach of managing complexity that has produced several theories explaining it and many techniques for containing it. Although this thesis poses that complexity is more than the structure, this chapter highlights and reviews some of the bloat related literature in detail because of the ubiquity of bloat-control in GP; moreover, bloat and the impact of its control becomes a very important benchmark for what this thesis proposes later.

Theories Explaining Bloat

Several theories have emanated that explain the propensity of the GP solutions to grow unproductively (bloating); some of them overlap. An overview of the leading ideas is as follows:

Hitchhiking: An early theory explains bloat as simply the occurrence of ineffective building blocks within a fit individual [124, 125]; the unproductive components of the individual are called *introns* (a term borrowed from genetics). Thus, being part of an individual with high fitness allows introns to propagate – the theory is termed *hitchhiking*. This theory maintains that unproductive components can be part of an individual by chance without adversely affecting its fitness. Other theories attribute bloat to the effect of GP operators and the characteristics of the individuals and the population.

Defense Against Crossover: Another theory explains that individuals with introns proliferate because they are more resilient against the adverse effect of the crossover operator [126–130]; the crossover operator is the most used genetic operator but it often degrades the fitness [129, 131]. Therefore, introns represent regions in a model where alterations do not affect the fitness of the whole. Consequently, a large-sized individual with many introns will thrive better than those that experience non-neutral variations that disrupt fitness.

Removal Bias: Similar to the defence against crossover theory, the *removal bias theory* asserts that evolution in GP favours individuals with regions that are not affected by addition or removal to their representation [132, 133]. Removal and additions to an individual occur during crossover or mutation operations. Hence, the large-sized individuals that are more likely to have such regions will proliferate.

Modification Point Depth: Another bloat theory is modification point depth [134–136], which associates bloating with the depth at which modifications of the tree representation occur; they observed that the deeper the modification point, the smaller the change in fitness. Further, the deeper the modification point, the smaller the removed branch, thus creating a removal bias [137]. Accordingly, deeper trees, which tend to be larger-sized, proliferate more during the evolutionary process.

Fitness Causes Bloat: The *fitness causes bloat* theory argues that the tendency of solutions to bloat is inherent in a variable-length representation when fitness-based selection is used [81, 84, 138]. The theory builds on the idea that for a given function, there are more ways to represent it using a larger-sized expression than with a concise expression. Therefore, the lengthy representations will occur more often during GP runs; furthermore, the lengths are likely to increase with time because lengthier but equally fit solutions can emerge as they change during the evolution. In essence, fitness-based selection leads to bloat.

Crossover bias: A relatively recent bloat theory is the *Crossover Bias* theory, which relates bloat to the effect of the crossover operator on the distribution of tree sizes in the population [139–142] (also known as *operator length bias*). The theory backed by empirical evidence shows that there is a bias in favour of larger-sized individuals in the population. Theoretically, after crossover (the exchange of subtrees), the average size of individuals is expected to remain the same; however, in practice, the average size of the population increases after repeated crossover operations. Hypothesis and experimental evidence show that this distribution, without the effect of selection, is a Lagrange distribution of the second kind [143–145], where small individuals are much more frequent than the larger ones. For example, crossover generates many single-node individuals that are generally unfit for most non-simple applications. Selection tends, therefore, to reject them in favour of the larger-sized individuals, causing an increase in mean tree size in the population.

These theories have inspired a variety of techniques for controlling bloat. The following section discusses representative examples of them.

Bloat Control Methods

Traditionally, controlling complexity in GP means controlling the size of the representation of the evolved models (bloat-control), such as by limiting the number of nodes, encapsulated sub-trees and layers. Bloat-control can ease the challenge associated with the unwanted growth in the structures of GP individuals that can exhaust computational resources and severely stifle the search for solutions [1, 146–151]. However, bloat-control ignores the underlying functional or computational complexity of the solutions.

Many bloat-control techniques either set an arbitrary size (or depth) limit for models or penalise large models [152]; the intron-based bloat theories and tree-depth-based one inspire this practice. However, studies show that such setups also encourage the models to grow to the size limits because that guarantees their survival after crossover [153, 154]. Other bloat-control techniques either limit the search space or lessen the likelihood of generating large models through customisation of the evolutionary process [152]. Another practice is to use multi-objective genetic programming (MOGP) [123], which optimises the twin objectives of fitness and size to obtain Pareto optimal solutions. Some instances of MOGP combine fitness and other measures of complexity.

Dynamic Operator equalisation (DynOpEQ) [155], a recent and advanced bloat-control technique, dynamically changes the distribution of size in a population to admit more individuals in those size ranges that are producing fitter models; DynOpEQ is inspired by crossover bias theory. First, DynOpEQ segments the individuals of the population into bins according to their sizes. As GP creates individuals for a new generation, DynOpEQ uses a quota system to decide whether to keep them in the new population or discard them. DynOpEQ sets quotas for size ranges (bins that correspond to those from the parent generation); it sets quotas for each bin based on the average fitness score of the corresponding bin in the parent generation. As a result, bins with higher average fitness scores are allowed more individuals. Still, DynOpEQ will always admit an offspring if its fitness score is higher than the average score of the bin it belongs to, even if the bin's quota has been exhausted. Further, if the fitness of an offspring is the new best and its size does not fit into any existing bin then a new one is created for it. Effectively, DynOpEQ somewhat controls the growth in size in the population unless fitness is improving; however, it does not guarantee that bloated individuals will not exist in the population. Furthermore, DynOpEQ is inefficient as it discards a significant number of evaluated candidates; an evaluated offspring is likely to be discarded if the bin it belongs to is full. To alleviate this problem, Mutation Operator Equalisation [155] was introduced; to avoid discarding of evaluated individuals, it mutates candidates (in small steps) to fit into the nearest bin that has available space. However, the required changes may be exceedingly destructive to the model's fitness when the distance between the individual's original size and the size it needs to be is too large.

Some studies have explored the concept of Kolmogorov complexity [156, 157] to manage complexity in GP. This concept is adopted from algorithmic information theory and relates to specifying an object such as a binary string or a sequence of numbers. The Kolmogorov complexity of such an object is the length of the shortest computer program that can produce it and nothing more. An abstract coding language (universal Turing machine) provides a reference. However, this measure is uncomputable [158, 159] because it is impractical to determine the shortest such program definitively. Therefore, the minimum description length (MDL) [160, 161] – a computable form of the Kolmogorov complexity – has been applied in GP instead [162]. The study [162] calculated the MDL value of an individual by summing the length of code required to encode it (e.g. tree size) and to encode its classification errors. Then, a scaling technique

[163] is used to transform the MDL value to a windowed MDL value; the windowed MDL value is relative to the maximum MDL value of the individuals produced up to that point of the GP run. The MDL-based fitness function (configured to minimise the windowed MDL value) controlled the growth of the individuals. However, the implementation only works on types of problems that have two features: (1) the performance (fitness) of candidate solutions increases with the growth of the trees (e.g., SR), and (2) where the fitness of the substructures of the trees are well-defined.

Despite all the discussed advances, restricting the structural representation of GP solutions is not an effective way of managing their complexity. For example, the size of the representation of a symbolic regression model may not represent its underlying functional and computational complexity. For instance, restricting size would mean penalising the large yet linear expression $9x + 6x + 3x + 2x + x$, which is computationally less complex than the smaller expression $sin(x)$ [8], which is computationally equivalent to the implementation of its Taylor series expansion $\sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$. Moreover, the response surface of $sin(x)$ is more complex than that of the linear function that is larger in size. Furthermore, two expressions with the same size may have different response surfaces; the response of the function $sin(9x)$ has more oscillations than $sin(2x)$. Therefore, model complexity in GP is more than their representation. The finding of studies that bloat-control does not automatically lead to models that generalise may be explained by the understanding that size and functional complexity are not the same [7, 8, 164].

Considering the motivations for managing the complexity of GP solutions, approaches that focus on the structure of models are limited. While it alleviates the computational constraint challenge, it does not effectively manage functional complexity to address overfitting. This observation suggests that, despite the ease of quantifying and hence controlling bloat, alternatives to bloat-control methods must be considered that look into functional characteristics of the evolving structures. While the proposed evaluation time-based method of controlling functional and computational complexity promises to be easy to implement, and broadly applicable, this review first look into some existing approaches from the literature that have gone some way down this route.

3.3.2 Functional-Based Approaches

Instead of working on the structure of expressions that represent GP models, another approach for managing complexity considers the functionality of the models. Methods in this category aim to detect and control the functional complexity of models; they try to evolve simpler functional forms in a bid to elicit data models that explain the phenomenon that generated the data and not the noise in the data. That is, they seek models that generalise on new data well, which bloat-control does not do effectively.

A method for quantifying functional complexity approximates the evolved expressions with Chebyshev polynomials such that the more functionally complex expressions have polynomials of higher degrees [165]. The degree of the approximating polynomials is termed as the *order of non-linearity* of the corresponding GP expressions [166]; it is minimised during the evolution to produce functionally simpler models. However, applying this method sometimes fails because it requires the evolved expressions to be twice differentiable, which is a property that is not always guaranteed. For expediency, the study combined and optimised the order-of-non-linearity and two other objectives: accuracy and expressional complexity (size). Furthermore, the study introduced a framework that alternates between optimising two sets of objectives during the evolution: (1) order of non-linearity with accuracy and (2) size with accuracy.

This optimisation framework (referred to as 2-D) showed some improvement in managing the size of evolved models and their generalisation ability.

To avoid the failings that requiring models to be twice-differentiability can present (in [165]), the study in [7] defined a less rigorous measure of functional complexity. In the study, the proposed method approximates the slope of an expression along each feature dimension with a simpler but error-prone measure that approximates second-order partial derivatives with a finite difference method that uses unequal intervals. Further, the eventual measure of complexity that the work proposes is mathematically questionable because it simply averages these approximations to partial derivatives across all the feature dimensions to get an average complexity. To what extent that average diverges from a Hessian is not discussed. Furthermore, to avoid the computational expense, they only computed complexity for the best individual in each generation, thus not using the measure as a complexity control but merely as a complexity indicator. Crucially, however, the paper reported that a decrease in this measure did not necessarily improve the generalisation ability of the models.

Furthermore, the work in [167] offered other complexity measures that still relate to the curvature of the response surface of the models. They quantified the *degree of curvature* by examining the output of the pairs of close training points for a data model. To indicate oscillations in the curvature, they counted the number of pairs of outputs with very different values. The result formed the basis of two measures: (1) *graph-based complexity* to measure functional complexity and (2) *graph-based learning ability* to quantify the ability to learn difficult training points. The outcome of the experiments showed some generalisation gains over standard GP. Although the authors did not report statistics describing the execution time differences, the proposed methods likely introduce computational overheads.

Some methods for managing functional complexity in GP exploit the knowledge available from statistical learning theory. Notable examples include the generalisation error-bound Vapnik–Chervonenkis (VC) theory and Rademacher complexity theory [168] [169], which aim to find a balance between model complexity and its generalisation capability. The VC dimension is a general measure of the capacity or complexity of a learning machine [170, 171]; that is, it is a measure of the expressive power of a set of functions that a statistical binary classification algorithm can learn [170]. According to the original definition (proposed for a set of indicator functions), the VC dimension is the maximum number of vectors that can be separated (shattered) into two classes in all possible ways by a set of functions [170]. The VC dimension enables various estimations of generalisation errors.

Structural Risk Minimisation (SRM) is a framework that uses the VC dimension to assess the generalisation ability of a learning machine [172]. The assessment involves predicting the distance between the training and test errors. To achieve this, SRM defines the upper bound of the generalisation error based on the empirical risk (training error) and confidence interval. The confidence interval measures the difference between the empirical risk (training error) and the expected risk (generalisation error); it depends on (1) the VC dimension and (2) the size of the training dataset. If the size of the dataset is fixed (as is typically done), the generalisation bound is indicated by the VC dimension only; therefore, it is also referred to as VC generalisation bound or VC bound. SRM is used to produce an optimal model that finds a balance between minimising the upper bound of the generalisation error and the training error. For example, SRM was used to manage the complexity of evolved models in [173]. While the proposed method outperformed standard GP by producing smaller sized models with better generalisation, the authors acknowledged the expensive computational cost of the method and the lack of exploration of the parameters used in measuring the VC dimension.

Rademacher complexity extends the VC dimension to handle real-valued functions; therefore, it is applicable in both classification and regression types of problems. This measure of complexity has a tighter bound on the generalisation error than the VC dimension, and unlike the VC dimension, it depends on the data distribution. For example, a study [174] used the Rademacher complexity in the fitness function to steer the evolution and control the functional complexity of its models. The method produced models with better generalisation ability than those generated by standard GP and Support Vector Regression (SVR) [175]. Also, the solutions the method produced were smaller-sized than the models standard GP yielded. However, the implementation has a lengthy process of tuning the parameters used to increase the pressure when overfitting occurs. Like the VC bound method, the Rademacher is computationally expensive.

Furthermore, the work in [176] used the *variance* of the output values of evolving expressions to infer their complexity. The study combined the variance and fitness as twin objectives to optimise. Although variance differs with mathematical smoothness (a straight line can have more variance than a sinusoid), its combination with fitness means that expressions within a similar functional space may normally be compared in later generations, where greater genetic convergence usually occur. However, this needs further verification. The method improved generalisation. Moreover, since the proposed method does not require specialised multi-objective optimisation methods, it is simple to implement and computationally inexpensive.

Tikhonov Regularization (also known as ridge regression) is a simple and common L^2 parameter regularisation strategy (for managing complexity). The work in [177] proposed a measure of complexity that combines the Tikhonov regularisation (as a functional complexity indicator) and size (a structural complexity measure). This proposed measure is motivated by the understanding that addressing structural complexity does not necessarily address functional complexity and vice versa – while addressing both is important in GP. Therefore, they proposed a method that registers the complexity of individuals in a two-dimensional (2-D) vector that consists of (1) Tikhonov Regularizer, which indicates functional complexity through the smoothness of response of the function, and (2) the size of the model, which represents its structural complexity; the Pareto optimal individual is the preferred. Then, traditional *multiobjective GP* optimises both the proposed measure of complexity and accuracy of the models. The results show that the generalisation ability of models produced by this method improved over GP with bloat-control; also, they attain higher accuracy over GP with Tikhonov Regularization. Like the discussed *order of non-linearity* method, this method has to contend with the differentiability of the models and the related computational costs.

The functional complexity methods generally offer some gains in the generalisation ability of models but come with associated challenges. The computational effort of some of these methods is so expensive that in some cases, such as the work in [7] only employed the measure for the best individual, thus rendering it impractical for complexity control. Some have stringent requirements like twice differentiability, while others use questionable approximations. Moreover, unlike bloat-control, they are non-trivial tasks to implement.

Furthermore, the functional complexity methods do not transfer from one GP application to another. These methods – most of which apply in regression problems, some in classification, but seldom both – do not work in other GP applications such as automatic programming and design. This limitation is consequential for a versatile tool like GP.

Therefore, despite the limitations of bloat-control (as highlighted in Section 3.3.1), such as its inability to manage functional complexity effectively, it remains the most popular form of complexity control in the GP literature. Although limited in its ability to address the various motivations for managing complexity, its simplicity and ease of use across GP applications contribute to this popularity.

The GP community needs an approach for managing functional complexity that is effective and simple. The proposed time-based measure of complexity of GP portends to be equally simple, broadly applicable, but more effective than controlling size.

3.3.3 Time Can Indicate Complexity

As indicated in the previous sections, there is a need for a measure of complexity that can effectively characterise complexity to address the motivations behind the complexity control when applied. Also, a measure of complexity that is broadly applicable and simple to manage is desirable. Therefore, this thesis proposes the use of the *evaluation time* of the GP solutions – the computational time required to evaluate a GP solution (compute its fitness) – as an indicator of their complexity. For example, the time it takes to test a data model with the given data; and the time it takes to execute an evolved programme. This idea is inspired by the observation in reality and computing that simple tasks take less time to perform than more complex ones. Accordingly, models within a GP application can have different evaluation times based on differences in their computational complexity, which may also relate to their functional complexity. For example, executing transcendental functions can be computationally more complex than simple arithmetic operations.

Furthermore, a data model constituted from computationally-expensive building blocks or carrying a large structure takes a long time to be evaluated and hence is computationally complex. Therefore, curbing the evaluation time can discourage the growth of both the structural and functional complexity.

In addition, evaluation time can detect efficiency in evolved programs and designs (e.g., digital circuits). The efficiency of these solutions is a form of complex behaviour because simple programs require less computational effort to execute than complex ones.

Furthermore, the evaluation time can address the computational constraint better than using size; as discussed in section 3.2.1 and 3.3.1, a small-sized representation of GP solutions can be computationally more expensive than a larger-sized one. Therefore, constraining the evaluation times of GP solutions will reflect more positively on the execution times of the individuals and the runtimes of GP than if their sizes are constrained.

The examples of the potential of the evaluation time presented in this section suggest that it will be a versatile and broadly applicable measure of complexity. However, the next chapter discusses this matter in detail.

3.4 Conclusion

This chapter examined the popular measures of the complexity of GP solutions and the techniques used to manage them and highlighted the need for a broadly-applicable indicator of the complexity. Essentially, it demonstrated that given the diversity of GP applications, the very definition of complexity is context-dependent, multifaceted and a reoccurring challenge across GP applications. In addition, it highlights the challenge of defining the complexity in GP and points out that because a formal and adequate definition of complexity is pending, the common practice is to resort to proxies. However, the measures of complexity that serve as proxies of true complexity are limited.

While reviewing existing complexity control methods, this chapter underscored the need to consider the various motivations that drive a need for complexity control. The most popular category of complexity control methods discourages the sizes of the expressions representing GP solutions (structural). These methods reduce the structural complexity of the representation of the solutions but ignore their computational or functional complexity. As the functional complexity of data models is associated with overfitting, this approach does not address the overfitting challenge effectively. The second category of methods (functional), which aim to address the shortcomings of the structural complexity method, offers some gains but with additional challenges. They introduce significant and sometimes prohibitive overheads, and at other times demand constraints like twice-differentiability of randomly evolving expressions; in any case, they are not broadly applicable beyond regression. However, this thesis notes that GP applies beyond regression and that the complexity control methods do not transfer from one GP application to another.

Therefore, the review in this chapter recommends building towards a broadly applicable measure of complexity that is aptly nuanced. Consequently, the chapter merely introduces *evaluation time* as a measure of complexity; however, the next chapter will detail it. Also, the next chapter will address the key questions and practical challenges associated with using the evaluation time. These include: (1) a demonstration of how to measure the evaluation time reliably and (2) validation of time's ability to effectively detect differences in both the sizes and the computational complexity of individuals.

Chapter 4

Time as a Measure of Complexity

"Time is the most valuable thing that a man can spend."

Diogenes

4.1 Introduction

This chapter examines the viability of *evaluation time* as an indicator of complexity. Therefore, it addresses questions related to the practical use of evaluation time to measure and control complexity.

The chapter details how – without using complicated programming structures (such as loops for components) and instead by using mathematical functions only – the evaluation times of models can still vary significantly. Hence, it shows that this variation can be related to the smoothness of the response function of the evolving functions.

In addition to building an intuitive case for evaluation time as a complexity indicator, this chapter also demonstrates this empirically. This demonstration is essential because it practically shows that the differences in computational complexity show up in evaluation times and thus can help GP differentiate between different kinds of complexity in models.

Furthermore, this chapter demonstrates that although evaluation time depends upon the size of an expression, contrary to some existing arguments in

GP literature, it is more than just a proxy for measuring size. In fact, this thesis argues that evaluation time can vary between differentiating size and functional complexity, and outlines the evolutionary conditions that dictate this variation.

Finally, a key question in measuring evaluation times is their reliability: evaluation times vary even for an identical process. Hence, for GP to use evaluation time to differentiate complexity, the variation must be curbed to bring some tolerable degree of certainty. Accordingly, this chapter proposes strategies to improve the reliability of the evaluation times measurements and demonstrates this reliability with experiments.

The rest of this chapter is organised as follows: Section 4.2 provides a closer look at evaluation time as a measure of complexity; Section 4.3 details experiments and analysis to show that the evaluation time can detect more than the sizes of GP individuals; Section 4.4 presents strategies for measuring the evaluation time reliably, and empirically demonstrates their effect; and finally, Section 4.5 concludes the chapter.

4.1.1 Time Complexity

Although the use of time to measure the complexity of GP solutions is new, the idea of using *time* to indicate complexity in computing is not. For instance, *Big O time complexity* is used in software design to measure the complexity of algorithms and describe their limiting behaviour [178–180]. Furthermore, machine learning fields have sparingly used the Big O time complexity [181–183]. Although different to the proposal of this thesis, Big O time complexity serves as a precedent showing that time can indeed reflect at least a form of complexity.

The runtime of computer programs and algorithms has always been a concern in computing. Hence, developers in computing strive to produce solutions with the shortest possible runtimes to meet many objectives, such as minimising the processing effort required, power consumption, and response time. To manage some of these challenges, the Big O time complexity – a time-based indicator of the complexity of algorithms – is used. Big O time complexity tells how quickly the runtime of an algorithm grows as the size of its input increases. Therefore, the Big O notation is used to communicate algorithmic complexity and is in the form $O(n)$, where O stands for *order of magnitude* and n signifies the complexity; the calculations consider the worst-case scenario. For example, $O(1)$ denotes a *constant time complexity*, which means that the time complexity of the algorithm stays constant regardless of the size of the input. In addition, $O(N)$ denotes a *linear time complexity*, which means that the time complexity grows in direct proportion to the size of the input (N). Also, $O(N^2)$ denotes a *quadratic time complexity*, which means the time complexity is directly proportional to the square of the input size. FIGURE 4.1 shows an illustration of the Big O time complexity.

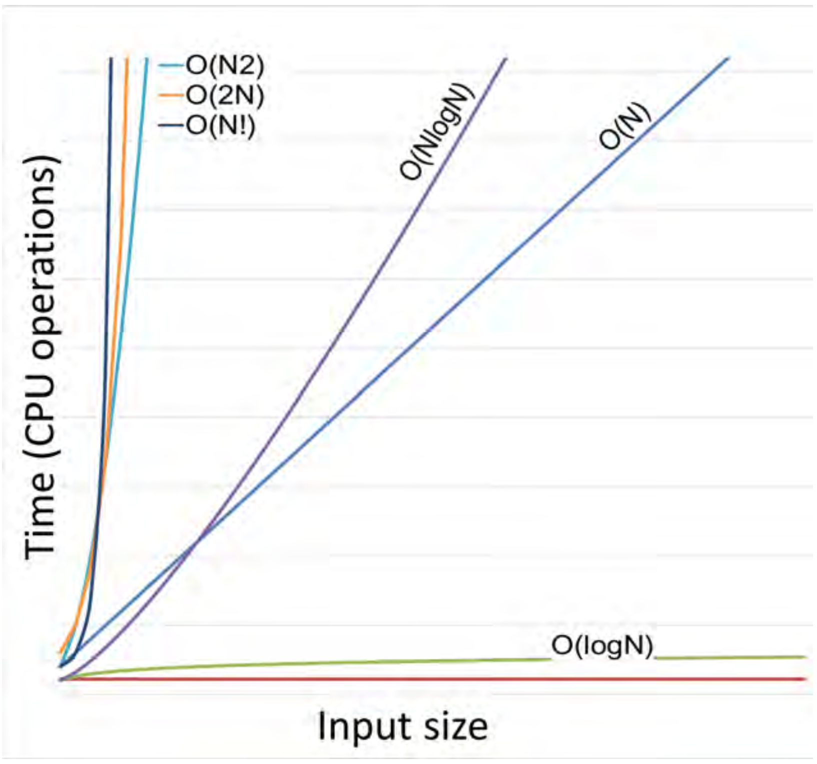


FIGURE 4.1: The line graphs show how time changes with input size for different Big O time complexities.

Big O time complexity is decided after a thorough understanding of the behaviour of a computational process; this is something infeasible when thousands of processes (varying GP individuals) are evolving. Therefore, this thesis resorts to a more practical measure, that is, the actual run time (or the *evaluation time*) of the evolving expressions.

4.2 The Evaluation Time

In this thesis, *evaluation time* is defined as the time it takes to evaluate GP solutions. Usually, GP must evaluate all candidate solutions the same way to assign fitness scores fairly. For example, GP uses the same dataset to evaluate data models in a population, and their evaluation times may vary due to differences in their make-up. Similarly, programs evolved by GP may have different evaluation times based on their efficiency.

Around the time of publishing the proof of concept of using evaluation time to control complexity in this thesis [184], another study [185] also used evaluation times to discourage growth in the size of GP individuals (bloat-control). Although this other study corroborates the premise in this thesis that time is a viable measure of complexity, the work presented some empirical results to reason that evaluation times behaved as merely a proxy for measuring expression sizes. However, the investigations in this thesis dispel this notion, and this chapter shows that evaluation time indeed varies for same sized individuals, even in symbolic regression. The following section argues that due to the evolutionary dynamics – especially during the later GP generations where the population has typically converged to very similar individuals – evaluation time may not vary much and thus may resign to being merely a proxy for sizes. However, this behaviour is not constant and is actually a function of the state of the evolution, as explained in the next section.

Therefore, this thesis has argued that the evaluation time can reflect more than the sizes of the evaluated individuals; this thesis poses that it can reflect the computational and functional complexity of the GP solutions and their efficiency. Therefore, this chapter empirically demonstrates that the evaluation times of GP individuals can detect more than their sizes in this chapter.

4.3 Time Detects Size and Functional Complexity

Since Chapter 3 explains that the model size does not necessarily represent functional complexity, this section empirically demonstrates how, instead, the evaluation time can discriminate between different functional complexities of models (through their computational complexities). After all, the evaluation time is also a function of model size and if functional differences of identically sized expressions do not show up in their evaluation times then measuring time is just another way of measuring expression size. Clearly, this is undesirable.

As discussed in Section 3.3.1, one can expect the linear expression $x + x + x + x + x$ to be both computationally and functionally simpler than a shorter expression $sin(x + x)$, which functionally represents an oscillating behaviour and computationally requires the corresponding Taylor series to be computed. If the expected computational differences exist and computational complexity reflects functional complexity, then a GP system can exploit these differences to promote functionally simple models. Accordingly, the experiments in this section verify whether considerable time differences exist between different functional complexities of identically sized expressions.

Although the verification experiments are limited to mathematical functions, the approach is widely applicable; the verification establishes that time can reflect more than the sizes of the models. Chapters 8 and 9 confirm that this understanding also applies in other GP applications such as robot control and Boolean circuit design. Accordingly, the results in the chapters show that, when exploited, the nuanced nature of evaluation times can produce simple solutions that offer qualitative gains.

4.3.1 Experimental Setup

The experiment in this section demonstrates that evaluation time can detect more than size. To this end, four different *function sets* were used to generate symbolic regression models of different complexities. Functional complexity of these sets decreases in the following order: $\{cos, sin\}, \{cos, sin, +, -\}, \{\times, \div, +, -\}$ and $\{+, -\}$.

Then, differently sized expressions (10, 20, 30, ..., 300) were generated for each function set; for each size within the function sets, 30 random expressions were generated.

For adequate sampling and to generate enough data to analyse, the models (expressions) were evaluated 50 times each, using the same data.

4.3.2 Experimental Results

The four line-graphs in FIGURE 4.2 represent the average evaluation times of the individuals according to their size and functional complexity.

Two trends are clearly visible in Figure 4.2: (1) for a given size, the higher the functional complexity, the greater are the evaluation times; and (2) the evaluation times are strongly correlated with the expression sizes, as expected. Hence, the evaluation time indeed discriminates between different functional complexities.

However, if a simple function is represented inefficiently by an excessively large expression, it evaluates slowly. Therefore, evaluation time control impacts conditionally: it curbs functional complexity when individual sizes are similar and allows greater sizes for functionally simpler models up to a certain tolerance (or range); however, in the presence of very different sizes (such as during early generations) it curbs growth in size (controls bloat).

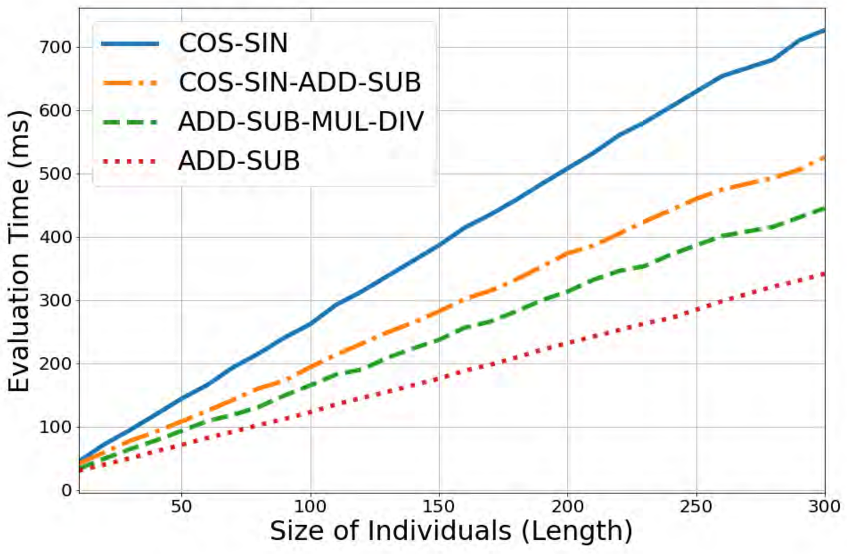


FIGURE 4.2: The line graphs illustrate the relationship between the evaluation time and features of a model. The graphs show the mean evaluation times of models made up of mathematical operators of different computational complexity. Individuals made up of COS/SIN operators have higher evaluation times than same-sized ones made up of less complex operators. Moreover, the size of models correlates with their evaluation time.

To estimate the tolerance for greater sizes of simpler models, compare different curves at identical values along the y-axis of Figure 4.2: for example, at evaluation time = 300 ms, ADD-SUB models can be more than twice as large as COS-SIN models.

The above findings also reveal the limiting behaviour of evaluation time control in GP. In a functionally diverse but size-converged population – where the bloat-control is impotent —, evaluation times discriminate between functional complexities; whereas, in a functionally converged but size-diverse population, evaluation times discriminate between sizes.

4.3.3 Challenges with Administering Evaluation Time

On the basis that any computational work takes time on a granular basis, the evaluation time should closely reflect the computational complexity of the task in an ideal situation. However, a question arises as to whether the measurement of time can be reliable and fine enough.

The evaluation time measurements of an individual can vary significantly from one computer system to another; however, this variance is inconsequential because we are concerned with the relative complexity of contending GP solutions within a GP run. However, within a system, multiple time measurements of an individual may vary due to excessive resource sharing by the operating system. Such variations within the system will make the estimations of the complexity of the solutions unreliable. Therefore, this thesis must find ways to address this challenge before proceeding with the study.

Accordingly, the following section proposes measures that improve the evaluation time measurements and details experiments to verify their efficacy.

4.4 Measuring Evaluation Time Reliably

This section demonstrates how to improve the consistency of the evaluation time measurements. Consistency in the evaluation time measurements is critical to estimate the complexity of models reliably. However, evaluation times vary across multiple executions, and if this variability is high, the reliability of the complexity estimate is low.

Noting this challenge, the work in [186] proposed a relatively expensive measure whereby they made multiple measurements of times and used the average (treating it as the first quartile) and left tasks with very long run times as is. Another study [185], which avoids such computational expense, ignored the impact of variability altogether and measured the time just once. This thesis takes a more careful approach whereby the time variation is minimised significantly without incurring extra overhead.

Although the variability may not be eliminated absolutely – because of the CPU scheduling decisions being beyond our control – later sections of this chapter show that the proposed CPU management options adequately improve the time measurements.

4.4.1 CPU Management Options

Careful consideration and some testing showed that specific CPU management options minimise variations of the evaluation time measurements. The CPU management options are as follows:

1. *Stop background services:* A typical office or home system has several background services running, which means processing will switch from one task to another. Furthermore, because some background services are triggered by events or scheduled to activate periodically, the sharing may not be predictable. This unpredictable sharing of the CPU means that the evaluation time of an individual may vary within a system. Therefore, stopping background services avoids excessive CPU time sharing.

2. *Lock processor speed:* Modern computer systems usually have advanced power management systems that regulate the processor (CPU); for example, to dynamically manage the power consumption and conserve battery life. This change in CPU speed may affect the evaluation time measurements. Therefore, locking the processor speed prevents the power management feature from making these changes.

3. *Run on dedicated processors:* The popular operating systems (e.g., Windows 10 and Linux Ubuntu 18) permit the running of programs on dedicated

processors. Pinning a GP run on a CPU means that the expected evaluation time of an evaluation task will not vary because of load differences across the CPU cores.

4. *Execute with high CPU priority:* Setting a higher than a normal priority for the GP runs further ensures that another task will not relegate an evaluation to the background and affect the time measurements. Testing of this idea showed that setting the CPU priority of the GP run just above normal is better than to the highest.

4.4.2 Experimental Setup

The experiment in this section compares evaluation time measured with and without the CPU management options listed above. Therefore, the experiment ran tests under the following two conditions:

1. *Condition 1:* This setting simulates a typical level of utilisation of an office system by simultaneously running multiple applications as the evaluation times are measured. Several applications run simultaneously on the test system to raise its processor utilisation to between 20-30%; examples of the typical applications include email, browsers, office applications, and periodic synchronisation of network drives.

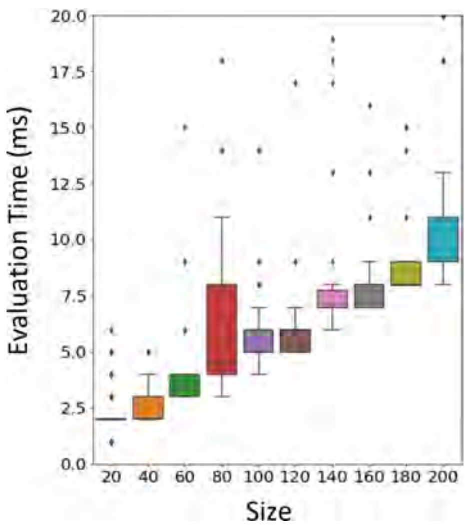
2. *Condition 2:* All the CPU management options were applied.

First, the experiment starts by creating ten individuals of varying sizes ranging from 10 to 200 nodes in steps of 20. Next, under the two conditions described above, the evaluation times of each individual are measured and recorded 50 times. Then, the results are analysed to determine the level of fluctuations in the measurements of the individuals.

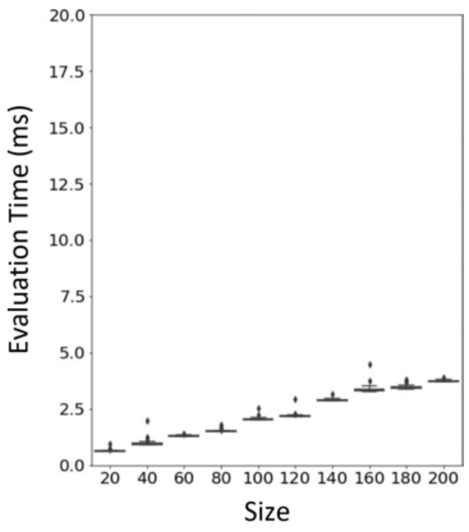
For the evaluation time measurements, the experiment used a CPU-time-based function that employs CPU performance counters [187]. This function is available in Python 3.3 and above. The function offers high resolution (in nanoseconds) across platforms, while the returned values are in fractional seconds.

4.4.3 Experimental Results

Figure 4.3 illustrates the impact of these options. Each boxplot in the figure shows multiple evaluation times of an individual of a given size. Figure 4.3a shows that under Condition 1 (no CPU management options applied), the variation in the evaluation times was high. In contrast, Figure 4.3b shows that under Condition 2 (CPU management options applied), the variation clearly decreased. As expected, the evaluation times have significantly dropped because of the elimination of background services that demand CPU resource sharing. In addition, adjacent boxplots of Condition 2 (the improved readings) do not overlap. Hence, differences between the computational complexity of the individuals are distinguished consistently.



(A) Condition 1 (Before)



(B) Condition 2 (After)

FIGURE 4.3: Result of improving the reliability of evaluation time measurements with CPU management options. Applying the option reduced the variability in the time measurements.

Based on the improvements achieved by applying these CPU management options, single measurements of evaluation time can be used reliably in later experiments. The multiple evaluations of an individual in the experiments in this section are for the sole purpose of studying variations in readings.

The techniques used to improve the evaluation time readings may not work in exceptional situations. For example, stabilising the time measurements becomes more challenging when the models are exceptionally large-sized, such as in [186], where the evolved trees have millions of nodes. In this case, CPU memory caching comes into play; the caching occurs when the processor in the middle of evaluating an individual is forced to access CPU memory caches (L1, L2, and L3) having different speeds. Moving from one type of cache to another may introduce delays and inconsistencies in the time measurements. However, the experiments in this thesis do not have the challenges of the described exceptional circumstance; typically, the experiments here use individuals with tens to hundreds of nodes and not millions.

Overall, applying the measure made the evaluation time readings consistent enough to explore the proposed evaluation time schemes.

4.5 Conclusion

This chapter empirically demonstrated that the evaluation time characterises the complexity of GP models in a more nuanced way than size, which is currently the most popular measure of complexity in GP. The investigation in this chapter shows that the complexity based on evaluation times even distinguishes between the models based on the mathematical functions (e.g., SIN vs ADD) that constitute them and, therefore, will help detect differences in their functional behaviours. Furthermore, the analysis of the results also shows the expected behaviour of the evaluation time. Since the evaluation time can detect such minor differences, it will productively detect pronounced differences in complexity between GP individuals across applications (e.g., efficiencies of programs and designs). This seemingly obvious finding dispels the understanding in some existing GP literature that sees the evaluation time as no more than just a proxy for measuring size. Subsequent chapters will test this indicated ability of the evaluation time in various applications.

To use evaluation times to measure complexity, the reliability of the measurements is critical. The chapter discussed some past precedent in using evaluation times; however, this work either achieved reliability at the cost of a great computational expense (a high number of repeated evaluations) or simply ignored the question of reliability. In contrast, this thesis propose a strategy to measure the evaluation times, which successfully achieves the twin target of attaining reliability without incurring additional expense.

The outcome from this chapter sets the stage appropriately for investigating the value of the evaluation time in managing complexity. The following two chapters propose and assess methods for controlling the complexity of GP models by controlling their evaluation times.

Chapter 5

Explicit Control of Evaluation Time

5.1 Introduction

This chapter empirically demonstrates the viability and merit of using *evaluation time* as an indicator of complexity. Therefore, it compares the effect of controlling evaluation time (time-control) with that of controlling *size* (bloat-control). The methods proposed in this chapter use explicit penalties on high evaluation times, much like how existing methods penalise other complexity measures. This approach differs from a second proposed approach (detailed in Chapter 6) that implicitly controls the evaluation time.

Furthermore, the study in this chapter leverages the findings in Chapter 4 that show that the evaluation time differentiates functional complexity even better in a functionally diverse population of models to create an environment that allows it to do so. To build such a diverse population, a new population initialisation scheme is proposed that is termed the *Fixed Length Initialisation scheme (FLI)*. The FLI produces an identically sized but functionally diverse population. The results of the experiments not only confirm that understanding but also show that FLI generally improves the performance of various GP systems. This additional finding is a surprising but welcomed result.

The outcome of the comparison shows that time-control produces more accurate solutions on both training and test data; however, in terms of complexity control, the result of the comparison is close. Time-control had significantly more accurate models in 17 out of 20 tests and significantly simpler models in 11 out of 20. This result takes us back to the argument in the GP literature as to whether bloat-control manages complexity appropriately, such as whether it improves the generalisation of the evolving model.

The rest of this chapter is organised as follows: Section 5.2 introduces the techniques that will be used to compare the effect of time-control with size-control (bloat-control); Section 5.3 details the need and mechanics of the proposed fixed-length initialisation scheme (FLI); Section 5.4 details the experiments; Section 5.5 presents the results; and finally, Section 5.6 concludes this chapter.

5.2 The Explicit Time-Control Techniques

This chapter employs well-established bloat-control techniques (which traditionally control size) to control the evaluation times of the GP solutions. They were adapted to do so by simply replacing the size of the evolving tree as a measure of complexity with their evaluation times and using the remaining mechanics of the bloat-control techniques as is.

The experiments in this chapter compare time-control with bloat-control because size is the most researched measure of complexity in GP [123]. This extensive research has led to a plethora of bloat-control techniques. However, these techniques may set arbitrary size limits for models or penalise large models without considering their functional behaviour [152]. Other approaches utilise multi-objective genetic programming (MOGP) [123] to optimise the twin objectives of fitness and size to produce Pareto optimal solutions.

The experiments in this chapter use a selection of four well-known bloat-control techniques, which are popular in the literature and are known for balancing accuracy and size-complexity. The selected techniques are as follows:

1. *Death by Size* (DS) [188] discourages size-growth by simply increasing the probability of replacing the large-sized individuals from the breeding population with the newly created offspring. Once a crossover event produces offspring, DS randomly selects two individuals from the existing population and probabilistically replaces the larger one with the offspring. Typically, the probability of substituting the larger-sized individual is 0.7; the experiments in this chapter use the same setting. To implement time-control with DS (and the other bloat-control techniques described below), the evaluation times measurements replace the sizes of the respective individuals.

2. *Double Tournament* (DT) [188, 189] discourages size-growth by selecting relatively smaller individuals as parents. DT uses two rounds of tournaments to select the parents. The first round runs n probabilistic tournaments, each of which pitches two randomly-selected contenders from the population against each other; in each competition, the smallest-sized individual wins with a probability of 0.7. At the end of the first round, a set of n individuals emerge. Then, in the second round, DT selects the fittest out of the n individuals.

3. *Operator Equalisation* (OpEq) [164, 190] discourages size-growth by identifying and preferring those sizes that on-average produce better fitness values. To do this, OpEq bins the present population according to different sizes and computes the average fitness of each size range (bin). Then in the following generation, each size is allocated a quota of individuals; this quota is proportional to the average fitness of the individuals of this

size. Therefore, the sizes that produce better average fitness values get more representation in future generations.

As a parameter setting, bin widths of 1 to 10 have been used successfully in the previous work [164]; therefore, the experiments here use a bin width of 5. Also, for performance reasons, OpEq was later replaced with a better version called Dynamic OpEq [164]; the Dynamic OpEq version is the choice here.

In addition to replacing size with time (to adapt OpEq to control evaluation time), the time equivalent of the bin width is estimated. The modified OpEQ uses this estimate to create bins that group individuals according to their evaluation times as the original OpEq groups them by size. To estimate the time equivalent of bin width 5, the procedure generates 30 random individuals of size 5; then, their average evaluation time is noted and used as the equivalent of bin width for the OpEq version with time-control. Note, however, that the evaluation times depend upon the number of data points, which vary for each problem. Therefore, the bin-width equivalent is computed separately for each time-control experiment with OpEq.

4. *The Tarpeian* (TP) method [191] discourages size-growth by killing-off a portion of the large-sized individuals in the population regardless of their accuracy. TP realises this by calculating the average size of the breeding population at every generation and then assigning the worst fitness score to a fraction W of the individuals with the above-average size (recommended $W = 0.3$; the experiments here use the same).

Therefore, the selected techniques control size (or time) in a variety of ways. DS disadvantages size during replacement; DT disadvantages size during selection for breeding; OpEq manipulates the distribution of the population to favour a size that is producing the more accurate (fitter) individuals (OpEq); and TP excludes a portion of large-sized individuals. In addition, note that DS and DT use the steady-state replacement strategy and TP and OpEq use the generational replacement strategy. Therefore, these techniques are a diverse set and are commonly cited in the GP literature; they have not been optimised to suit this comparison. The papers that introduced these techniques have been cited in other publications as follows: DS [188] in 226; DT [189] in 99; OpEq [190] in 62; and TP [191] in 213.

5.3 Time Control for Functionally Diverse Populations

Chapter 5 showed that the evaluation time detects both the size and the computational (functional) complexity of models. This finding implies that when the individuals are identically sized but functionally diverse, the variations in evaluation times will mainly be the differences in the functional or computational complexity of the components that make up the models. Therefore, restraining the evaluation time in this environment will restrict the complexity of the components (functions) that make up the individuals; this restriction, in turn, may then lead to a simpler functional behaviour and better generalisation of the models. Therefore, this thesis proposes the *Fixed-Length Initialisation (FLI)* scheme, which starts the evolution with a same-size but functionally diverse population; the details of the FLI are given in Section 5.3.1. Before applying FLI to the experiment that compares time-control with size-control, Section 5.5.1 tests its impact on each method.

5.3.1 The Fixed Length Initialisation Scheme (FLI)

The proposed Fixed Length Initialisation scheme produces an initial population of unique individuals with uniform sizes (or lengths). To further ensure functional diversity in the population, FLI considers two individuals that only differ by numeric constants as not unique. Algorithm 2 shows the pseudocode of the FLI. A fixed length of 10 nodes is set as the default.

Given the function set that the experiments use (7 mathematical operators), FLI can produce a large number of unique individuals with ease. To illustrate, consider the tree in Figure 5.1 which represents an individual made up of only binary functions and terminals. Given that there are 4 options for the binary operators and 5 variables as terminals, the number of unique individuals that can be produced can be calculated as follows:

$$4 \times 4 \times 4 \times 4 \times 5 \times 5 \times 5 \times 5 \times 5 = 160,000. \tag{5.1}$$

Therefore, this tree with 9 nodes, which represents only a subset of the possibilities, can have 160,000 unique individuals.

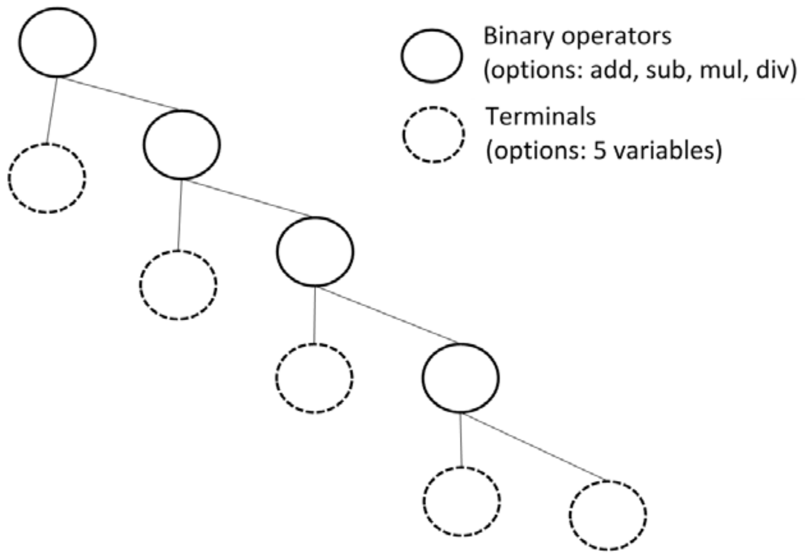


FIGURE 5.1: A tree representing an individual made up of binary functions and terminals.

Algorithm 2: The Fixed Length Initialisation (FLI)

```
/* Initialisation                                              */
$ln \leftarrow$ set length of individual;
$ps \leftarrow$set population size;
$pln \leftarrow 0$ ;/* empty population                        */
/* Create a population                                         */
while $\|pln\| < ps$ do
    newind $\leftarrow$ random individual of size $ln$ ;
    if $CheckUnique(newind, pln) = True$ then
        pln $\leftarrow$ pln + newind;
end
/* Function to check if an individual is unique;               */
/* constants are treated as same.                              */
Function CheckUnique($newind, pln$):
    $isUnique \longleftarrow True$;
    $nodesNew \longleftarrow$ list of nodes in $newind$ ;
    foreach $ind \in pln$ do
        $nodesInd \longleftarrow$ list of nodes in $ind$;
        $different \longleftarrow False$;
        for $i \leftarrow 1$ to $ln$ do
            if $nodesNew_i \neq nodesInd_i$ then
                if $(nodesNew_i \neq numeric) \vee (nodesInd_i \neq numeric)$ then
                    $different \longleftarrow True$;
                    continue to next $ind$;
        end
        if $different = False$ then
            $isUnique \longleftarrow False$; /* match found for $newind$ in $pln$ */
            return $isUnique$ ;
    end
return $isUnique$ ;
```

5.4 Experimental Setup

5.4.1 Test Problems

The experiments in this chapter use symbolic regression (SR) problems, which is a popular GP application. Unlike linear regression, which assumes that the relationship between the features is linear and thus only optimises the parameters of the features, symbolic regression (SR) makes no such assumptions and, instead, discovers both the structure and parameters of an appropriate model from data. SR uses the evolution-inspired process of GP to combine mathematical building blocks (such as operators, functions, variables, and constants) to make data models. Furthermore, because SR often produces large and nonlinear monolithic models, managing the complexity of these models is important. Therefore, SR is an appropriate testbed for studying complexity management in GP.

The experiments use five challenging regression problems for which GP produces models with low accuracy scores; the results in section 5.5 show that the accuracy scores of all of them are less than 28%. On challenging problems, GP will have to run for a long time to improve the fitness scores of the solutions; moreover, long runs are associated with growth in complexity. Therefore, the longer that GP runs, the more opportunity for bloating (growth in the size of the expression without a corresponding improvement in fitness). Bloat is a well-studied phenomenon for which numerous theories have been proposed. For example, a theory explains that small trees are likely to be simple and have relatively low fitness scores for challenging problems [81–83]. Consequently, the selection operator favours the large-sized (relatively more complex) individuals in the population, which results in further bloating; furthermore, large-sized individuals are more likely to produce bloated offspring [84–87]. Therefore, the challenging SR problems are suitable for experiments that manage complexity in GP.

In addition to choosing challenging problems, the recommendations in some benchmarking studies in GP [192, 193] were considered. These studies surveyed the GP community and reviewed current practices. Instead of producing a recommended list of benchmark problems, they collated an exclusion list of unreliable ones (that exist in GP literature) to avoid. Therefore, the choice for the experiments in this study avoids problems on the exclusion list. Furthermore, the selection criteria include the following: relevance to real applications and research in the GP field; variety in the class of problems; execution that is fast enough to allow for the large runs that GP requires; and ease to implement and reproduce without a need for specialised domain expertise.

Table 5.1 shows a summary of the benchmark problems (datasets). Four of these problems (problems 1 - 4) have five or more input variables and are available at [194]. Problem 5 is a bi-variate version of the function used in [195].

ID	Problem label	Number of Variables	Number of Instances
1	Airfoil	5	1503
2	Boston Housing	13	506
3	Concrete Strength	8	1030
4	Energy Efficiency	8	768
5	$y^2x^6 - 2.13y^4x^4 + y^6x^2$	2	250 (x=min:-0.3, step: 0.012; y=x + 0.03)

TABLE 5.1: Test problems for Explicit Time Control experiment.

5.4.2 Experimental Settings

This section presents the basic parameters in Table 5.2 and discusses other experimental decisions.

First, it is common for GP to evolve mathematical models that produce divide-by-zero errors upon evaluation. However, as recommended in [196] the experiments in this study simply kill off such models (individuals) by awarding them the worst possible fitness because *protecting* them via the so-called protected operators produces overfitting. Note, protected operators such as protected-division, protected-log, and protected-exponential detect when an incomputable value (e.g. NaN or infinity) is generated and replace this value with a default value. For example, protected-division avoids divide-by-zero errors by replacing the result of the offending division operation with just the value of its numerator or a constant value of 1.0. Despite the popularly cited and well-motivated warning as in [196] protected operators are still used as a legacy convention of GP since the 1990s; however, this study does not use them here. Second, the datasets are randomly split (without replacement) in an 80% and 20% ratio for training and testing, respectively. Third, the fitness score is computed as the normalised mean square error (NMSE) as follows:

$$\frac{1}{1 + \frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}. \tag{5.2}$$

Therefore, the experimental setup configures GP to maximise the fitness scores (prefer high values).

The experiments are run on a system with Windows 10 (64-bit), 32GB of RAM, and an Intel Quad-Core processor (Core i7-6700 CPU @ 3.40GHz).

Parameter	Setting
Number of runs	50
Population size	500
Run terminates	After 70 generations $\equiv$ 35,000 evaluations
Random tree generation	Fixed Length Initialisation (see Section 5.3.1)
Subtree generation	Ramped half-and-half ($1 =< depth =< 4$)
Operators & probabilities	One point crossover = 0.9; Point mutation = 0.1
Depth Limit	17
Function set	$+, -, *, /, \sin, \cos$, neg
Constants (ERC)	$\|ERC\|$ = 100 (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	Tournament of size = 3
Replacement	Steady-state/generational as per each method

TABLE 5.2: Experimental settings for Explicit Time Control.

5.5 Experimental Results

Before applying FLI to the experiment that compares time-control with size-control, its impact on the selected GP methods is examined. The aim, at this point, is to determine if introducing FLI will unduly favour or disadvantage the contending methods as they are used conventionally.

5.5.1 The Impact of the Fixed Length Initialisation

Section 5.3 motivated FLI, which initialises the population with functionally diverse individuals of the same size, to increase the focus of the time-control on differentiating functional complexity. To check how it affects the selected bloat-control techniques on their default, FLI is compared with the popularly used Ramped-Half-and-Half initialisation scheme; henceforth, the term Variable Length Initialisation (VLI) will refer to the Ramped-Half-and-Half.

The charts in Figure 5.2 and Figure 5.3 show the mean test fitness accuracy by generation for time-control with and without FLI and size-control with and without FLI. The figure shows that FLI generally improves the test fitness scores of both time-control and size-control; the results in Figure 5.4 are from analysing the final population.

Figure 5.4 details the result of the test for significance in the difference that FLI makes on both time-control and size-control; the test is Mann-Whitney U with an alpha of 0.05. The figure is colour coded so that green indicates where FLI makes a positive and significant difference, brown where the difference is negative and significant, and yellow where the difference is not significant. FLI produced significantly better results in 16 out of 20 for time-control and 11 out of 20 for size-control.

The results show that when using OpEq, size-control with VLI was better than size-control with FLI on all the test problems. This exception can be attributed to the mechanism OpEq uses. To improve accuracy, the OpEq bins the population of models by size; and then allows the bins with the higher average accuracy to produce more offspring. While OpEq with VLI (variable length) will naturally start with multiple bins, OpEq with FLI (fixed length) begins the evolution with only one bin. As a result, OpEq with FLI does not have the means to favour the more accurate solutions until later generations when the sizes of individuals begin to differ. However, when OpEq with FLI is used to

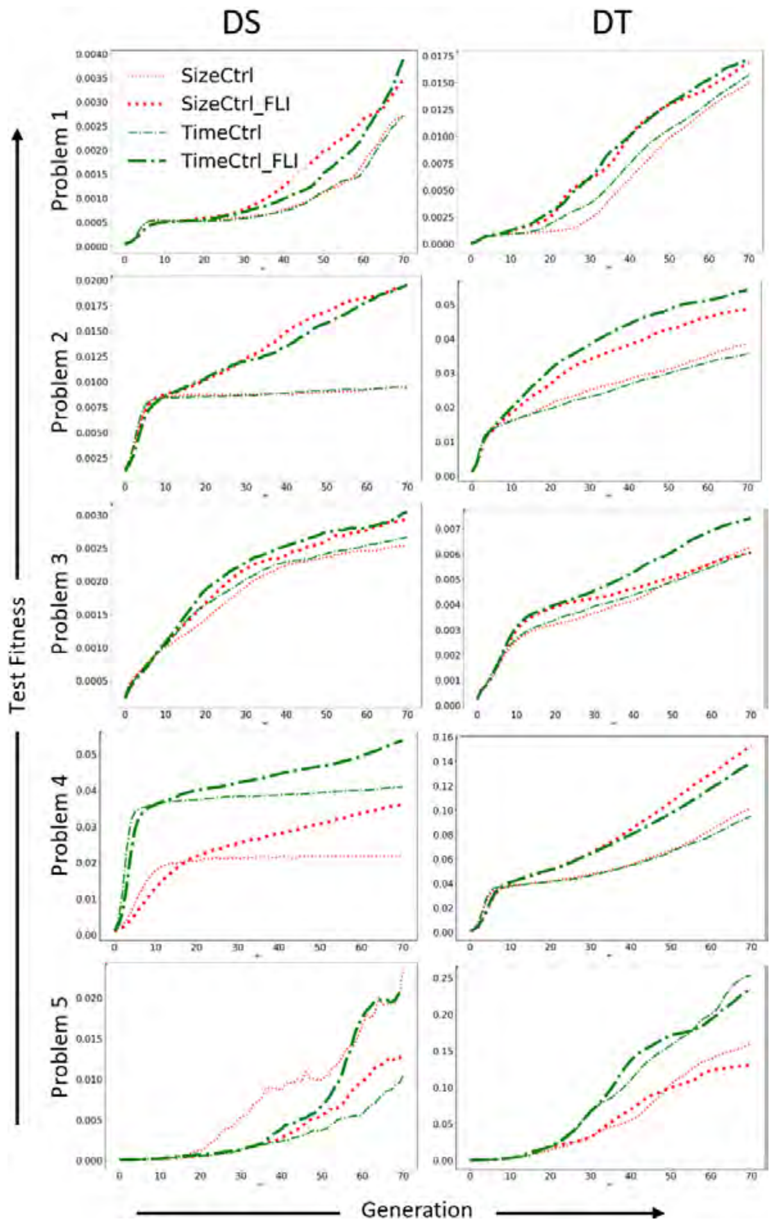


FIGURE 5.2: Test fitness scores of both size-control and time-control (using DS and DT) with and without FLI. The red and green lines represent size-control and time-control, respectively; the bold and thin lines indicate FLI and VLI, respectively

control time, the binning is done with time instead of size. Therefore, despite a uniform size in the population, their evaluation times will differ and enable OpEq with FLI to create multiple time-based bins, which allows it to favour the time-based bins that are more accurate. Consequently, the better performing

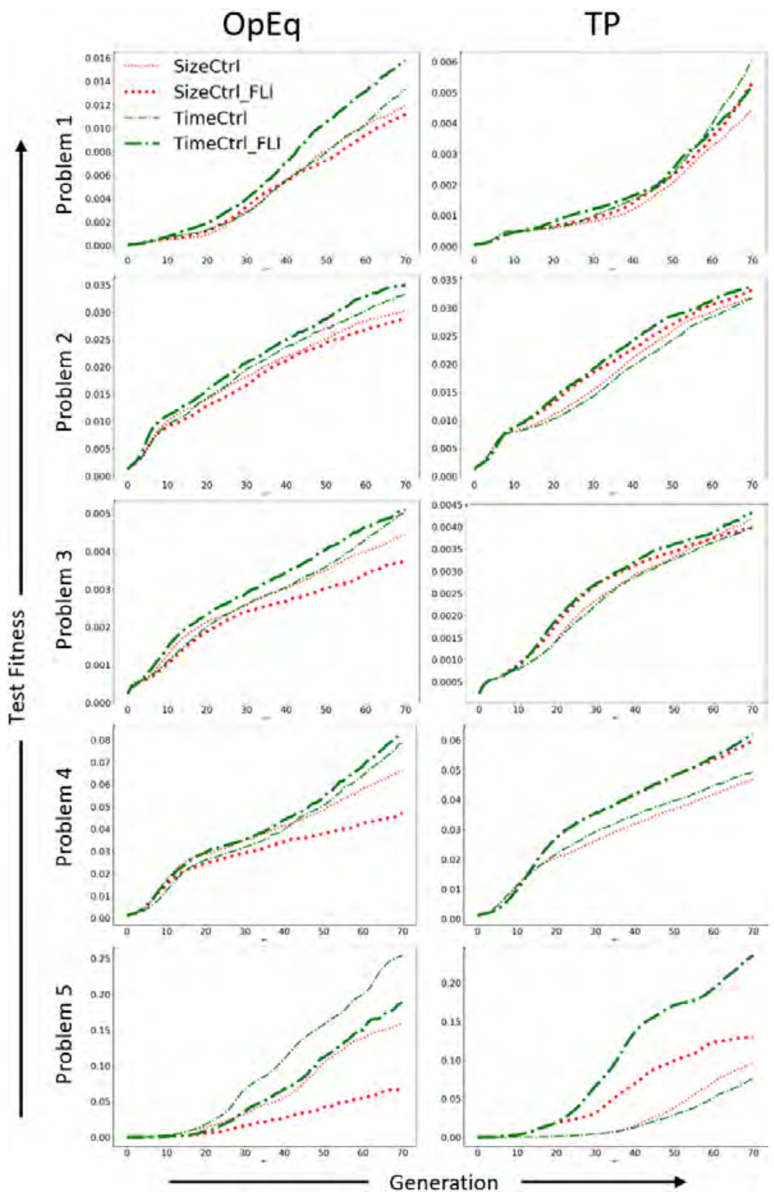
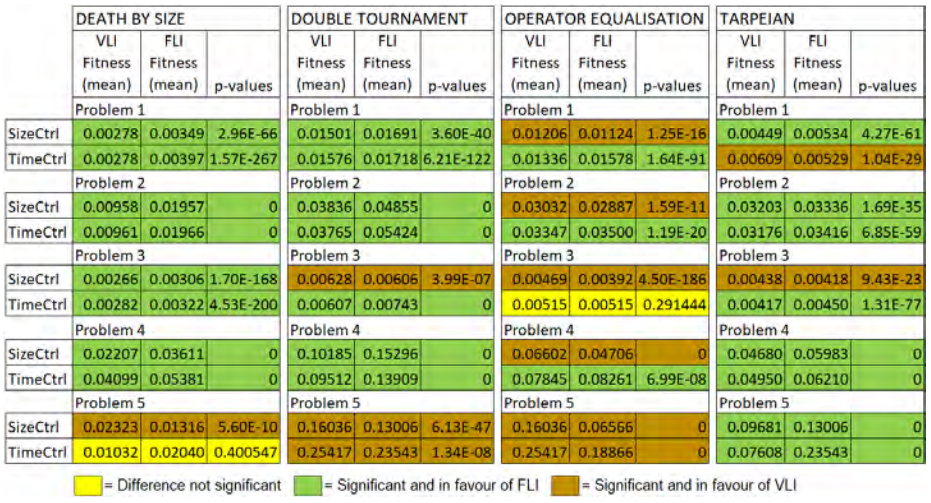


FIGURE 5.3: Test fitness scores of both size-control and time-control (using OpEq and TP) with and without FLI. The red and green lines represent size-control and time-control, respectively; the bold and thin lines indicate FLI and VLI, respectively

size control initialisations will be the benchmark: OpEq with VLI and all other methods with FLI.



	DEATH BY SIZE			DOUBLE TOURNAMENT			OPERATOR EQUALISATION			TARPEIAN		
	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values
	Problem 1			Problem 1			Problem 1			Problem 1		
SizeCtrl	0.00278	0.00349	2.96E-66 🟩	0.01501	0.01691	3.60E-40 🟩	0.01206	0.01124	1.25E-16 🟧	0.00449	0.00534	4.27E-61 🟩
TimeCtrl	0.00278	0.00397	1.57E-267 🟩	0.01576	0.01718	6.21E-122 🟩	0.01336	0.01578	1.64E-91 🟩	0.00609	0.00529	1.04E-29 🟧
	Problem 2			Problem 2			Problem 2			Problem 2		
SizeCtrl	0.00958	0.01957	0 🟩	0.03836	0.04855	0 🟩	0.03032	0.02887	1.59E-11 🟧	0.03203	0.03336	1.69E-35 🟩
TimeCtrl	0.00961	0.01966	0 🟩	0.03765	0.05424	0 🟩	0.03347	0.03500	1.19E-20 🟩	0.03176	0.03416	6.85E-59 🟩
	Problem 3			Problem 3			Problem 3			Problem 3		
SizeCtrl	0.00266	0.00306	1.70E-168 🟩	0.00628	0.00606	3.99E-07 🟧	0.00469	0.00392	4.50E-186 🟧	0.00438	0.00418	9.43E-23 🟧
TimeCtrl	0.00282	0.00322	4.53E-200 🟩	0.00607	0.00743	0 🟩	0.00515	0.00515	0.291444 🟨	0.00417	0.00450	1.31E-77 🟩
	Problem 4			Problem 4			Problem 4			Problem 4		
SizeCtrl	0.02207	0.03611	0 🟩	0.10185	0.15296	0 🟩	0.06602	0.04706	0 🟧	0.04680	0.05983	0 🟩
TimeCtrl	0.04099	0.05381	0 🟩	0.09512	0.13909	0 🟩	0.07845	0.08261	6.99E-08 🟩	0.04950	0.06210	0 🟩
	Problem 5			Problem 5			Problem 5			Problem 5		
SizeCtrl	0.02323	0.01316	5.60E-10 🟧	0.16036	0.13006	6.13E-47 🟧	0.16036	0.06566	0 🟧	0.09681	0.13006	0 🟩
TimeCtrl	0.01032	0.02040	0.400547 🟨	0.25417	0.23543	1.34E-08 🟧	0.25417	0.18866	0 🟧	0.07608	0.23543	0 🟩

🟨 = Difference not significant 🟩 = Significant and in favour of FLI 🟧 = Significant and in favour of VLI

FIGURE 5.4: Result of significance test for the difference FLI makes. The FLI test fitness accuracy improved 11 out of 20 for size-control and 16 out of 20 for time-control.

5.5.2 Comparing Time-Control and Size-Control

This section compares the effect of controlling complexity with evaluation time and controlling it with size (bloat-control). The bloat-control techniques described in Section 5.2 are used to compare the accuracy, complexity, and compositions of the models the two approaches produce. For accuracy, the test fitness (accuracy on out-of-sample data) is the key measure (high values preferred); however, the comparison also reports the training fitness scores. For complexity, the sizes and evaluation times of the models are compared (low values preferred). Finally, the comparison checks the composition of the final populations to determine the percentage of the genetic material comprised of more or less complex mathematical functions. This finding indicates that the evaluation time detects more than the sizes of the models.

Figures 5.5, 5.6, 5.7 and 5.8 (representing the results for DS, DT, OpEq and TP, respectively) compare the development of performance measurements by generation of time-control and size-control. A trend is observable where time-control generally gains accuracy (on all four methods) and simplicity (on DS and DT) over size-control. The test for significance of their differences uses their final populations; the charts show that all measures continuously increase through to the final generation and therefore is an appropriate common point.

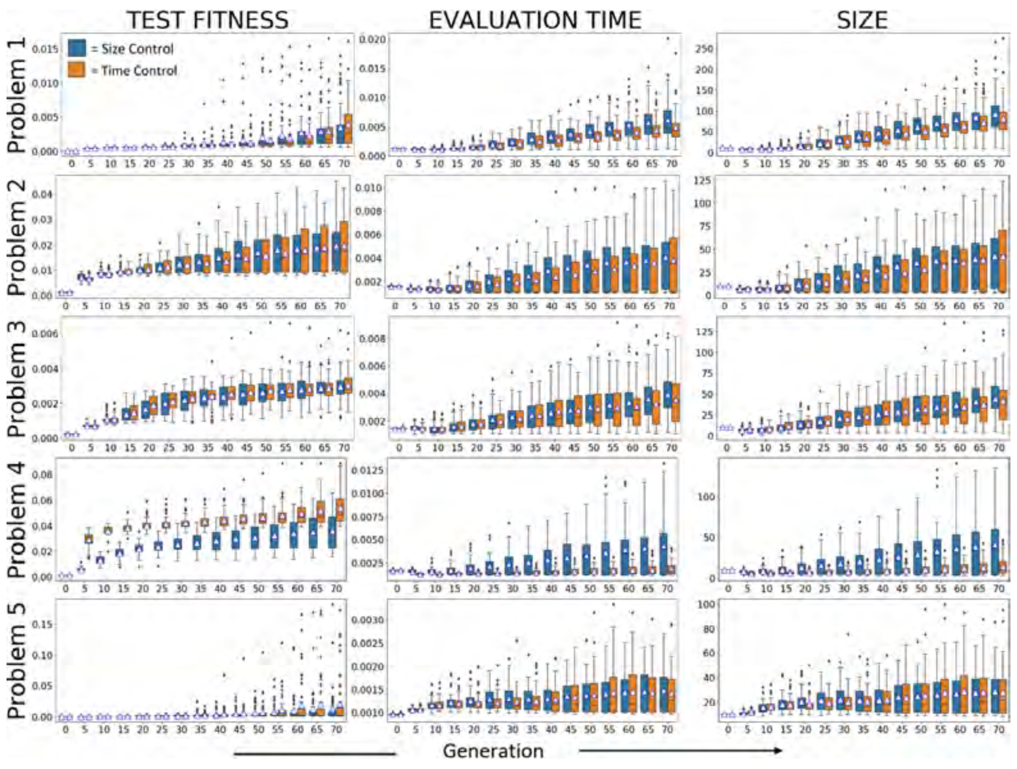


FIGURE 5.5: Comparison of size-control and time-control with Death By Size (DS).

Figure 5.9 shows the result of the significance test for the difference; also, unless stated otherwise, henceforth, the discussion about differences refers to this figure.

Difference in accuracy: Time-control produced significantly more accurate models (on both seen and unseen data) on all but 3 (out of 20) combinations of problems and control techniques; the difference is not significant on one of the three exceptions. The exceptions are Problem 1 on TP, where the difference is insignificant, and Problem 2 on DS and Problem 5 on TP, where size-control outperforms time-control. Overall, the result indicates that using evaluation time to control complexity produces significantly more accurate models than using size to control complexity.

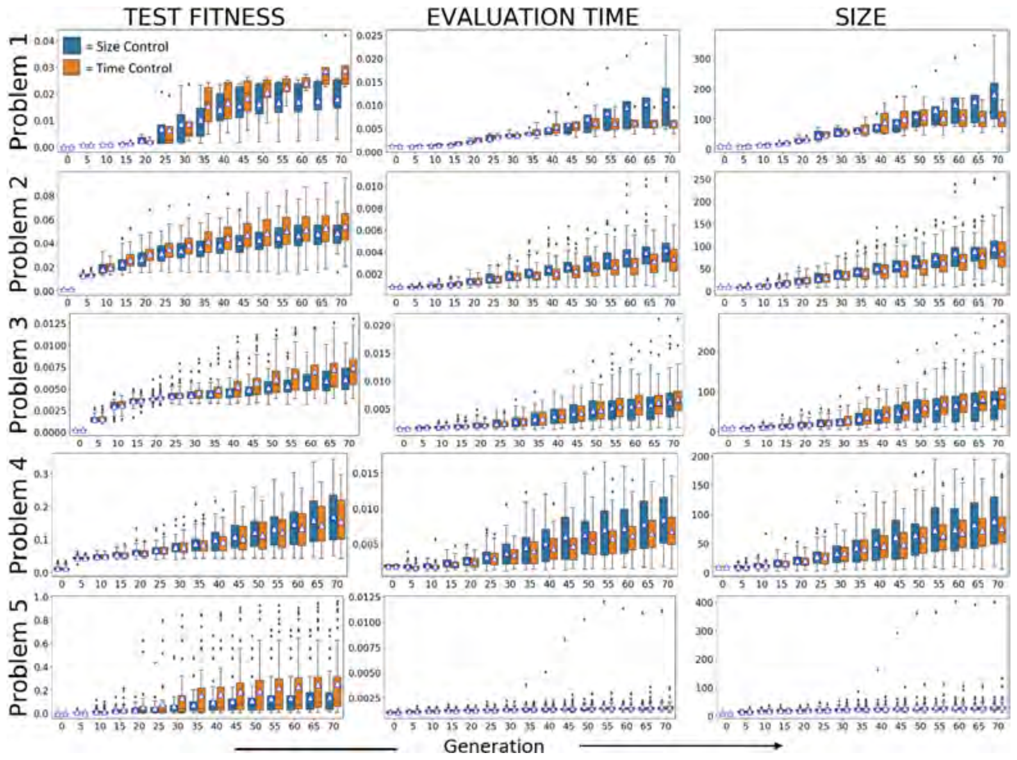


FIGURE 5.6: Comparison of size-control and time-control with Double Tournament (DT).

Difference in Complexity: Note, to compare complexity, the size of the expressions and their evaluation time is analysed. Therefore, for 5 test problems, 10 complexity results per complexity-control method are generated.

In terms of complexity control, the results appear to differ depending upon the replacement scheme employed by the time-control method. Time-control using steady state replacement with DS and DT created less complex models than size-control 9 out of 10 times each; however, time-control with generational replacement, that is, TP and OpEq created less complex models only 2 and 0 times respectively. While it is not clear as to why the complexity control of TP does not work well, the result in Section 5.5.1 indicates that OpEq behaves differently and does not benefit from FLI.

Although, this does not conclusively prove that steady state replacement is better suited for time-control, it does provide us with a benchmark for further

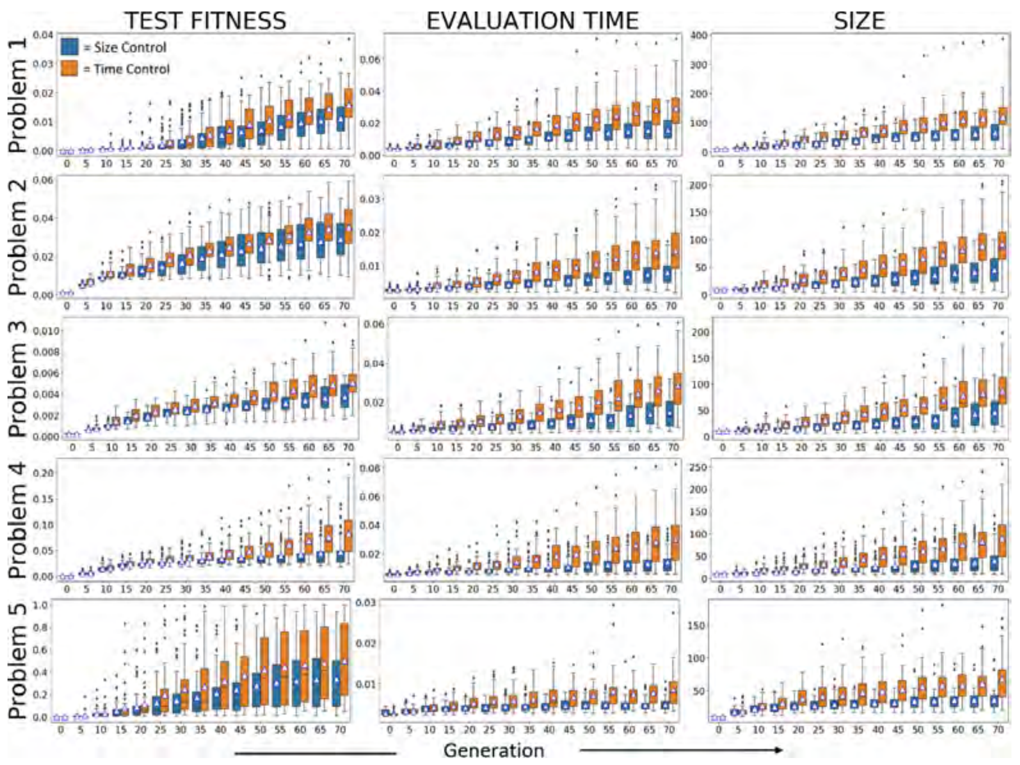


FIGURE 5.7: Comparison of size-control and time-control with Operator Equalisation (OpEq).

experiments in the thesis with time-control. Based on these results, from among the selection of complexity control techniques tried, DT and DS are better suited to time-control.

Difference in the Composition of Models: Table 5.3 shows the result of counting and differentiating the nature of nodes constituting the trees of the models in the final populations to understand the composition of the genetic material therein. Consistent with the complexity results (evaluation time and size), time-control with DS and DT used smaller percentages of complex mathematical functions (the COS and SIN operators) in the population they produced than the respective figures for size-control. Also consistent with their complexity results, TP and OpEq used higher percentages of complex mathematical functions in the population they produced than those by size-control.

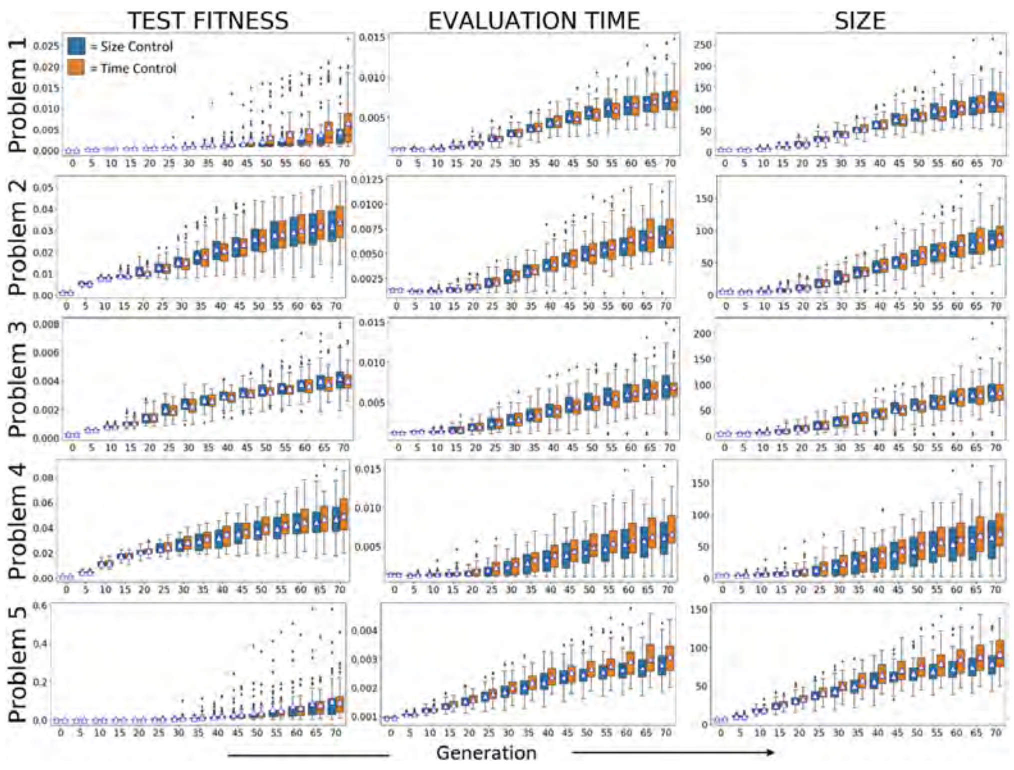
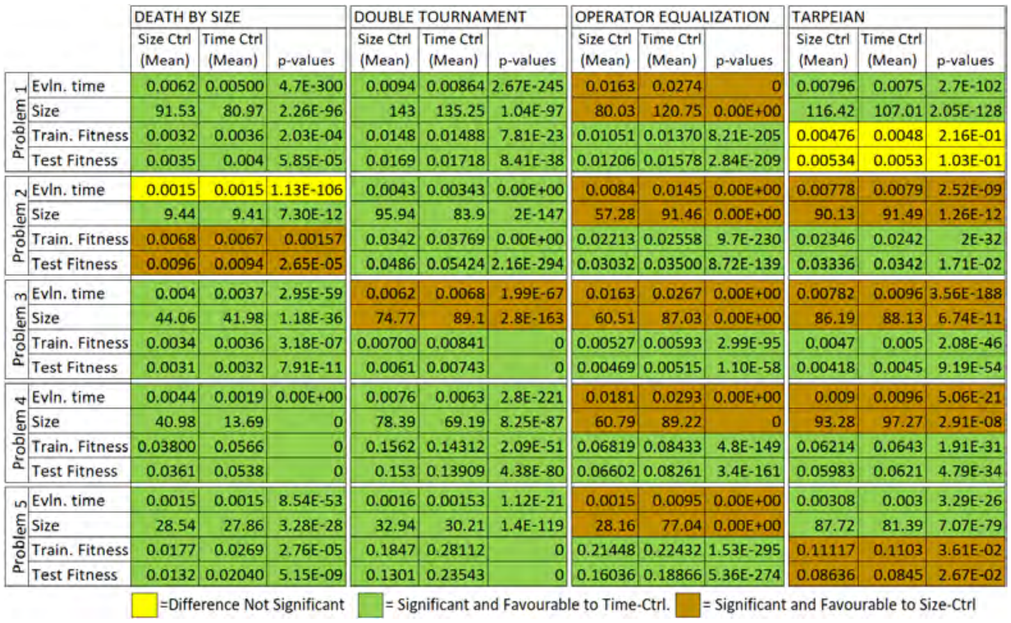


FIGURE 5.8: Comparison of size-control and time-control with The Tarpeian (Tp).

		Problem 1		Problem 2		Problem 3		Problem 4		Problem 5	
DT	Component Type	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS	19.15%	17.55%	17.67%	9.15%	10.38%	6.82%	18.90%	13.14%	8.02%	5.23%
	ADD & SUB	33.28%	34.27%	27.80%	30.17%	27.24%	29.72%	24.40%	27.94%	14.84%	12.11%
DS	Component Type	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS	15.29%	8.89%	11.25%	10.42%	9.78%	7.69%	16.88%	5.16%	6.66%	4.42%
	ADD & SUB	36.31%	37.95%	30.80%	28.77%	32.82%	36.03%	26.17%	30.77%	14.45%	14.55%
OpEq	Component Type	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS	11.95%	15.85%	12.06%	16.64%	9.42%	13.28%	18.26%	20.73%	9.38%	12.11%
	ADD & SUB	27.18%	24.98%	24.82%	23.23%	23.53%	21.71%	22.76%	21.46%	19.14%	18.04%
TP	Component Type	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS	14.05%	15.70%	17.29%	17.34%	15.81%	15.21%	18.07%	20.31%	8.47%	9.90%
	ADD & SUB	29.86%	30.46%	26.49%	25.01%	25.02%	24.03%	23.16%	23.29%	20.04%	19.26%

TABLE 5.3: Composition of the populations produced by Time-control and Size-control are compared.



		DEATH BY SIZE			DOUBLE TOURNAMENT			OPERATOR EQUALIZATION			TARPEIAN		
		Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values
Problem 1	Evln. time	0.0062	0.00500	🟩 4.7E-300	0.0094	0.00864	🟩 2.67E-245	0.0163	0.0274	🟧 0	0.00796	0.0075	🟩 2.7E-102
	Size	91.53	80.97	🟩 2.26E-96	143	135.25	🟩 1.04E-97	80.03	120.75	🟧 0.00E+00	116.42	107.01	🟩 2.05E-128
	Train. Fitness	0.0032	0.0036	🟩 2.03E-04	0.0148	0.01488	🟩 7.81E-23	0.01051	0.01370	🟩 8.21E-205	0.00476	0.0048	🟨 2.16E-01
	Test Fitness	0.0035	0.004	🟩 5.85E-05	0.0169	0.01718	🟩 8.41E-38	0.01206	0.01578	🟩 2.84E-209	0.00534	0.0053	🟨 1.03E-01
Problem 2	Evln. time	0.0015	0.0015	🟨 1.13E-106	0.0043	0.00343	🟩 0.00E+00	0.0084	0.0145	🟧 0.00E+00	0.00778	0.0079	🟧 2.52E-09
	Size	9.44	9.41	🟩 7.30E-12	95.94	83.9	🟩 2E-147	57.28	91.46	🟧 0.00E+00	90.13	91.49	🟧 1.26E-12
	Train. Fitness	0.0068	0.0067	🟧 0.00157	0.0342	0.03769	🟩 0.00E+00	0.02213	0.02558	🟩 9.7E-230	0.02346	0.0242	🟩 2E-32
	Test Fitness	0.0096	0.0094	🟧 2.65E-05	0.0486	0.05424	🟩 2.16E-294	0.03032	0.03500	🟩 8.72E-139	0.03336	0.0342	🟩 1.71E-02
Problem 3	Evln. time	0.004	0.0037	🟩 2.95E-59	0.0062	0.0068	🟧 1.99E-67	0.0163	0.0267	🟧 0.00E+00	0.00782	0.0096	🟧 3.56E-188
	Size	44.06	41.98	🟩 1.18E-36	74.77	89.1	🟧 2.8E-163	60.51	87.03	🟧 0.00E+00	86.19	88.13	🟧 6.74E-11
	Train. Fitness	0.0034	0.0036	🟩 3.18E-07	0.00700	0.00841	🟩 0	0.00527	0.00593	🟩 2.99E-95	0.0047	0.005	🟩 2.08E-46
	Test Fitness	0.0031	0.0032	🟩 7.91E-11	0.0061	0.00743	🟩 0	0.00469	0.00515	🟩 1.10E-58	0.00418	0.0045	🟩 9.19E-54
Problem 4	Evln. time	0.0044	0.0019	🟩 0.00E+00	0.0076	0.0063	🟩 2.8E-221	0.0181	0.0293	🟧 0.00E+00	0.009	0.0096	🟧 5.06E-21
	Size	40.98	13.69	🟩 0	78.39	69.19	🟩 8.25E-87	60.79	89.22	🟧 0	93.28	97.27	🟧 2.91E-08
	Train. Fitness	0.03800	0.0566	🟩 0	0.1562	0.14312	🟩 2.09E-51	0.06819	0.08433	🟩 4.8E-149	0.06214	0.0643	🟩 1.91E-31
	Test Fitness	0.0361	0.0538	🟩 0	0.153	0.13909	🟩 4.38E-80	0.06602	0.08261	🟩 3.4E-161	0.05983	0.0621	🟩 4.79E-34
Problem 5	Evln. time	0.0015	0.0015	🟩 8.54E-53	0.0016	0.00153	🟩 1.12E-21	0.0015	0.0095	🟧 0.00E+00	0.00308	0.003	🟩 3.29E-26
	Size	28.54	27.86	🟩 3.28E-28	32.94	30.21	🟩 1.4E-119	28.16	77.04	🟧 0.00E+00	87.72	81.39	🟩 7.07E-79
	Train. Fitness	0.0177	0.0269	🟩 2.76E-05	0.1847	0.28112	🟩 0	0.21448	0.22432	🟩 1.53E-295	0.11117	0.1103	🟧 3.61E-02
	Test Fitness	0.0132	0.02040	🟩 5.15E-09	0.1301	0.23543	🟩 0	0.16036	0.18866	🟩 5.36E-274	0.08636	0.0845	🟧 2.67E-02

🟨 =Difference Not Significant 🟩 = Significant and Favourable to Time-Ctrl. 🟧 = Significant and Favourable to Size-Ctrl

FIGURE 5.9: The result of the significance test for the difference between size-control and time-control. Time-control methods have significantly higher accuracy scores (on both training and test) in 17 out of 20 test cases. Whereas DS and DT on time-control produced simpler models than on size-control in 9 out of 10 test cases, TP and OpEq on time-control produced more complex models than size-control in 8 out of 10 test cases.

5.5.3 Discussion

The discussion in Section 5.1 argues that a sensible control of complexity should provide models that are only complex enough to explain the phenomenon that generated the given data but not too complex. As a result of a faithful learning about the phenomenon, such models can predict accurately on unseen data. The results in this chapter show that time-control delivers superior accuracy on unseen data than size control; it does this almost consistently despite producing more complex solutions using the methods with generational replacement (OpEq and TP). Still, the increased complexity with time-control on OpEq and TP is not as excessive as is typically the case with unrestrained GP.

Unlike the steady-state replacement methods where a new individual displaces the loser of a tournament, the generational replacement methods (OpEq and TP) compute the evaluation times of individuals and their distribution in the current generation before producing the next. However, as discussed in Section 5.5.1, the increased complexity of OpEq may be attributed to the impact of FLI on it.

To distinguish the functionally complex individuals, the experiments leverage a fixed-length initialisation (FLI) scheme, which keeps the individual sizes constant but promotes functional (compositional) diversity in the initial population. The expectation is that this would suit the time-control techniques; however, the results indicated that FLI improves all methods that use time-control and size-control; OpEq with size-control is the only exception.

The results support further research into this initialisation method. Also, as FLI only enforces the functional diversity in the initial generation, future work must study whether promoting the functional diversity in the population during the remaining evolution further enhances the effect of time control.

The techniques used to control size and time in this chapter, and the GP literature, explicitly penalise or discourage the measure of complexity. However, the next chapter shows that using time as a measure of complexity can also enable a more natural means of controlling complexity. Instead of explicit penalties, the evolutionary process can itself confer an advantage to the simpler solutions such that they participate in further evolution before their more complex (and slow evaluating) counterparts.

5.6 Conclusion

This chapter shows that using the evaluation time as a measure of complexity is viable and beneficial. Time-control almost consistently produces models with greater accuracy on both training and test data than size-control. In the instances where time-control produces slightly greater sizes or evaluation times, the corresponding greater accuracy compensates for the increases. Hence, the increases are not unwanted complexity; after all, the end goal is not to avoid complexity per se, but to have complexity that brings accuracy. Therefore, the result of this study and the idea that the evaluation time goes beyond the size of a model to reflect its computational complexity advocate for evaluation time as a better alternative to size.

This chapter also shows that time-control can distinguish more than size; it can differentiate functional complexity, particularly in a functionally diverse population. Therefore, the proposed Fixed Length Initialisation (FLI), which creates an identically sized but functionally diverse population, improved the time-control methods and most of those of size-control.

Overall, the results in this chapter back the evaluation time as a promising alternative to counting nodes in GP. Moreover, it promises to be a measure that can characterise complexity in many domains. Also, as indicated at the end of the last section, use of evaluation times opens up different opportunities for complexity control beyond merely replacing size with time in the bloat-control methods. The next chapter expands on one such opportunity.

Chapter 6

Implicit Control of Evaluation Time

"Everything should be made as simple as possible, but no simpler."
Albert Einstein

6.1 Introduction

All the techniques that were used to manage the complexity of models through controlling their size and evaluation times in the previous chapter – and those commonly found in GP literature – actively penalise complexity to various degrees. The degree to which a method penalises complexity is a design choice and acts as a parameter to the system.

The proposal in this chapter takes a radically different approach by taking the view that while a model is undergoing evaluation it is consuming valuable computational resources; therefore, a slow-evaluating model must not impede the progress of a fast-evaluating model that can evaluate much faster and thus help evolution continue. This is similar to the competition in product innovation where a product that takes longer to reach the market does not stop the competing products from taking a share of – or even taking over – the market.

Therefore, the onus is on the competing products to arrive faster in the market and there is no pressure *by design* on them to be ready before a certain time [1]. The products essentially *race* against each other to take over the market.

Following that analogy, this chapter incorporates a race in the evolutionary process that lets multiple models to *asynchronously* evaluate in parallel. Also, instead of penalising the computationally complex models by design, the process allows the fast evaluating models to join the breeding population (effectively our market here) as soon as they are ready, provided they pass a performance threshold. As a result, this process allows simple (fast evaluating) and accurate solutions to gain an evolutionary advantage. Aptly, the proposed process is called *Asynchronous Parallel Genetic Programming (APGP)*. Thus, unlike the complexity control employed explicitly by the techniques introduced in the previous chapter, the complexity control in this method is relatively implicit and naturally adapts to the given situation of a specific GP run.

Since the APGP lets multiple models evaluate in parallel with different finish times, this allows the simpler models who finish earlier to potentially join the breeding population in a *steady-state*[2] fashion earlier than the complex ones. Once such an individual enters the breeding population, it may be selected to breed while its counterparts are still evaluating. This behaviour is natural as well; after all, breeding is not synchronised in nature. Although unnatural, synchronised breeding has been the norm in evolutionary algorithms. Therefore, the APGP borrows an extra leaf from nature.

As stated already, unlike the time control methods discussed in Chapter 5, the APGP does not explicitly exclude or penalise the complex and slowly evaluated individuals; the sole criterion for joining the population is still the fitness

[1]except where the product becomes useless after a certain deadline.

[2]The steady-state replacement scheme allows GP to update the breeding population with new individuals one at a time instead of in batches.

of the new offspring – every new offspring must be fitter than an individual selected (via an inverse tournament) from the existing population. However, evaluation speed (a proxy for simplicity) is still an advantage because that means the fast-evaluating offspring only competes with the older (and generally less fit and simpler) models that exist in the present population. In contrast, the task for the late-comers is harder because, by the time they arrive, the population may have got fitter because of all those that joined the population earlier. However, the simpler pre-existing models can survive only until a more accurate yet complex model arrives. In fact, the validation of APGP presented later in Section 6.4 and its analysis in Chapter 7 confirm that APGP does line up the simpler solutions for an opportunity to join the population; however, whether they actually make it into the population depends on their fitness. Thus, a dynamic interplay between simplicity and accuracy happens continuously during the evolution in the APGP. Therefore, unlike the explicit time control methods, the complexity control in the APGP may be *gentler* due to the absence of explicit, time-based penalties.

A set of experiments in this chapter compares the APGP with standard GP (GP) and GP with a very effective bloat-control mechanism (GP+BC). Another set compares the APGP with the time control methods introduced in Chapter 5; henceforth, those methods are called *explicit* time control methods. A further set of experiments checks the impact of the Fixed Length Initialisation scheme (FLI) on APGP and the other methods; FLI was introduced in Chapter 5 as a means to create an initial population of individuals that are functionally diverse to encourage the evaluation time to distinguish functional complexity.

The results show that the APGP produces the most accurate models (on both training and test data) that are also simple; moreover, it trains the fastest in that it uses the fewest evaluations to match a target accuracy (the average that GP achieves), whereas GP+BC takes the most. Against the explicit time control methods, APGP still produces the most accurate solutions with a gentler complexity control. In addition, like other methods, the accuracy of APGP solutions improves with FLI. Overall, the results suggest that the complexity control in APGP is a viable method to balance the accuracy-simplicity dilemma. Thus, APGP adds to a family of algorithms that are possible due to adopting evaluation time as a proxy for measuring complexity.

The rest of this chapter is organised as follows: Section 6.2 discusses the works that are related to the APGP; Section 6.3 details the workings of the APGP; Section 8.4 details the experimental setup; Section 6.6 presents the results; and finally, Section 6.7 concludes this chapter.

6.2 Parallelism in Genetic Programming

Since the proposed APGP leverages parallel computing, this section first reviews parallel computing in GP and contrasts its use in existing GP literature with the objectives of this study.

Parallel computing in GP [197–199] is typically used to improve the run times [200], as opposed to reducing the complexity of individual models, which is the target of this study. Reducing the overall runtime is prudent because GP runs can take a long time to complete. Most commonly, generational replacement schemes – that evaluate the entire set of offspring produced by the present population to create the next population – parallelise the evaluation of the offspring. However, the generational replacement requires the *entire* offspring population to be ready before the next set of breed operations can proceed; this means that all evaluation threads join at a single *point of synchronisation* before the evolution proceeds further. Therefore, this parallelisation gives no advantage to simpler (faster) individuals. Moreover, this approach is inefficient in terms of resource utilisation - while a complex individual is taking an excessive amount of time to evaluate on a CPU, the remaining CPUs stay idle after the other faster (less complex) individuals have completed evaluation.

Some evolutionary algorithms (EA) have used asynchronous parallel computing in non-GP setups to alleviate the idle CPU time challenge. The examples [198, 201] observed an evaluation time bias that favours individuals with shorter evaluation times. However, these studies are concerned with optimising parameters using fixed-length chromosomes; therefore, they do not study the impact of asynchronous parallelism on the functional complexity of variable-length structures in GP.

EAs also use parallel computing in the so-called *island model* [202, 203], where the evolving populations divide into multiple distinct islands, and the *sub-population* in each island can be evolved in a separate parallel thread. Through this parallelisation, the island model offers advantages such as greater diversity in the overall population; the diversity arises because each island remains isolated from other islands except at discrete intervals when selected individuals are exchanged across the islands. This work does not use island models; instead, this thesis uses a single panmictic population.

6.3 Asynchronous Parallel Genetic Programming (APGP)

The APGP algorithm lets GP individuals evaluate asynchronously to allow simple individuals (that evaluate quickly) to get into the breeding population earlier than their more complex (slow evaluating) counterparts. This practice is natural in natural evolution, where individuals of a population continue breeding while one of them is still testing (evaluating) against the environment. Yet, the contrary is precisely what happens in traditional GP; while the GP individual (of any complexity) is testing against its natural environment, the evolution waits regardless of how long it takes. Thus, traditional GP does not confer any advantage to fast evaluation (and potentially low complexity). However, APGP strives to leverage speed (quick evaluation) as an advantage to breed simple yet accurate models.

APGP works with the *Steady State* [77] replacement scheme, which allows an offspring to compete for a place in the breeding population as soon as it completes its evaluation. This scheme enables APGP to allow a set number of multiple breeding operations and fitness evaluations to be executed in parallel and asynchronously. For example, configure APGP to run 50 of these breed operations (followed immediately by the corresponding fitness evaluation; note the setup here produces one offspring per breed operation)[3] simultaneously and asynchronously in the same population. As soon as one of the 50 operations finishes, another starts; this keeps the total number of simultaneous operations fixed. Still, the evaluations may complete at different times due to the varying times taken to evaluate models that differ by their make-up; since the evaluation of models is done the same way (using the same dataset).

Section 6.4 details the experiments carried out to validate the race condition in APGP. The results confirm that the simpler models, having shorter evaluation times, finish before the more complex counterparts. Therefore, when a batch or a stream of evaluation tasks runs asynchronously, the simpler models often overtake the more complex ones.

Thus, a race condition establishes in the evolutionary process of APGP such that less complex individuals can *contest* for a place in the breeding population before their slower counterparts. Contesting for a place in the population means checking if the new model is fitter than the "winner"[4] of an inverse tournament and if so, replacing that winner with the new model. Hence, if the fast evaluating individuals enter the breeding population, they may reproduce earlier than the more complex individuals taking longer to evaluate.

[3]50 produces reasonable results; however, later Chapter 7 analyses the impact of varying the number of parallel threads.

[4]The *winner* of an inverse tournament is the worst member; the experiment in this chapter uses a tournament size of 5.

6.3.1 The APGP Algorithm

Algorithm 3: Asynchronous Parallel Genetic Programming (APGP) Algorithm

```
/* Initialise                                                    */
$N \leftarrow$ set total number of offspring to produce;
$threadpool \leftarrow$ set number of concurrent operations allowed;
$popsize \leftarrow$ set population size;
/* Generate and evaluate initial population                      */
$population \leftarrow$ generate initial population of size $popsize$;
Evaluate($population$);
/* Generate and evaluate offspring in parallel                   */
$count = 0$; while $count < N$ do
    if $threadpool > 0$ then
        Thread($threadpool \leftarrow threadpool - 1$ &&
        $offspring \leftarrow$ Breed_and_evaluate() );
        $count \leftarrow count + 1$;
        if offspring_evaluated then
            $replace \leftarrow$ To_Replace($population$);
            if $fitness(offspring) > fitness(population[replace])$ then
                Lock $population[replace]$ memory position;
                $population[replace] \leftarrow offspring$;
                Release locked position;
            else
                Discard $offspring$;
            Release thread: $threadpool \leftarrow threadpool + 1$;
    else
        Wait
end
```

The APGP algorithm is outlined in the pseudo-code in **Algorithm** 3. It begins by setting the number of concurrent breed operations (evaluations) allowed, the population size, and the total number of offspring to produce (total number of fitness evaluations in the run). Next, the initial population is produced and evaluated. Then, the parallel breeding begins by launching multiple breed operations up to the allowed limit. A breed operation evaluates its offspring within its independent thread. As soon as the evaluation of an offspring completes, it tries to get into the breeding population; it replaces the winner of an inverse tournament in the current population if it is fitter than the winner. Immediately after that, the corresponding resources (for the thread) are freed, and a new breed operation commences. As these parallel operations work on the same breeding population, a temporary lock is necessary on the memory position of the individual that is being replaced; this prevents clashes that may occur from multiple threads trying to replace the same individual.

As discussed earlier, the criterion when an offspring tries to find a place in the population is its accuracy only. Therefore, for an offspring to take advantage of its speed, its accuracy needs to be competitive. When the more complex candidates become more accurate, they will eventually get in, get selected and propagate. As such, APGP does not exclude the more complex individuals from the contest. Therefore, the possibilities within a specific problem will determine how simple the accurate models can be.

Before evaluating the APGP on various problems (in Section 6.5) as proof of concept, the following section first validates that a race condition indeed exists in the experimental settings used; a race condition – where simple solutions finish evaluating before their more complex counterparts – is essential in the APGP.

6.4 Validating the Race in APGP

In order to demonstrate that APGP can give an advantage to simpler individuals, this section simply validates as to whether the simpler individuals do finish evaluation earlier. Thus, here the objective is not to trace the end results of evolution but only to practically demonstrate that the simpler individuals do get a time-advantage that can give them an opportunity to join the breeding population earlier.

6.4.1 Experimental Setup

To ascertain as to whether fast-evaluating models actually finish evaluations earlier than their slower counterparts, evaluations tasks that take verifiably different times to complete are designed. Thus, the test in this section creates ten types of evaluation tasks of varying complexity; they all involve executing an identical model but on varying dataset sizes. The dataset sizes ranged from 10,000 to 190,000 in steps of 20,000 (this makes 10 different tasks). Since increasing dataset sizes increases the computational load, this setup can check if a parallel race indeed suits the computationally cheaper tasks.

Two thousand such tasks are sent to evaluate in parallel and asynchronously (a race to completion) while tracking the order they started and the order they finished. Also tracked are their evaluation times and complexity (as designed). Then, using the collected data, different stages of the race are analysed; for example, from the beginning to the 100th, to the 200th, and so forth.

6.4.2 Results

The results show that the race condition (that APGP also uses, though APGP was not used here) allows simpler evaluation tasks to finish earlier. FIGURE 6.1 shows periodical snapshots of the completed evaluations by the dataset sizes and FIGURE 6.2 shows them by their evaluation times; the charts result from using 50 threads. At the earliest stage of the race (as illustrated by the graph at the top of the figure, which corresponds to the first 100 completed evaluations), a high number of simple evaluations have been completed; thus, the peak is around dataset size of 25,000 data points. At the later stages of the race (bottom chart in the figure), the spread increasingly evens out (by dataset size and evaluation time, respectively) in both FIGURES 6.1 and 6.2; this shows that the slower (complex) evaluations do also complete as time goes on.

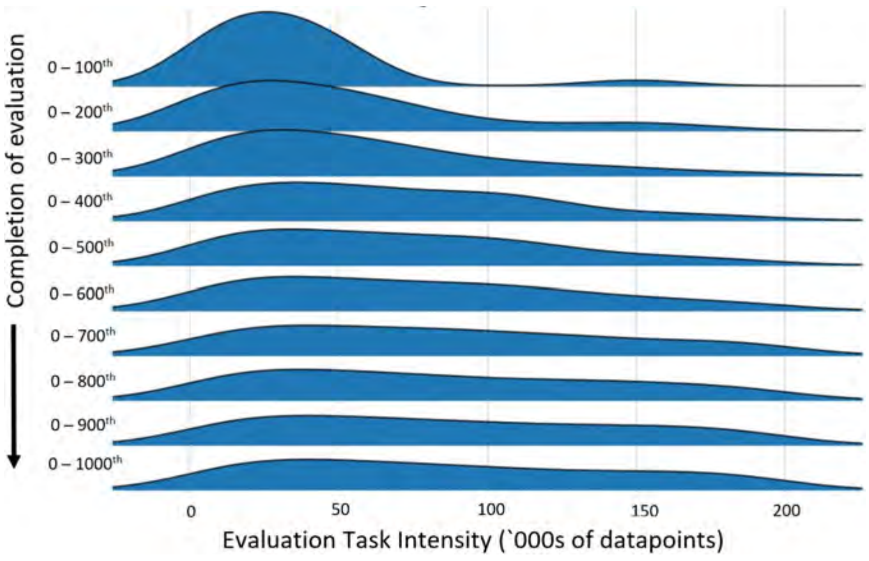


FIGURE 6.1: The race enables the tasks with smaller dataset sizes to complete earlier than those with larger datasets. Note, x-axis shows the dataset sizes in thousands. Y-axis shows the number of completed evaluations; thus, the distribution of the first 100 completions is at the top.

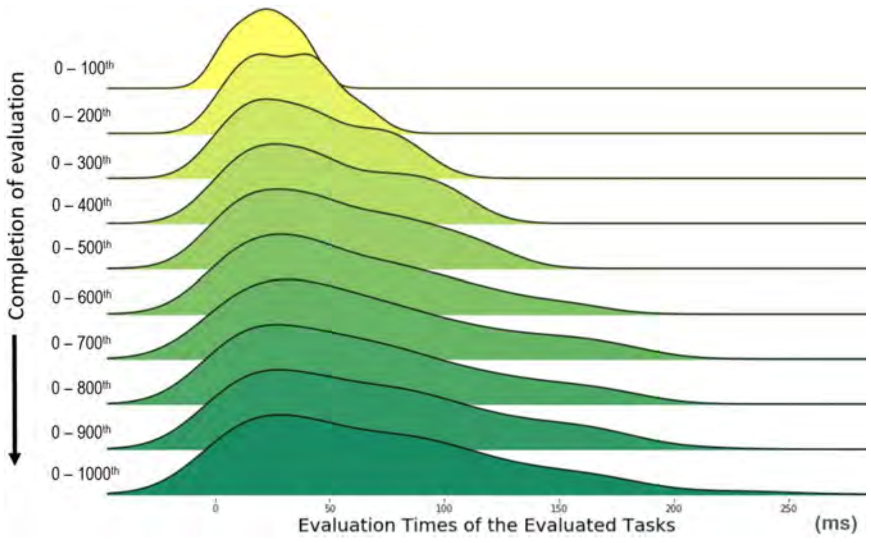


FIGURE 6.2: The race enables tasks with smaller evaluation times to complete earlier than those with larger evaluation times.

Since factors (such as accuracy scores) beyond the speed of evaluation determine the result of the APGP, the race condition is validated separately. The results demonstrate that simpler individuals finish earlier than their more complex counterparts. Thus, the method for inducing the race condition in APGP is validated. While the validation test in this section used a set degree of concurrency (50 threads), Chapter 7 examines the effect of varying it.

6.5 Proof of Concept: Experimental comparison of APGP with Other Methods

Using the selected suite of symbolic regression problems, three sets of experiments are run in this chapter, as follows:

1. APGP is compared with GP with bloat-control (GP+BC) and standard GP (GP). This serves as a proof of concept for APGP and further compares bloat-control with time control.

2. APGP is compared with the time control methods that were introduced in Chapter 5. This is done to compare the implicit time-control technique that APGP offers with the explicit time control techniques from Chapter 5. The explicit time control methods are:

    - *Death by Size* (DS) [188], which raises the likelihood of displacing large-sized individuals from the breeding population.
    - *Double Tournament* (DT) [188, 189], which increases the likelihood of producing simple offspring by raising the probability of selecting simple parents to breed.

- *Operator Equalisation* (OpEq) [164, 190], which prevents unnecessary growth in size by controlling the distribution of size in the population to increase opportunities for more accurate sizes to produce more offspring.
- *The Tarpeian* (TP) [191], which penalising a portion of the large-sized individuals in the population and makes them noncompetitive; thus, it discourages growth in size.

Like the experiments in Chapter 5, size is replaced by time in these methods to effect time-control.

3. APGP with the Fixed Length Initialisation (APGP-FLI) is compared with APGP without FLI. This experiment examines the behaviour of the APGP in a functionally diverse environment, which the FLI facilitates.

6.5.1 Test Problems

The experiments in this chapter use six test problems: five multidimensional problems and a bi-dimensional one. The data and description for Problems 1 - 5 are available at [194]; Problem 6 is a bivariate variant of the mathematical function used in [195]. As shown in the summary of the problems in Table 6.1, all but Problem 4 are the same carefully selected problems from Chapter 5; the criteria for the selection are discussed in Section 5.4.1.

ID	Problem label	No. of variables	No. of instances
1	Airfoil	5	1503
2	Boston Housing	13	506
3	Concrete Strength	8	1030
4	Dow Chemical	57	1066
5	Energy Efficiency	8	768
6	$y^2x^6 - 2.13y^4x^4 + y^6x^2$	2	250 (x= min: -0.3, step: 0.012; y = x + 0.03)

TABLE 6.1: Test problems for Implicit Time Control Experiment.

6.5.2 Experiment Settings

Table 6.2 provides the basic GP settings the experiments use. Other configurations and parameters are as follows:

- *Bloat-control for GP+BC* was implemented with the *Double tournament* [188], which has been effective on a variety of benchmark problems [188][189]. Furthermore, the results in Chapter 5 show that this method restricts the sizes of the models aggressively while producing accurate solutions.
- *Degree of Concurrency:* For the experiments in the chapter, APGP uses 50 parallel threads; however, Chapter 7 investigates other settings.

Some of the settings are the same as Chapter 5 (Section 5.4.2); these include the system configuration, handling of divide-by-zero errors, dataset splitting, and the fitness function.

Parameter	Setting
Number of runs	50
Population size	500
Run terminates	After 35,000 evaluations ($\equiv$ 70 generations)
Random tree/ subtree generation	Ramped half-and-half (depth $min = 1, max = 4$)
Tree depth limit	17
Operators & probabilities	One point crossover = 0.9 Point mutation $= 0.1$
Function set	$+, -, *, /,$ sin, cos, neg
Constants	Ephemeral random constants (ERC) $\lvert ERC \rvert = 100$ (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament size = 3
Replacement	steady state, inverse tournament size = 5

TABLE 6.2: Experimental settings for Implicit Time Control.

6.6 Experimental Results

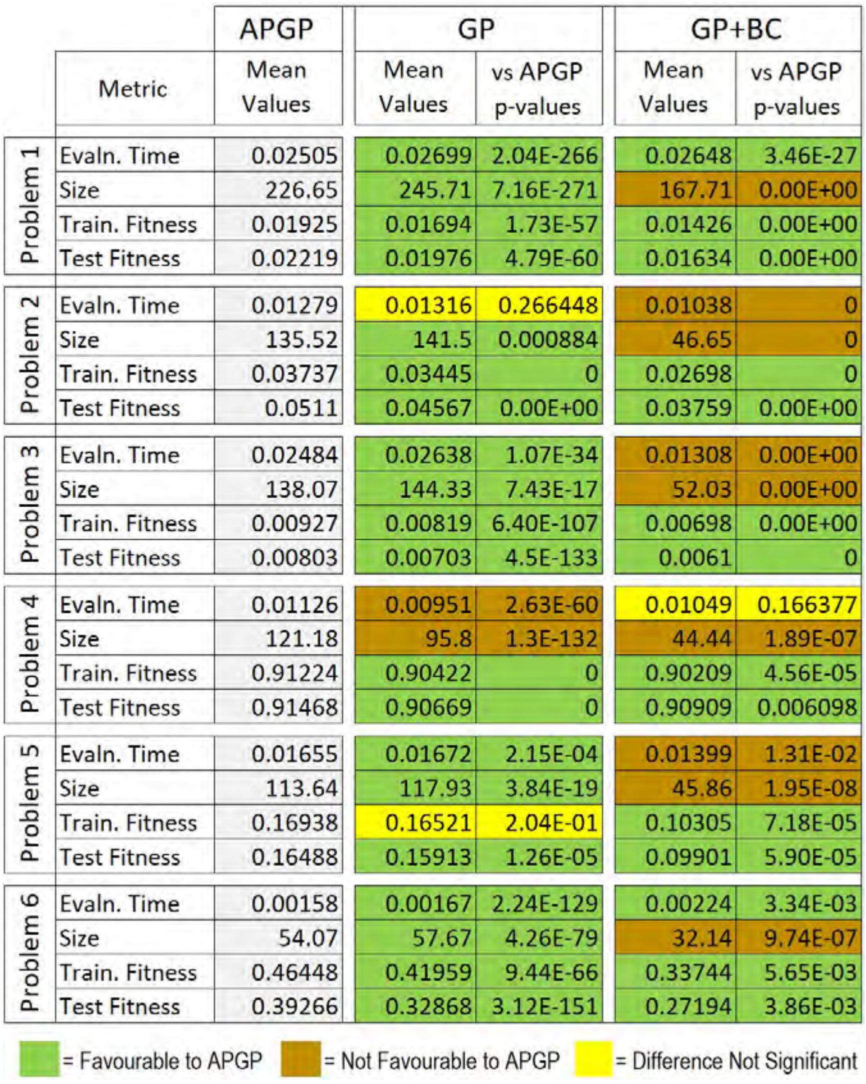
6.6.1 APGP compared with GP and GP+BC

The experiments enable comparing the accuracy (on training and test data) and the simplicity of the models that APGP produced with those of the contending methods. Furthermore, the number of evaluations the compared methods perform to reach a target accuracy (the average accuracy that standard GP achieves) are analysed to determine the efficiency of the APGP and GP+BC relative to GP.

Accuracy and Complexity Control

As the fitness in the experiments in this thesis is a maximisation function (where a higher score is preferred), the terms *fitness* and *accuracy* are interchangeable here. Figure 6.3 shows a colour-coded table of results of the Mann-Whitney U statistical test on the final populations of APGP against those of GP and GP+BC. The table includes the mean evaluation times, sizes, fitness scores (training and test); also, the table shows the p-values of each test. The green coloured cells show instances where APGP is significantly better (i.e., higher accuracy scores or lower complexity) than GP or GP+BC and brown where it is significantly worse; the yellow cells indicate an insignificant difference.

The dominance of the green coloured cells in Figure 6.3 indicates that APGP is significantly better generally. This dominance by APGP is greater against GP. In terms of accuracy (training and test fitness scores), the APGP scores are significantly better than those of GP and GP+BC in all tests, except for the insignificant difference (against GP) seen in the training fitness scores on Problem 5. Moreover, as the APGP test fitness values are consistently and significantly better than those of GP and GP+BC, this indicates that APGP models tend to



		APGP	GP		GP+BC	
	Metric	Mean Values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values
Problem 1	Evaln. Time	0.02505	0.02699	2.04E-266	0.02648	3.46E-27
	Size	226.65	245.71	7.16E-271	167.71	0.00E+00
	Train. Fitness	0.01925	0.01694	1.73E-57	0.01426	0.00E+00
	Test Fitness	0.02219	0.01976	4.79E-60	0.01634	0.00E+00
Problem 2	Evaln. Time	0.01279	0.01316	0.266448	0.01038	0
	Size	135.52	141.5	0.000884	46.65	0
	Train. Fitness	0.03737	0.03445	0	0.02698	0
	Test Fitness	0.0511	0.04567	0.00E+00	0.03759	0.00E+00
Problem 3	Evaln. Time	0.02484	0.02638	1.07E-34	0.01308	0.00E+00
	Size	138.07	144.33	7.43E-17	52.03	0.00E+00
	Train. Fitness	0.00927	0.00819	6.40E-107	0.00698	0.00E+00
	Test Fitness	0.00803	0.00703	4.5E-133	0.0061	0
Problem 4	Evaln. Time	0.01126	0.00951	2.63E-60	0.01049	0.166377
	Size	121.18	95.8	1.3E-132	44.44	1.89E-07
	Train. Fitness	0.91224	0.90422	0	0.90209	4.56E-05
	Test Fitness	0.91468	0.90669	0	0.90909	0.006098
Problem 5	Evaln. Time	0.01655	0.01672	2.15E-04	0.01399	1.31E-02
	Size	113.64	117.93	3.84E-19	45.86	1.95E-08
	Train. Fitness	0.16938	0.16521	2.04E-01	0.10305	7.18E-05
	Test Fitness	0.16488	0.15913	1.26E-05	0.09901	5.90E-05
Problem 6	Evaln. Time	0.00158	0.00167	2.24E-129	0.00224	3.34E-03
	Size	54.07	57.67	4.26E-79	32.14	9.74E-07
	Train. Fitness	0.46448	0.41959	9.44E-66	0.33744	5.65E-03
	Test Fitness	0.39266	0.32868	3.12E-151	0.27194	3.86E-03

🟩 = Favourable to APGP 🟧 = Not Favourable to APGP 🟨 = Difference Not Significant

FIGURE 6.3: Result of significance test in the differences between the APGP populations and those of GP and GP+BC. The APGP models are more accurate (on both training and test data) than those of GP and GP+BC in all tests. Also, APGP models are simpler than GP. However, APGP models showed a greater complexity than GP+BC that is associated with higher accuracy.

generalise better; test fitness score is the main concern because it reflects generalisation. Moreover, the reported p-values are very small, which indicates very significant differences.

In terms of managing complexity, APGP produced significantly simpler models than GP but more complex than GP+BC. The APGP models were significantly simpler than those of GP in all but Problem 4. Although APGP produced more complex individuals, in this exception, it produced models that have greater accuracy scores. Therefore, the increase in complexity comes with an associated gain in fitness; consequently, the greater sizes and evaluation times do not necessarily indicate bloat – bloat is a growth in size without a correlated improvement in accuracy. However, APGP did not produce simpler solutions than GP+BC in most cases. Instead, GP+BC generated significantly smaller sized models in 6 out of 6 tests and shorter evaluation times in 3 out of 6 tests; note, GP+BC aggressively and solely targets the model size. But, the simplicity of models that GP+BC achieved is associated with significantly lower training and test accuracy values. In other words, simplicity in GP+BC comes at the expense of accuracy.

Training Efficiency of the Methods

As the APGP allows simple and accurate models to push through, the experiment examines if APGP affects the overall training speed and how that compares with the effect of bloat-control. To that end, the number of evaluations it takes for each run of the methods to meet a set target training fitness – the average training score that GP usually achieves. Runs that do not reach the target are assigned the maximum number of budgeted evaluations.

As detailed in Table 6.3, APGP used between 10% to 40% fewer evaluations than GP, and 15% to 84% fewer than GP+BC to meet the same target training scores. Therefore, APGP is the fastest to train. Note, GP+BC is the slowest to train despite producing smaller individuals. These observations tally with the consistently less accuracy that GP and GP+BC models show against APGP models in Figure 6.3.

Problem ID	GP Mean	GP+BC Mean	APGP Mean	Difference APGP v. GP	Difference APGP v. GP+BC
1	29407	33041	23941	18.59 %	38.01 %
2	20761	27792	17762	14.44 %	56.47 %
3	31668	33595	20422	35.51 %	64.50 %
4	17861	19762	10719	39.99 %	84.36 %
5	28852	34842	25658	11.07 %	35.79%%
6	33077	33895	29546	10.68 %	14.72 %

TABLE 6.3: The relative training speeds of the methods as determined by the number of evaluations needed to meet a set training accuracy target. Columns 2 - 4 show the average number of evaluations the methods used. Column 5 - 6 show the percentage differences of APGP relative to GP and GP+BC, respectively. APGP uses significantly fewer evaluations than the others.

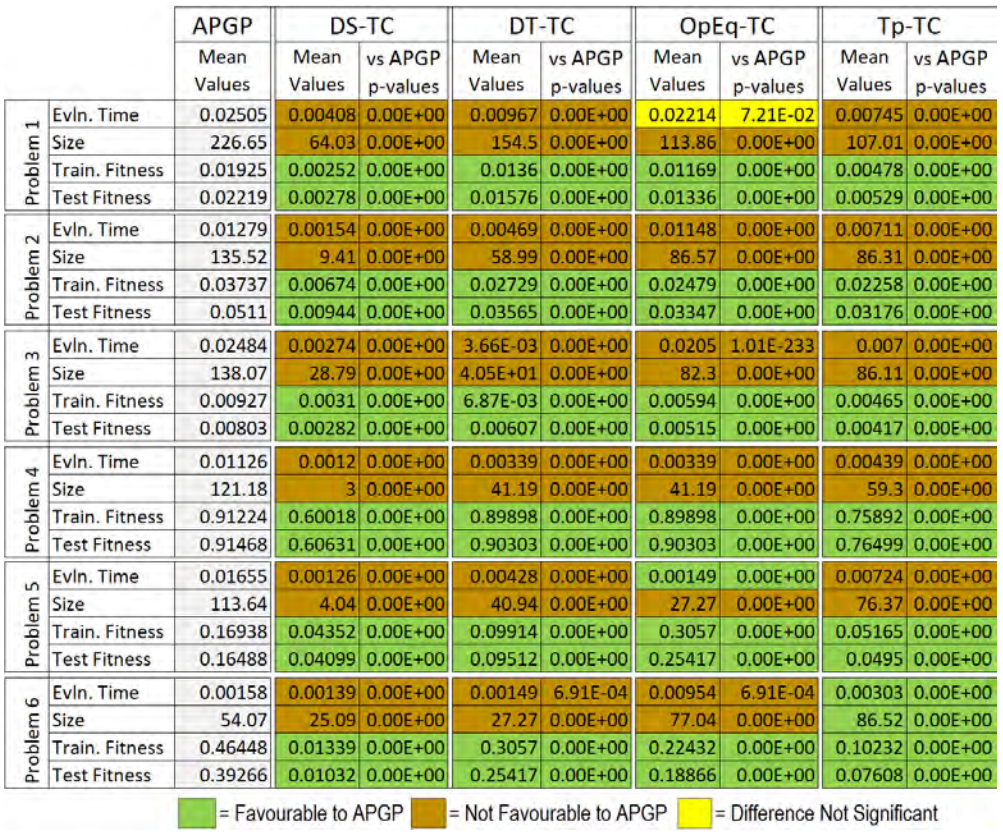
6.6.2 Comparison of APGP with Explicit Time Control

APGP consistently produced significantly more accurate models than the explicit time control methods (DS, DT, OpEq and Tp) on both training and test data; Figure 6.4 shows the results of the significance tests. Moreover, similar to its performance against GP+BC in Section 6.6.1, APGP produced more complex models against the time control methods; the exceptions are Problem 5 on OpEq and Problem 6 on Tp. Therefore, the complexity control of APGP is not as aggressive as the explicit time-control methods.

Despite producing more accurate solutions than size-control in Chapter 5, the time-control methods do not meet the accuracy of APGP. APGP remains the most accurate on both training and test data. Also, like GP+BC, the explicit time-control methods show more aggressive management of complexity than APGP that costs accuracy.

6.6.3 APGP in a Functionally Diverse Population

The experiments examine how the APGP performs in the functionally diverse initial populations that FLI creates. Therefore, results of APGP with and without FLI are compared, and Figure 6.5 details the significance of their difference.

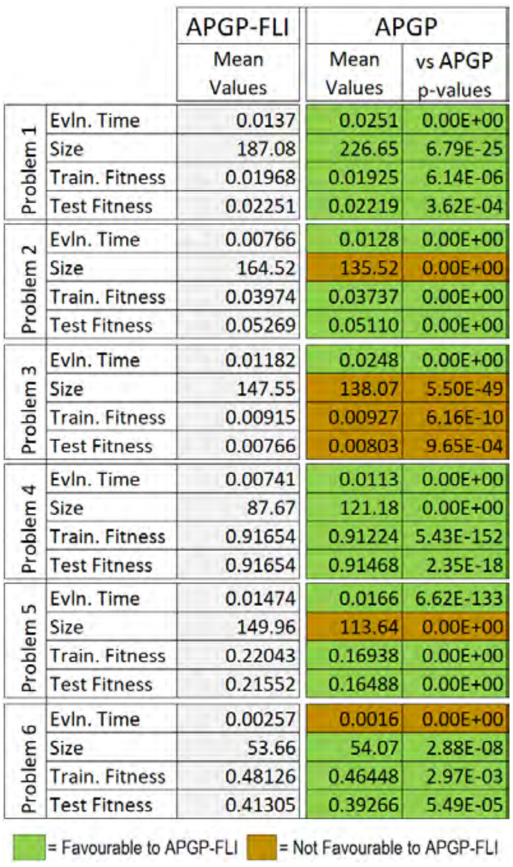


		APGP	DS-TC		DT-TC		OpEq-TC		Tp-TC	
		Mean Values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values
Problem 1	Evln. Time	0.02505	0.00408	0.00E+00	0.00967	0.00E+00	0.02214	7.21E-02	0.00745	0.00E+00
	Size	226.65	64.03	0.00E+00	154.5	0.00E+00	113.86	0.00E+00	107.01	0.00E+00
	Train. Fitness	0.01925	0.00252	0.00E+00	0.0136	0.00E+00	0.01169	0.00E+00	0.00478	0.00E+00
	Test Fitness	0.02219	0.00278	0.00E+00	0.01576	0.00E+00	0.01336	0.00E+00	0.00529	0.00E+00
Problem 2	Evln. Time	0.01279	0.00154	0.00E+00	0.00469	0.00E+00	0.01148	0.00E+00	0.00711	0.00E+00
	Size	135.52	9.41	0.00E+00	58.99	0.00E+00	86.57	0.00E+00	86.31	0.00E+00
	Train. Fitness	0.03737	0.00674	0.00E+00	0.02729	0.00E+00	0.02479	0.00E+00	0.02258	0.00E+00
	Test Fitness	0.0511	0.00944	0.00E+00	0.03565	0.00E+00	0.03347	0.00E+00	0.03176	0.00E+00
Problem 3	Evln. Time	0.02484	0.00274	0.00E+00	3.66E-03	0.00E+00	0.0205	1.01E-233	0.007	0.00E+00
	Size	138.07	28.79	0.00E+00	4.05E+01	0.00E+00	82.3	0.00E+00	86.11	0.00E+00
	Train. Fitness	0.00927	0.0031	0.00E+00	6.87E-03	0.00E+00	0.00594	0.00E+00	0.00465	0.00E+00
	Test Fitness	0.00803	0.00282	0.00E+00	0.00607	0.00E+00	0.00515	0.00E+00	0.00417	0.00E+00
Problem 4	Evln. Time	0.01126	0.0012	0.00E+00	0.00339	0.00E+00	0.00339	0.00E+00	0.00439	0.00E+00
	Size	121.18	3	0.00E+00	41.19	0.00E+00	41.19	0.00E+00	59.3	0.00E+00
	Train. Fitness	0.91224	0.60018	0.00E+00	0.89898	0.00E+00	0.89898	0.00E+00	0.75892	0.00E+00
	Test Fitness	0.91468	0.60631	0.00E+00	0.90303	0.00E+00	0.90303	0.00E+00	0.76499	0.00E+00
Problem 5	Evln. Time	0.01655	0.00126	0.00E+00	0.00428	0.00E+00	0.00149	0.00E+00	0.00724	0.00E+00
	Size	113.64	4.04	0.00E+00	40.94	0.00E+00	27.27	0.00E+00	76.37	0.00E+00
	Train. Fitness	0.16938	0.04352	0.00E+00	0.09914	0.00E+00	0.3057	0.00E+00	0.05165	0.00E+00
	Test Fitness	0.16488	0.04099	0.00E+00	0.09512	0.00E+00	0.25417	0.00E+00	0.0495	0.00E+00
Problem 6	Evln. Time	0.00158	0.00139	0.00E+00	0.00149	6.91E-04	0.00954	6.91E-04	0.00303	0.00E+00
	Size	54.07	25.09	0.00E+00	27.27	0.00E+00	77.04	0.00E+00	86.52	0.00E+00
	Train. Fitness	0.46448	0.01339	0.00E+00	0.3057	0.00E+00	0.22432	0.00E+00	0.10232	0.00E+00
	Test Fitness	0.39266	0.01032	0.00E+00	0.25417	0.00E+00	0.18866	0.00E+00	0.07608	0.00E+00

🟩 = Favourable to APGP 🟧 = Not Favourable to APGP 🟨 = Difference Not Significant

FIGURE 6.4: Comparison of APGP and the time-control methods (DS, DT, OpEq and Tp). APGP consistently produced significantly more accurate models (on both training and test data) than all the time-control methods. The time-control methods tend to produce simple models at the expense of accuracy.

APGP with FLI (APGP-FLI) produced significantly more accurate models (on both training and test data) on 5 out of 6 tests. The sizes of the models produced by APGP-FLI were significantly better (smaller) than those produced by APGP. Furthermore, the evaluation times of the models produced by APGP-FLI were significantly better (less) than those of APGP. Therefore, functionally diverse initial populations enable APGP to improve the accuracy and simplicity of the models it generates.



		APGP-FLI	APGP	
		Mean Values	Mean Values	vs APGP p-values
Problem 1	Evln. Time	0.0137	🟩 0.0251	🟩 0.00E+00
	Size	187.08	🟩 226.65	🟩 6.79E-25
	Train. Fitness	0.01968	🟩 0.01925	🟩 6.14E-06
	Test Fitness	0.02251	🟩 0.02219	🟩 3.62E-04
Problem 2	Evln. Time	0.00766	🟩 0.0128	🟩 0.00E+00
	Size	164.52	🟧 135.52	🟧 0.00E+00
	Train. Fitness	0.03974	🟩 0.03737	🟩 0.00E+00
	Test Fitness	0.05269	🟩 0.05110	🟩 0.00E+00
Problem 3	Evln. Time	0.01182	🟩 0.0248	🟩 0.00E+00
	Size	147.55	🟧 138.07	🟧 5.50E-49
	Train. Fitness	0.00915	🟧 0.00927	🟧 6.16E-10
	Test Fitness	0.00766	🟧 0.00803	🟧 9.65E-04
Problem 4	Evln. Time	0.00741	🟩 0.0113	🟩 0.00E+00
	Size	87.67	🟩 121.18	🟩 0.00E+00
	Train. Fitness	0.91654	🟩 0.91224	🟩 5.43E-152
	Test Fitness	0.91654	🟩 0.91468	🟩 2.35E-18
Problem 5	Evln. Time	0.01474	🟩 0.0166	🟩 6.62E-133
	Size	149.96	🟧 113.64	🟧 0.00E+00
	Train. Fitness	0.22043	🟩 0.16938	🟩 0.00E+00
	Test Fitness	0.21552	🟩 0.16488	🟩 0.00E+00
Problem 6	Evln. Time	0.00257	🟧 0.0016	🟧 0.00E+00
	Size	53.66	🟩 54.07	🟩 2.88E-08
	Train. Fitness	0.48126	🟩 0.46448	🟩 2.97E-03
	Test Fitness	0.41305	🟩 0.39266	🟩 5.49E-05

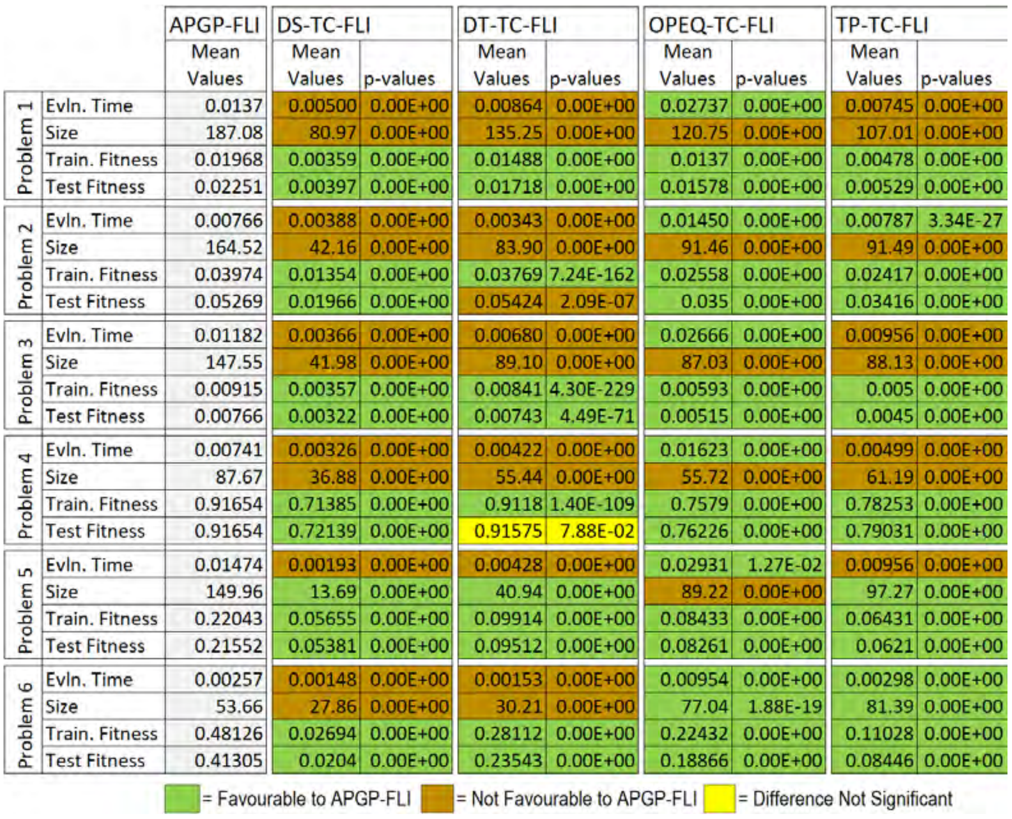
🟩 = Favourable to APGP-FLI 🟧 = Not Favourable to APGP-FLI

FIGURE 6.5: Result of test for significance of the impact of FLI on APGP. FLI significantly improved the accuracy scores (on both training and test data) of APGP on 5 out of 6 problems. Also, FLI reduced size (on 4 out of 6 problems) and evaluation times (on 5 out of 6 problems).

6.6.4 APGP with FLI vs. Time Control with FLI

As FLI improves the performance of both APGP and the explicit time-control methods, this section examines if FLI has tilted the balance. Therefore, the results of APGP-FLI with those of time-control methods with FLI are compared; Figure 6.6 details the results of the significance testing.

The results indicate that APGP-FLI produces significantly more accurate individuals than all the compared time-control methods despite introducing FLI in all methods. APGP-FLI generated models with significantly better test fitness scores in 23 out of 24 tests and 23 out of 24 training test scores; the exceptions are on DT. Furthermore, like earlier comparisons with time-control methods, APGP-FLI tends to produce more complex individuals that improve accuracy scores.



		APGP-FLI	DS-TC-FLI		DT-TC-FLI		OPEQ-TC-FLI		TP-TC-FLI	
		Mean Values	Mean Values	p-values	Mean Values	p-values	Mean Values	p-values	Mean Values	p-values
Problem 1	Evln. Time	0.0137	🟧 0.00500	0.00E+00	🟧 0.00864	0.00E+00	🟩 0.02737	0.00E+00	🟧 0.00745	0.00E+00
	Size	187.08	🟧 80.97	0.00E+00	🟧 135.25	0.00E+00	🟧 120.75	0.00E+00	🟧 107.01	0.00E+00
	Train. Fitness	0.01968	🟩 0.00359	0.00E+00	🟩 0.01488	0.00E+00	🟩 0.0137	0.00E+00	🟩 0.00478	0.00E+00
	Test Fitness	0.02251	🟩 0.00397	0.00E+00	🟩 0.01718	0.00E+00	🟩 0.01578	0.00E+00	🟩 0.00529	0.00E+00
Problem 2	Evln. Time	0.00766	🟧 0.00388	0.00E+00	🟧 0.00343	0.00E+00	🟩 0.01450	0.00E+00	🟩 0.00787	3.34E-27
	Size	164.52	🟧 42.16	0.00E+00	🟧 83.90	0.00E+00	🟧 91.46	0.00E+00	🟧 91.49	0.00E+00
	Train. Fitness	0.03974	🟩 0.01354	0.00E+00	🟩 0.03769	7.24E-162	🟩 0.02558	0.00E+00	🟩 0.02417	0.00E+00
	Test Fitness	0.05269	🟩 0.01966	0.00E+00	🟧 0.05424	2.09E-07	🟩 0.035	0.00E+00	🟩 0.03416	0.00E+00
Problem 3	Evln. Time	0.01182	🟧 0.00366	0.00E+00	🟧 0.00680	0.00E+00	🟩 0.02666	0.00E+00	🟧 0.00956	0.00E+00
	Size	147.55	🟧 41.98	0.00E+00	🟧 89.10	0.00E+00	🟧 87.03	0.00E+00	🟧 88.13	0.00E+00
	Train. Fitness	0.00915	🟩 0.00357	0.00E+00	🟩 0.00841	4.30E-229	🟩 0.00593	0.00E+00	🟩 0.005	0.00E+00
	Test Fitness	0.00766	🟩 0.00322	0.00E+00	🟩 0.00743	4.49E-71	🟩 0.00515	0.00E+00	🟩 0.0045	0.00E+00
Problem 4	Evln. Time	0.00741	🟧 0.00326	0.00E+00	🟧 0.00422	0.00E+00	🟩 0.01623	0.00E+00	🟧 0.00499	0.00E+00
	Size	87.67	🟧 36.88	0.00E+00	🟧 55.44	0.00E+00	🟧 55.72	0.00E+00	🟧 61.19	0.00E+00
	Train. Fitness	0.91654	🟩 0.71385	0.00E+00	🟩 0.9118	1.40E-109	🟩 0.7579	0.00E+00	🟩 0.78253	0.00E+00
	Test Fitness	0.91654	🟩 0.72139	0.00E+00	🟨 0.91575	7.88E-02	🟩 0.76226	0.00E+00	🟩 0.79031	0.00E+00
Problem 5	Evln. Time	0.01474	🟧 0.00193	0.00E+00	🟧 0.00428	0.00E+00	🟩 0.02931	1.27E-02	🟧 0.00956	0.00E+00
	Size	149.96	🟩 13.69	0.00E+00	🟩 40.94	0.00E+00	🟧 89.22	0.00E+00	🟩 97.27	0.00E+00
	Train. Fitness	0.22043	🟩 0.05655	0.00E+00	🟩 0.09914	0.00E+00	🟩 0.08433	0.00E+00	🟩 0.06431	0.00E+00
	Test Fitness	0.21552	🟩 0.05381	0.00E+00	🟩 0.09512	0.00E+00	🟩 0.08261	0.00E+00	🟩 0.0621	0.00E+00
Problem 6	Evln. Time	0.00257	🟧 0.00148	0.00E+00	🟧 0.00153	0.00E+00	🟩 0.00954	0.00E+00	🟩 0.00298	0.00E+00
	Size	53.66	🟧 27.86	0.00E+00	🟧 30.21	0.00E+00	🟩 77.04	1.88E-19	🟩 81.39	0.00E+00
	Train. Fitness	0.48126	🟩 0.02694	0.00E+00	🟩 0.28112	0.00E+00	🟩 0.22432	0.00E+00	🟩 0.11028	0.00E+00
	Test Fitness	0.41305	🟩 0.0204	0.00E+00	🟩 0.23543	0.00E+00	🟩 0.18866	0.00E+00	🟩 0.08446	0.00E+00

🟩 = Favourable to APGP-FLI 🟧 = Not Favourable to APGP-FLI 🟨 = Difference Not Significant

FIGURE 6.6: Result of significance test for differences in the final populations of APGP with FLI and the time-control methods with FLI. APGP-FLI produced models with higher accuracy (on both training and test data) in all cases. The models produced by time-control methods with FLI were simpler (smaller size or evaluation time) than APGP in 34 out of 48 tests.

6.7 Conclusion

As a novel method that evaluation time enables, the APGP introduces a simple yet significant change in the evolutionary process of GP to offer several gains. The APGP induces a race between concurrent executions of multiple breed operation (evaluations) to allow simple and accurate models to get a competitive advantage. Therefore, APGP removes the point of synchronisation that enforces a lock-step evolution of models. Instead, the race condition in APGP allows models that finish evaluating earlier than their counterparts to join a steady-state population if their accuracy is competitive enough. APGP thus questions the conventional, but ultimately unnatural, practice of evolving individuals in a lock-step manner or one after the other.

Contrary to bloat-control techniques that target and penalise sizes while sacrificing accuracy and training time, APGP produces models that are significantly more accurate than standard GP and GP with bloat-control in all the tests (100%). The APGP complexity control (size and evaluation times as complexity) was significantly better in 5 out of 6 tests against GP; however, GP+BC produced smaller sizes in 6 out of 6 tests and smaller sizes in 3 out of 6. Therefore, APGP produces significant gains in accuracy and manages complexity in a way that does not compromise accuracy.

The race condition in APGP that allows simple and accurate solutions to push through, has the added advantage of improving training speed. On all six problems, APGP used fewer evaluations to match the training accuracy of GP (11% to 40% fewer) and that of GP+BC (15% to 84% fewer).

APGP consistently produced significantly more accurate solutions than the explicit time control methods introduced in Chapter 5; it prevailed in 24 out of 24 tests. Compared with the explicit time control methods, the APGP complexity control was less aggressive. This shows the advantage of APGP's ability to simultaneously encourage simplicity and accuracy without subjectively targeting and penalising complexity.

The Fixed Length Initialisation (FLI), whose introduction was motivated by the understanding that evaluation time can differentiate functional complexity, improved the performance of the APGP; FLI creates an initial population made up of same-sized individuals that are functionally diverse. The accuracy of APGP-FLI on the test data improved significantly in 5 out of 6 cases; the complexity improved (decrease) in 8 out of 12 tests (6 tests for size and 6 for evaluation time). Moreover, despite introducing FLI, APGP retained its lead in accuracy scores; APGP-FLI results remained significantly more accurate than the time-control that improved with FLI.

Ultimately, the results presented in this chapter show that the proposed APGP manages the challenge of balancing the simplicity and accuracy of the models it generates well. Although the experiments in this chapter used regression problems, in principle, the evaluation time can also characterise complexity in other applications. Therefore, it promises to be a broadly applicable means to achieve simple and accurate solutions fast.

As a novel system, which uses a new measure of complexity and a novel approach that implicitly controls it, the APGP needs to be analysed. The analysis will validate the assumptions behind it, answer questions that have arisen about it, and explore ways it may be optimised. The next chapter (Chapter 7) presents the results of the analysis of the APGP.

Chapter 7

Analysis of the APGP

7.1 Introduction

The asynchronous parallel GP (APGP) is a novel GP method that introduces a simple yet fundamental change in the evolutionary process. The rationale for this change and the result it produced (in Chapter 6) challenge the common practice of executing evolutionary algorithms in a lock-step manner, where the evaluation of all existing individuals is completed (in batches or sequentially) before the evolution proceeds to produce new offspring [184, 204]. Although the results in Chapter 6 attest that the APGP works as expected, some further analysis in this chapter helps understand the impact of the key APGP hyperparameter: the number of threads in the parallel race.

Furthermore, the analysis examines the following assumptions about the APGP: (1) increasing the degree of concurrency (the number of individuals that are allowed to race at a time) will increase the number of simpler solutions that become available early to contest for a place in the breeding population; (2) if the complex solutions do well they will not be excluded simply because they are complex; (3) and the fitness scores of simple (fast-evaluating) solutions must be competitive for them to both get into the population and reproduce further.

In concordance with the assumptions behind the APGP, the analysis in this chapter shows that the APGP offers increased opportunities for simple (fast-evaluating) individuals to get ahead during the evolution. Furthermore, the analysis shows that increasing the degrees of concurrency can increase the complexity control without compromising training accuracy scores. Furthermore, it confirms that simple individuals have to be competitive (in terms of accuracy) to take advantage of the opportunities.

The rest of this chapter is organised as follows: Section 7.2 analyses the effect of the degree of concurrency on APGP. This section examines the effect of concurrency in both a simplified setting (without selection and further breeding) and when APGP is applied to real problems with selection and breeding back in play. Section 7.3 examines the interplay between the accuracy and the complexity of models during the evolution; Section 7.4 examines how concurrency affects the APGP's ability to manage complexity; Section 7.5 examines how the fixed-length initialisation (FLI) can be used to improve APGP further, and; finally, Section 7.6 concludes this chapter.

7.2 Effect of Degree of Concurrency in APGP

This section examines whether increasing the degree of concurrency will increase the opportunity for the simpler solutions to finish early, which, in turn, means they have more opportunities to get into the breeding population to gain an evolutionary advantage. To this end, the analysis starts by examining the impact of varying the degrees of concurrency in a simplified setting as used in Section 6.4, where evolutionary processes like selection and replacement are not involved. Subsequently, the analysis examines the effect of concurrency when all evolutionary processes are at play.

7.2.1 Testing Concurrency in a Simplified Environment

Using the same experimental setup from Section 6.4 (where evolutionary processes like selection and replacement are not involved), tests were carried out and monitored with varying degrees of concurrency (threads). Similarly to the earlier experiments, the order evaluations started and finished were recorded; likewise, their evaluation times and complexity (as designed). Using the collected data, different stages of the race are analysed to study the completed evaluations at different points. For example, from the beginning to the 100th, to the 200th, and so forth.

Results: Higher Degree of Concurrency Increase the Opportunity for Simple Solutions to Finish Early

Figure 7.1 illustrates the impact of the degree of concurrency on the rate that simpler solutions (with smaller dataset sizes and evaluation times) complete their concurrent evaluations. The higher the degree of concurrency (threads), the higher the number of simple evaluations that return earlier. For example, at the early stages of the race (top charts of the figure), the completion of simple evaluations peaks in the same order as their degrees of concurrency; over time (charts at the bottom of the figure), the differences between the different threads reduces as expected. This result indicates that a higher degree of concurrency increases the opportunity for the simpler evaluation tasks to complete earlier.

In Sections 6.3, this thesis theorised that the APGP allows simple solutions to complete evaluation earlier than their more complex counterpart and that a higher degree of concurrency in APGP increases the opportunity to do so. Further, it posited that this does not necessarily mean that the simpler individuals will continue to thrive because it also matters as to how fit they are. Thus, by delineating task completions with subsequent evolution, the result in this section shows that at least the simpler individuals complete earlier than their

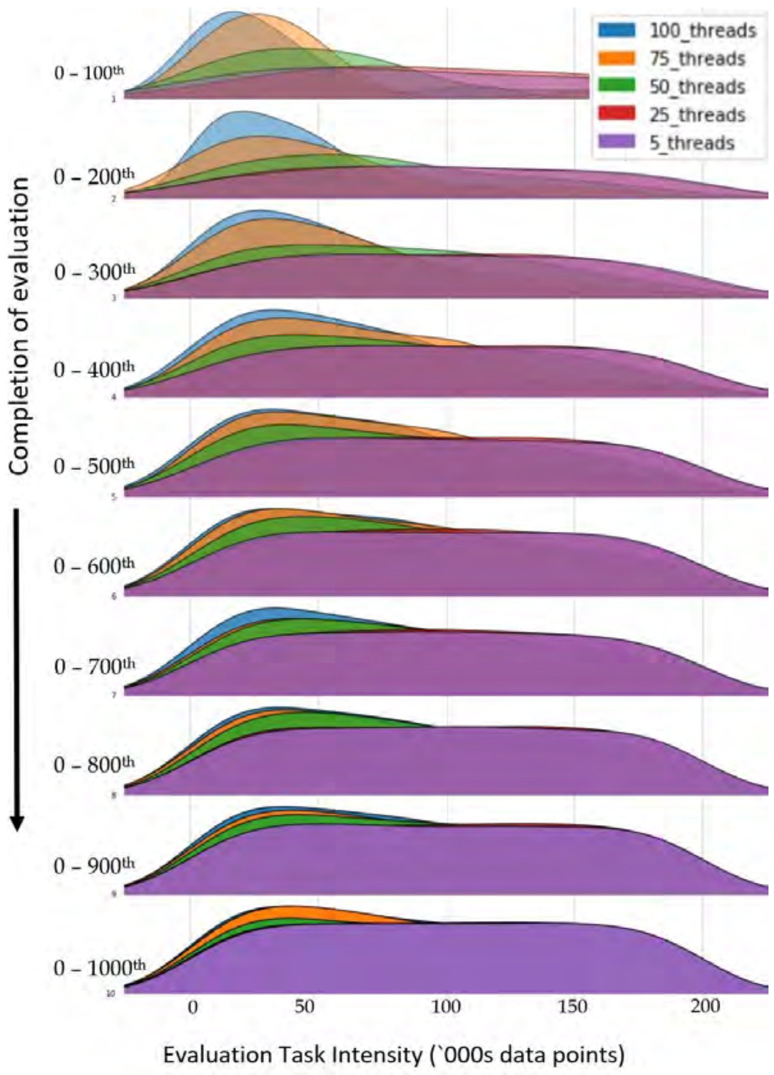


FIGURE 7.1: Illustrated is the effect of degree of concurrency on the rate that simple evaluation tasks (with small dataset sizes) complete their concurrent evaluations. Higher degrees of concurrency (threads) allow the simpler tasks to complete early more frequently.

complex counterparts and increasing the degree of concurrency does increase the opportunity for the simple solutions to finish early. This is not necessarily a trivial result considering the experiments were not run on specialised systems with advanced parallelisation; instead, the experiments were on a Quad-Core processor where some noise in the measurements of evaluation time can not be completely eliminated despite the techniques introduced that substantially improve the reliability of time measurements.

7.2.2 Testing Concurrency on Real Problems

While Section 7.2.1 examines how the degree of concurrency affects evaluation tasks outside the evolution, this section investigates how concurrency affects the APGP, where all evolutionary dynamics are at play.

Although a higher concurrency allows a more frequent early completion for simpler individuals, it does not automatically mean that simpler solutions will thrive more during actual evolution. This is because fitness is key to participating in subsequent evolution. Moreover, the makeup of the parent population that the subsequent breed operation selects from may have been shaped by the result of previous replacement attempts. Therefore, the effect of breeding and replacement means that the relationship between the degree of concurrency and the performance of the APGP is expected to be more complex in this case, and therefore merits the investigation reported in this section.

The experiment in this section tests the degrees of concurrency 5, 25, 50, 75 and 100 on the regression problems from the experiments in Section 6.5. This time, in addition to tracking the order the individuals are sent for evaluation and their return, the data collected includes how successfully they replace other individuals in the population. Tracking the success at replacement enables verifying the assumption that merely completing an evaluation does not guarantee that an individual will successfully get into the breeding population. Furthermore, while tracking the successful individuals, changes in the population are observed to study how the incoming individuals influence the characteristics of the population; for example, the changes in the distribution of size in the population.Section 7.3 analyses the interplay between speed and accuracy.

Results: Higher concurrency still suits simpler solutions in APGP

The analysis starts by checking if the simple solutions finish evaluating early in APGP (inside evolution) as observed in Section 7.2 (outside evolution).

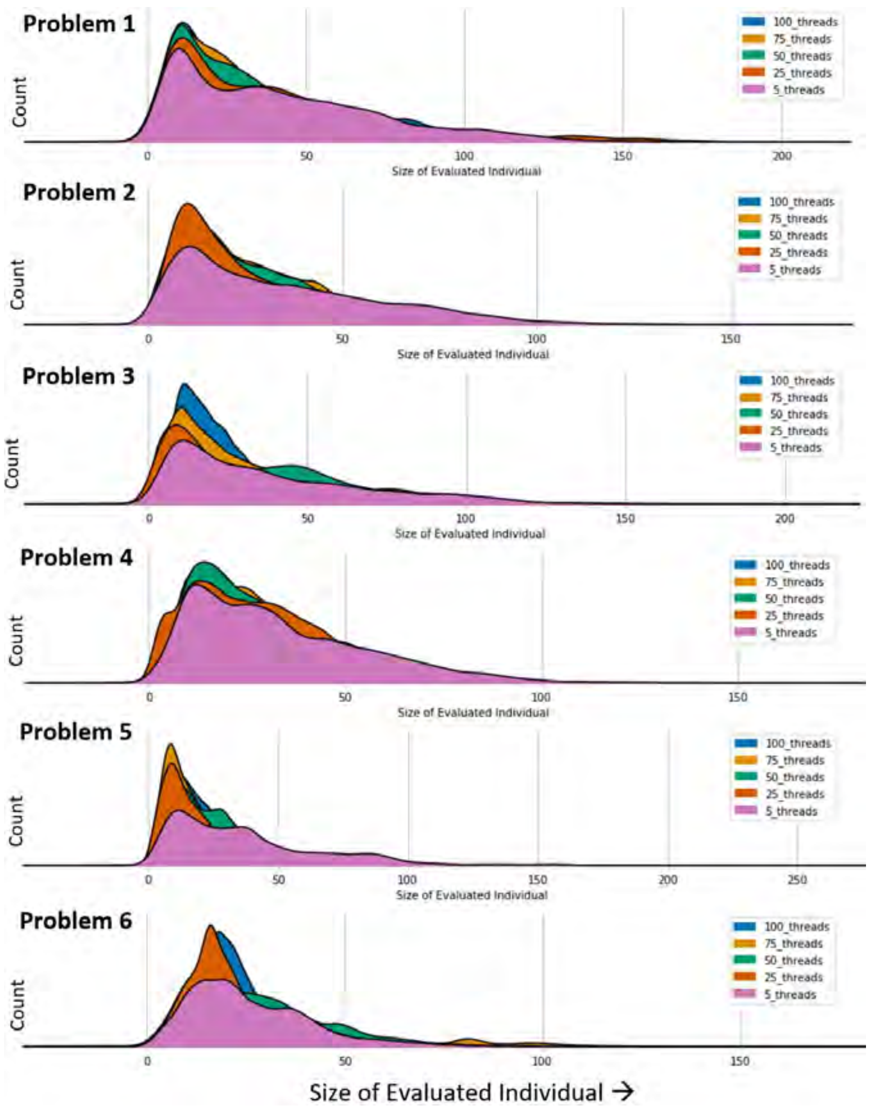


FIGURE 7.2: Effect of the Degree of Concurrency of APGP on Real Problems. The charts in each row show the distribution of size in the population at the 1,000th evaluation that different degrees of concurrency produced.

Figure 7.2 visualises the distribution of size in the populations after the 1,000th evaluation of degrees of concurrency 5, 25, 50, 75 and 100. Each row represents a test problem, and the overlapping charts within a row represent the result from the different degrees of concurrency. Further, each chart is an aggregation of the first 1,000 evaluations of 30 runs.

Two key points are notable in the charts in Figure 7.2. First, concurrency still enables simple evaluation tasks to finish earlier in APGP; the distribution of size is skewed to show that the simpler solutions are finishing earlier than the more complex ones across all six problems. Second, higher degrees of concurrency tend to create more opportunities for simple individuals; for example, concurrency 5 shows a lower number of simple (smaller) individuals that have finished than do the higher concurrencies.

As expected, the distribution is not as predictable as the charts in Figure 7.2, where the difference from one thread to another is proportional. Note, in the proposed APGP, the impact of the parallelisation and the degree of concurrency is conditional: while simple solutions may complete evaluations early, they are discarded unless their accuracy is competitive. The next section examines whether the conditional impact of parallelisation is true.

7.3 A dynamic complexity control based on the accuracy of simple models

According to the formulation of the APGP, evaluated individuals must compete for a place in the breeding population based on training accuracy (fitness score); therefore, the fast individuals are not guaranteed places in the population. Unlike techniques that control complexity by penalising, APGP simply creates more opportunities for the fast evaluating individuals to join the population but to actually join the population they must be accurate as well. Therefore, it is important to: (1) verify if this assumption is true, and (2) to understand what happens when the opportunity for simple individuals is increased by raising the degree of concurrency.

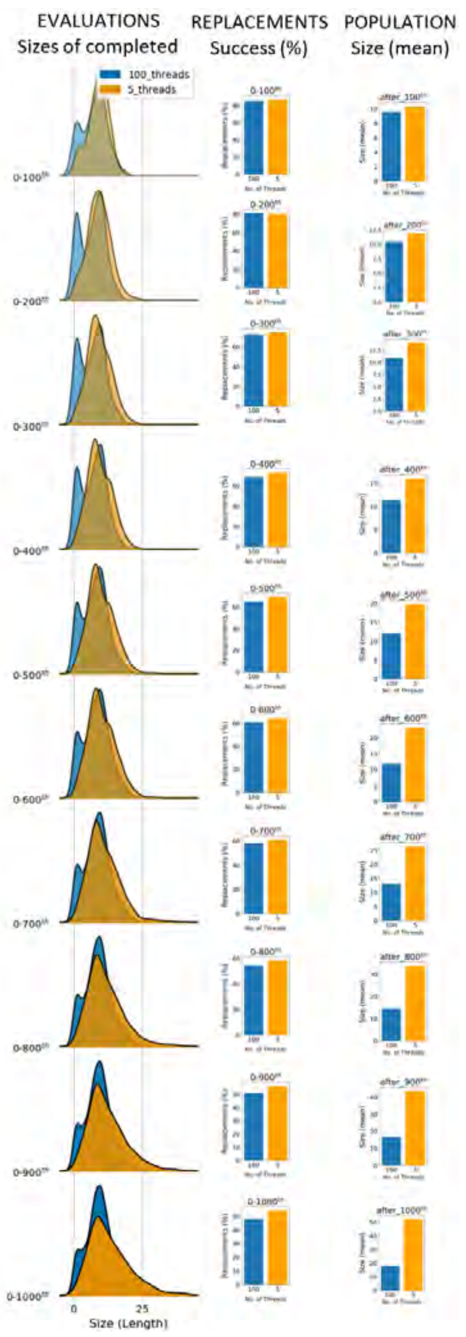
Therefore, the experiments from Section 7.2.2 are re-run; in addition to the data previously collected, here the experiments also record whether the evaluated individuals succeed at getting into the breeding population or not.

As empirically demonstrated in Section 7.2, an increase in the degree of concurrency increases the frequency of simple solutions finishing evaluations early; therefore, the subsequent analysis compares the result of two extreme degrees of concurrency – that is, 5 and 100. Furthermore, the analysis includes the results from several problems because the expectation is that the success of simple (fast evaluating) individuals at getting into the breeding population is problem-specific. Accordingly, the analysis examines whether or not the relatively simple individuals fail to win out consistently.

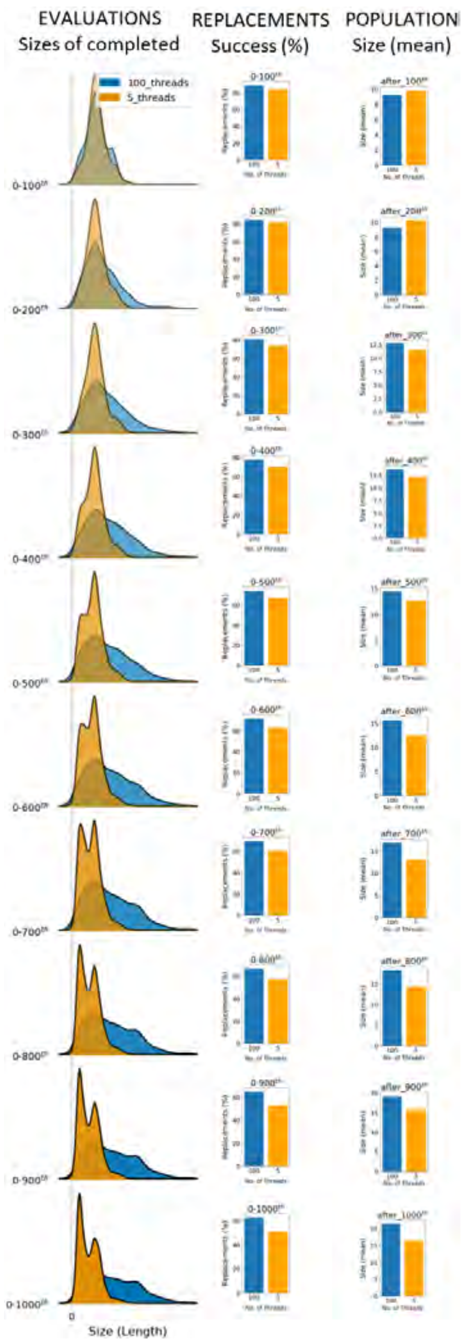
A close look at the first 1,000 evaluations of the test problems shows that the replacement process (which is based on training fitness) plays an important role. APGP with 100 threads (APGP-100) was compared with APGP with 5 threads (APGP-5); the results are visualised in figures 7.3, 7.4 and 7.5. Each figure reports the results from two problems.

For each of the test problems, the first column shows the distribution of size of all the models that have completed evaluations at 10 different intervals, (top row for 0 to 100^{th} and bottom for 0 to $1{,}000^{th}$), the second column shows how successful the evaluated individuals are at replacement (percentage), and the third column compares the mean size of the population at the corresponding stage of the evolution.

The mean size of the population for APGP-100 and APGP-5 change based on how well their individuals are succeeding at replacement; as expected, this is problem specific because while simpler solutions may suit one problem, it may not do so for another problem to the same degree. Note, it is important



(A) Problem 1



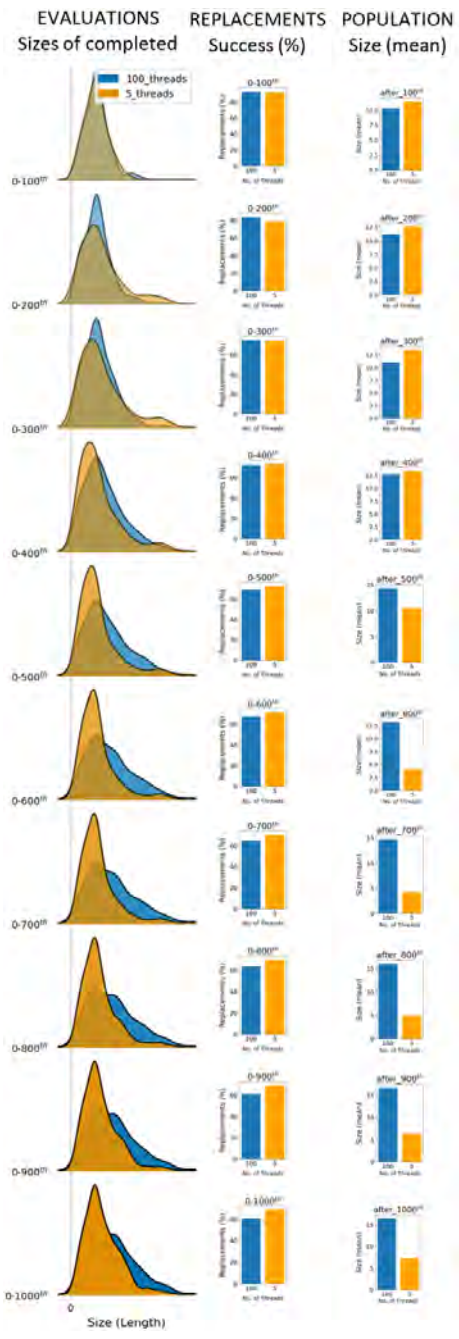
(B) Problem 2

FIGURE 7.3: APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100^{th} and bottom for 0 to $1{,}000^{th}$), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes.

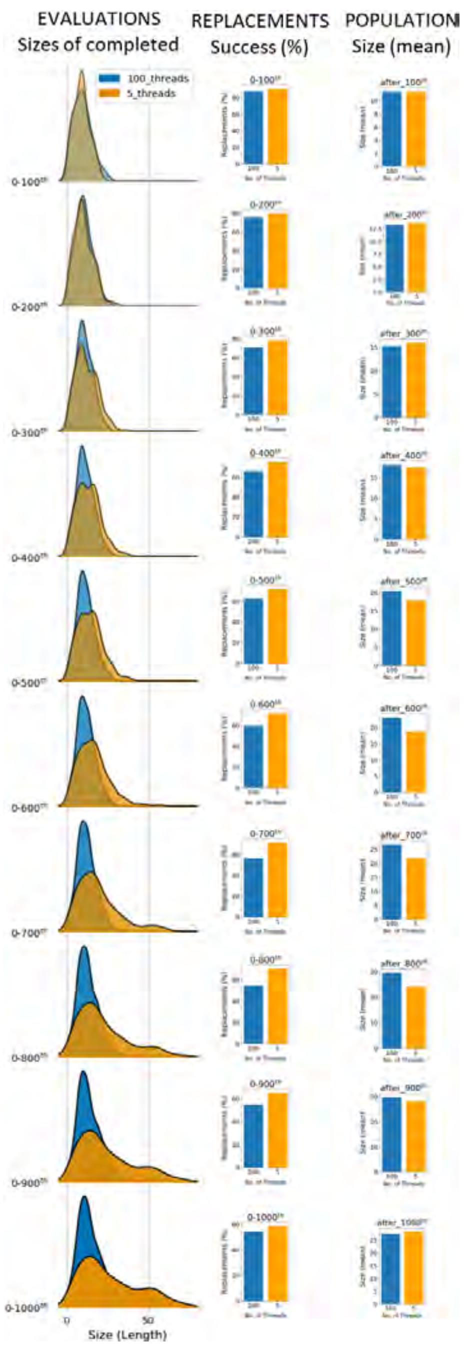
for APGP allow a dynamic interplay between accuracy and simplicity and not obstruct accuracy by over-hindering complexity.

Therefore, when APGP-100 produces a higher number of simple solutions than APGP-5, the mean size of its population becomes lower than that of APGP-5 *only* when its individuals replace into the population at least as often as those produced by APGP-5. For example, the results for Problem 1 and Problem 5, visualised in Figure 7.3a and Figure 7.5a respectively, show that APGP-100 creates more opportunities for simpler solutions than APGP-5. For both problems, APGP-100 and APGP-5 were succeeding in replacement at similar levels and this resulted in lower mean sizes for 100 threads.

In contrast, APGP-5 produced greater numbers of small individuals for Figure 7.3b (Problem 2) and Figure 7.4a (Problem 3); and, because they were more successful at replacement, the mean size of the population of APGP-5 reduced. Therefore, a higher degree of concurrency alone does not reduce the mean size of the population. The simple solutions need to be accurate enough to take advantage of the increased opportunity that higher degrees of concurrency provides. This relationship is observable in the results of all test problems.

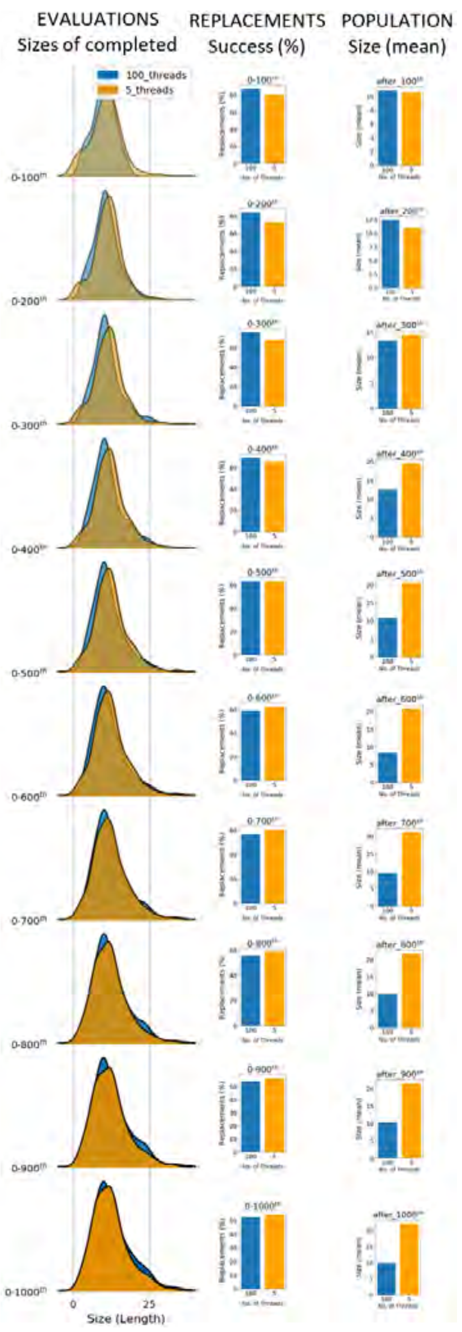


(A) Problem 3

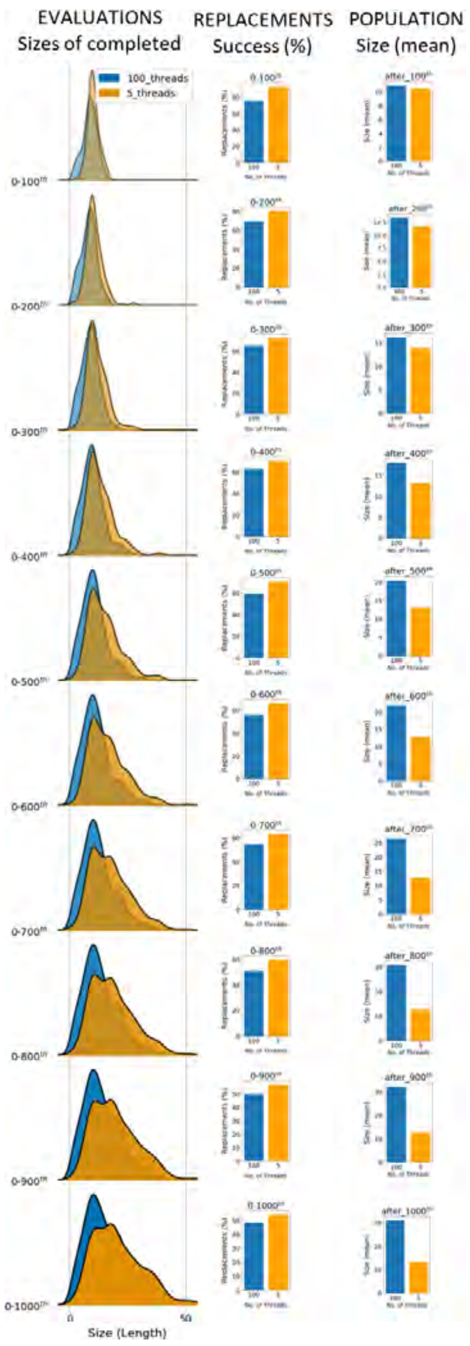


(B) Problem 4

FIGURE 7.4: APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100^{th} and bottom for 0 to $1{,}000^{th}$), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes.



(A) Problem 5



(B) Problem 6

FIGURE 7.5: APGP runs with 100 and 5 threads are compared. The first 1,000 evaluations were monitored; column 1 shows their distribution by size as they complete (top charts is for 0 to 100^{th} and bottom for 0 to $1{,}000^{th}$), column 2 shows their success (%) at replacement, and Column 3 shows how the mean size of the population changes.

7.4 Balancing Accuracy with Simplicity Across a Wide Range of Concurrency

The experiment in this section explores degrees of concurrency of a wider range than those used before to determine how this may affect the accuracy and complexity control of APGP and search for an optimal value. To this end, the experiment uses multiple runs of the six problems with different degrees of concurrency 5, 100, 250, and 500; these values correspond to 1%, 20%, 50% and 100% of the population size, respectively. Other settings include a population of 500, 25,000 evaluations (50 generations) and 50 runs.

Figure 7.6 shows the development by the generation of the mean size and training fitness for the degrees of concurrency. The first column in Figure 7.6 compares how the size changes across generations with different degrees of concurrency (threads); size is an indicator of complexity here as simple, intuitive, and easy to quantify measure. The results show that the higher degrees of concurrency have a greater effect on suppressing complexity than the lower degrees. On all the problems, 5 threads produced the highest mean sizes across the generations and 500 threads the lowest. Thus, it shows that a very high degree of currency does create more opportunities for the simpler individuals, which subsequently makes the population simpler. Although 100 threads generally produce a greater mean size than 250 threads, their mean sizes converge to similar values at some point of the evolution for some of the problems. Therefore, a clear difference is seen when comparing the effect of concurrency values from the extremes but not with the middle and closer concurrency values.

While the higher degree of concurrency (500 threads) suppresses the size of the individuals predictably, it does not compromise the training fitness. As shown in column two of Figure 7.6, no clear trend as to which concurrency level produces better fitness is consistently better than the others is observable. Also, generally, the training fitness values converge by the end of evolution.

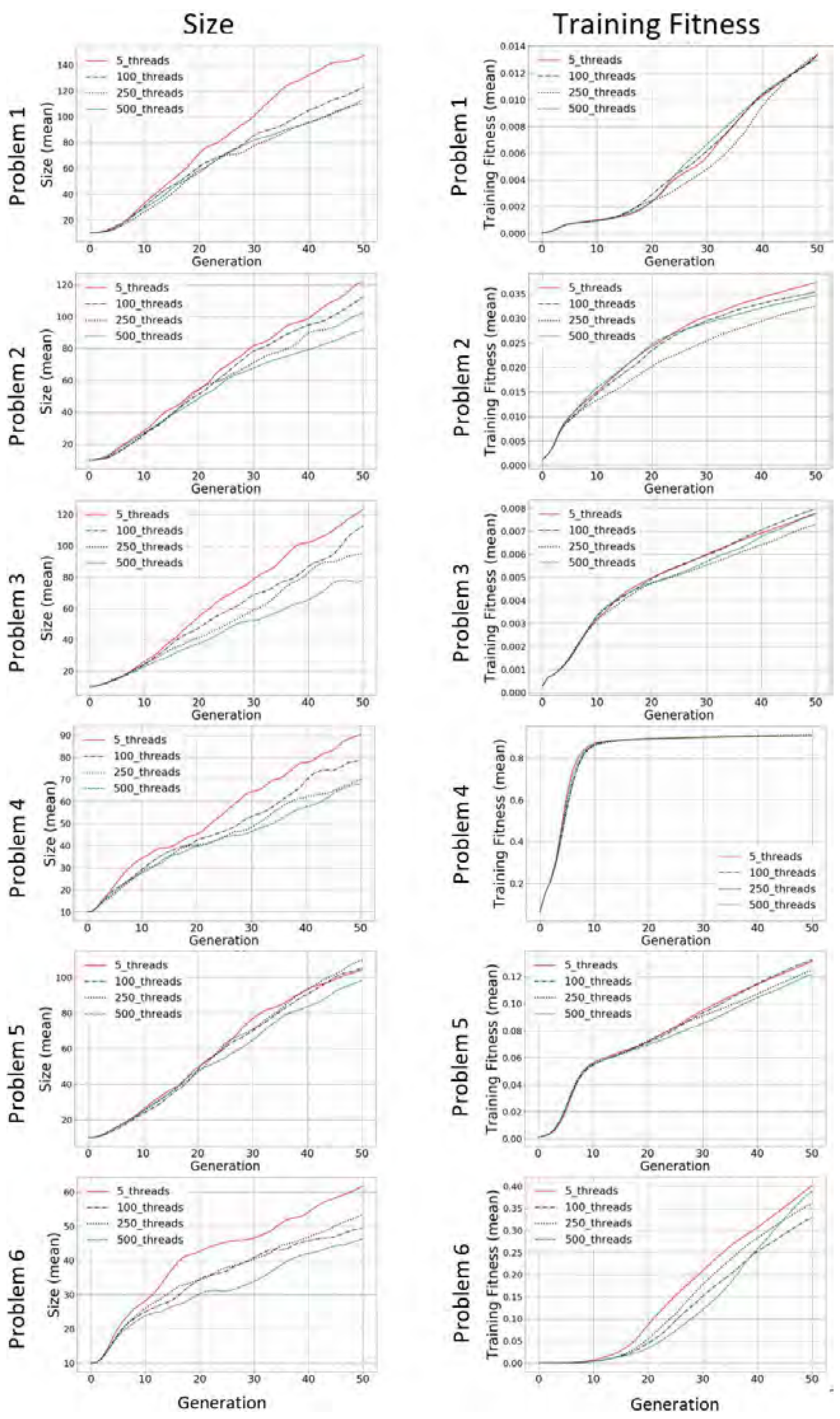


FIGURE 7.6: The development in size and training fitness (by generations) in the population of APGP with 5, 100, 250 and 500 parallel threads are on the plots. The higher degree of concurrency (threads) increases the opportunities for the smaller sized individuals to lower the average population sizes; yet, as preferred, their average training fitness is generally not reduced.

7.5 Varying the Lengths in the Fixed Length Initialisation

Chapter 6 showed that the Fixed Length Initialisation (FLI) improves the performance of the APGP; also, FLI generally improves the explicit time control methods in Chapter 5. FLI raises functional diversity (mathematical operators) in the population to let evaluation time can characterise this complexity, and size can not. To do that, FLI initialises the population with random and unique individuals of uniform size. Thus far, FLI has initialised with a population of individuals with a length of ten nodes each. Therefore, this section explores as to whether initialising with other lengths affects the performance of APGP.

The experiments in this section involve re-running the APGP-FLI experiments from Section 6.5 with ten different sizes: from five to fifty, in steps of 5 (FLI-5 to FLI-50); other settings remain the same.

7.5.1 Result: Effect of varying the length in the FLI

The results of varying the fixed-length of FLI are illustrated in Figure 7.7. For each problem, a box-plot for the mean test fitness values and another for the mean sizes of individuals in the final populations are plotted; the horizontal axis shows the variation of the fixed-length parameter.

The variation did not significantly change the sizes of individuals in the final populations. However, there was some activity in test-fitness scores: except for Problem 2, the results show that setting a length of five (5 nodes) for FLI appears disadvantageous. In addition, the setting that produced the best results is problem-specific. However, it is reassuring to see that the relatively low length settings – FLI-10, FLI-15 and FLI-20 – are at least competitive with the higher lengths.

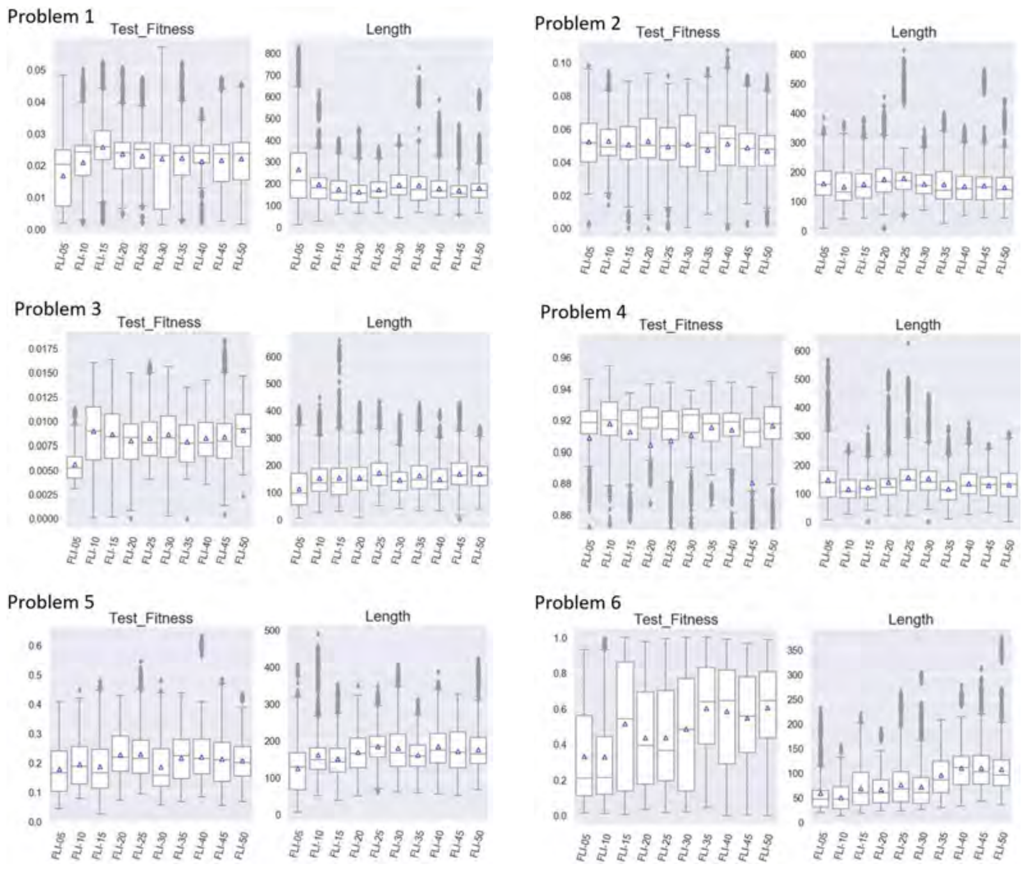


FIGURE 7.7: Result of varying the fixed length in FLI. The variation did not significantly change the sizes of individuals in the final populations. However, their test-fitness accuracy showed some differences.

7.6 Conclusion

The outcome of this chapter confirms that increasing the degree of concurrency increases the opportunity for simpler individuals to finish their evaluation early. However, the increased opportunity for simple solutions does not offer any advantage for them unless they are competitive enough to get into the population; replacement is based on accuracy. The success in replacement by simple candidates is problem specific and dependent on the current state of the population. When the simple candidates are competitive and are succeeding at getting into the current population, the constitution of the population becomes simpler and increases the chances of breeding simple offspring.

However, if the simple solutions are not as competitive (in terms of accuracy) as the complex ones, then the more complex candidates are admitted into the population and the population changes accordingly. When the simple and complex individuals are competitive and are getting into the population, then increasing the degree of concurrency has a greater effect of discouraging complexity; note, in this case, increasing the degree of concurrency has the effect of encouraging simplicity without compromising accuracy. Therefore, increasing the degree of concurrency will increase complexity control where possible.

Next, the analysis considered what may be the optimal degree of concurrency for APGP. Given that the possibility of producing very simple and accurate solutions is problem-specific, it may not be easy to specify an optimal degree of concurrency. However, since the higher degree of concurrency tends to control complexity better than the lower values, they may be preferred. This is supported by the observation that the higher values do not stop the development of training accuracy. In addition, the simpler solutions that the degree of concurrency produces may improve generalisation. Furthermore, the higher degree of concurrency offers the added advantage of improving the overall runtime.

Finally, the investigation of an optimal length for FLI indicates that a moderately small value is sufficient: node length of 10 to 20. Though the analysis shows that initialising with large-sized individuals does not negatively impact the final solutions, initialising with the smaller (10 to 20 nodes) sizes is preferred. The smaller sizes are sufficient to produce ample numbers of unique individuals (as discussed in Section 5.3.1) and will save some computing resources at the beginning of the APGP run.

In addition to the explicit time control methods from Chapter 5, this thesis presents the APGP as a promising method that automatically manages the complexity of the solutions that GP produces. Fundamental to these systems is the evaluation time as an indicator of complexity; thus far, this thesis shows time can characterise the size of an individual and the computational complexity of its constituent components. The next chapter explores how the evaluation time may detect other ideas of the complexity of GP solutions. The investigation will be in the context of a practical GP application; thus, the next chapter assesses the usefulness of the proposed system.

Chapter 8

Application of Evaluation Time Schemes in GP with Multiple Linear Regression

8.1 Introduction

This chapter practically demonstrates that *time* and *size*, as measures of complexity, are not the same. At the time of the publication of the proof of concept of this thesis [184], another paper [185] hypothesised that time and size are no more than two sides of the same coin. In contrast, this thesis shows that the evaluation time can detect and leverage other notions of complexity to offer benefits that size is unable to. These findings represent a novel contribution to the debate. Before now, studies of time in GP focus on managing the overall run-time and not using it to determine and manage the complexity of GP solutions (models).

Although like [185], this thesis also observes that the evaluation time increases with size, but goes further to investigate and show that there is more to time than size. Chapter 4 theoretically argues why time as a measure of complexity is different from size. Chapter 5 and Chapter 6 demonstrate the benefit of using the evaluation time to control complexity over using size. This chapter uses a practical application of GP to quantify the differences between size-based and time-based methods of managing complexity.

This chapter uses a novel hybridisation of GP and multiple linear regression (MLR) as a platform to show that time can decisively reflect several notions of complexity. The so-called MLR-GP system [205] is made of two components: GP and MLR. In MLR-GP, the GP component evolves a set of features instead of a monolithic model and then hands this set over to MLR that optimally combines these features to produce a model that is a linear combination of the features (illustrated in FIGURE 8.1). The results indicate that the MLR-GP hybrid significantly outperforms both its constituent components (GP and MLR). As the MLR-GP feature adds to a model's learning capacity and their numbers can easily grow in a GP setting, managing this complexity (number of features) becomes necessary. Furthermore, the much-improved training performance, which increases the likelihood of overfitting, makes MLR-GP an appropriate platform to gain insight into the impact of the evaluation time schemes on overfitting.

With the MLR-GP application, this chapter further examines how the evaluation time characterises complexity. In addition, it compares the effect of controlling model complexity using time with that of doing so using size (bloat-control). The choice of bloat-control technique is the Double Tournament [189], which is the best performing technique from Chapter 4, to control size (BC) and control time (TC); using the same mechanism and settings give a fair comparison of the measures of complexity. Additionally, the APGP is used, which was introduced in Chapter 6 and analysed in Chapter 7, to control the evaluation time implicitly. Instead of subjectively penalising the slow evaluation times, the APGP induces a *race* among competing models that offers a competitive advantage to the simpler (fast evaluating) models because they may enter the breeding population (if their accuracy permits) and reproduce before their expensive (slow) counterparts. Altogether, the evaluation time schemes (APGP and TC) on MLR-GP are benchmarked against MLR-GP with bloat-control (titled BC in the results) and MLR-GP without bloat-control (titled STD in the results).

The results, computed over 10 datasets, indicate that although evaluation time correlates with size, its correlation with the number of features is much stronger. This shows that a GP scheme that exploits the evaluation times to reduce complexity in models is *effectively* different from the one that uses size to do the same. This is because each additional feature significantly increases the expense of computing the MLR model - computing the MLR model is a major component of the total computational expense per fitness evaluation. While increasing the number of nodes in the features also increases the computational cost, the results indicate that this cost is smaller than that of adding an extra feature. This delineation of respective expenses is possible for the evaluation time based complexity control but not so for the traditional, size-based complexity control in GP. Consequently, the evaluation time schemes were able to effectively control the number of features in models but the size-based method could not.

These findings practically support the qualitative view that containing evaluation time is about more than just containing size. Therefore, these findings steer the discussion in the field that the evaluation time control can draw benefits, that standard approaches like bloat-control cannot.

The rest of this chapter is organised as follows: Section 8.2 introduces MLR-GP and argues why such systems are effective for GP based regression; Section 8.3 then empirically exemplifies this efficacy; Section 8.4 details the experiments; Section 8.5 presents the results; and finally, Section 8.7 concludes the chapter.

8.2 Enhancing GP with Multiple Linear Regression

This section reviews the literature that shows that hybridisation of GP with various approaches of linear regression produces effective systems for GP based regression. The review justifies the choice of an MLR based GP system later in the chapter to investigate the efficacy of evaluation-time based complexity-control. After reviewing the literature in this section, the chapter later (in Section 8.3.1) also empirically demonstrates that hybridised MLR-GP outperforms both GP and MLR when used individually.

GP systems that are hybridised with statistical and machine learning techniques [205–208] have become increasingly popular lately because they improve the training performance significantly; this is because the traditional GP often underfits the data [209]. After all, it can not efficiently generate numeric constants [210, 211], that is, the coefficients of the evolving models. This happens because, unlike the statistical and numeric methods that use numerical methods to tune the parameters of an otherwise fixed model, GP traditionally relies on evolution alone to evolve both the model as well as its parameters, which is a significant undertaking. The history of GP's struggle with manufacturing the requisite constants is well-documented [196, 210] and dates back to Koza's early work [19]. Hybridising GP with statistical and machine learning (ML) methods help circumvent the problem of tuning constants in GP [212]; in fact, the performance boost can be significant.

Instead of evolving monolithic models as in traditional GP, the GP component of MLR-GP here evolves a set of features: each individual is a collection of features. MLR-GP then hands each such collection over to the MLR component, which now combines these features into a regression model and optimises the coefficients of each feature. Thus the representation of a model in MLR-GP is different from that in standard GP; in fact, *representation learning* [213] is a well-known topic in machine learning. Before providing the details of the adopted

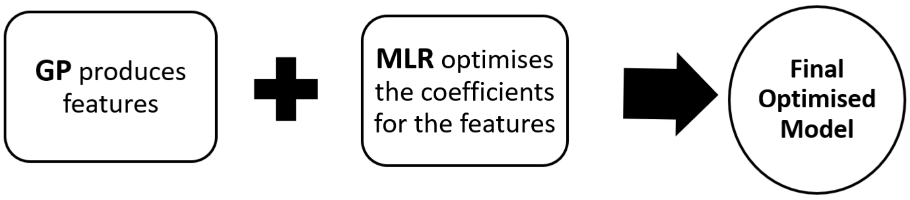

GP produces features
MLR optimises the coefficients for the features
Final Optimised Model

FIGURE 8.1: The MLR-GP process flow.

approach, this section briefly reviews some approaches in the spirit of MLR-GP that have been proposed for addressing the challenge of tuning constants in GP but do so to varying degrees only.

Linear Scaling [211] treats the whole model as a feature and uses linear regression (LR) to find the best coefficient and intercept to minimise the error of a model. Thus, GP finds the correct structure of the model, while LR optimises its overall slope and intercept. This approach improves the model accuracy; however, because linear scaling only optimises the two coefficients (slope and intercept) that are attached to the model, the improvement is limited [205]. Linear Scaling is implemented as follows. Suppose y = the output of a candidate model f on the given data, $\bar{y}$ = mean of y, t = the target (true y) and $\bar{t}$ = mean of t, then linear scaling optimises the slope (b) and intercept a of the model f via the deterministic computations as follows:

$$b = \frac{\sum(t-\bar{t})(y-\bar{y})}{\sum(y-\bar{y})^2}; \tag{8.1}$$

$$a = \bar{t} - b\bar{y}; \tag{8.2}$$

such that the optimised function (f') is:

$$f' = bf + a \tag{8.3}$$

Another approach combines some or all of the models in a population to form an ensemble-model [207, 214–216]; therefore, this approach also treats the individuals in the population as features. Typically, one ensemble-model is produced per generation and its quality heavily depends on the makeup of the current population. However, the relationship between the evolutionary processes (selection and variation) that are used to improve an individual and how the ensemble model improves is unclear. For example, the evolutionary search process tends to make the individuals of a population converge to have a similar makeup; thus, this leads to collinearity of the features, which in turn limits the representation learning. Therefore, several approaches have been proposed to overcome the challenge of collinearity in the produced features and for enabling the evolutionary process to update the features predictably and productively. For example, [215] uses the ϵ-lexicase selection to build a method that encourages the production and survival of diverse features; as a result, semantically unique individuals are preserved and a population of uncorrelated features is maintained. Another intervention avoids or limits the use of the evolutionary updating of the models [214, 216]; for instance, [214] simply produces a randomly generated population of models (features) and formulates the final model without evolutionary updating. However, the capacity of the individual features is limited because they do not get developed by the evolutionary process.

Instead of working with a population of models, another set of approaches looks within a model (program) to identify useful building blocks (features) that can in turn be exchanged during crossover for producing high-quality offspring [207, 217, 218]. For example, some approaches of GP-based Program Synthesis identify useful features (blocks within evolving programs) with the help of Machine-Learning based classification methods such as Decision Trees [217][218]. However, these approaches typically work with problems of non-continuous (e.g. Boolean) nature. Instead, an example that is a close contender to the proposed MLR-GP method, is the multiple regression GP (MRGP) [207].

MRGP considers the output of every tree node as a distinct feature for MLR to optimise. Therefore, a GP tree contains many *overlapping* features because the output of one feature inputs into another feature.

The MRGP [207] method assumes that all subexpressions of a model can be treated as features. Thus, the method involves stepping through a hierarchical tree, stopping at each node, evaluating and recording the output; each node is treated as a feature and its output is the corresponding feature value. MLR is then applied to a matrix containing all such feature values and the target variable values. Note, every feature, a node in the GP tree, hierarchically depends upon the lower level features (that is, subtrees located beneath the node); in such a system crossover can be disruptive because swapping a feature out in turn impacts the evaluation of all features higher up the hierarchy. Thus the features in such a system are hierarchically *tangled*.

Instead, the proposed MLR-GP method in this study keeps the features untangled and allows GP to designate different parts of the tree structure as distinct and transferable features; the details of the MLR-GP are provided in Section 8.2.1. With MLR-GP, the subtree crossover can swap both complete features or subtrees within the features. This makes the MLR-GP more robust to the destructive effect of crossover [68, 219]; the results of the validation and analysis in Section 8.3 confirms that MLR-GP improves crossover performance.

Note, the purpose of using MLR-GP here is to showcase how containing evaluation time practically realises benefits that containing sizes cannot. Therefore, Section 8.5.1 shows that the time cost of a model with a high number of features is different from that of a model that is just big in size (contains more nodes). Therefore, a complexity control method that can differentiate between time and size is better able to control these various aspects of complexity of such a system, but a size-based control cannot.

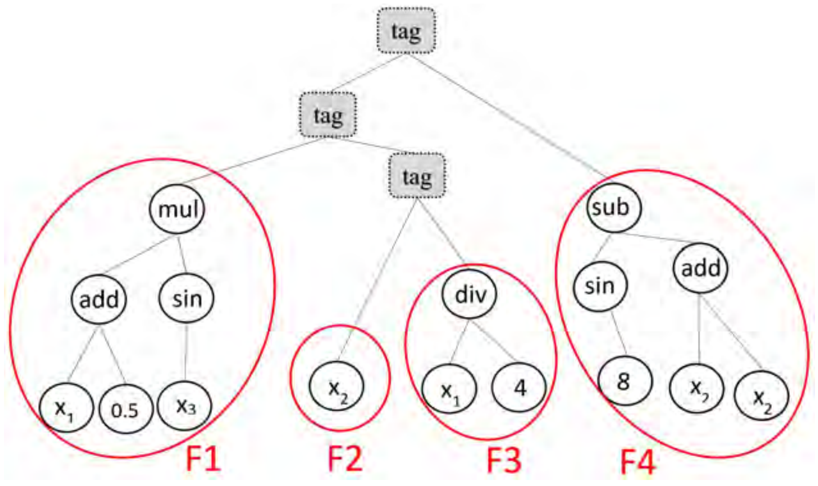


FIGURE 8.2: An MLR-GP representation of a model. The nodes labeled *tag* are placeholders that are not evaluated; and the subtrees F1 - F4 are treated as features.

8.2.1 The Proposed Multiple Linear Regression Genetic Programming (MLR-GP)

Key aspects of the proposed MLR-GP method include: (1) changes to the tree structure of the expressions representing models and (2) changes to the way the expressions are evaluated.

MLR-GP Representation and Genetic Operators

The MLR-GP uses a modified version of the standard tree representation of symbolic regression models; an example is shown in FIGURE 8.2. A new type of node, labeled *tag*, is introduced to act as a placeholder, that is not evaluated. The placeholder was defined with an arity of two. This enables it to either branch out further creating more placeholders or contain a feature directly below it. As marked in FIGURE 8.2, F1, F2, F3 and F4 are subtrees that act as independent features.

The only constraint required for the MLR-GP tree to remain valid is that the *tag* nodes can only have *tag* nodes as parent nodes. The genetic operators (mutation and crossover) must obey this constraint.

Besides respecting the above constraint, genetic operators (crossover and mutation) proceed as usual. This means that mutations can change a subtree within a feature or replace an entire feature (or even a set of features that are all rooted together). Regardless of whether the replacing structure is a feature or a standard subtree, a valid position is randomly selected. Similarly, the crossover is enabled on all nodes that will lead to a valid structure. Using FIGURE 8.2 as an example, all the features (F1 to F4) can be replaced with a constant, a standard subtree, another feature, or even a set of features. Inside the features, all changes are allowed as long as they obey the one constraint mentioned earlier, that is, a tag node can only be placed at a position where its parent is another tag node.

Evaluating an MLR-GP Model

The fitness evaluation in MLR-GP begins by identifying the distinct features in a tree; these are the subtrees rooted at a *tag* node. Then, these features are evaluated for each input data point. The set of these evaluations is recorded in a matrix; the final column of the matrix contains the target values (for the outcome variable as specified in the problem dataset). This matrix then becomes the *design matrix* for MLR; MLR is applied next to produce a linear model that uses all of the features in the design matrix.

For example, four features appear in the example in FIGURE 8.2. Each identified feature is then evaluated against the training data. The expression in the example will return the following as the final model:

$$\mathrm{Y} = \beta_0 + \beta_1 \mathrm{F}_1 + \beta_2 \mathrm{F}_2 + \beta_3 \mathrm{F}_3 + \beta_4 \mathrm{F}_4 \tag{8.4}$$

where Y is the output of the model, β_0 is the intercept, β_1 to β_4 are the coefficients of features F_1 to F_4 respectively. The training and test-set accuracy of the final model is then recorded.

8.3 Validation of the MLR-GP

As mentioned earlier in Section 8.2, before experimenting with the time-control methods, the impact of hybridising MLR with GP is examined; that is, to determine if MLR-GP significantly outperforms both standard GP and MLR individually [205]. Therefore, the performance of MLR-GP is compared with standard GP (GP), MLR, and the popular Linear Scale GP (LS-GP) [211]; LS-GP (introduced in Section 8.2) is included because it is a popular hybridisation of GP with regression that improves upon GP's performance with minimal overhead. The comparison of performances is done on some challenging test problems from Chapter 6 where standard GP could achieve no more than 17% training accuracies. The experiments use the same settings as described in Chapter 6.

Moreover, the results include a report on the performance of the crossover operator on the MLR-GP system. The report shows that an extra benefit of using MLR-GP is that the normally destructive nature of crossover in standard GP significantly improves with MLR-GP. As such, the MLR-GP system provides a viable medium for exchanging the building blocks in an evolving population; Section 8.3.2 gives the background to the disruptive nature of crossover in GP and explains the significance of the results in that background. Overall, the rationale behind this section is to show that MLR-GP is a worthy system to investigate with time-control methods.

8.3.1 Accuracy Improvement in MLR-GP

FIGURE 8.3 compares the training accuracy of the models produced by MLR-GP, standard GP (GP) and GP with linear scaling (LS-GP). The results clearly show that MLR-GP significantly improves the training fitness scores; in fact, MLR-GP at 5^{th} generation outperforms GP at the 30^{th} generation, that is, GP at a far more advanced stage. This supports the earlier statement that standard GP often underfits and MLR-GP can ameliorate that.

Table 8.1 also records the performance of MLR and the percentage improvement produced by MLR-GP over all benchmarks considered here. MLR-GP improved the average training scores by: between 178% to 226% over MLR; between 261% to 730% over LS-GP; and between 448% to 2,750% over GP methods.

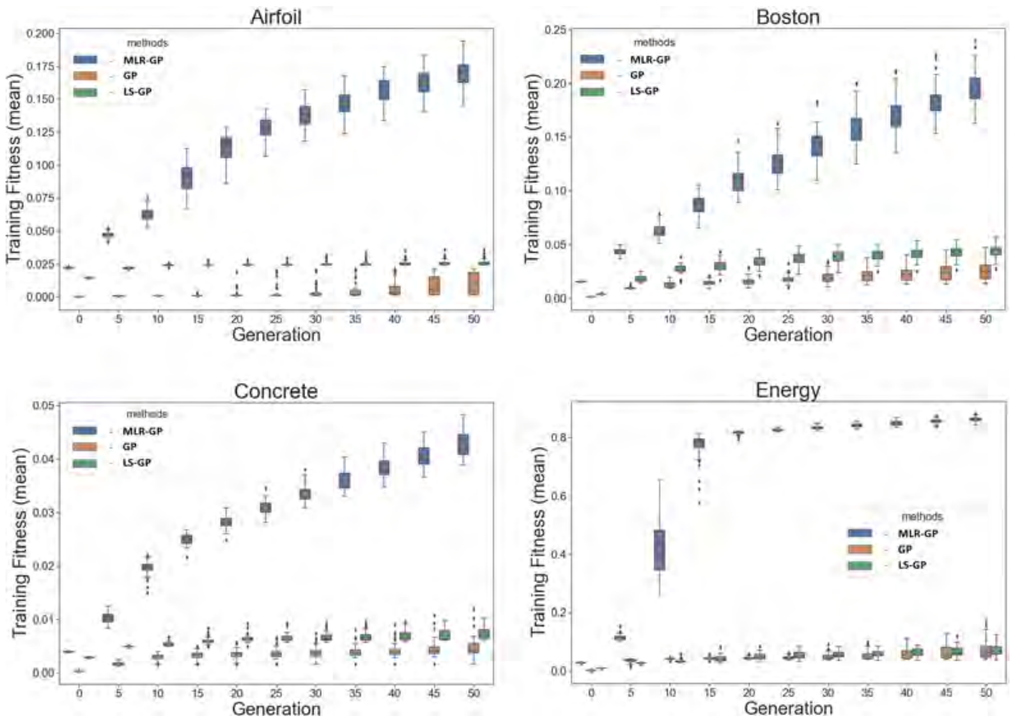


FIGURE 8.3: The training fitness values by generation of MLR-GP, GP, and LS-GP are compared. MLR-GP has improved training to the point of overfitting the data.

	Mean Training Fitness				MLR-GP Improvement		
Problem ID	MLR-GP	MLR	LS-GP	GP	over MLR	over LS-GP	over GP
Airfoil	0.114	0.041	0.024	0.004	178 %	375 %	2,750%
Boston	0.119	0.041	0.033	0.017	190 %	261 %	600%
Concrete	0.029	0.009	0.006	0.003	222 %	383 %	867%
Energy	0.307	0.094	0.037	0.056	226 %	730 %	448%

TABLE 8.1: The training accuracy gain by MLR-GP over MLR, LS-GP and GP; both the mean training fitness scores and the percentage improvement by MLR-GP are detailed.

8.3.2 Crossover Performance in MLR-GP

To improve the population of candidate solutions, GP relies on operators inspired by natural evolution. Crossover is one of the most frequently used operators. It exchanges genetic material between better performing individuals (models) from the current generation to produce the next generation. However, previous work [219][68] has shown that crossover in GP is quite inefficient because it produces a high percentage of poor offspring (offspring that are significantly worse than their parents). However, as explained below, MLR-GP can significantly improve the nature of crossover.

Unlike the standard GP tree that is a monolithic structure, MLR-GP uses independent sub-structures (features) to make up a model. These substructures, when exchanged into another tree, are then *tuned* by the MLR to according to the needs of the model encoded by that tree. This is unlike crossover in standard GP where the incoming tree is simply patched-in at a random location without any tuning; naturally, such a random patching is unlikely to produce a better offspring. Therefore, MLR-GP can be expected to be more robust against the destructive effect of the crossover operator. To gain insight into this important operator and the MLR-GP,monitored the effect of crossover during the evolution. The effect of crossover is measured as the percentage difference in the fitness of an individual before and after crossover:

$$\frac{F_{After} - F_{Before}}{F_{Before}} \times 100, \tag{8.5}$$

where F_{Before} is the fitness (training accuracy) before crossover of the parent that accepts an incoming subtree and F_{After} is its fitness after crossover. This fitness differential is computed for both offspring resulting from the crossover.

Table 8.2 compares the percentages of *negative* crossover operations, which degrade the fitness values (less than zero percent improvement). As anticipated, MLR-GP significantly improves the crossover efficiency.

The effects of crossover in GP, LS-GP and MLR-GP are visualised in FIGURE 8.4 in the form of 3D histograms that illustrate how crossover affects fitness (accuracy). The results are plotted for all the generations: For all the test problems, crossover with MLR-GP was clearly less disruptive because the incidences of disimprovements with MLR-GP are clearly less than those with the counterparts. While the MLR-GP crossover differentials are mostly centred around the neutral point (zero), the corresponding results for standard GP and LS-GP have a larger proportion in the negative region.

Problem ID	GP	LS-GP	MLR-GP
Airfoil	73%	69.0%	55%
Boston	71%	76.0%	51%
Concrete	68%	70.0%	50%
Energy	72%	78.0%	35%

TABLE 8.2: Comparison of percentage of crossover with negative improvements (a deterioration). MLR-GP is having fewer negative improvements on all test problems.

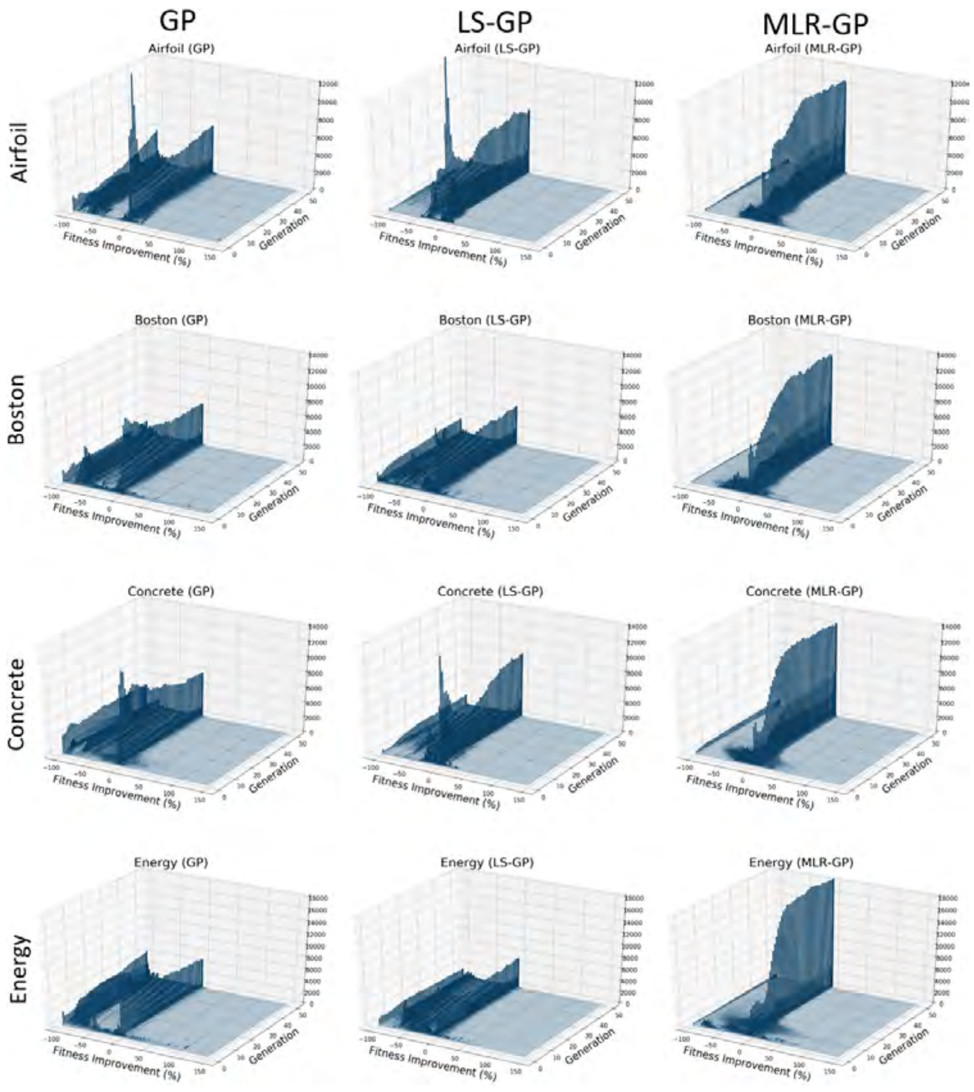


FIGURE 8.4: The 3D histograms show fitness (accuracy) improvement after crossover by generation. MLR-GP (column 3) shows fewer negative improvements (deterioration) than GP and LS-GP.

8.4 Experiments with Time-Control Methods

Using MLR-GP as a benchmark, the experiments compare the performance of the methods that contain evaluation times (APGP and TC) with that of MLR-GP without complexity control (MLR-GP) and MLR-GP with an effective bloat-control technique (MLR-GP-BC).

8.4.1 Methods for Controlling Complexity Using Evaluation Time

The two approaches that are used to control complexity using the evaluation times are as follows:

1. *Explicit Time Control (TC)*: As introduced in Chapter 5, Time-control (TC) leverages the mechanisms of established bloat-control techniques but instead of containing size growth, it contains the evaluation times. *Double Tournament* (DT) version is the chosen method; it has been the best performing bloat-control technique so far in the experiments in this thesis. Note, the same technique will be used to control size in a contending method.

2. *Implicit Time Control with Asynchronous Parallel GP (APGP):* As introduced in Chapter 6 and analysed in Chapter 7, the APGP implicitly manages the evaluation times of models by inducing a race condition in the evolution process; it does not aggressively target models with high evaluation times but gently encourages simplicity where possible. The previous results show that APGP attained the highest training accuracy against its counterparts.

8.4.2 The Contending GP Methods

The two evaluation time methods, that are detailed in Section 8.4.1, will be compared with the following methods in the experiments:

1. *MLR-GP without Bloat-control (STD):* The MLR-GP with no complexity control (no bloat-control nor time-control) is run on the problems for comparison.

2. *MLR-GP with Bloat-Control (BC):* The choice of bloat-control technique is Double Tournament [188, 189]. From the results of Chapter 5, this is the

best bloat-control technique that balances accuracy and bloat-control; it is also considered a good technique in [188][189][220].

3. *MLR-GP with Adjusted R^2 for Fitness (AR2):* Since the complexity control with the evaluation time schemes in MLR-GP contains the growth in the number of features (amongst other things), this study also considers other statistical measures that similarly control the growth of features; the Adjusted R^2 as a fitness measure is an established statistical measure that has that effect. The purpose of this inclusion is not to outperform the use of Adjusted-R^2 but to show how time-control methods, that are not designed just for the regression domains, can still produce an effect that is similar to that produced by the conventional human wisdom specialised in a particular problem domain. Note, rediscovery of such human-designed effects has been a popular application of Genetic Programming, and one of the principal reasons behind its early popularity [221].

   Unlike the well-known R^2, the Adjusted R^2 [222] is a performance measure used in MLR that balances out the accuracy of a model with the number of features it uses. R^2 (also known as the coefficient of determination) is a measure of how well a model explains the response variable. The closer its value is to 1.0, the better the model is. A shortcoming of using R^2 as a fitness measure in MLR-GP is that it allows adding features that contribute very little or nothing to the fitness of a model. Adjusted R^2 overcomes this shortcoming by considering the contribution of additional features. Whenever a new feature is added to a linear model, the value of Adjusted R^2 increases only when the additional feature enhances the R^2 more than one would expect to see by chance; otherwise, Adjusted R^2 decreases. Using the Adjusted R^2 in the fitness function of MLR-GP discourages the growth in the number of features (subtrees) unless these features significantly improve the model performance.

The selected version of Adjusted R^2 used here is the Wherry/McNemar version [223]; the formula is as follows:

$$1 - \frac{(1 - R^2)(n - 1)}{(n - k - 1)}, \tag{8.6}$$

where k denotes the number of features and n denotes the number of instances in the data.

To compare the accuracy of AR2 with with other methods, the Normalised Mean Squared Error (NMSE) is also captured.

8.4.3 Test Problems

Ten widely used and publicly available datasets were selected as test problems. As in Section 5.4.1, the selection is based on the recommendations in the studies [192, 193] that surveyed the common practice of selecting test problems in GP and collected feedback from the GP community. The recommendation includes the sources of benchmarking datasets, guidelines for choosing benchmarks (for example, variety, relevance, and fast runtime) and problems to avoid (provided in the blacklist they collated). The data for problems 1 - 7 are available at the Penn machine learning benchmark (PMLB) [224] and the data for problems 8 - 10 are available at [194]. The problems are summarised in Table 8.3.

Before using the selected problems to study time control on the MLR-GP, the investigation checks to determine whether MLR-GP outperforms standard GP on the selected problems. The results show that MLR-GP increases fitness scores and complexity. FIGURE 8.5 shows the mean training and test fitness scores by generation for both GP and MLR-GP. The figure shows that MLR-GP significantly improves the training accuracy over all the test problems. However, the results also reveal that this improvement in training accuracy also brings overfitting well before the 30th generation (see FIGURE 8.5). Therefore, and also due to a much-improved training performance achievable within 30 generations, the subsequent experiments in this chapter only run until the 30th generation.

Problem ID	Data-Set Name	Variables	Instances
1	027 ESL	3	488
2	207 AutoPrice	14	159
3	522 pm10	6	500
4	547 NO2	6	500
5	560 bodyfat	13	252
6	666 rmftsa ladata	9	508
7	690 Visualizing galaxy	3	323
8	Boston Housing	13	506
9	Concrete Strength	8	1030
10	Diabetes	10	442

TABLE 8.3: Test problems for the MLR-GP experiment.

8.4.4 Configurations and Settings

The basic parameters are detailed in Table 8.4. Other settings are as follows:

- *Population Initialisation*: The Fixed Length Initialisation (FLI) that was introduced in Chapter 5 is used. FLI creates a population that contains individuals of identical size but of diverse make-up. This enables the evaluation time methods to discriminate between the functional complexities of individuals. However, this does not hamper other GP methods: as seen in Chapter 5 and Chapter 6, FLI improves the performance of a variety of GP methods.

- *Divide-by-zero error*: Individuals that contain fractions that result in divide-by-zero errors are assigned a zero fitness-score to make them noncompetitive; previous studies [196] have shown that the standard practice of using *protected* operators (that replace the offending fraction with just the numerator or some other constant) can lead to overfitting.

- *Data Splitting*: 70% of a dataset is allocated for training and 30% for testing using random selection without replacement.
- *Concurrency of APGP*: 250 parallel threads, representing 50% of the population size, are used.
- *Replacement Scheme*: As APGP uses a steady-state replacement scheme, all the contending methods use same.
- *Fitness function*: is a maximisation fitness function that uses the normalised mean squared error (NMSE); the NMSE formula is as follows:

$$\frac{1}{1+\frac{1}{n}\Sigma_{i=1}^{n}(y_i-\hat{y}_i)^2}. \tag{8.7}$$

Parameter	Setting
No. of runs	50
Size of population	500
Generations	30 (15,000 evaluations)
Initialisation	Fixed Length Initialisation (FLI) (length = 10)
Tree depth	max = 17
Operator types	crossover = One point; Point mutation
Operator settings	crossover = 0.9 ; mutation = 0.1
Function set	$+, -, *, /,$ sin, cos, neg
Constants (ERC)	∣ERC∣ = 100 (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament (size = 3)
Replacement	steady state; inverse tournament (size = 5)

TABLE 8.4: Experimental settings for the MLR-GP experiment.

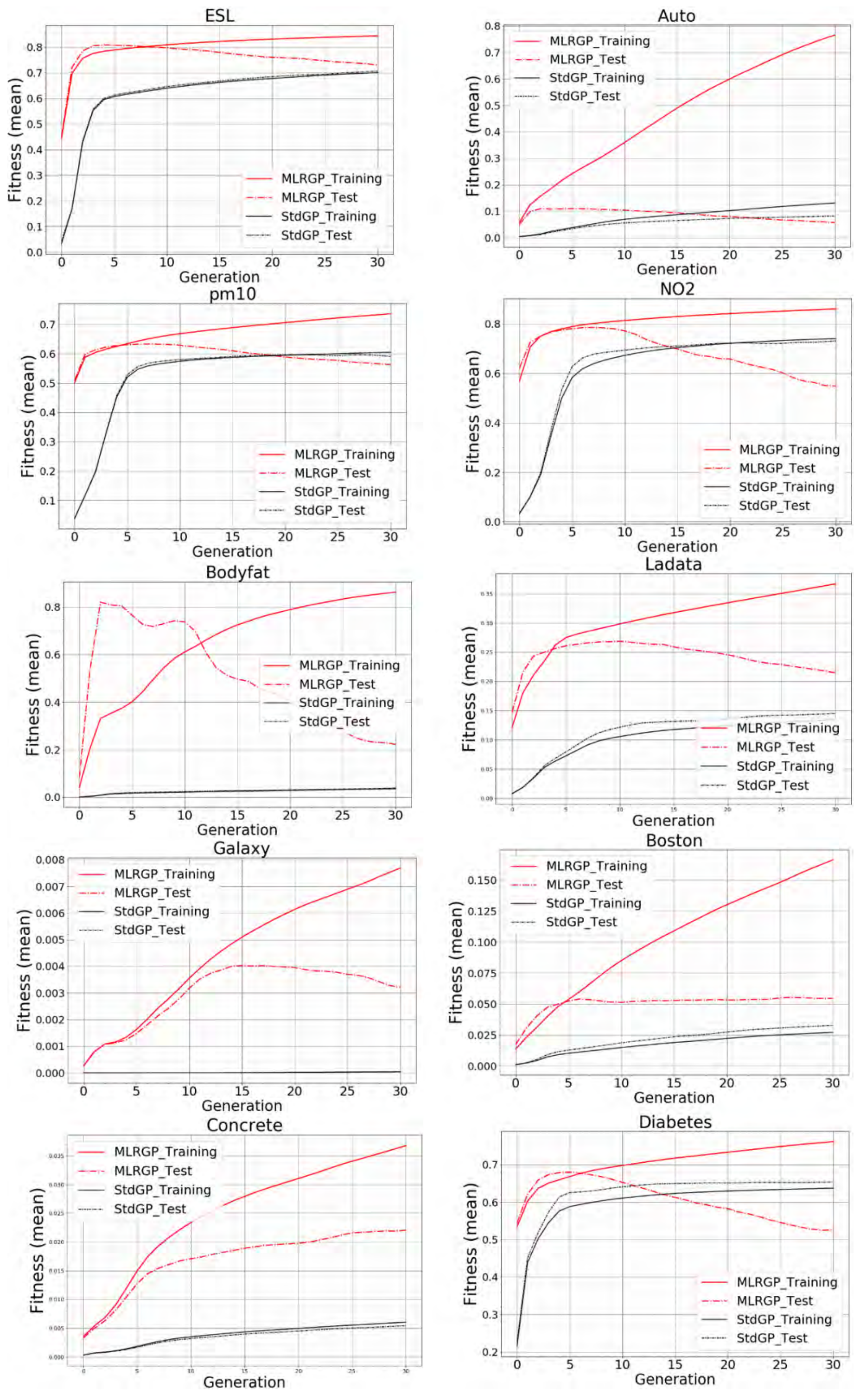


FIGURE 8.5: The training and test fitness values of MLR-GP and standard GP are compared. For all the test problems, MLR-GP has improved training to the point of overfitting the data.

8.5 Results and Discussion

Before discussing the effect of using evaluation time to control complexity in the MLR-GP application, this section analyses the result to answer the question: Does the evaluation time reflect more than size?

8.5.1 Evaluation Time Characterises Complexity Beyond Size

As discussed in Section 8.2, the MLR-GP method finds the features that make up an individual model. Also, the method evaluates each feature independently on the training data and stores the evaluation results in a column of the design matrix; MLR then uses this design matrix to generate the regression model. This means that increasing the number of features increases the size of the design matrix, which must increase the evaluation times.

An MLR-GP feature can add to a model's learning capacity. Unlike standard GP where adding a node (or a subtree) to a monolithic tree can deteriorate its fitness (because genetic operators do not test the suitability of the additional nodes to the receiving tree), MLR can cancel out the effect of any additional feature if it decreases R^2 by simply assigning it a zero coefficient. While this ensures that crossover in MLR-GP is more constructive than in standard GP, it also means that MLR-GP models can grow unnecessarily complex because unproductive features can be added without negatively affecting the fitness of a model. Therefore, managing the number of features is important. However, a bloat-control method that discourages the addition of nodes can be oblivious to it because a small tree can still contain a large number of features and vice versa.

The results show that, unlike the bloat-control methods, the time control methods detect the contribution of the number of features and therefore by containing the evaluation times, it contains the number of features as well. The final populations that MLR-GP produced show a stronger correlation between the evaluation time and the number of features (average of 0.996) than between evaluation time and the sizes (average of 0.843); see the plots and correlation values in FIGURE 8.6 and FIGURE 8.7.

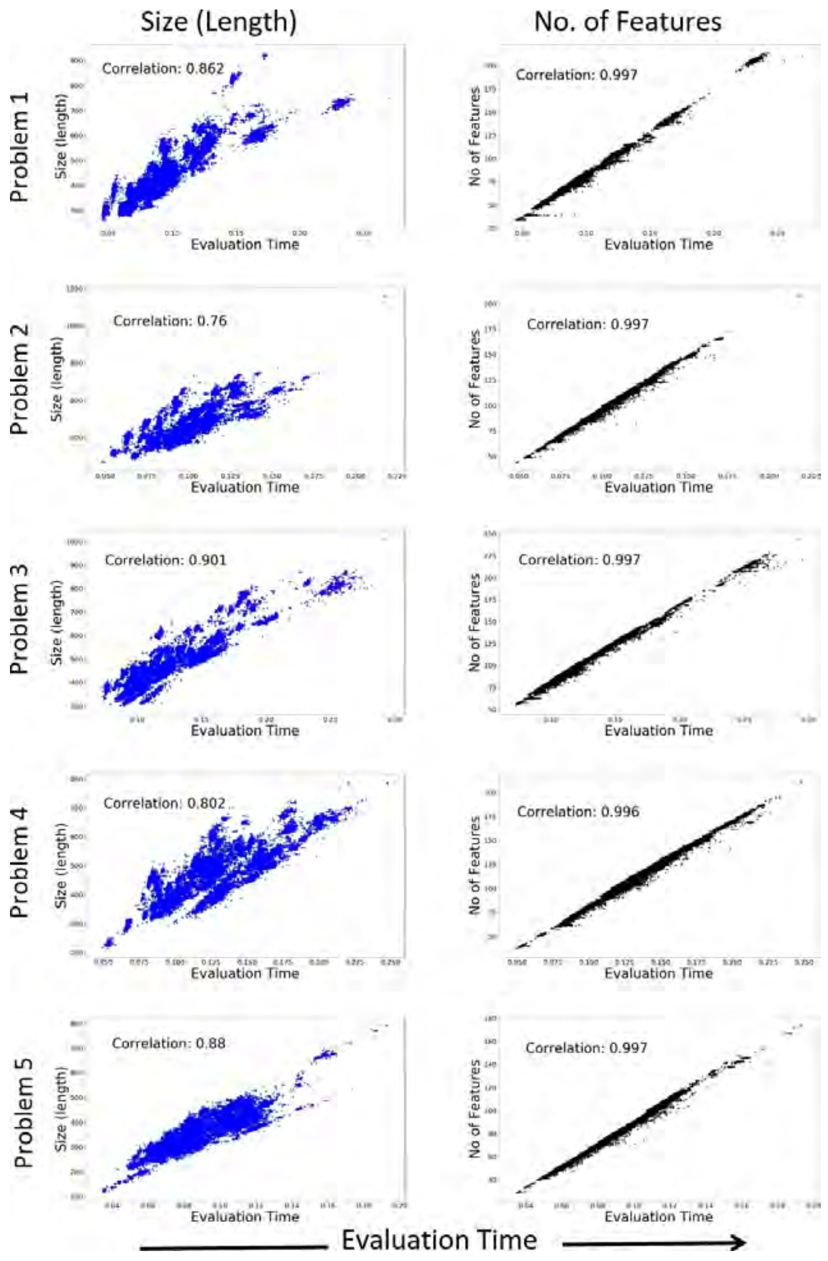


FIGURE 8.6: Correlations between evaluation time and size and between evaluation time and the number of features for Problem 1 - 5.

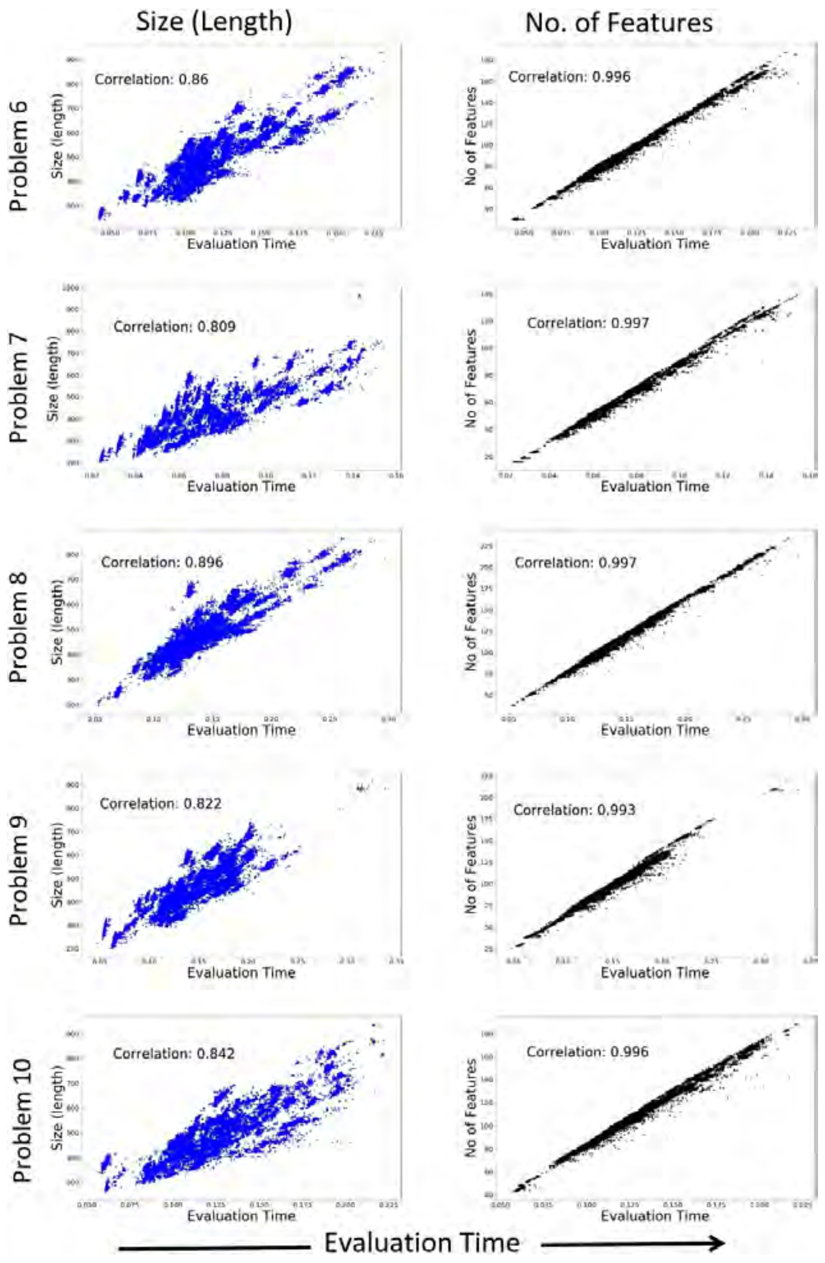


FIGURE 8.7: Correlations between evaluation time and size and between evaluation time and the number of features for Problem 6 - 10.

Therefore, this finding implies that discouraging the evaluation time in the MLR-GP will discourage all that contributes to the evaluation times, including the size, the number of features, and the computational complexity of the components (as discussed in Section 4). Instead, discouraging size alone (bloat-control) ignores the complexity of the components and the number of features within the individual. After all, two individuals of the same size can have different numbers of features and exhibit different functional behaviours. This underscores that time is not size, as motivated in Chapter 4; instead, it is a richer characterisation of complexity in GP.

The following section analyses the practical implication of using the evaluation to control complexity in the MLR-GP application.

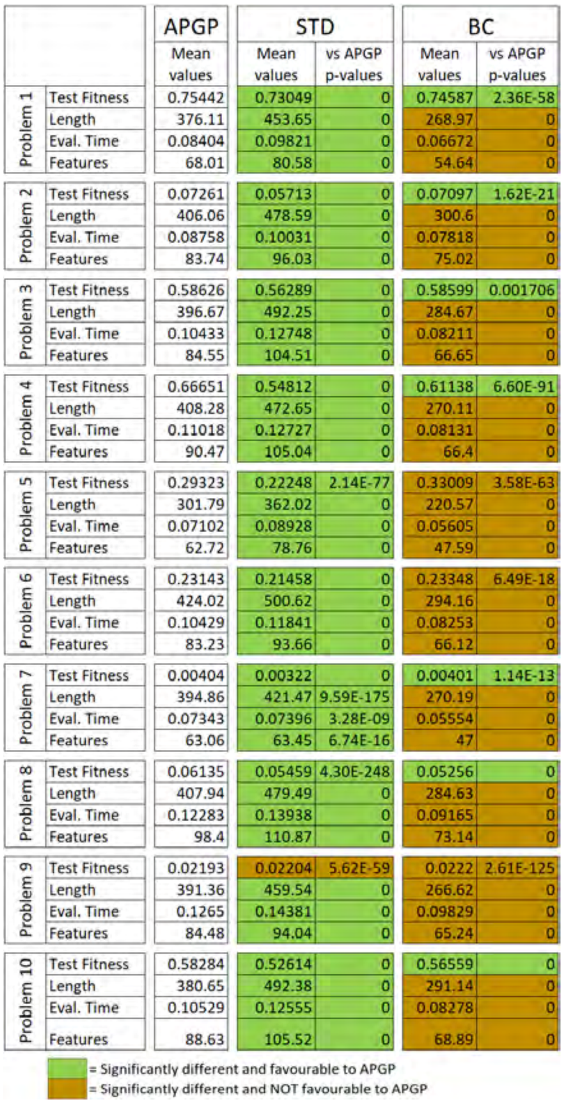
8.5.2 Complexity Control and Generalisation Ability of Models

The effects of time-control and size-control to manage the accuracy and complexity of models are analysed next. For accuracy, the test fitness scores are used, which reflect the generalisation ability of the models; for complexity, the analyses compare differences in size, evaluation times, and the number of features.

The test for the significance of the differences in the final populations of the competing methods uses the Mann-Whitney U test; the details (including mean values and p-values) are captured in the colour coded tables in FIGURE 8.8 and FIGURE 8.9. The green coloured cells represent the results that favour the time-control method and where the difference is significant. The brown coloured cells indicate where time-control is significantly worse than the other methods. The yellow coloured cells indicate that the difference is not statistically significant.

In terms of accuracy (test fitness), the time-control methods (TC and APGP) outperformed STD and BC on 9 out of 10 and 7 out of 10 problems, respectively. In addition, when comparing the solution simplicity, both TC and APGP consistently outperformed STD (evaluation time, size, and the number of features).

However, while TC also outperformed BC (bloat-control method), APGP could not.

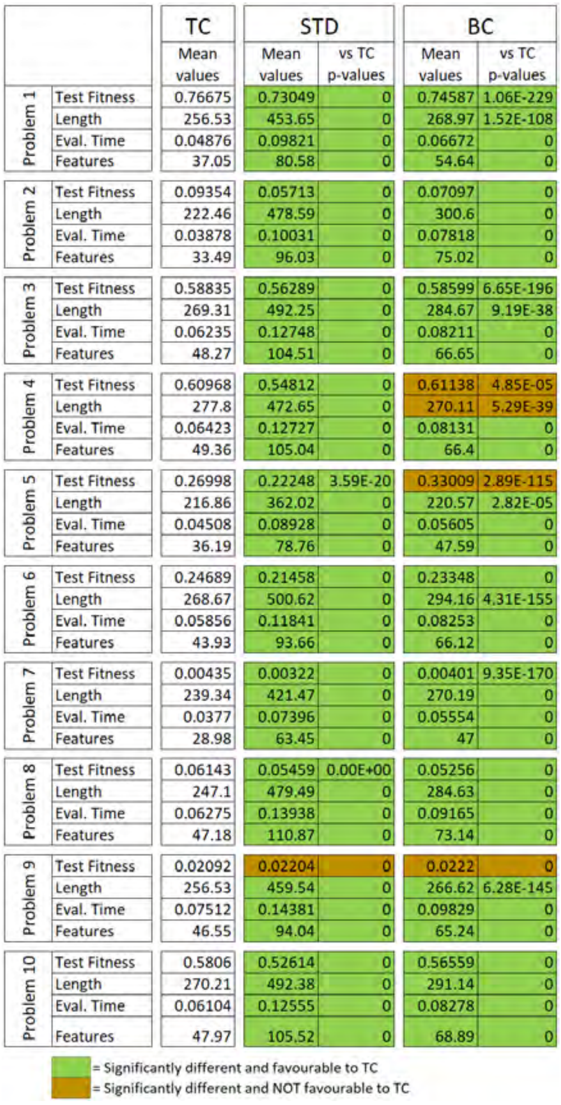


		APGP	STD		BC	
		Mean values	Mean values	vs APGP p-values	Mean values	vs APGP p-values
Problem 1	Test Fitness	0.75442	0.73049	0	0.74587	2.36E-58
	Length	376.11	453.65	0	268.97	0
	Eval. Time	0.08404	0.09821	0	0.06672	0
	Features	68.01	80.58	0	54.64	0
Problem 2	Test Fitness	0.07261	0.05713	0	0.07097	1.62E-21
	Length	406.06	478.59	0	300.6	0
	Eval. Time	0.08758	0.10031	0	0.07818	0
	Features	83.74	96.03	0	75.02	0
Problem 3	Test Fitness	0.58626	0.56289	0	0.58599	0.001706
	Length	396.67	492.25	0	284.67	0
	Eval. Time	0.10433	0.12748	0	0.08211	0
	Features	84.55	104.51	0	66.65	0
Problem 4	Test Fitness	0.66651	0.54812	0	0.61138	6.60E-91
	Length	408.28	472.65	0	270.11	0
	Eval. Time	0.11018	0.12727	0	0.08131	0
	Features	90.47	105.04	0	66.4	0
Problem 5	Test Fitness	0.29323	0.22248	2.14E-77	0.33009	3.58E-63
	Length	301.79	362.02	0	220.57	0
	Eval. Time	0.07102	0.08928	0	0.05605	0
	Features	62.72	78.76	0	47.59	0
Problem 6	Test Fitness	0.23143	0.21458	0	0.23348	6.49E-18
	Length	424.02	500.62	0	294.16	0
	Eval. Time	0.10429	0.11841	0	0.08253	0
	Features	83.23	93.66	0	66.12	0
Problem 7	Test Fitness	0.00404	0.00322	0	0.00401	1.14E-13
	Length	394.86	421.47	9.59E-175	270.19	0
	Eval. Time	0.07343	0.07396	3.28E-09	0.05554	0
	Features	63.06	63.45	6.74E-16	47	0
Problem 8	Test Fitness	0.06135	0.05459	4.30E-248	0.05256	0
	Length	407.94	479.49	0	284.63	0
	Eval. Time	0.12283	0.13938	0	0.09165	0
	Features	98.4	110.87	0	73.14	0
Problem 9	Test Fitness	0.02193	0.02204	5.62E-59	0.0222	2.61E-125
	Length	391.36	459.54	0	266.62	0
	Eval. Time	0.1265	0.14381	0	0.09829	0
	Features	84.48	94.04	0	65.24	0
Problem 10	Test Fitness	0.58284	0.52614	0	0.56559	0
	Length	380.65	492.38	0	291.14	0
	Eval. Time	0.10529	0.12555	0	0.08278	0
	Features	88.63	105.52	0	68.89	0

🟩 = Significantly different and favourable to APGP
🟧 = Significantly different and NOT favourable to APGP

FIGURE 8.8: Result of the significance test for the difference between the MLR-GP with APGP and MLR-GP with bloat-control (BC) and without (STD).

However, note that – besides using different complexity measures – TC and BC use the same mechanics; they both use the same complexity-control technique and settings to control their respective complexity measures. Therefore, their comparison is more equitable, and any difference in their results is attributable to the complexity measure. As shown in FIGURE 8.9, TC outperformed BC by containing the number of features and evaluation times in all 10 problems and containing the size in 9 out of 10 problems. In addition, TC produced solutions with better test scores (reflecting generalisation) in 7 out of 10 problems. Hence, the results show that the complexity-control with evaluation time serves its purpose, that is, it produces simple yet accurate models.



		TC	STD		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
Problem 1	Test Fitness	0.76675	0.73049	0	0.74587	1.06E-229
	Length	256.53	453.65	0	268.97	1.52E-108
	Eval. Time	0.04876	0.09821	0	0.06672	0
	Features	37.05	80.58	0	54.64	0
Problem 2	Test Fitness	0.09354	0.05713	0	0.07097	0
	Length	222.46	478.59	0	300.6	0
	Eval. Time	0.03878	0.10031	0	0.07818	0
	Features	33.49	96.03	0	75.02	0
Problem 3	Test Fitness	0.58835	0.56289	0	0.58599	6.65E-196
	Length	269.31	492.25	0	284.67	9.19E-38
	Eval. Time	0.06235	0.12748	0	0.08211	0
	Features	48.27	104.51	0	66.65	0
Problem 4	Test Fitness	0.60968	0.54812	0	0.61138	4.85E-05
	Length	277.8	472.65	0	270.11	5.29E-39
	Eval. Time	0.06423	0.12727	0	0.08131	0
	Features	49.36	105.04	0	66.4	0
Problem 5	Test Fitness	0.26998	0.22248	3.59E-20	0.33009	2.89E-115
	Length	216.86	362.02	0	220.57	2.82E-05
	Eval. Time	0.04508	0.08928	0	0.05605	0
	Features	36.19	78.76	0	47.59	0
Problem 6	Test Fitness	0.24689	0.21458	0	0.23348	0
	Length	268.67	500.62	0	294.16	4.31E-155
	Eval. Time	0.05856	0.11841	0	0.08253	0
	Features	43.93	93.66	0	66.12	0
Problem 7	Test Fitness	0.00435	0.00322	0	0.00401	9.35E-170
	Length	239.34	421.47	0	270.19	0
	Eval. Time	0.0377	0.07396	0	0.05554	0
	Features	28.98	63.45	0	47	0
Problem 8	Test Fitness	0.06143	0.05459	0.00E+00	0.05256	0
	Length	247.1	479.49	0	284.63	0
	Eval. Time	0.06275	0.13938	0	0.09165	0
	Features	47.18	110.87	0	73.14	0
Problem 9	Test Fitness	0.02092	0.02204	0	0.0222	0
	Length	256.53	459.54	0	266.62	6.28E-145
	Eval. Time	0.07512	0.14381	0	0.09829	0
	Features	46.55	94.04	0	65.24	0
Problem 10	Test Fitness	0.5806	0.52614	0	0.56559	0
	Length	270.21	492.38	0	291.14	0
	Eval. Time	0.06104	0.12555	0	0.08278	0
	Features	47.97	105.52	0	68.89	0

🟩 = Significantly different and favourable to TC
🟧 = Significantly different and NOT favourable to TC

FIGURE 8.9: Result of the significance test for the difference between MLR-GP with Time-control (TC) and MLR-GP with bloat-control (BC) and without (STD).

While the TC and APGP both produce more accurate models than do STD and BC, TC controls complexity more aggressively and – on these problems – more successfully. These results contrast with those in Chapter 6 where APGP was both more accurate and faster to train. Therefore, while one cannot conclusively pinpoint a winner among the two methods, it is encouraging to note that the two methods generalise consistently better than their counterparts.

8.5.3 Comparing Adjusted R^2 with the Evaluation Time Schemes

FIGURE 8.11 shows the result of the significance test for the difference between *MLR-GP with Adjusted* R^2 (AR2) and the time-control methods (TC and APGP). Both TC and APGP produced significantly better test fitness scores in 6 out of 10 tests each. However, AR2 produced the simple solutions; it produced simpler solutions (size, evaluation time, and number of features) than APGP in 10 out of 10 problems. Against TC, AR2 produced simpler solutions as follows: 9 out of 10 problems for size, 7 out of 10 problems for evaluation times and 8 out of 10 problems for number of features.

Note, however, as illustrated in FIGURE 8.10, AR2 and TC are closer than AR2 and APGP; in fact, there is some overlap in the error bars of AR2 and TC (Problem 2, 7 and 8) charts. This is because much like AR2, TC effectively discourages the addition of new features because they increase the computational expense unless they significantly improve the training accuracy. The sensitivity of the evaluation time (as employed by time control methods) to the number of features of a model is supported by the very strong positive correlation between the evaluation time and the number of features in all test problems (reminder: FIGURE 8.6 - 8.7). The slightly more aggressive suppression of the number of features by AR2 is not a surprise: AR2 by design penalises additional features unless they significantly improve R^2. Instead, TC and – more gently – APGP promote accuracy and simplicity simultaneously without reducing the fitness score of a model at any point.

Essentially, the time control methods with MLR-GP have produced a behaviour that approximates the effect of Adjusted R^2 by employing the evaluation time to control complexity. Note, this is significant because while the Adjusted R^2 is the result of a carefully crafted human design to achieve a specific aim, the time control methods are general. By design, Adjusted R^2 targets features in a manner similar to how bloat-control only targets size; instead, time-control targets several notions of complexity and is broadly applicable. Moreover, the ability of GP to produce phenomena resembling or outperforming human interventions has already popularised GP [225] [226], and has in fact inspired annual competitions [227]. Consequently, the result in this section adds to the conversation.

Moreover, the time-control methods on MLR-GP can incorporate Adjusted R^2. Section 8.6, examines how leveraging Adjusted R^2 impacts the performance of time-control.

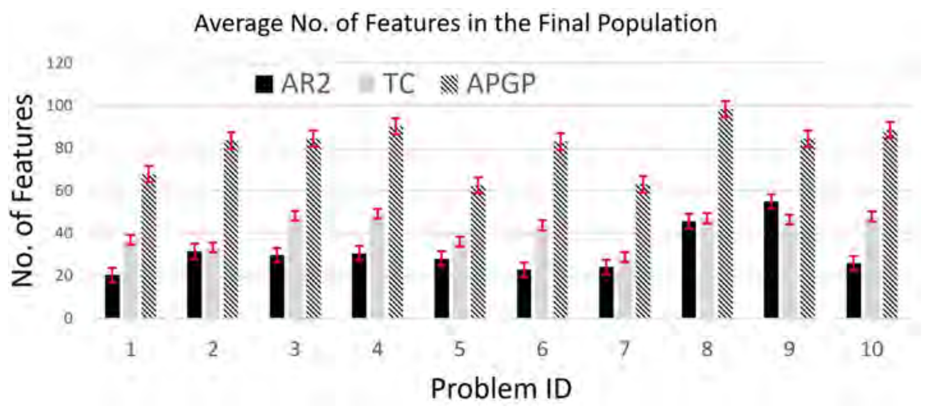
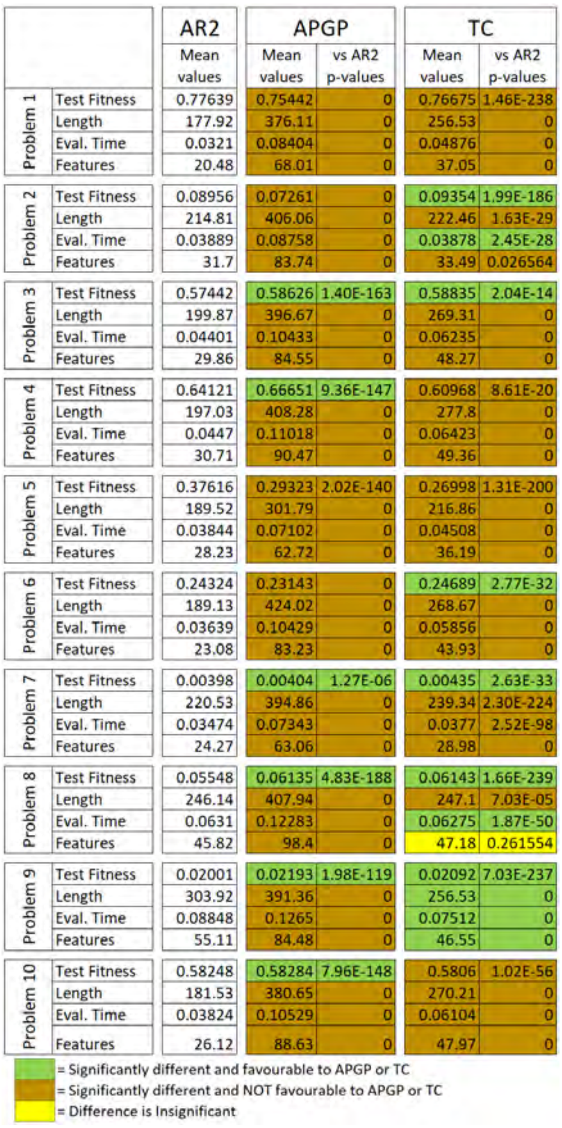


FIGURE 8.10: Comparison of the average number of features in the final populations of AR2, TC and APGP.



		AR2	APGP		TC	
		Mean values	Mean values	vs AR2 p-values	Mean values	vs AR2 p-values
Problem 1	Test Fitness	0.77639	🟧 0.75442	0	🟧 0.76675	1.46E-238
	Length	177.92	🟧 376.11	0	🟧 256.53	0
	Eval. Time	0.0321	🟧 0.08404	0	🟧 0.04876	0
	Features	20.48	🟧 68.01	0	🟧 37.05	0
Problem 2	Test Fitness	0.08956	🟧 0.07261	0	🟩 0.09354	1.99E-186
	Length	214.81	🟧 406.06	0	🟧 222.46	1.63E-29
	Eval. Time	0.03889	🟧 0.08758	0	🟩 0.03878	2.45E-28
	Features	31.7	🟧 83.74	0	🟧 33.49	0.026564
Problem 3	Test Fitness	0.57442	🟩 0.58626	1.40E-163	🟩 0.58835	2.04E-14
	Length	199.87	🟧 396.67	0	🟧 269.31	0
	Eval. Time	0.04401	🟧 0.10433	0	🟧 0.06235	0
	Features	29.86	🟧 84.55	0	🟧 48.27	0
Problem 4	Test Fitness	0.64121	🟩 0.66651	9.36E-147	🟧 0.60968	8.61E-20
	Length	197.03	🟧 408.28	0	🟧 277.8	0
	Eval. Time	0.0447	🟧 0.11018	0	🟧 0.06423	0
	Features	30.71	🟧 90.47	0	🟧 49.36	0
Problem 5	Test Fitness	0.37616	🟧 0.29323	2.02E-140	🟧 0.26998	1.31E-200
	Length	189.52	🟧 301.79	0	🟧 216.86	0
	Eval. Time	0.03844	🟧 0.07102	0	🟧 0.04508	0
	Features	28.23	🟧 62.72	0	🟧 36.19	0
Problem 6	Test Fitness	0.24324	🟧 0.23143	0	🟩 0.24689	2.77E-32
	Length	189.13	🟧 424.02	0	🟧 268.67	0
	Eval. Time	0.03639	🟧 0.10429	0	🟧 0.05856	0
	Features	23.08	🟧 83.23	0	🟧 43.93	0
Problem 7	Test Fitness	0.00398	🟩 0.00404	1.27E-06	🟩 0.00435	2.63E-33
	Length	220.53	🟧 394.86	0	🟧 239.34	2.30E-224
	Eval. Time	0.03474	🟧 0.07343	0	🟧 0.0377	2.52E-98
	Features	24.27	🟧 63.06	0	🟧 28.98	0
Problem 8	Test Fitness	0.05548	🟩 0.06135	4.83E-188	🟩 0.06143	1.66E-239
	Length	246.14	🟧 407.94	0	🟧 247.1	7.03E-05
	Eval. Time	0.0631	🟧 0.12283	0	🟩 0.06275	1.87E-50
	Features	45.82	🟧 98.4	0	🟨 47.18	0.261554
Problem 9	Test Fitness	0.02001	🟩 0.02193	1.98E-119	🟩 0.02092	7.03E-237
	Length	303.92	🟧 391.36	0	🟩 256.53	0
	Eval. Time	0.08848	🟧 0.1265	0	🟩 0.07512	0
	Features	55.11	🟧 84.48	0	🟩 46.55	0
Problem 10	Test Fitness	0.58248	🟩 0.58284	7.96E-148	🟧 0.5806	1.02E-56
	Length	181.53	🟧 380.65	0	🟧 270.21	0
	Eval. Time	0.03824	🟧 0.10529	0	🟧 0.06104	0
	Features	26.12	🟧 88.63	0	🟧 47.97	0

🟩 = Significantly different and favourable to APGP or TC
🟧 = Significantly different and NOT favourable to APGP or TC
🟨 = Difference is Insignificant

FIGURE 8.11: Result of the test for significance of the difference in the final populations of AR2 against APGP and TC.

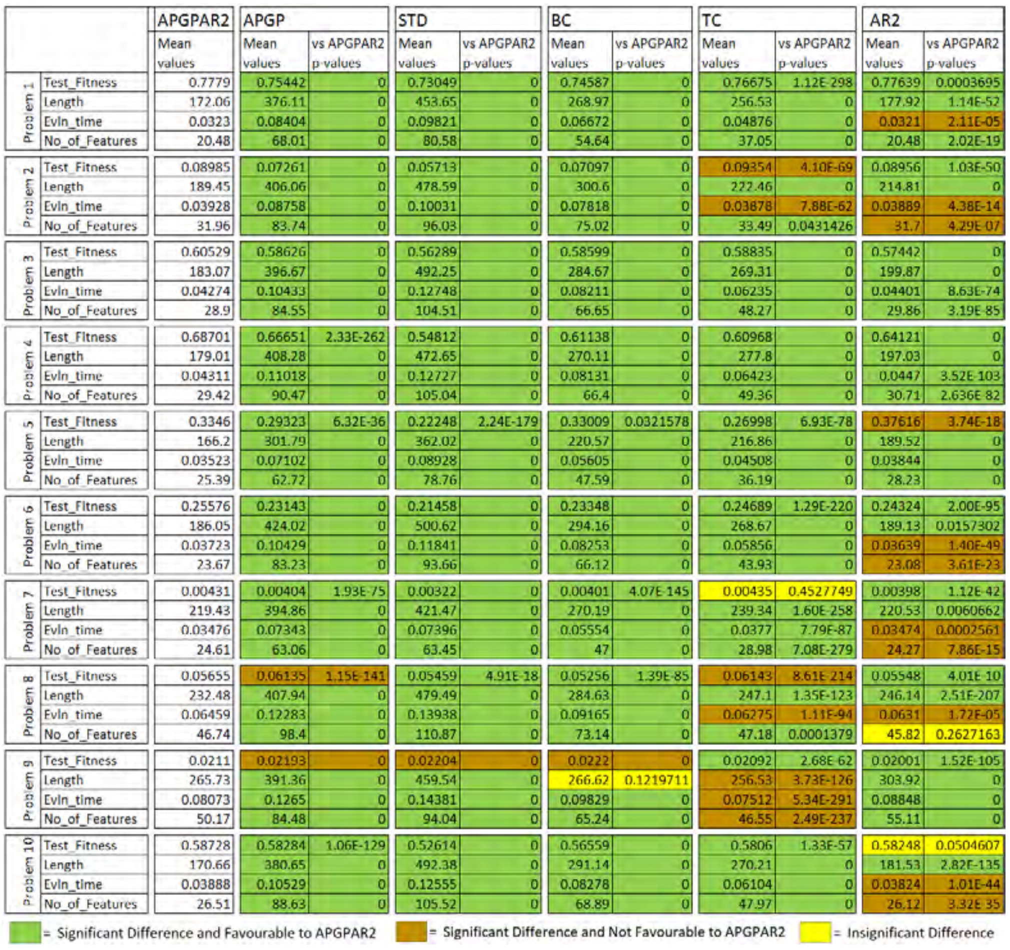
8.6 Time Control Schemes with Adjusted R^2

Because MLRGP with Adjusted R^2 (AR2) outperforms the time control techniques on complexity measures (as seen in Section 8.5.3), it makes sense to try Adjusted R^2 with the time control techniques as well. Thus, this section compares APGP and TC with Adjusted R^2 (APGPAR2 and TCAR2, respectively) with the other MLRGP methods. To produce APGPAR2 and TCAR2, the fitness is changed from normalised mean square error (NMSE) to Adjusted R^2 but also captured the NMSE, which provides a means to compare with other methods that do not use Adjusted R^2. The following sections report the differences.

8.6.1 APGP with Adjusted R^2

As detailed in FIGURE 8.12, using the Adjusted R^2 as the fitness function improves both the simplicity and accuracy of APGP. Therefore, APGPAR2 produced a final populations that were significantly simpler (size, evaluation times, and number of features) than those from APGP 10 out of 10 times and were significantly more accurate test scores 8 out 10 times. Against the other methods the test fitness improved as follows: 9 out of 10 times against STD, 9 out of 10 against BC, 7 out of 10 (plus an insignificant difference in 1 problem) against TC, 9 out of 10 against AR2. In terms of complexity control (size, evaluation time, and the number of features), APGPAR2 produced less complex solutions than STD, BC and APGP *all* the time; only once (out of 90 comparisons) was the difference insignificant. Also against TC APGPAR2 produced solutions with significantly fewer features 9 times out of 10, smaller sized solutions 9 times out of 10, and shorter evaluation times in 7 times out of 10. Futhermore, against AR2, APGPAR2 produced solutions with fewer features in 5 out 10 tests, smaller sized solutions in 10 out 10 and shorter evaluation times in 4 out 10 tests.

So overall, using Adjusted R^2 in APGP (APGPAR2) led to more accurate (test fitness scores) and simpler (in terms of size, evaluation times, and number of features).



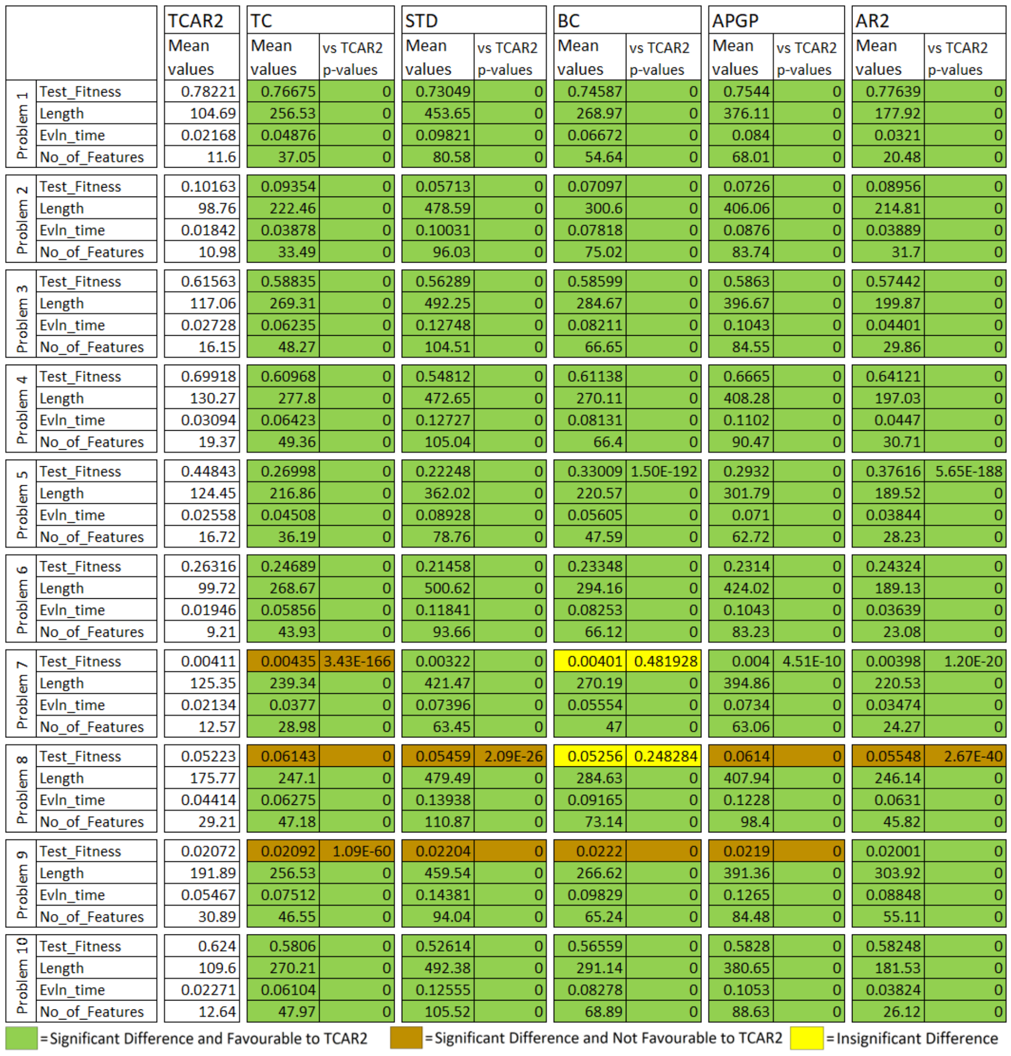
		APGPAR2	APGP		STD		BC		TC		AR2	
		Mean values	Mean values	vs APGPAR2 p-values	Mean values	vs APGPAR2 p-values	Mean values	vs APGPAR2 p-values	Mean values	vs APGPAR2 p-values	Mean values	vs APGPAR2 p-values
Problem 1	Test_Fitness	0.7779	🟩 0.75442	0	🟩 0.73049	0	🟩 0.74587	0	🟩 0.76675	1.12E-298	🟩 0.77639	0.0003695
	Length	172.06	🟩 376.11	0	🟩 453.65	0	🟩 268.97	0	🟩 256.53	0	🟩 177.92	1.14E-52
	Evln_time	0.0323	🟩 0.08404	0	🟩 0.09821	0	🟩 0.06672	0	🟩 0.04876	0	🟧 0.0321	2.11E-05
	No_of_Features	20.48	🟩 68.01	0	🟩 80.58	0	🟩 54.64	0	🟩 37.05	0	🟩 20.48	2.02E-19
Problem 2	Test_Fitness	0.08985	🟩 0.07261	0	🟩 0.05713	0	🟩 0.07097	0	🟧 0.09354	4.10E-69	🟩 0.08956	1.03E-50
	Length	189.45	🟩 406.06	0	🟩 478.59	0	🟩 300.6	0	🟩 222.46	0	🟩 214.81	0
	Evln_time	0.03928	🟩 0.08758	0	🟩 0.10031	0	🟩 0.07818	0	🟧 0.03878	7.88E-62	🟧 0.03889	4.38E-14
	No_of_Features	31.96	🟩 83.74	0	🟩 96.03	0	🟩 75.02	0	🟩 33.49	0.0431426	🟧 31.7	4.29E-07
Problem 3	Test_Fitness	0.60529	🟩 0.58626	0	🟩 0.56289	0	🟩 0.58599	0	🟩 0.58835	0	🟩 0.57442	0
	Length	183.07	🟩 396.67	0	🟩 492.25	0	🟩 284.67	0	🟩 269.31	0	🟩 199.87	0
	Evln_time	0.04274	🟩 0.10433	0	🟩 0.12748	0	🟩 0.08211	0	🟩 0.06235	0	🟩 0.04401	8.63E-74
	No_of_Features	28.9	🟩 84.55	0	🟩 104.51	0	🟩 66.65	0	🟩 48.27	0	🟩 29.86	3.19E-85
Problem 4	Test_Fitness	0.68701	🟩 0.66651	2.33E-262	🟩 0.54812	0	🟩 0.61138	0	🟩 0.60968	0	🟩 0.64121	0
	Length	179.01	🟩 408.28	0	🟩 472.65	0	🟩 270.11	0	🟩 277.8	0	🟩 197.03	0
	Evln_time	0.04311	🟩 0.11018	0	🟩 0.12727	0	🟩 0.08131	0	🟩 0.06423	0	🟩 0.0447	3.52E-103
	No_of_Features	29.42	🟩 90.47	0	🟩 105.04	0	🟩 66.4	0	🟩 49.36	0	🟩 30.71	2.636E-82
Problem 5	Test_Fitness	0.3346	🟩 0.29323	6.32E-36	🟩 0.22248	2.24E-179	🟩 0.33009	0.0321578	🟩 0.26998	6.93E-78	🟧 0.37616	3.74E-18
	Length	166.2	🟩 301.79	0	🟩 362.02	0	🟩 220.57	0	🟩 216.86	0	🟩 189.52	0
	Evln_time	0.03523	🟩 0.07102	0	🟩 0.08928	0	🟩 0.05605	0	🟩 0.04508	0	🟩 0.03844	0
	No_of_Features	25.39	🟩 62.72	0	🟩 78.76	0	🟩 47.59	0	🟩 36.19	0	🟩 28.23	0
Problem 6	Test_Fitness	0.25576	🟩 0.23143	0	🟩 0.21458	0	🟩 0.23348	0	🟩 0.24689	1.29E-220	🟩 0.24324	2.00E-95
	Length	186.05	🟩 424.02	0	🟩 500.62	0	🟩 294.16	0	🟩 268.67	0	🟩 189.13	0.0157302
	Evln_time	0.03723	🟩 0.10429	0	🟩 0.11841	0	🟩 0.08253	0	🟩 0.05856	0	🟧 0.03639	1.40E-49
	No_of_Features	23.67	🟩 83.23	0	🟩 93.66	0	🟩 66.12	0	🟩 43.93	0	🟧 23.08	3.61E-23
Problem 7	Test_Fitness	0.00431	🟩 0.00404	1.93E-75	🟩 0.00322	0	🟩 0.00401	4.07E-145	🟨 0.00435	0.4527749	🟩 0.00398	1.12E-42
	Length	219.43	🟩 394.86	0	🟩 421.47	0	🟩 270.19	0	🟩 239.34	1.60E-258	🟩 220.53	0.0060662
	Evln_time	0.03476	🟩 0.07343	0	🟩 0.07396	0	🟩 0.05554	0	🟩 0.0377	7.79E-87	🟧 0.03474	0.0002561
	No_of_Features	24.61	🟩 63.06	0	🟩 63.45	0	🟩 47	0	🟩 28.98	7.08E-279	🟧 24.27	7.86E-15
Problem 8	Test_Fitness	0.05655	🟧 0.06135	1.15E-141	🟩 0.05459	4.91E-18	🟩 0.05256	1.39E-85	🟧 0.06143	8.61E-214	🟩 0.05548	4.01E-10
	Length	232.48	🟩 407.94	0	🟩 479.49	0	🟩 284.63	0	🟩 247.1	1.35E-123	🟩 246.14	2.51E-207
	Evln_time	0.06459	🟩 0.12283	0	🟩 0.13938	0	🟩 0.09165	0	🟧 0.06275	1.11E-94	🟧 0.0631	1.72E-05
	No_of_Features	46.74	🟩 98.4	0	🟩 110.87	0	🟩 73.14	0	🟩 47.18	0.0001379	🟨 45.82	0.2627163
Problem 9	Test_Fitness	0.0211	🟧 0.02193	0	🟧 0.02204	0	🟧 0.0222	0	🟩 0.02092	2.68E-62	🟩 0.02001	1.52E-105
	Length	265.73	🟩 391.36	0	🟩 459.54	0	🟨 266.62	0.1219711	🟧 256.53	3.73E-126	🟩 303.92	0
	Evln_time	0.08073	🟩 0.1265	0	🟩 0.14381	0	🟩 0.09829	0	🟧 0.07512	5.34E-291	🟩 0.08848	0
	No_of_Features	50.17	🟩 84.48	0	🟩 94.04	0	🟩 65.24	0	🟧 46.55	2.49E-237	🟩 55.11	0
Problem 10	Test_Fitness	0.58728	🟩 0.58284	1.06E-129	🟩 0.52614	0	🟩 0.56559	0	🟩 0.5806	1.33E-57	🟨 0.58248	0.0504607
	Length	170.66	🟩 380.65	0	🟩 492.38	0	🟩 291.14	0	🟩 270.21	0	🟩 181.53	2.82E-135
	Evln_time	0.03888	🟩 0.10529	0	🟩 0.12555	0	🟩 0.08278	0	🟩 0.06104	0	🟧 0.03824	1.01E-44
	No_of_Features	26.51	🟩 88.63	0	🟩 105.52	0	🟩 68.89	0	🟩 47.97	0	🟧 26.12	3.32E-35

🟩 = Significant Difference and Favourable to APGPAR2 🟧 = Significant Difference and Not Favourable to APGPAR2 🟨 = Insignificant Difference

FIGURE 8.12: Result of significance test for difference between the final populations of APGPAR2 (APGP with Adjusted R2 as fitness) and those of the other Methods.

8.6.2 Time Control (TC) with Adjusted R^2

This section compares time-control (TC) with Adjusted R2 (TCAR2) against all the methods except APGPR2; TCAR2 is compared with APGPAR2 later in this section. FIGURE 8.13 shows that TCAR2 produced more accurate scores on test sets. On 10 problems TCAR2 outperformed TC, STD, BC, APGP and AR2 7, 8, 7, 8 and 9 times respectively. On the three complexity scores, TCAR2 outperformed all the benchmark methods all the time.



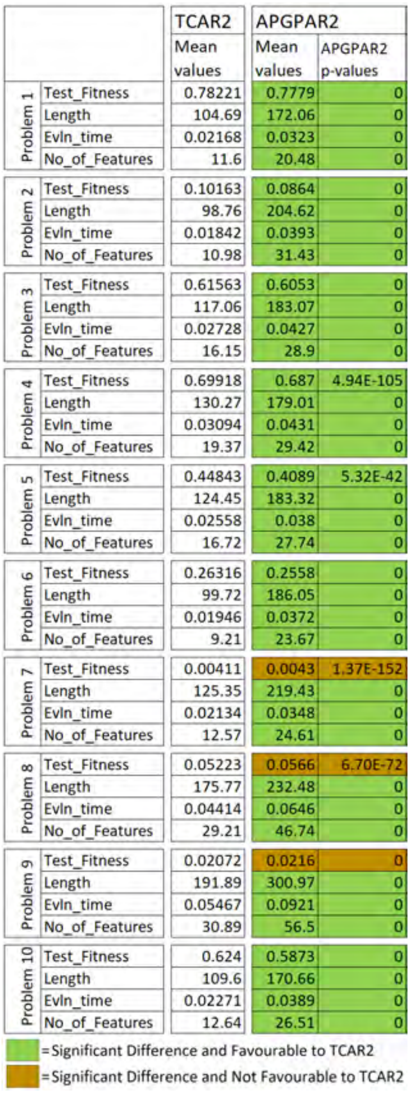
		TCAR2	TC		STD		BC		APGP		AR2	
		Mean values	Mean values	vs TCAR2 p-values	Mean values	vs TCAR2 p-values	Mean values	vs TCAR2 p-values	Mean values	vs TCAR2 p-values	Mean values	vs TCAR2 p-values
Problem 1	Test_Fitness	0.78221	0.76675	0	0.73049	0	0.74587	0	0.7544	0	0.77639	0
	Length	104.69	256.53	0	453.65	0	268.97	0	376.11	0	177.92	0
	Evln_time	0.02168	0.04876	0	0.09821	0	0.06672	0	0.084	0	0.0321	0
	No_of_Features	11.6	37.05	0	80.58	0	54.64	0	68.01	0	20.48	0
Problem 2	Test_Fitness	0.10163	0.09354	0	0.05713	0	0.07097	0	0.0726	0	0.08956	0
	Length	98.76	222.46	0	478.59	0	300.6	0	406.06	0	214.81	0
	Evln_time	0.01842	0.03878	0	0.10031	0	0.07818	0	0.0876	0	0.03889	0
	No_of_Features	10.98	33.49	0	96.03	0	75.02	0	83.74	0	31.7	0
Problem 3	Test_Fitness	0.61563	0.58835	0	0.56289	0	0.58599	0	0.5863	0	0.57442	0
	Length	117.06	269.31	0	492.25	0	284.67	0	396.67	0	199.87	0
	Evln_time	0.02728	0.06235	0	0.12748	0	0.08211	0	0.1043	0	0.04401	0
	No_of_Features	16.15	48.27	0	104.51	0	66.65	0	84.55	0	29.86	0
Problem 4	Test_Fitness	0.69918	0.60968	0	0.54812	0	0.61138	0	0.6665	0	0.64121	0
	Length	130.27	277.8	0	472.65	0	270.11	0	408.28	0	197.03	0
	Evln_time	0.03094	0.06423	0	0.12727	0	0.08131	0	0.1102	0	0.0447	0
	No_of_Features	19.37	49.36	0	105.04	0	66.4	0	90.47	0	30.71	0
Problem 5	Test_Fitness	0.44843	0.26998	0	0.22248	0	0.33009	1.50E-192	0.2932	0	0.37616	5.65E-188
	Length	124.45	216.86	0	362.02	0	220.57	0	301.79	0	189.52	0
	Evln_time	0.02558	0.04508	0	0.08928	0	0.05605	0	0.071	0	0.03844	0
	No_of_Features	16.72	36.19	0	78.76	0	47.59	0	62.72	0	28.23	0
Problem 6	Test_Fitness	0.26316	0.24689	0	0.21458	0	0.23348	0	0.2314	0	0.24324	0
	Length	99.72	268.67	0	500.62	0	294.16	0	424.02	0	189.13	0
	Evln_time	0.01946	0.05856	0	0.11841	0	0.08253	0	0.1043	0	0.03639	0
	No_of_Features	9.21	43.93	0	93.66	0	66.12	0	83.23	0	23.08	0
Problem 7	Test_Fitness	0.00411	0.00435	3.43E-166	0.00322	0	0.00401	0.481928	0.004	4.51E-10	0.00398	1.20E-20
	Length	125.35	239.34	0	421.47	0	270.19	0	394.86	0	220.53	0
	Evln_time	0.02134	0.0377	0	0.07396	0	0.05554	0	0.0734	0	0.03474	0
	No_of_Features	12.57	28.98	0	63.45	0	47	0	63.06	0	24.27	0
Problem 8	Test_Fitness	0.05223	0.06143	0	0.05459	2.09E-26	0.05256	0.248284	0.0614	0	0.05548	2.67E-40
	Length	175.77	247.1	0	479.49	0	284.63	0	407.94	0	246.14	0
	Evln_time	0.04414	0.06275	0	0.13938	0	0.09165	0	0.1228	0	0.0631	0
	No_of_Features	29.21	47.18	0	110.87	0	73.14	0	98.4	0	45.82	0
Problem 9	Test_Fitness	0.02072	0.02092	1.09E-60	0.02204	0	0.0222	0	0.0219	0	0.02001	0
	Length	191.89	256.53	0	459.54	0	266.62	0	391.36	0	303.92	0
	Evln_time	0.05467	0.07512	0	0.14381	0	0.09829	0	0.1265	0	0.08848	0
	No_of_Features	30.89	46.55	0	94.04	0	65.24	0	84.48	0	55.11	0
Problem 10	Test_Fitness	0.624	0.5806	0	0.52614	0	0.56559	0	0.5828	0	0.58248	0
	Length	109.6	270.21	0	492.38	0	291.14	0	380.65	0	181.53	0
	Evln_time	0.02271	0.06104	0	0.12555	0	0.08278	0	0.1053	0	0.03824	0
	No_of_Features	12.64	47.97	0	105.52	0	68.89	0	88.63	0	26.12	0

🟩 = Significant Difference and Favourable to TCAR2 🟧 = Significant Difference and Not Favourable to TCAR2 🟨 = Insignificant Difference

FIGURE 8.13: Result of significance test for difference between the final populations of TCAR2 (TC with Adjusted R2 as fitness) and those of the other Methods.

8.6.3 TCAR2 vs. APGPAR2

Next, FIGURE 8.14 shows the difference between TCAR2 and APGPAR2. TCAR2 produced significantly simpler solutions across all the problems; also the solutions produced by TCAR2 were more accurate 7 out of 10 times.



		TCAR2	APGPAR2	
		Mean values	Mean values	APGPAR2 p-values
Problem 1	Test_Fitness	0.78221	0.7779	0
	Length	104.69	172.06	0
	Evln_time	0.02168	0.0323	0
	No_of_Features	11.6	20.48	0
Problem 2	Test_Fitness	0.10163	0.0864	0
	Length	98.76	204.62	0
	Evln_time	0.01842	0.0393	0
	No_of_Features	10.98	31.43	0
Problem 3	Test_Fitness	0.61563	0.6053	0
	Length	117.06	183.07	0
	Evln_time	0.02728	0.0427	0
	No_of_Features	16.15	28.9	0
Problem 4	Test_Fitness	0.69918	0.687	4.94E-105
	Length	130.27	179.01	0
	Evln_time	0.03094	0.0431	0
	No_of_Features	19.37	29.42	0
Problem 5	Test_Fitness	0.44843	0.4089	5.32E-42
	Length	124.45	183.32	0
	Evln_time	0.02558	0.038	0
	No_of_Features	16.72	27.74	0
Problem 6	Test_Fitness	0.26316	0.2558	0
	Length	99.72	186.05	0
	Evln_time	0.01946	0.0372	0
	No_of_Features	9.21	23.67	0
Problem 7	Test_Fitness	0.00411	0.0043	1.37E-152
	Length	125.35	219.43	0
	Evln_time	0.02134	0.0348	0
	No_of_Features	12.57	24.61	0
Problem 8	Test_Fitness	0.05223	0.0566	6.70E-72
	Length	175.77	232.48	0
	Evln_time	0.04414	0.0646	0
	No_of_Features	29.21	46.74	0
Problem 9	Test_Fitness	0.02072	0.0216	0
	Length	191.89	300.97	0
	Evln_time	0.05467	0.0921	0
	No_of_Features	30.89	56.5	0
Problem 10	Test_Fitness	0.624	0.5873	0
	Length	109.6	170.66	0
	Evln_time	0.02271	0.0389	0
	No_of_Features	12.64	26.51	0

🟩 = Significant Difference and Favourable to TCAR2

🟧 = Significant Difference and Not Favourable to TCAR2

FIGURE 8.14: Result of significance test for difference between the final populations of APGPAR2 and those of TCAR2.

8.6.4 Summary of Time Control Schemes with Adjusted R^2

These results show that using Adjusted R^2 benefits the evaluation time schemes. The resulting solutions both usually generalise better and are simpler in terms of size, evaluation times and the number of features.

8.7 Conclusion

While earlier chapters showed that the evaluation time reflects the size of a model and the functional complexity of its components, this chapter practically demonstrates an additional notion of complexity that the evaluation time reflects - the number of features. This finding further counters the argument that evaluation time is the same as size.

To examine how the proposed complexity measure – that is, evaluation time – encompasses further notions of complexity, this chapter leverages a novel hybridisation of MLR and GP that discovers features within a model and effectively exploits them to produce efficient regression models. This hybridisation, termed MLR-GP, improves training fitness remarkably – within 5 generations, MLR-GP outperforms the training accuracy of standard GP after 50 generations; however, MLR-GP also overfits the data easily. Thus, a natural follow-up question is to examine how this overfitting can be improved via effective complexity-control that the time-control methods offer.

Examining how time characterises complexity in MLR-GP shows that it detects the number of features very effectively; in fact, time correlates more strongly with the number of features than it does with size. Therefore, managing complexity by restricting the evaluation time is effectively different from restricting just size.

The time-control methods (TC and APGP) generalise better than the benchmark methods; they both outperformed standard GP (STD) (on 9 out of 10 problems) and BC (7 out of 10 problems). However, on simplicity (evaluation time, size, and the number of features) while both TC and APGP outperformed STD, only TC outperformed BC (GP with bloat-control method). Given that TC and BC use the same technique to manage time and size, respectively, their comparison is a truer reflection of the difference that using evaluation time makes. Using time as a measure of complexity is the better option because TC prevails over BC as follows: accuracy (7 out of 10), size (9 out of 10), evaluation time (10 out of 10) and the number of features (10 out of 10).

Because time-control manages the number of features effectively, it is compared with an effective human-designed method that specifically manages the number of features – Adjusted R^2. The results show that while the accuracy scores on test sets were mixed, the use of Adjusted R^2 generally outperformed APGP on simplicity; however, TC performed similarly. This was a useful result in that unlike AR2, the time-control methods are not limited to regression but are general methods with potentially broader applications. However, GP with time control still managed to approximate the behaviour of Adjusted R^2. Also, there is no reason why Adjusted-R^2 can not be combined with the evaluation time based methods.

Therefore, when the Adjusted R^2 was used with the evaluation time schemes, they produced the overall winners, that is, simpler solutions that generalise better than all the benchmark methods. TC with Adjusted R^2 also outperformed APGP with Adjusted R^2.

Overall, this chapter reasserts that not only does the evaluation-time controls complexity more effectively than the size control methods in GP but also that these methods (based on evaluation time control) can also synergise with domain-specific methods such as Adjusted R^2 to yield the most optimised results.

The next chapter further demonstrates the versatility of the evaluation time schemes. Accordingly, they are employed to solve other classical GP problems from robot control, classification, and Boolean logic applications.

Chapter 9

Applications - Beyond Regression

9.1 Introduction

Before this chapter, the experiments to assess the evaluation time schemes used only regression problems. However, this thesis argues that time is a versatile measure of complexity; therefore, it is imperative to test how containing evaluation times unearths different notions of complexity across some other highly popular GP applications. In doing so, this lets us further explore the forms of complexity the evaluation time can characterise.

This chapter applies the evaluation time schemes to some well-known GP applications. The problem domains are Boolean logic, robot control and classification. While later sections give details of these problems, this section provides a brief overview.

For the Boolean logic applications, the multiplexer and even-parity problems are used. The challenge in the multiplexer problem (henceforth termed Multiplexer) [228, 229] is for GP to use Boolean logic operators to reproduce the behaviour of an electronic multiplexer given all its possible input and output values. The challenge in the even-parity problem (Parity) [229, 230] is for GP to use Boolean logic operators to find a solution that produces the value of the Boolean even parity given n independent Boolean inputs.

For the robot control application, GP is used to evolve solutions to the well known artificial ant problem (ANT) [229, 231]. The challenge in ANT is to evolve a routine for navigating a robotic ant on a virtual field to help it find food items within a given time limit.

The classification application uses a hybridisation of GP with machine learning (GPML). Like MLR-GP in Chapter 8, the hybridisation involves GP and machine learning (GPML), whereas here it is for classification; GP engineers a set of features and then logistic regression [232] uses these features to build a classification model.

When reviewing the outcome of the complexity control, the analyses consider the common motives for managing complexity. As discussed in Chapter 4, the incentive for managing complexity varies; thus, the motive can be to produce models that generalise well, utilise computing resources efficiently, or have short run times. The problem domains selected in this chapter help us explore these various notions of complexity. Much like the regression applications, the generalisation ability of models continues to be important in the classification applications. The results in this chapter indicate that time control indeed improves generalisation.

However, generalisation is not relevant in the robot control (ANT) and Boolean logic problems (Multiplexer and Parity) because the inputs are completely specified during training for both these problems. However, in the ANT problem, it is advantageous to produce a routine that both navigates correctly and does so in a short amount of time. A question arises as to whether time control can confer an advantage over standard GP or GP with bloat-control. The results show that indeed time control helps evolve accurate routines that are significantly more efficient than those produced by the counterpart GP systems.

Similarly, in the Boolean problems, the benefits of simplicity differ. If the computational complexity of the Boolean constructs used as building blocks for the solutions differ significantly, the results of the time control systems and others may differ. Otherwise, the time control systems will simply act as bloat-control systems. The results indicate that time control systems act as bloat-control systems for the Boolean problems.

From the above discussion it appears that, unlike with bloat-control that is popularly used in GP to control complexity, time control is versatile in that it exploits different notions of complexity depending upon the problem domain. The investigation in this chapter verifies this.

The rest of this chapter is organised as follows: Section 9.2 investigates the ANT problem; sections 9.3 and 9.4 investigate the Boolean problems; Section 9.5 investigates classification with GP aided by machine learning; Section 9.6 summarises and discusses the results of this chapter; and Section 9.7 concludes this chapter.

9.2 Robot Control Application - The Artificial Ant

For the robot control application, a classical GP problem – *The Artificial Ant* (ANT) – is used. The challenge here is to evolve a program that navigates an artificial ant on a virtual field to eat up all the available food within a time constraint. The food is distributed along a trail, well known as the *Santa Fe Ant* trail, which contains 89 pieces of food spread along a given path within a toroidal grid of 1024 squares. The ant is only allowed 600 moves to consume as many food items as possible.

Koza introduced this problem in [229, p. 147–155] and it has been studied extensively in the GP community [18, 229, 231, 233–236]. Previous studies consider the ANT problem hard [237]; this is because the search space is characterised by multiple plateaus, deep valleys, and many local and global optima.

The experiment builds on the implementation of this problem that is available in DEAP (a GP framework) [47]. FIGURE 9.1 illustrates the Santa Fe trail.

As stated earlier, the robotic ant is only allowed a limited number of moves to pick up the food available (600); therefore, the fitness of a navigating routine is the number of food items picked within 600 moves. This limit encourages the programs to navigate smartly and precludes excessively long running routines.

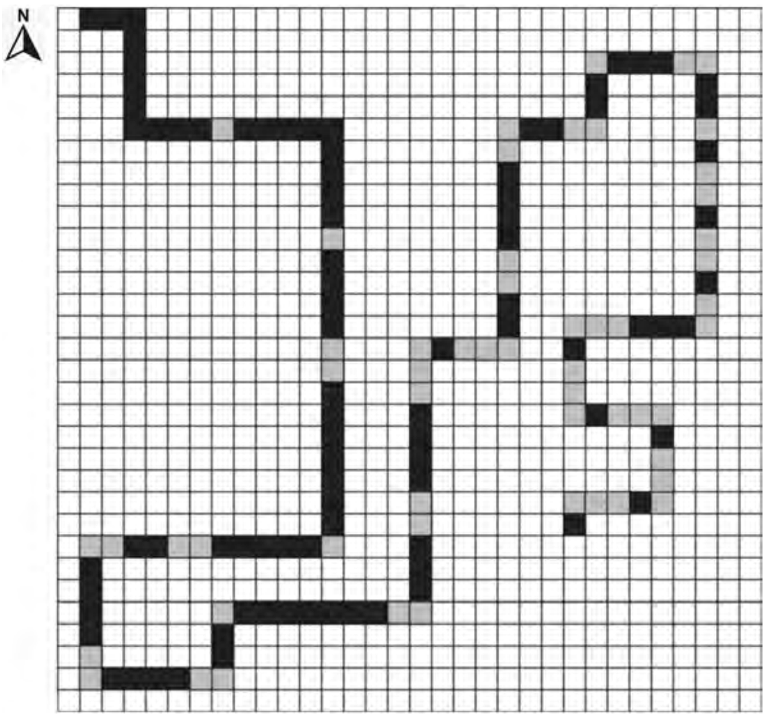


FIGURE 9.1: Santa Fe food trail for the Artificial Ant problem (ANT). The black cells represent the food items, and the grey cells are gaps in the trail. The artificial ant begins in the top left corner facing eastward.

Three operations are available for the programs to navigate the ant across the field: `MoveForward`, `TurnRight`, and `TurnLeft` move the ant in the direction it presently faces, turns the ant right and turns the ant left respectively; executing any of these actions consumes a unit from the budgeted number of moves allowed. Note these operations do not need any input arguments (they access the current position and direction of the ant from a global memory); therefore, these operations act as the terminals in a GP tree, that is, the GP tree can not extend any further after one of these operations appears in the tree. Therefore, GP also needs a function set to grow trees.

Thus, besides the operations that indicate movement, an individual can also leverage three functions: `Prog2`, `Prog3` and `IfFoodAhead`. The `Prog2`, and, `Prog3` functions enable producing a sequence of 2 and 3 operations or functions respectively; these sequences can also include `Prog2` and `Prog3` themselves. This allows creating longer sequences of instructions in a GP tree that encodes the navigation routine. Figure 9.2 shows an example tree representation of an ANT program.

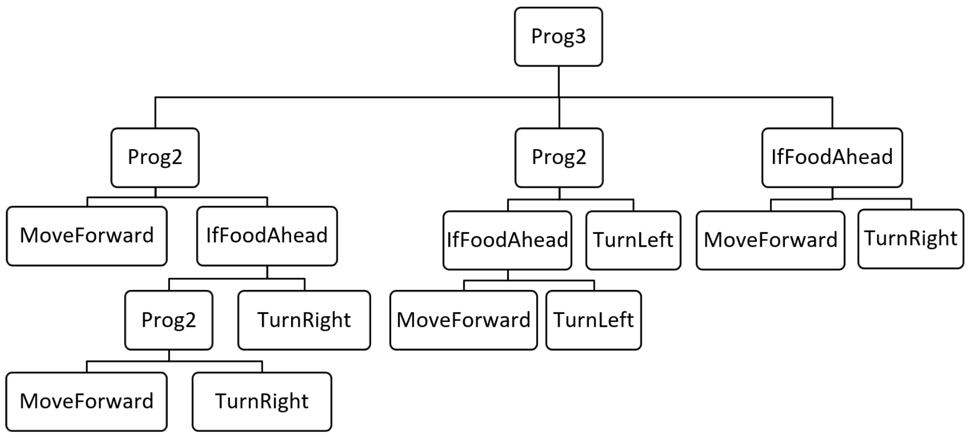


FIGURE 9.2: An example of ANT solutions.

`IfFoodAhead` is a function that detects the presence of food in the square immediately in front of the ant; it has an arity of two and can execute either of two child nodes (arguments) based on the Boolean output (whether the food is directly in front or not) of the function. The `IfFoodAhead` function does not automatically consume the food it detects. Consequently, part of the learning is to know to move forward into a square with food to consume.

Note during the execution of a navigating program `IfFoodAhead`, `Prog2` and `Prog3` do not consume a move.

Naturally, while GP is evolving fit solutions (programs that can eat an increasingly large number of food items), it may also produce unnecessarily complex and inefficient ones. This challenge makes this problem appropriate for studying time control as to whether it can induce efficiency in a way that standard GP approaches cannot.

9.2.1 Experiment

The experiments compare the time schemes (APGP and TC) with standard GP (GP) and GP with bloat-control (BC).

ANT Fitness Evaluation: The evaluation function of ANT assesses a program and gives a fitness score as follows. First, it compiles the program to translate it to a routine that will navigate the artificial ant. Next, the ant navigates through the field by continuously using the routine until it either finds all the food or reaches the maximum number of moves allowed (600). All ants begin in the top left corner facing eastward. When the ant enters a square with food, the food item is considered eaten and is no longer available on the field. The fraction of food eaten from the available is the fitness score of the control program. Therefore, the fitness score is simply the amount of eaten divided by the total available. In other words, fitness $f = x \div y$, where x is the number of eaten items and y is the total number of items on the trail.

The parameters for the ANT experiments are summarised in TABLE 9.1.

9.2.2 Size is Not Time in ANT

In the ANT application, the final populations of all the contending methods show a weak correlation between size and evaluation times. As illustrated in FIGURE 9.3, the correlation scores between the size and the evaluation times are 0.149 for TC, 0.183 for APGP, 0.16 for BC, and 0.493 for GP. Note that two individuals with very different sizes may have similar evaluation times in these

Parameter	Setting
Terminal set:	TurnLeft, TurnRight, MoveForward
Function set:	IfFoodAhead, Prog2, Prog3
Fitness:	Amount of Food eaten
Selection:	Tournament size of 7, non-elitist
Wrapper:	Program repeatedly executed up to 600 moves
Population Size:	500
Max program size:	32,767
Initial population:	"ramped half-and-half" (max. depth = 6)
Probabilities:	mutation = 0.3, crossover rate = 0.7
Termination:	After 70 generations
Probabilities:	crossover = 0.7, mutation = 0.3

TABLE 9.1: Experimental settings for the Ant problem.

populations. This similarity exists because the evaluation function demands that all the candidate solutions make 600 moves (steps) unless they find all the food items before then. Therefore, a routine that comprises a few steps (e.g. 50) will be rerun until it makes 600 moves; and another that implements more than 600 steps (e.g. 900) will stop before a single run of the routine is completed. Thus, the evaluation times of very different sized individuals may be similar.

These findings further counter the assertion of another study [185] that evaluation time is another side of the same coin as size. Furthermore, the results later in Section 9.2.3 show that using evaluation time to manage complexity in the ANT application does offer a qualitative advantage over using size.

Despite this weak correlation, the evaluation time schemes discourage growth in the sizes of their solutions because after all size can contribute to the evaluation time. Note, on the correlation charts in on FIGURE 9.3, the individuals produced by TC and APGP are less than 500 nodes in size, and those of GP (without complexity control) are up to 1,400. Furthermore, the results in (Section 9.2.3 confirms this observation.

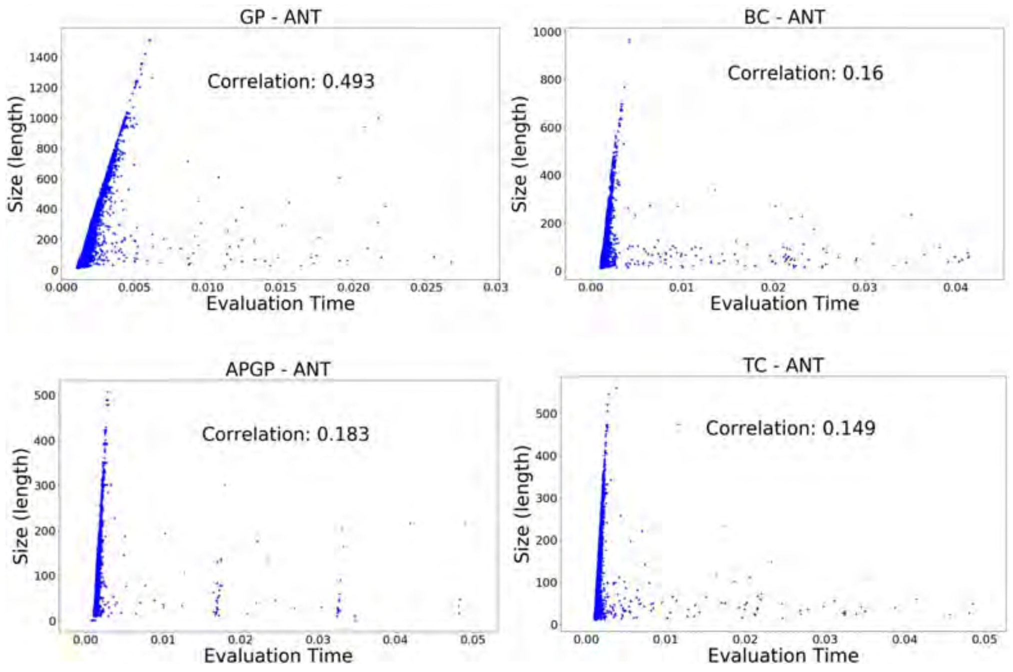


FIGURE 9.3: The correlation in ANT between evaluation time and size in the final populations of the contending methods.

Factors that affect Evaluation Times in ANT

A series of tests were used to determine the factors that significantly contribute to the evaluation times. The tests involve evaluating individuals of different makeup (detailed below) 200 times each and testing the significance of the difference in their evaluation times.

Efficient solutions that find all food items before using up the budgeted number of moves (steps) can have significantly shorter evaluation times than similar-sized solutions that use up theirs. The mean evaluation times of an individual that uses 475 moves differed by 13.35% from one that uses 309. This difference is despite the individual with the shorter evaluation time being the larger-sized one (length 34 vs 23). In this example, the evaluation time can easily spot the more efficient program. Therefore, the efficiency of individuals affects their evaluation time.

However, evaluation time detects size because of other reasons. For example, the duration for the compilation of a shorter solution may be significantly less than that of a much larger-sized one. As part of the evaluation, the evaluation function compiles an evolved solution to get an executable routine before navigation begins. The tests confirm that the compilation times of very different-sized individuals are significantly different. This significant contribution by compilation time helps evaluation time to detect differences in size. Therefore, size still influences the evaluation time.

In summary, the difference in sizes impacts compile time; however, the execution time of different sizes may be similar because each is allowed the same number of maximum moves, that is 600. Therefore, it is the internal logic of the program that can gain a program a time advantage (shorter duration) during execution. So an ideal program according to the evaluation time schemes will be the one that is both small in size (hence incurs small compile time) and in execution (hence intelligent logic). In contrast, bloat-control decreases the compile-time while maintaining accuracy; but it does not target execution time, which reflects intelligent logic.

9.2.3 Results

According to the results, the evaluation time schemes can produce efficient solutions by minimising evaluation times. As detailed in Section 9.2.1, a solution translates into a routine that is allowed to make up to 600 moves (move forward, turn left or right). Therefore, efficient solutions are routines that find all the 89 food items making fewer moves than the 600 limit; the fewer, the more efficient.

The efficiency of the best performing individuals in each generation is analysed. Figure 9.4 shows that the average number of moves used by the contending GP methods by generation. Generally, the average number of moves that all the methods use reduces from the first generation to later ones. However, while standard GP (GP) and GP with bloat-control (BC) stopped reducing the

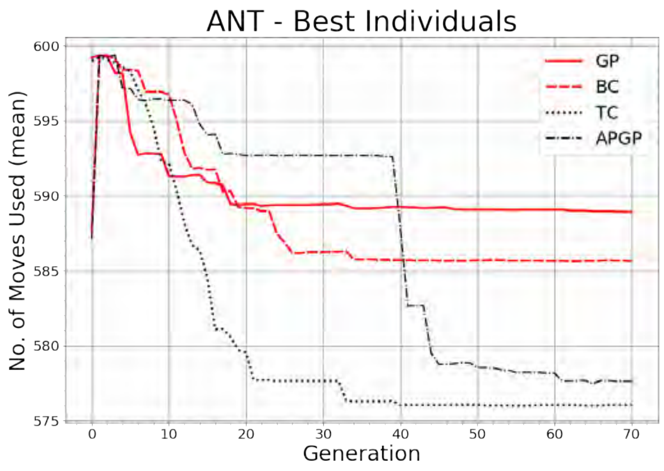


FIGURE 9.4: The figure shows the average number of moves used by the ANT solutions for the different methods. The time schemes (APGP and TC) effectively reduce the average number of moves required by their solutions; therefore, their solutions are more efficient than those of the others.

number of moves after the 34th generation, the time control schemes (APGP and TC) continue to reduce the number they used. Thus, the time schemes produce increasingly efficient solutions. They can do this because the evaluation time detects the difference in the number of moves used to find all food (solutions that use significantly more steps take a longer time); therefore, time schemes encourage efficiency by promoting solutions with shorter evaluation times.

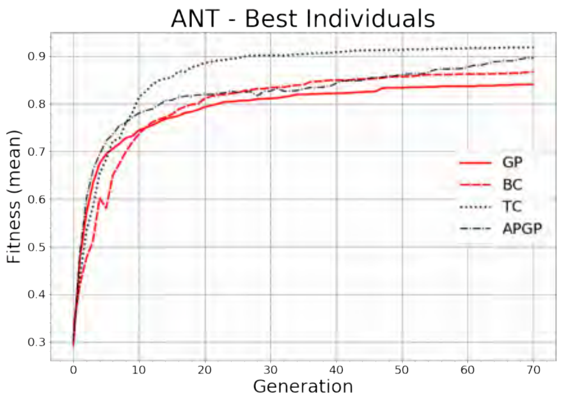


FIGURE 9.5: The average fitness scores of the ANT solutions produced by the methods by generation.

In addition to producing efficient solutions, the evaluation time schemes also produce more accurate and simpler solutions than GP and BC. The figures 9.5, 9.7 and 9.6 show the evolution of fitness, evaluation times, and size respectively. TC controls complexity (evaluation time and size) most effectively and produces the fittest solutions. Similar to some previous results, APGP shows a milder complexity control and produces the next best fitness scores.

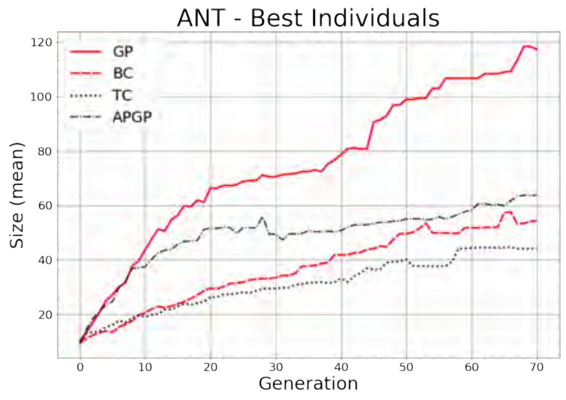


FIGURE 9.6: The average size (length of expressions) of the ANT solutions produced by the methods by generation.

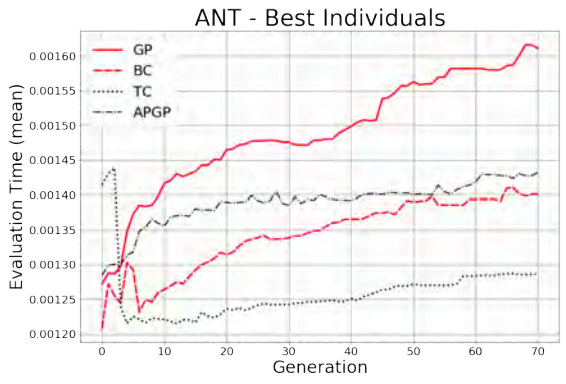
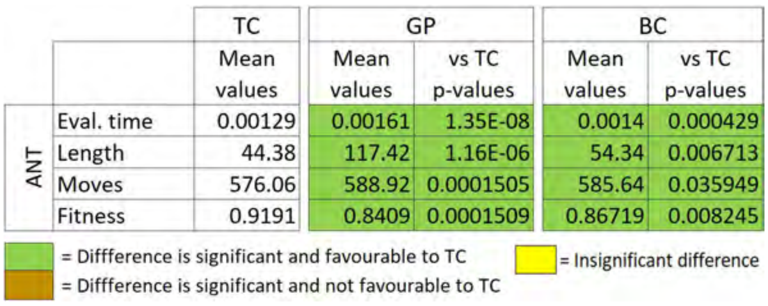


FIGURE 9.7: The average evaluation times of the ANT solutions produced by the methods by generation.

The significance in the differences between the reported results is captured in figures 9.9 and 9.8. The tests compare the best individuals of the methods at the end of the runs. FIGURE 9.8 shows the result of the test for significance of the differences between the output of TC and GP and between TC and BC. The results show that TC produces significantly more accurate, less complex, and more efficient solutions than both GP and BC; improved efficiency is indicate by the fewer number of moves taken to find all the available food items.

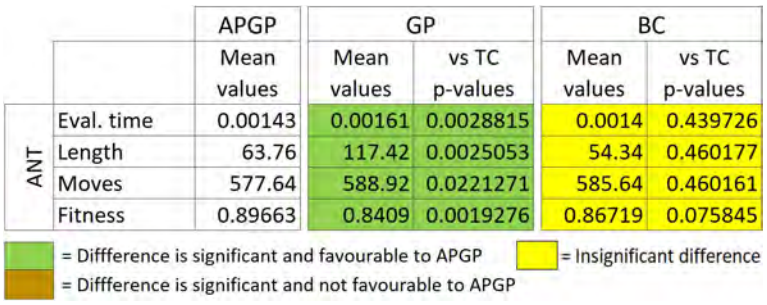


		TC	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
ANT	Eval. time	0.00129	0.00161	1.35E-08	0.0014	0.000429
	Length	44.38	117.42	1.16E-06	54.34	0.006713
	Moves	576.06	588.92	0.0001505	585.64	0.035949
	Fitness	0.9191	0.8409	0.0001509	0.86719	0.008245

🟩 = Diffference is significant and favourable to TC
🟫 = Diffference is significant and not favourable to TC
🟨 = Insignificant difference

FIGURE 9.8: ANT application result of the test for significance in the difference between Time-control (TC) and standard GP (GP), and between TC and GP with bloat-control (BC).

As detailed in FIGURE 9.9, APGP produces significantly more efficient solutions (that use fewer moves) than both GP and BC. Moreover, the solutions that APGP produces are more accurate and simpler (evaluation times and size) than those produced by GP. Furthermore, against BC, APGP produces solutions that are not significantly different in terms of evaluation time, size and fitness.



		APGP	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
ANT	Eval. time	0.00143	0.00161	0.0028815	0.0014	0.439726
	Length	63.76	117.42	0.0025053	54.34	0.460177
	Moves	577.64	588.92	0.0221271	585.64	0.460161
	Fitness	0.89663	0.8409	0.0019276	0.86719	0.075845

🟩 = Diffference is significant and favourable to APGP
🟫 = Diffference is significant and not favourable to APGP
🟨 = Insignificant difference

FIGURE 9.9: ANT application result of the test for significance in the difference between APGP and standard GP (GP), and between APGP and GP with bloat-control (BC).

These results show that time is much more than size in the ANT application. Furthermore, the time schemes deliver qualitative improvements – accuracy and efficiency.

9.3 Boolean Logic Application - Even-Parity

The Even-Parity problem (Parity) is one of the two Boolean logic applications used. Like ANT, Parity is a classic GP challenge that was introduced by Koza [229]. The goal is to find a program that computes the Boolean even parity given n independent Boolean inputs. TGP uses standard Boolean operators (AND, OR, XOR, and NOT) as building blocks (primitives). In addition, it uses two constant terminals: 0 and 1, which represent a true and false state, respectively.

The difficulty of this challenge is tunable. Empirical evidence from the GP literature shows that increasing the number of Boolean inputs makes the problem harder for GP [238, 239]. However, using 6 Boolean inputs is common, which means that the target is to produce a program that computes the parity value for each of the 64 (2^6) possible input combinations. However, here the Parity experiments use two challenging settings – with 9 (Parity9) and 10 (Parity10) inputs that require 2^9 and 2^{10} input combinations, respectively.

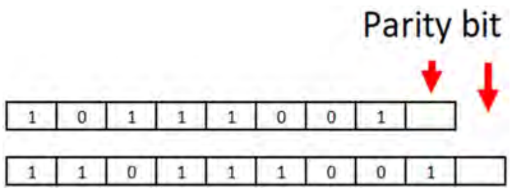


FIGURE 9.10: Parity bits of binary strings.

9.3.1 Experiment

The evaluation function of the Parity application calculates fitness based on the number of times the evolved program successful computes the even parity out of all the input combinations. Therefore, the fitness score of an evolved Parity solution (program) is simply the number of successes represented as a decimal fraction (a value between 0 and 1) of the total input combinations. The experimental settings are detailed in TABLE 9.2.

Parameter	Setting
Number of runs	50
Population size	500
Function set	AND, OR, XOR, and NOT
Terminal set	0 (false), 1 (true)
Initial population	Ramped-half-and-half; depth: min. = 3 , max. = 5
Selection:	Tournament size = 3
Replacement	Steady-state
Probabilities	Crossover = 0.7, mutation = 0.3
Subtree generation	Ramped-half-and-half; depth min. = 0 , max. = 3

TABLE 9.2: Experimental settings for the Parity problems.

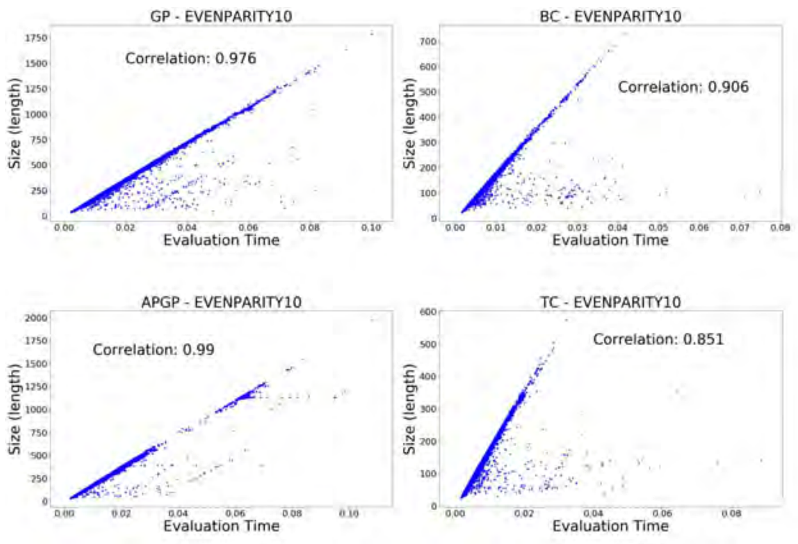
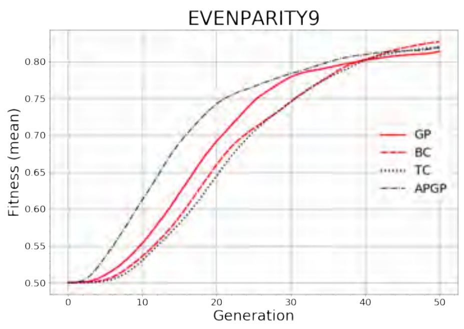


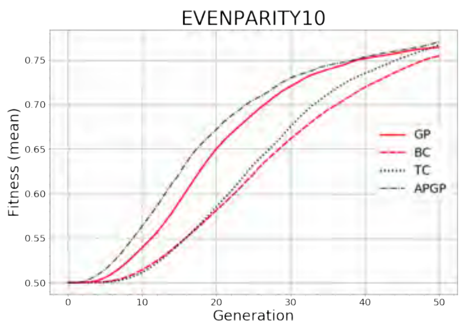
FIGURE 9.11: Correlations between size and evaluation time in Even Parity (Parity10).

9.3.2 Results

The components of Boolean logic solutions do not vary in their computational complexity greatly like those of symbolic regression in Chapters 5 – 6 (e.g., *sin* and *add* require significantly different computation time). In addition, FIGURE 9.11 shows a very high correlation between size and time in the populations of this application; the correlation scores are 0.976 for GP, 0.906 for BC, 0.99 for APGP, and 0.851 for TC. However, it is noteworthy that TC reports a



(A) Parity 9



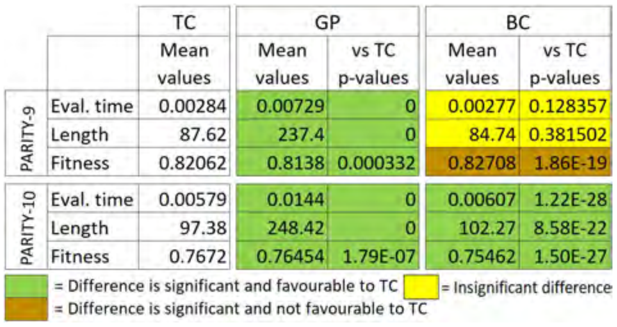
(B) Parity 10

FIGURE 9.12: The charts show the evolution of fitness in the parity problems by the contending methods.

comparatively lower correlation value of 0.851. The results later in this section, as well as those from the Multiplexer problem (another Boolean problem, in Section 9.4) indicate that on occasions, TC *possibly* exploits this fact to exhibit better complexity control.

As expected, the results of TC and BC were similar in this application; these methods explicitly control time and size, respectively. FIGURE 9.12, which represents the evolution of fitness, shows that BC and TC overlap several times (in both Parity9 and Parity10). Furthermore, comparisons of the final solutions of TC and BC show insignificant differences in some performance indicators and divided performance in others. As detailed in FIGURE 9.13, the complexity measures (evaluation time and size) of TC and BC do not differ significantly in Parity9 but do so in Parity10 in favour of TC. In terms of accuracy, TC was better in one (Parity10), and BC was better in the other (Parity9).

In line with previous results, the complexity control that APGP offers in this application is milder than TC and BC. Therefore, the solutions that APGP produces are less complex (both size and time) than GP's but more complex than BC's. Furthermore, the final population of APGP is more accurate than that of GP; also, it was more accurate than that of BC in Parity10 but less than that of BC in Parity9.

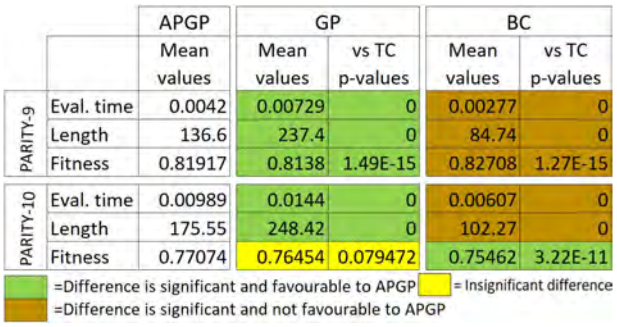


		TC	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
PARITY-9	Eval. time	0.00284	0.00729	0	0.00277	0.128357
	Length	87.62	237.4	0	84.74	0.381502
	Fitness	0.82062	0.8138	0.000332	0.82708	1.86E-19
PARITY-10	Eval. time	0.00579	0.0144	0	0.00607	1.22E-28
	Length	97.38	248.42	0	102.27	8.58E-22
	Fitness	0.7672	0.76454	1.79E-07	0.75462	1.50E-27

🟩 = Difference is significant and favourable to TC 🟨 = Insignificant difference
🟧 = Difference is significant and not favourable to TC

FIGURE 9.13: Even Parity: significance test result for the difference between TC vs GP and TC vs BC.

The graphs in FIGURE 9.12 show that APGP trains the fastest. For example, at the 10th generation of Parity9, APGP reached a fitness score of 0.61 while GP, TC, and BC were at 0.51, 0.53, and 0.53, respectively. Similarly, at the 10th generation of Parity10, APGP reached a fitness score of 0.56 while GP, TC, and BC were at 0.54, 0.51, and 0.51, respectively. This trend continues to the later generations (generation 40), where the accuracy values converge to similar values. This observation tallies with the findings in Chapter 6 that measured and compared the training speed of the methods – the number of evaluations used to meet the average fitness of GP.



		APGP	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
PARITY-9	Eval. time	0.0042	0.00729	0	0.00277	0
	Length	136.6	237.4	0	84.74	0
	Fitness	0.81917	0.8138	1.49E-15	0.82708	1.27E-15
PARITY-10	Eval. time	0.00989	0.0144	0	0.00607	0
	Length	175.55	248.42	0	102.27	0
	Fitness	0.77074	0.76454	0.079472	0.75462	3.22E-11

🟩 =Difference is significant and favourable to APGP 🟨 = Insignificant difference
🟧 =Difference is significant and not favourable to APGP

FIGURE 9.14: Even Parity: significance test result for the difference between APGP vs GP and APGP vs BC.

In summary, although in Parity problems the evaluation time of individuals is highly correlated with their size, TC still managed to control complexity *at* *least* as well as BC. The following section further investigates this possibility with another Boolean problem.

Furthermore, consistent with the results from previous chapters, APGP exerts a gentler complexity control, but there is some indication that it trains faster.

9.4 Boolean Logic Application - Multiplexer

The multiplexer problem (Multiplexer, henceforth) is another Boolean logic application that has been extensively studied in GP [229]. The challenge is for GP to use Boolean logic to evolve a program that replicates the behaviour of an electronic multiplexer given all its possible input and output values. A 3-8 multiplexer setting is commonly used, which means three address lines (labelled A0 to A2), and eight data lines (labelled D0 to D7) together make 11 input lines. Therefore, the data will consist of 2^{11} input combinations and their corresponding outputs.

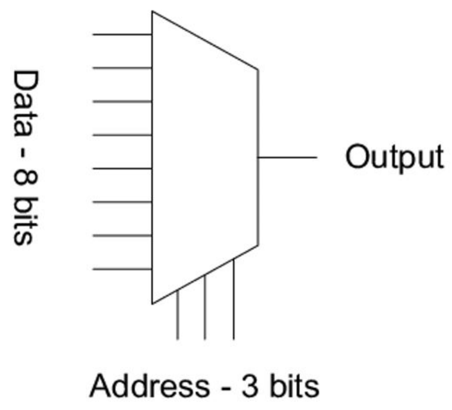


FIGURE 9.15: The Multiplexer with 3 address and 8 data lines.

The multiplexer problem falls under the GP applications area that designs electronic circuits. While the design of simple digital circuits is easy using conventional methods, it is difficult and time-consuming for complex circuits. Therefore, researchers and designers are turning to GP for automation of the design of such circuits; for example, the challenging task of designing asynchronous electronic circuits is leveraging GP [103]. Moreover, in addition to the functionality and accuracy of the automatically generated solutions that GP offers, the proposed complexity control methods in this thesis can provide simplicity and efficiency.

9.4.1 Experiment

With two exceptions, the experimental settings are the same as the Parity application; TABLE 9.2 details the parameters. The additional settings for Multiplexer are: (1) the `XOR` logical operator is excluded, (2) the logical `If-Then-Else` is included. The training data is all the possible inputs of the 3-8 multiplexer and the corresponding outputs.

9.4.2 Results

Like the Parity application (Section 9.3), which is also a Boolean logic application, the evaluation times and size are practically the same in Multiplexer. From their results, TC and BC behave similarly; these methods use identical mechanisms to manage complexity by controlling evaluation time and size. The overlapping of their fitness graphs in FIGURE 9.16 indicates the similarity.

In addition, a high correlation is observed between the size and evaluation times of individuals in the final population of the methods; FIGURE 9.17 shows correlation scores of 0.985, 0.885, 0.982, and 0.883 for GP, BC, APGP, and TC, respectively. Again, as with the Parity problem earlier on, TC reports a relatively lower correlation. Correspondingly, the results later show that TC possibly exploits this lower correlation to produce slightly but significantly better results.

Note, in FIGURE 9.16, TC and BC – which control complexity more aggressively (explicitly) than APGP – train slower (from generation 1 to 34) than APGP and GP.

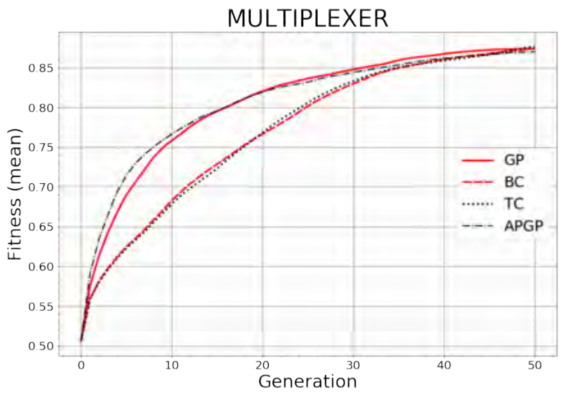


FIGURE 9.16: Multiplexer: The mean fitness scores by generation of the contending methods.

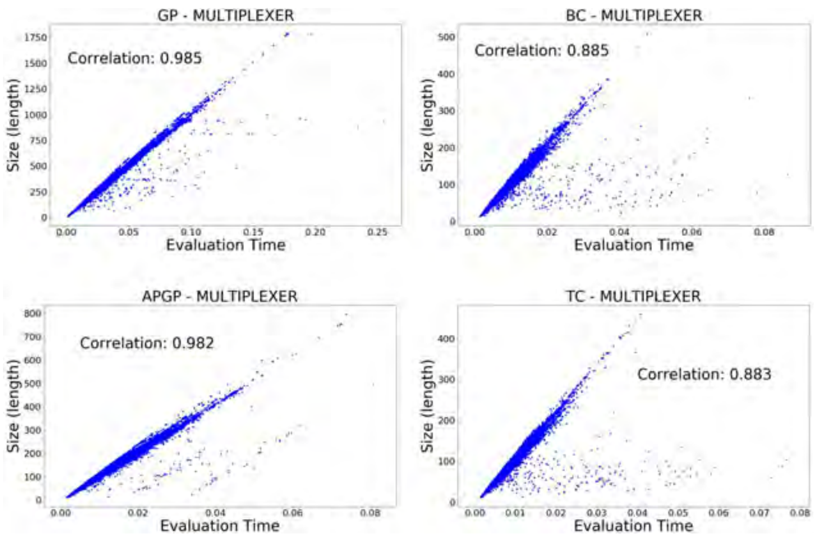
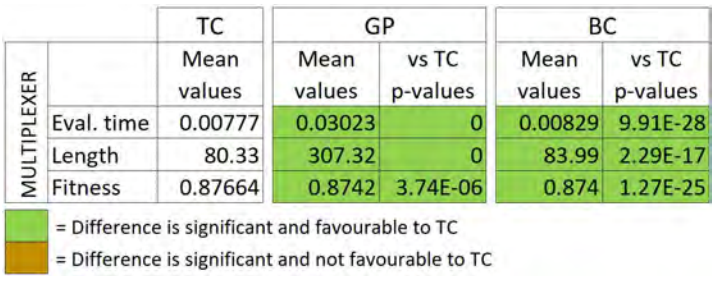


FIGURE 9.17: Correlations between size and evaluation time in Multiplexer applications.

However, a small but significant difference in the final populations they produced is observed. As detailed in FIGURE 9.18, TC delivers significantly simpler (shorter evaluation times and smaller size) and more accurate solutions than BC and GP.

As commented earlier, it is possible that TC exploits a relatively lower size-time correlation to differentiate its results from the other methods. As to why TC and (to an extent) BC report lower correlations are not entirely clear; however, one clue may be that these methods, on average, produce smaller sizes, which deviate from an otherwise linear relationship; this deviation decreases the correlation.

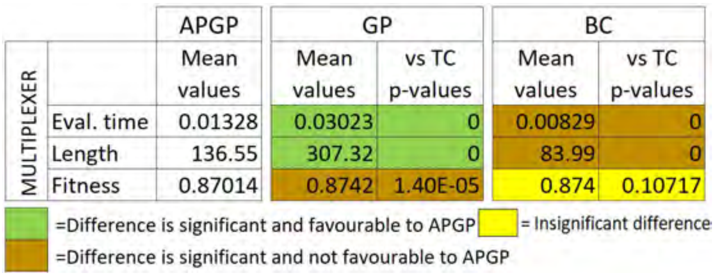


		TC	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
MULTIPLEXER	Eval. time	0.00777	🟩 0.03023	🟩 0	🟩 0.00829	🟩 9.91E-28
	Length	80.33	🟩 307.32	🟩 0	🟩 83.99	🟩 2.29E-17
	Fitness	0.87664	🟩 0.8742	🟩 3.74E-06	🟩 0.874	🟩 1.27E-25

🟩 = Difference is significant and favourable to TC
🟧 = Difference is significant and not favourable to TC

FIGURE 9.18: Multiplexer: significance test result for the difference between Time-control (TC) vs standard GP and GP with bloat-control.

As observed in previous results, the complexity control of APGP is gentler than TC in Multiplexer. As detailed in FIGURE 9.18, APGP produces solutions that have significantly shorter evaluation times and sizes than GP but not BC. In terms of accuracy, the difference between APGP and BC is not significant; APGP produces significantly less accurate solutions than GP.



		APGP	GP		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
MULTIPLEXER	Eval. time	0.01328	🟩 0.03023	🟩 0	🟧 0.00829	🟧 0
	Length	136.55	🟩 307.32	🟩 0	🟧 83.99	🟧 0
	Fitness	0.87014	🟧 0.8742	🟧 1.40E-05	🟨 0.874	🟨 0.10717

🟩 =Difference is significant and favourable to APGP 🟨 = Insignificant difference
🟧 =Difference is significant and not favourable to APGP

FIGURE 9.19: Multiplexer: significance test result for the difference between APGP vs standard GP and GP with bloat-control.

9.4.3 Summary of Boolean Logic Results

On both the Boolean logic applications (Multiplexer and Parity), in terms of complexity control, TC performed at least as well as BC (GP with bloat-control) and strictly better than GP. TC also produced the best fitness values two out of three times. As to why TC outperforms other GP methods is perhaps rooted in the relatively lower size-time correlations that it induces in its population. As to what causes this lower correlation is not entirely clear; however, the results indicated one clear differentiating factor – on average a lower size in TC populations. Overall, the improvement that TC realises over BC was not as clear as against GP.

Again, APGP did not outperform BC or TC in terms of complexity control but it did outperform GP. In terms of accuracy the results were mixed: better in one, worse in another and similar in the third.

Overall, the results suggest that time-control (especially TC) continues to be a versatile complexity control mechanism in GP applications: where possible, it exploits the time-size disparity to curb complexity, but even otherwise, it is no worse than GP with or without the traditional complexity control.

9.5 Classification with GP Aided by Machine Learning

This section uses GP as a machine learning tool for classification [96]. Although both regression and classification approximate functions (models) from historical data, they differ fundamentally. While regression models produce continuous output values, classification models provide discrete ones. The discrete output of a classification model reflects the categorisation of data instances into one of a given set of classes. Therefore, in symbolic regression, GP can identify and reward the minute improvement of one model over another. The minute improvement is detected because the fitness score of a regression model reflects how far its prediction is from the actual continuous value on each instance of the training data. Instead, the fitness score of a classification model is the fraction of the data instances that it has classified correctly. As such, classification in GP requires a change in the model that is significant enough to affect its behaviour and performance.

The classification application in this section uses a hybridisation of GP with machine learning. Like the hybridisation of MLR-GP in Chapter 8, again here

GP finds features while logistic regression [232] classifies the data instances by building a model based on the evolved features; this GP hybrid is named *GP with machine learning* (GPML).

The rationale for using GPML is to attain a GP based classification system that trains better than standard GP. Such a system increases the likelihood of model overfitting; as discussed in Chapter 8.2, traditional GP often underfits the data [209]. As preventing overfitting (to maintain generalisation) is the motive for managing complexity in classification, GPML allows us to assess the time schemes in this regard.

Before using the GPML application to assess the time schemes, it is benchmarked with standard GP and with a selection of popular machine learning algorithms for classification. The result shows that GPML generally outperforms the other methods.

Furthermore, the results show that managing complexity with the time control schemes benefits GPML: complexity and overfitting are both reduced.

9.5.1 Experiment

The classification experiments in this section use all the basic settings as MLRGP (in Chapter 8, Table 8.4) and add logical functions. The additional functions and terminals are: logical `If-Then-Else` , `Less-Than`, and `Greater-Than`; Boolean `AND`, `OR`, and `NOT`; and the terminals `True` and `False`. The experiments compare APGP and TC with GPML without complexity control (GPML) and GPML with bloat-control (BC).

GPML Evaluation function

Like the evaluation function of MLRGP (Chapter 8), the GPML evaluation function starts by extracting the features of the individual. Next, the evaluation function evaluates the features with the training data to populate a design matrix with the values of the evaluated features. Thereafter, the design matrix is used as training data to build a classification model with logistic regression (a machine learning algorithm). Finally, the evaluation function calculates the fitness score of the model as a fraction of all instances in the data that it classifies correctly. Therefore, the fitness score is a decimal fraction that ranges between 0 and 1 inclusively.

The Classification Problems

The datasets used for the classification problems are available at [194]; Table 9.3 details the datasets. Brief descriptions of the datasets are as follows:

1. *Bank Note:* The banknote dataset (BankNote) contains digital images from banknote-like specimens. The instances are labelled as either forged or genuine notes.
2. *Blood Transfusion:* The blood transfusion dataset (Blood) consists of 748 donor data from a blood transfusion service centre. The challenge is to build a model for determining if a donor donated within a period.
3. *Pima Indians diabetes:* The Pima Indians diabetes dataset (Diabetes) consist of diagnostic measurements that predict whether or not a patient has diabetes; examples of the measures include the patients' age, glucose level, and body mass index.
4. *Indian Liver Disease:* The Indian liver disease dataset (Liver) contains the labelled data of healthy and liver disease patients (such as age, gender, protein levels, and bilirubin characteristics).
5. *Ozone:* This dataset contains environment measurements that predict air quality. The dataset includes the ozone level and other environmental measurements such as temperature, wind speed, and precipitation.

6. *Spam Mail:* In the Spam Mail dataset (Spam), each row is a Boolean vector of size 57 and each column flags the presence or otherwise of a specific word in the contents of an email. The rows are labelled as either spam emails or not.

ID	Data-Set Name	Variables	Instances	Class Distribution
1	Bank Note	3	1371	1: 44%, 0: 56%
2	Blood Transfusion	4	748	1: 76%, 0: 24%
3	Diabetes	8	767	1: 35%, 0: 65%
4	Liver Disease	10	583	1: 71%, 0: 29%
5	Ozone	72	2534	1: 93%, 0: 6.3%
6	Spam Mail	57	4601	1: 39%, 0: 60.6%

TABLE 9.3: Datasets for the classification experiments.

9.5.2 Benchmarking GPML with other Classification Algorithms

The results of GPML are compared with those from standard GP and six popular machine learning algorithms for classification. The machine learning algorithms include logistic regression, which is the algorithm used to hybridise GP to produce GPML. The others are Decision Tree Classifier (DecisionTree), Support Vector Classifier (SVC), Random Forest Classifier (RandomForest), Gaussian Naive Bayes (GaussianNB), and K-Neighbors Classifier (KNeighbors); the implementations in the SKLearn Library [240].

As illustrated in Figure 9.20, GPML outperformed the other machine learning algorithms. GPML produced significantly more accurate solutions (both training and test fitness scores) in 5 out of 6 problems; the non-overlapping error bars indicate significance. On the sixth (SPAM), GPML did not do better than RandomForest, LogisticRegression, and KNeighbors.

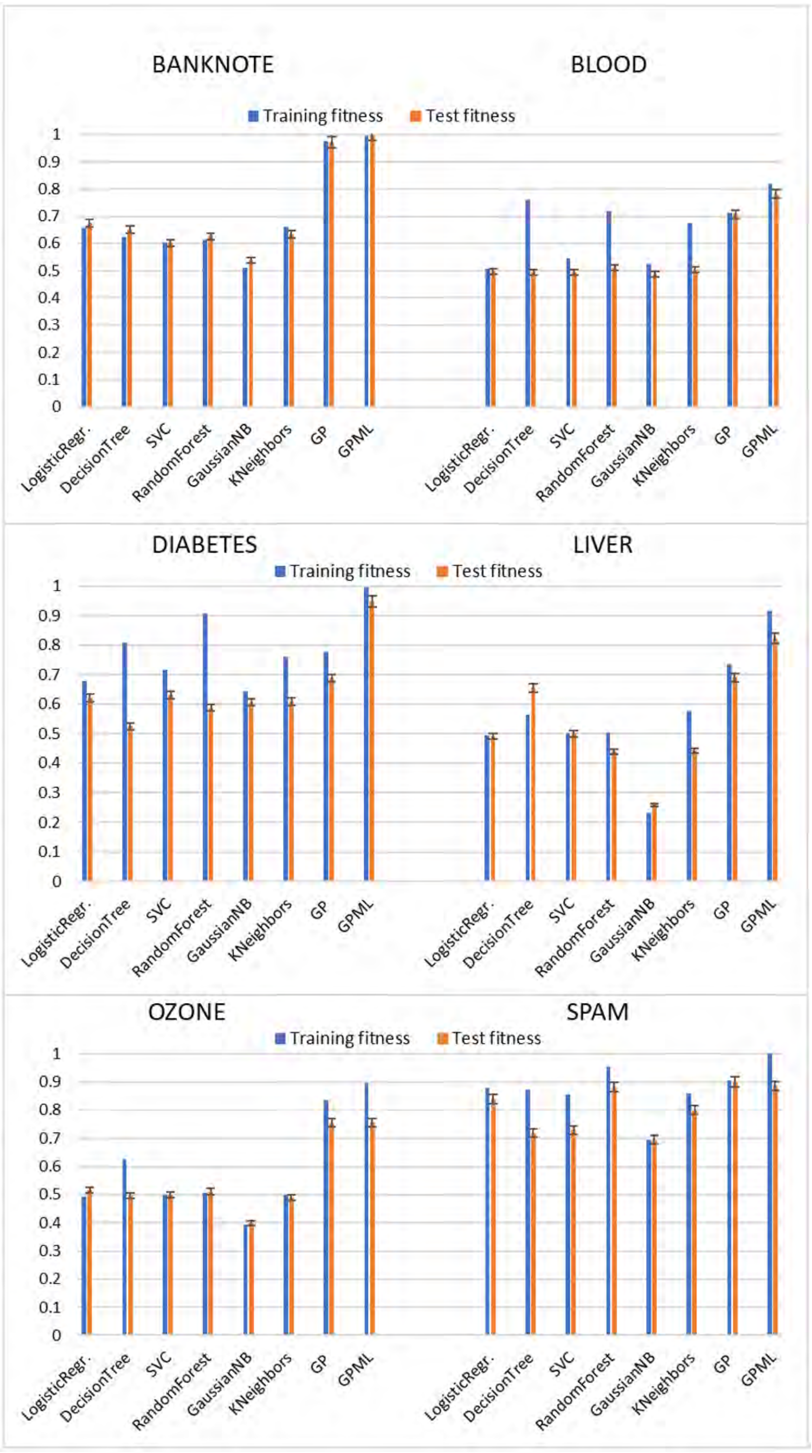


FIGURE 9.20: Comparing GP and GPML with other machine learning classification algorithms.

Furthermore, GPML performed at least as well as GP on all the problems; it was significantly better in three out of six. Therefore, GPML (the hybrid) offers a performance advantage over the constituent components (GP and LogisticRegression).

9.5.3 Time Detects More than Size in Classification with GPML

Like the results in Chapter 8, the evaluation time reflects the size, number of features, and components of individuals. Similarly, the correlation between evaluation time and the number of features is stronger than between evaluation time and size for all but the SPAM problem; see Figure 9.21 for details.

9.5.4 Results: Complexity Control

To assess how the time schemes control complexity, the analysis considers the evaluation times, sizes, and the number of features of the models produced by the methods. Figures 9.22 and 9.23 summarise the result of the test for significance in the difference between the final populations.

APGP generally produces significantly simpler solutions than GPML (standard) but more complex than BC. Compared with GPML, APGP produced solutions with significantly shorter evaluation times in 4 out of 6 problems, smaller sizes in 4 out of 6, and fewer features in 5 out of 6. Compared with BC, APGP produced solutions with significantly shorter evaluation times in 1 out of 6 problems and fewer features in 2 out 6.

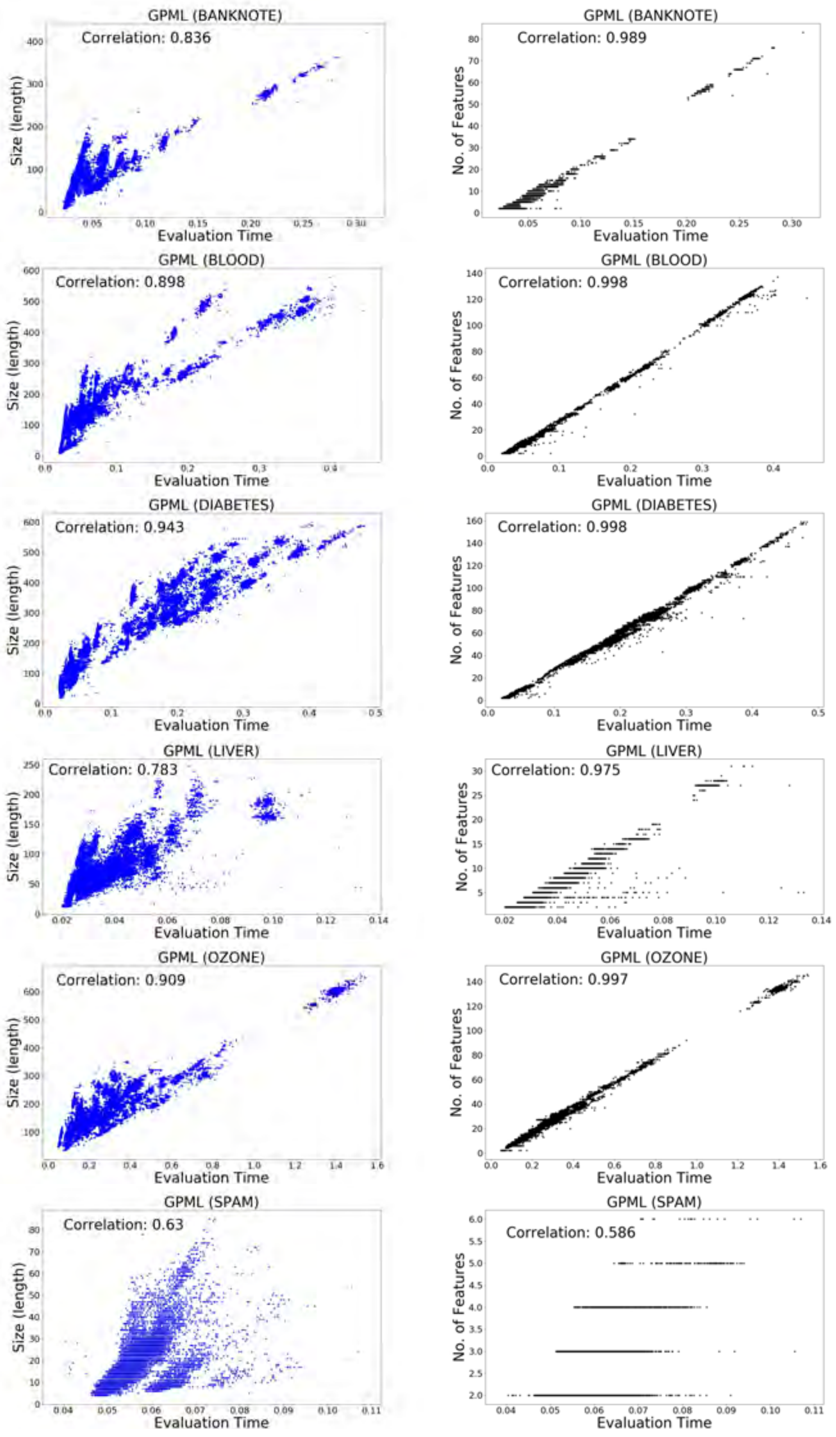
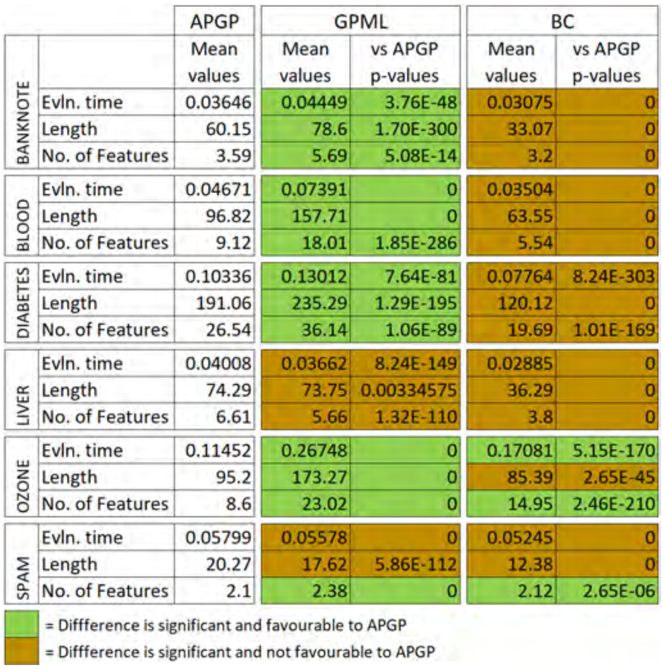


FIGURE 9.21: The figure shows the correlation of evaluation times with size (column 1) and the number of features (column 2) in the classification with GPML application. Evaluation time shows a stronger correlation with the number of features than with size, except for the SPAM problem.



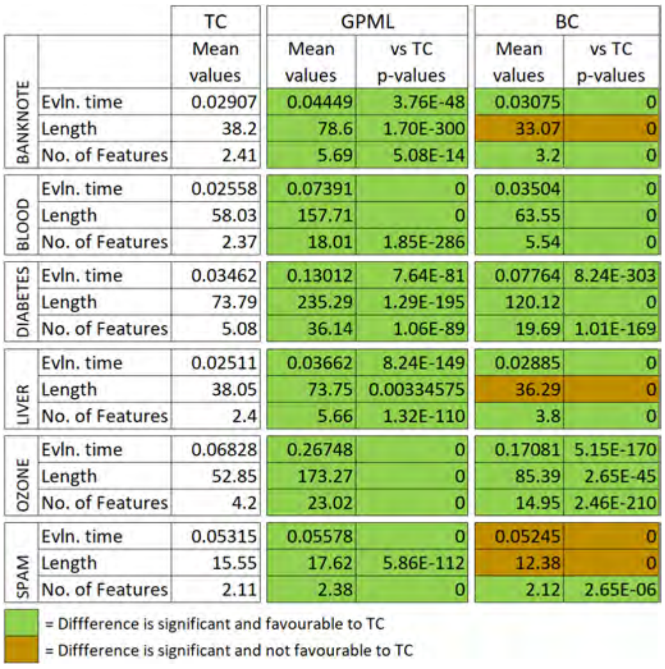
		APGP	GPML		BC	
		Mean values	Mean values	vs APGP p-values	Mean values	vs APGP p-values
BANKNOTE	Evln. time	0.03646	🟩 0.04449	🟩 3.76E-48	🟧 0.03075	🟧 0
	Length	60.15	🟩 78.6	🟩 1.70E-300	🟧 33.07	🟧 0
	No. of Features	3.59	🟩 5.69	🟩 5.08E-14	🟧 3.2	🟧 0
BLOOD	Evln. time	0.04671	🟩 0.07391	🟩 0	🟧 0.03504	🟧 0
	Length	96.82	🟩 157.71	🟩 0	🟧 63.55	🟧 0
	No. of Features	9.12	🟩 18.01	🟩 1.85E-286	🟧 5.54	🟧 0
DIABETES	Evln. time	0.10336	🟩 0.13012	🟩 7.64E-81	🟧 0.07764	🟧 8.24E-303
	Length	191.06	🟩 235.29	🟩 1.29E-195	🟧 120.12	🟧 0
	No. of Features	26.54	🟩 36.14	🟩 1.06E-89	🟧 19.69	🟧 1.01E-169
LIVER	Evln. time	0.04008	🟧 0.03662	🟧 8.24E-149	🟧 0.02885	🟧 0
	Length	74.29	🟧 73.75	🟧 0.00334575	🟧 36.29	🟧 0
	No. of Features	6.61	🟧 5.66	🟧 1.32E-110	🟧 3.8	🟧 0
OZONE	Evln. time	0.11452	🟩 0.26748	🟩 0	🟩 0.17081	🟩 5.15E-170
	Length	95.2	🟩 173.27	🟩 0	🟧 85.39	🟧 2.65E-45
	No. of Features	8.6	🟩 23.02	🟩 0	🟩 14.95	🟩 2.46E-210
SPAM	Evln. time	0.05799	🟧 0.05578	🟧 0	🟧 0.05245	🟧 0
	Length	20.27	🟧 17.62	🟧 5.86E-112	🟧 12.38	🟧 0
	No. of Features	2.1	🟩 2.38	🟩 0	🟩 2.12	🟩 2.65E-06

🟩 = Diffference is significant and favourable to APGP

🟧 = Diffference is significant and not favourable to APGP

FIGURE 9.22: Result of statistical tests of the difference of the complexity control in the GPML application between APGP and GPML (GPML without complexity control) and between APGP AND BC (GPML with bloat-control). The difference in the final populations of the methods were tested.

TC produced the simplest solutions. Compared with GPML, TC produced simpler solutions (smaller evaluation times, sizes, and number of features) on all 6 out of 6 problems. Compared with BC, TC produced solutions with shorter evaluation times in 5 out of 6 problems, smaller sizes in 3 out of 6, and fewer features in 6 out of 6. Note, in some instances (three in Figure 9.23 and one in Figure 9.22) the time schemes produced shorter evaluation times and fewer features but not smaller sizes; this observation further confirms that size is indeed not time in this application.



		TC	GPML		BC	
		Mean values	Mean values	vs TC p-values	Mean values	vs TC p-values
BANKNOTE	Evln. time	0.02907	0.04449	3.76E-48	0.03075	0
	Length	38.2	78.6	1.70E-300	🟧 33.07	🟧 0
	No. of Features	2.41	5.69	5.08E-14	3.2	0
BLOOD	Evln. time	0.02558	0.07391	0	0.03504	0
	Length	58.03	157.71	0	63.55	0
	No. of Features	2.37	18.01	1.85E-286	5.54	0
DIABETES	Evln. time	0.03462	0.13012	7.64E-81	0.07764	8.24E-303
	Length	73.79	235.29	1.29E-195	120.12	0
	No. of Features	5.08	36.14	1.06E-89	19.69	1.01E-169
LIVER	Evln. time	0.02511	0.03662	8.24E-149	0.02885	0
	Length	38.05	73.75	0.00334575	🟧 36.29	🟧 0
	No. of Features	2.4	5.66	1.32E-110	3.8	0
OZONE	Evln. time	0.06828	0.26748	0	0.17081	5.15E-170
	Length	52.85	173.27	0	85.39	2.65E-45
	No. of Features	4.2	23.02	0	14.95	2.46E-210
SPAM	Evln. time	0.05315	0.05578	0	🟧 0.05245	🟧 0
	Length	15.55	17.62	5.86E-112	🟧 12.38	🟧 0
	No. of Features	2.11	2.38	0	2.12	2.65E-06

🟩 = Diffference is significant and favourable to TC

🟧 = Diffference is significant and not favourable to TC

FIGURE 9.23: Result of statistical tests of the difference of the complexity control in the GPML application between TC and GPML (GPML without complexity control) and between TC AND BC (GPML with bloat-control). The difference in the final populations of the methods were tested.

9.5.5 Results: Accuracy and Generalisation

In the GPML classification application, the motivation for complexity control is to attain solutions that generalise well, which is reflected by test fitness scores. Therefore, the analysis considers the test fitness scores (from unseen data) as the measure of accuracy and the indicator of the generalisation ability of the models.

Figure 9.24 shows both the average training fitness (column 1) and test fitness (column 2) scores by generation. The average training fitness scores increase by generation as expected. However, the test fitness fluctuates, which indicates overfitting. As a result of the different fluctuations shown per method, the analysis does not compare the scores at a single point, such as the final population. Therefore, the peak average fitness values that each method attains are

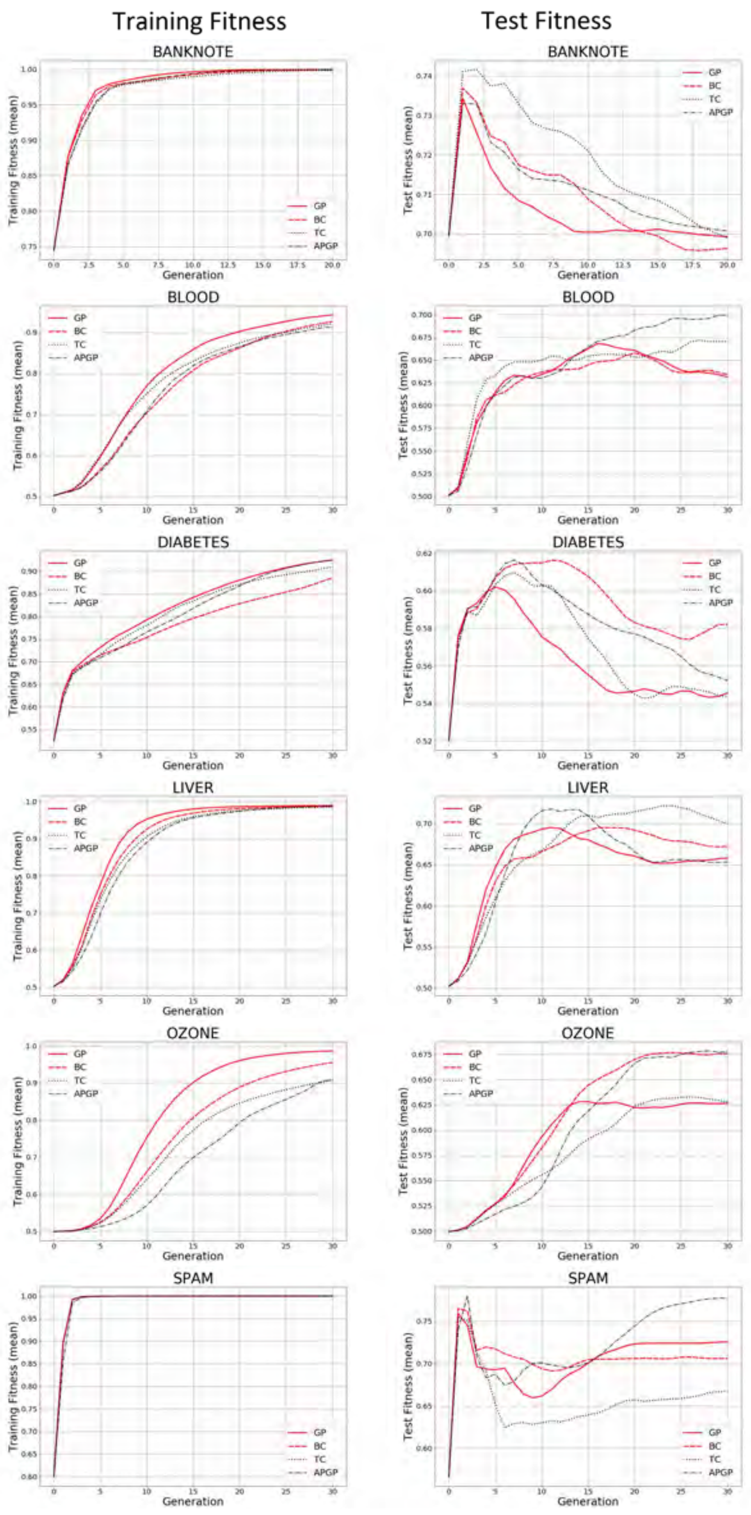


FIGURE 9.24: The mean training and test fitness (accuracy) by generation of the methods in GPML.

considered.

Table 9.4 shows a comparison of the peak test fitness values attained by the various method on all the problems. The highest test fitness scores per problem is indicate by the bold text in the table. APGP produced the highest peaks in 4 out of 6 problems and TC produced the highest in the other 2 out of 6. Therefore, the complexity control of the time schemes produce models that generalise the best.

Problem	APGP	TC	BC	GPML
BLOOD	*0.67084*	**0.67192**	0.65777	0.66825
LIVER	*0.71747*	**0.72187**	0.69504	0.695
BANKNOTE	**0.86512**	0.81278	0.84266	*0.84942*
DIABETES	**0.61631**	0.60975	*0.6161*	0.6019
SPAM	**0.77969**	0.75465	*0.76512*	0.75901
OZONE	**0.67913**	0.63282	*0.67667*	0.62802

TABLE 9.4: Peak average test fitness scores of GPML and the compared methods on classification problems.

9.6 Discussion

This chapter tested the versatility of complexity control via curbing the evaluation times on problems from several different domains. The experiments sought to ascertain whether time control effectively curbs complexity without significant side effects across various problem domains. In addition, the investigation wanted to explore the types of complexity that the evaluation time determines across the GP applications and, subsequently, examine how the proposed time-based system addresses the motivations for managing complexity.

In the robot control application (ANT), the time schemes can detect the solutions that find all the available food items using fewer moves and hence have shorter evaluation times. Therefore, evaluation time recognises efficiency – a non-functional requirement of a piece of code – in a way that size can not. Improving non-functional requirements can be sought-after in real-world applications. For example, in automatic programming, a small expression (such as a nested loop) may translate to a resource-intensive and inefficient program; time control can detect and discourage this type of inefficiencies.

In the classification application (GPML), the time schemes effectively detect and manage the number of features in a solution, which bloat-control can not match. This improved characterisation of complexity that time offers produced classification models that generalise better than those by bloat-control and standard GPML.

As anticipated, because of the similarity in the computational complexity of the logical operators that Boolean applications use, the effect of controlling complexity via time and via size were similar. However, the explicit time control (TC) was at least as good as bloat-control in terms of both accuracy and complexity control. The lower correlation between evaluation time and size that TC showed suggests that time can find and capitalise on some differences. As to why TC and (to an extent) BC report lower correlations is not entirely clear; however, one clue may be that these methods, on average, produce smaller sizes, which cause a greater time-scatter and hence a deviation from an otherwise linear relationship; this deviation decreases the correlation.

Generally across all the problem domains considered in this chapter, the results show that TC almost consistently yields solutions with lower evaluation times than both standard GP (GP) and GP with bloat-control (BC). Since small expressions can sometimes demand more resources than larger ones (as demonstrated in Chapter 4 and this chapter), discouraging evaluation time addresses the resource utilisation and run time challenge better than discouraging size. Therefore, lower evaluation time itself is an advantage for the time schemes.

Whilst shorter evaluation times of solutions are closely reflected by smaller sizes in the Boolean applications, it is very different in the others. Despite the differences across the various applications, the time schemes generally produced solutions with smaller sized expressions than those by standard GP (or its appropriate variants such as GPML), and TC did not produce more complex solutions than BC. Although the analysis did not investigate the interpretability of the evolved models, the time schemes also help with that *as long as* parsimonious expressions are more interpretable.

The results in this chapter show that evaluation time is a versatile measure of complexity. Unlike bloat-control that is popularly used in GP to control complexity, the evaluation time schemes manage the complexity of GP solutions in various ways to address the diverse motives behind the quest for simplicity.

9.7 Conclusion

The results in this chapter demonstrate that time characterises complexity in a resourceful way; when time is employed to manage complexity, several qualitative advantages are achieved. In the ANT application, time distinguishes between efficient solutions (those that find all food items using minimal moves) from the inefficient ones. Next, although the computational complexity of the operators in the Boolean applications may appear to be similar, TC still showed a difference; TC was at least as accurate and as simple as bloat-control. Moreover, In the GPML application (classification), time effectively detects the number of features of solutions, which bloat-control could not - hence leading to a more rounded complexity control that improves the generalisation capacity of the models. Therefore, the results indicate that the proposed time-based system of managing complexity is versatile, and it addresses the motivations behind managing the complexity of GP solutions well.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

This thesis presents a system for managing the complexity of the solutions GP produces. Unlike many other evolutionary algorithms that use fixed-length structures to represent the evolving solutions, GP evolves variable-length ones, which is, in part, driven by the idea that computer programs are not necessarily limited to fixed lengths when written. Consequently, GP has its own unique evolutionary dynamics, and one of the most commonly observed results is that the evolving solutions become increasingly more complex with time – even prohibitively so.

Crucially, because GP is applied to produce solutions in many domains, the notion of complexity differs from one application domain to another (e.g., machine learning, automatic programming and design), and so do the motivations for curbing the complexity of these solutions. As a result, producing a universal and formal definition of complexity is hard, and, therefore, one is yet to exist. Yet, to control the complexity of the evolving solutions, we must first quantify it objectively. Therefore, the common practice in GP is to resort to proxies for indicating complexity; these proxies address varying concerns from system resources required for executing the evolving solutions to their functional characteristics – whatever the implementer prioritises at the time. However, it is clear that these motivations are not always transferable across the various applications and sometimes fail to answer the myriad questions concerning complexity even within the same domain.

Therefore, this thesis presents a simple idea to measure the complexity of the evolving solutions in any GP application – the time it takes to evaluate a solution or *evaluation time*. The idea stems from the fact that a simple task takes a shorter time to perform than a more complex one. Hence, this evaluation time – even if not perfect in all aspects – can at least *differentiate* the complexity of GP solutions in a computational sense because GP evaluates all the evolving solutions to a given application in an identical environment. However, questions that arise are whether this idea will hold in practice and whether the evaluation time measurements can be reliable.

Therefore, before applying the proposed evaluation time to manage the complexity of the GP solutions, its viability and practical challenges are studied. The result of the study reveals that the evaluation time detects differences in both the structural and computational complexity of GP solutions. In addition, it shows that evaluation time is a versatile measure of complexity that, in a functionally diverse population, it can elicit functional or computational differences, whereas, in a functionally converged population, it switches to act as a bloat-control method. This versatility shows that when conditions permit, evaluation time control can capture various nuances of complexity as prevalent in the evolving population. Crucially, to ensure that the evaluation time measurements are reliable and not distorted by the effect of system decisions, this thesis proposes and tests strategies that effectively improve the consistency of the time measurements. Therefore, unlike other GP studies that use time measurement without ensuring consistency, this chapter sets an appropriate and reliable stage for investigating the value of the evaluation time as a measure of complexity. As previous work has found attaining reliable time measurements particularly challenging, the usefulness of this finding can go beyond this work.

The next question is how to manipulate the evaluation time of solutions to make them simple. The first approach leverages the machinery of the well-precedented bloat-control techniques to constrain (with explicit penalties) the evaluation times of the GP solutions. Identical mechanisms and settings control size on one set of experiments and evaluation time in another so that the results can show the benefit of using the proposed measure or otherwise. The outcomes decisively show that the time control methods manage the functional complexity of models to make them generalise better (higher test fitness scores) than the bloat-control methods. Also, the solutions were frequently simpler than those by bloat-control (smaller size and evaluation times). This result suggests that evaluation time can detect and prefer less computational complexity (and smaller sizes) to produce less functional complexity. As this approach only involves a relatively minor adjustment to a well-accepted practice, it can be adopted readily and widely by the GP users as a complexity control method. The next question is whether using time as a measure of complexity can open new opportunities for complexity control beyond merely replacing size with time in the bloat-control methods.

Therefore, this thesis presents a new approach for controlling complexity – an opportunity evaluation time opens – that implicitly discourages high evaluation time (APGP) and represents a significant change in the evolutionary dynamics of GP. Instead of explicit penalties, the APGP incorporates a race condition in the evolutionary process that allows simple individuals (with relatively shorter evaluation times) to gain an evolutionary advantage – this is a metaphor from natural evolution. In the APGP, the fast evaluating (simple) candidate solutions gain an evolutionary advantage when they finish evaluating and get into the breeding population earlier than their slower (more complex) counterparts. These simple solutions in the breeding population can breed (similarly simple and potentially more accurate offspring) while their slower peers are busy evaluating. The results in this chapter show that the APGP, like the explicit time control methods (TC), overwhelming outperforms bloat-control, and it is competitive with the TC. Moreover, the APGP offers a gentler complexity control than TC. Another significant finding is that APGP trains faster (uses fewer evaluations to meet accuracy targets) than all the other methods; on the contrary, bloat-control slows down training. As a general principle in evolutionary computing is to mimic concepts from nature to enhance the algorithms, the success of this system, which mimics nature further, thus designates a significant development.

Given that APGP is unstudied and represents a significant shift in the GP paradigm, analysing it is imperative. Findings from the analysis validate the assumptions made while conceptualising the idea of the APGP. Accordingly, the result confirms that the computationally efficient solutions outpace their expensive counterparts to enrol into the population; therefore, verifying that the APGP results are indeed induced by its internal evolutionary dynamics influenced by the evaluation time control method. Furthermore, the analysis shows that increasing the degree of concurrency opens up more opportunities for the simple solutions to finish evaluating early and try to get ahead. A welcome finding is that the high degrees of concurrency (e.g., the same number as the population size) significantly reduce the complexity (e.g., average population size) without negatively affecting fitness.

Following the forging, testing and analysis of the proposed methods, a GP application that produces solutions whose complexity adds another notion to the ones seen already is used as a platform to examine the proposed system further. The application is set up by hybridising GP with the well-known multiple linear regression (MLR) for generating regression models. While MLR works very well if the features (or system variables) are well-identified, it relies on the human user to identify these features. In this hybridisation, GP manufactures useful features and; MLR combines them optimally to produce a powerful regression model (MLR-GP). While this makes for an effective hybrid system (MLR-GP), this system runs the risk of generating very complex models with a very high number of complex features. Complexity, in this case, is twofold: the number of features and the complexity of the constitution of the GP evolved features themselves. The results indicate that time-based complexity control covers both these aspects of the complexity of the models much more effectively than bloat-control. The results show that evaluation time control leveraged this ability (to manage both features and size effectively) to prevail in producing models with higher test fitness scores (generalise better), fewer features, and smaller sizes. The results from this application reaffirm the versatility of the evaluation time and the associated methods.

As GP is a tool with broad application, it is vital to examine the proposed system on a diverse set of applications to find out how it may unearth various forms of complexity. Accordingly, the highly-popular application domains (classical GP problems) are selected; they include robot control, Boolean logic applications, and classification using a hybridisation designed for classification problems (GPML). The variety of applications means that context and motivation for managing the complexity of evolved solutions must be considered per application when assessing performance. The results produced in these applications show that time characterises complexity in a resourceful way to offer several qualitative advantages. In the ANT application (robot control), time control distinguishes between the efficiency of the solutions (efficient solutions find all food items using minimal moves). Next, although the computational complexity of the operators in the Boolean applications may appear to be similar, TC still showed a difference; TC was at least as accurate and as simple as bloat-control. Moreover, in the GPML application (classification), the time control effectively detects the number of features in models, which bloat-control could not; hence, time control leads to a more rounded complexity control that improves the generalisation capacity of the models. Therefore, the results indicate that the proposed time-based system of managing complexity is versatile, and it broadly addresses the motivations behind managing the complexity of GP solutions.

This thesis arguably addresses one of the most significant challenges in GP – managing the complexity of the GP solutions. GP presents an exciting system that harnesses the power of evolution to produce solutions to seemingly limitless types of problems; after all, natural evolution produced life and intelligence in myriad forms, and it continues to optimise both. However, the unconstrained complexity of GP solutions renders it impractical to use exhaustively in many applications or makes the quality of the solutions it produces unfit for purpose. Furthermore, the current methods are limited in how well they manage complexity. Thus, the success of the proposed system for managing complexity across a broad set of GP applications attests to its ability to characterise and manage complexity in a more nuanced way than the existing methods to offer qualitative gains and to its significance. Therefore, this thesis contributes to the pivotal efforts to make GP more accessible and prolific.

10.2 Future Work

Lines of further research include the following:

- *Applying the evaluation time schemes in processor-intensive problems.*
  Although the proposed strategies for improving the reliability of the evaluation time measurements improved it significantly, it is not perfect. Therefore, processor-intensive tasks may cancel out minor fluctuations, and they stand to gain more from the time schemes, given that evaluation time

can detect computational effort. Therefore, a compelling area of studying the time schemes is applications with intense computing resource utilisation and high evaluation times. For example, *neuro-evolution* – the automatic design of *artificial neural networks* with GP – is an appropriate application. Further, such an application can demonstrate the benefit of the time schemes in a challenging and high-in-demand application.

- *Exploring more opportunities in the Fixed Length Initialisation scheme (FLI).* As the proposed FLI initialisation scheme improved the performance of all the contending methods (with only one justifiable exception: OPEQ on time-control), further investigation of the initialisation scheme may provide a means to enhance GP.

- *Studying how the evaluation time schemes may be improved by encouraging functional diversity throughout the evolutionary process of GP.*From the results of using FLI and in theory, evaluation time distinguishes functional complexity better in a functionally and structurally diverse population. As improvements from FLI only comes from starting the evolution with a functionally diverse population, future work can explore the effect of maintaining functional diversity in the breeding population throughout the evolution. Therefore, such an environment, may allow evaluation time to continuously distinguish functional complexity productively.

- *Exploring the utility of the time schemes in applications where computational simplicity is essential.*Future work can explore using the time schemes in application areas where computationally simple solutions are highly valued. Therefore, they may serve as a general complexity management tool, similar to how evolutionary algorithms optimise accuracy in many domains. For example, as a well-established GP application, automatic programming stands to gain from the complexity control of APGP because it can improve accuracy, simplicity and training speed.

Bibliography

[1] Terence Soule, James A. Foster, and John Dickinson. "Code Growth in Genetic Programming". In: *Genetic Programming 1996: Proceedings of the First Annual Conference*. Ed. by John R. Koza et al. Stanford University, CA, USA: MIT Press, July 1996, pp. 215–223. URL: http://cognet.mit.edu/sites/default/files/books/9780262315876/pdfs/9780262315876_chap26.pdf.

[2] Gregory Paris, Denis Robilliard, and Cyril Fonlupt. "Exploring Overfitting in Genetic Programming". In: *Evolution Artificielle, 6th International Conference*. Ed. by Pierre Liardet et al. Vol. 2936. Lecture Notes in Computer Science. Revised Selected Papers. Marseilles, France: Springer, Oct. 2003, pp. 267–277. ISBN: 3-540-21523-9. DOI: 10.1007/b96080.

[3] Ashish Kumar, Saurabh Goyal, and Manik Varma. "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, Aug. 2017, pp. 1935–1944.

[4] Joseph Yiu. "CHAPTER 9 - Interrupt Behavior". In: *The Definitive Guide to the ARM Cortex-M3 (Second Edition)*. Ed. by Joseph Yiu. Second Edition. Oxford: Newnes, 2010, pp. 145–153. ISBN: 978-1-85617-963-8. DOI: https://doi.org/10.1016/B978-1-85617-963-8.00012-0.

[5] Kenneth O. Stanley et al. “Designing neural networks through neuroevolution”. In: *Nature Machine Intelligence* 1 (Jan. 2019), pp. 24–35. DOI: doi:10.1038/s42256-018-0006-z. URL: https://www.nature.com/articles/s42256-018-0006-z?fbclid=IwAR0v_oJR499daqgqiKCAMa-LHWAoRYuaiTpOtHCwsOWmc6vcbe5Qx6Yjils.

[6] Thanwa Sripramong and Christofer Toumazou. “The invention of CMOS amplifiers using genetic programming and current-flow analysis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.11 (Nov. 2002), pp. 1237–1252. ISSN: 0278-0070. DOI: doi:10.1109/TCAD.2002.804109.

[7] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. “Measuring bloat, overfitting and functional complexity in genetic programming”. In: *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*. Ed. by Juergen Branke et al. Portland, Oregon, USA: ACM, July 2010, pp. 877–884. DOI: 10.1145/1830483.1830643.

[8] Raja Muhammad Atif Azad and Conor Ryan. “A Simple Approach to Lifetime Learning in Genetic Programming based Symbolic Regression”. In: *Evolutionary Computation* 22.2 (June 2014), pp. 287–317. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00111. URL: http://www.mitpressjournals.org/doi/abs/10.1162/EVCO_a_00111.

[9] Charles Darwin. *On the origin of species: A facsimile of the first edition*. Harvard University Press, 1964.

[10] Sean Luke, Gabriel Catalin Balan, and Liviu Panait. “Population Implosion in Genetic Programming”. In: *Genetic and Evolutionary Computation – GECCO-2003*. Ed. by E. Cantú-Paz et al. Vol. 2724. LNCS. Chicago: Springer-Verlag, July 2003, pp. 1729–1739. ISBN: 3-540-40603-4. DOI: doi:10.1007/3-540-45110-2_65. URL: http://cs.gmu.edu/~lpanait/papers/luke03population.pdf.

[11] Lau Tung Leng. “Guided genetic algorithm”. In: *University of Essex, A thesis submitted for the degree of Ph. D in Computer Science, Department of Computer Science* (1999).

[12] Charles C Peck and Atam P Dhawan. “Genetic algorithms as global random search methods: An alternative perspective”. In: *Evolutionary Computation* 3.1 (1995), pp. 39–80.

[13] Mary Lou Maher and Douglas H Fisher. “Using AI to evaluate creative designs”. In: *DS 73-1 Proceedings of the 2nd International Conference on Design Creativity Volume 1*. 2012.

[14] Peter JM Van Laarhoven and Emile HL Aarts. “Simulated annealing”. In: *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[15] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[16] Mehrdad Dianati, Insop Song, and Mark Treiber. *An introduction to genetic algorithms and evolution strategies*. Tech. rep. Citeseer, 2002.

[17] Nichael Lynn Cramer. “A representation for the Adaptive Generation of Simple Sequential Programs”. In: *Proceedings of an International Conference on Genetic Algorithms and the Applications*. Ed. by John J. Grefenstette. Carnegie-Mellon University, Pittsburgh, USA, July 1985, pp. 183–187. URL: `https://dl.acm.org/doi/10.5555/645511.657085`.

[18] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5. URL: `http://mitpress.mit.edu/books/genetic-programming`.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

ISBN: 0-262-11170-5. URL: http://mitpress.mit.edu/books/genetic-programming.

[20] Nguyen Xuan Hoai, R. I. (Bob) McKay, and Daryl Essam. "Representation and Structural Difficulty in Genetic Programming". In: *IEEE Transactions on Evolutionary Computation* 10.2 (Apr. 2006), pp. 157–166. DOI: 10.1109/TEVC.2006.871252. URL: http://sc.snu.ac.kr/courses/2006/fall/pg/aai/GP/nguyen/Structdiff.pdf.

[21] Conor Ryan, Michael O'Neill, and J. J. Collins, eds. *Handbook of Grammatical Evolution*. Springer, Sept. 2018. DOI: 10.1007/978-3-319-78717-6.

[22] Raja Muhammad Atif Azad. "A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System". PhD thesis. Ireland: University of Limerick, Dec. 2003. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/azad_thesis.ps.gz.

[23] Gopinath Chennupati, Raja Muhammad Atif Azad, and Conor Ryan. "Performance Optimization of Multi-Core Grammatical Evolution Generated Parallel Recursive Programs". In: *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. Ed. by Sara Silva et al. Madrid, Spain: ACM, July 2015, pp. 1007–1014. DOI: 10.1145/2739480.2754746. URL: http://doi.acm.org/10.1145/2739480.2754746.

[24] Lee Spector and Alan Robinson. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language". In: *Genetic Programming and Evolvable Machines* 3.1 (Mar. 2002), pp. 7–40. ISSN: 1389-2576. DOI: 10.1023/A:1014538503543. URL: http://hampshire.edu/lspector/pubs/push-gpem-final.pdf.

[25] Ting Hu et al. “Evolutionary dynamics on multiple scales: a quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming”. In: *Genetic Programming and Evolvable Machines* 13.3 (Sept. 2012). Special issue on selected papers from the 2011 European conference on genetic programming, pp. 305–337. ISSN: 1389-2576. DOI: 10.1007/s10710-012-9159-4.

[26] James Alfred Walker and Julian Francis Miller. “The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming”. In: *IEEE Transactions on Evolutionary Computation* 12.4 (Aug. 2008), pp. 397–417. ISSN: 1089-778X. DOI: 10.1109/TEVC.2007.903549. URL: http://results.ref.ac.uk/Submissions/Output/3354578.

[27] Edmund K Burke, James P Newall, and Rupert F Weare. “Initialization strategies and diversity in evolutionary timetabling”. In: *Evolutionary computation* 6.1 (1998), pp. 81–103.

[28] Borhan Kazimipour, Xiaodong Li, and A Kai Qin. “A review of population initialization techniques for evolutionary algorithms”. In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2014, pp. 2585–2592.

[29] W. B. Langdon and J. P. Nordin. “Seeding GP Populations”. In: *Genetic Programming, Proceedings of EuroGP’2000*. Ed. by R Poli et al. Vol. 1802. LNCS. Edinburgh: Springer-Verlag, Apr. 2000, pp. 304–315. ISBN: 3-540-67339-3. DOI: doi:10.1007/978-3-540-46239-2_23. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL_eurogp2000_seed.pdf.

[30] R. Muhammad Atif Azad, David Medernach, and Conor Ryan. “Efficient Approaches to Interleaved Sampling of training data for Symbolic Regression”. In: *Sixth World Congress on Nature and Biologically Inspired Computing*. Ed. by Ana Maria Madureira et al. Porto, Portugal: IEEE, July 2014, pp. 176–183. DOI: doi:10.1109/NaBIC.2014.6921874.

[31] Patrick D Surry and Nicholas J Radcliffe. "Inoculation to initialise evolutionary search". In: *AISB Workshop on Evolutionary Computing*. Springer. 1996, pp. 269–285.

[32] Hammad Ahmad and Thomas Helmuth. "A comparison of semantic-based initialization methods for genetic programming". In: *GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Ed. by Carlos Cotta et al. Kyoto, Japan: ACM, July 2018, pp. 1878–1881. DOI: `doi:10.1145/3205651.3208218`.

[33] Sean Luke. "Two Fast Tree-Creation Algorithms for Genetic Programming". In: *IEEE Transactions on Evolutionary Computation* 4.3 (Sept. 2000), pp. 274–283. DOI: `doi:10.1109/4235.873237`. URL: `http://ieeexplore.ieee.org/iel5/4235/18897/00873237.pdf`.

[34] Dirk Schweim, David Wittenberg, and Franz Rothlauf. "On sampling error in genetic programming". In: *Natural Computing* (2021), pp. 1–14.

[35] Tiantian Dou and Peter Rockett. "Comparison of semantic-based local search methods for multiobjective genetic programming". In: *Genetic Programming and Evolvable Machines* 19.4 (2018), pp. 535–563.

[36] Hamidreza Abbasianjahromi, Emadaldin Mohammadi Golafshani, and Mehdi Aghakarimi. "A prediction model for safety performance of construction sites using a linear artificial bee colony programming approach". In: *International Journal of Occupational Safety and Ergonomics* (2021), pp. 1–16.

[37] Sibel Arslan and Celal Ozturk. "Multi hive artificial bee colony programming for high dimensional symbolic regression with feature selection". In: *Applied Soft Computing* 78 (2019), pp. 515–527.

[38] Nikola Anđelić et al. “Estimation of gas turbine shaft torque and fuel flow of a CODLAG propulsion system using genetic programming algorithm”. In: *Pomorstvo* 34.2 (2020), pp. 323–337.

[39] João Victor C Fracasso and Fernando J Von Zuben. “Multi-objective semantic mutation for genetic programming”. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2018, pp. 1–8.

[40] Sahar Mohammad-Azari, Omid Bozorg-Haddad, and Hugo A Loáiciga. “State-of-art of genetic programming applications in water-resources systems analysis”. In: *Environmental monitoring and assessment* 192.2 (2020), pp. 1–17.

[41] Wenbin Pei et al. “Reuse of program trees in genetic programming with a new fitness function in high-dimensional unbalanced classification”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, pp. 187–188.

[42] Michael Prince, Kristiian DeHaan, and Daniel Tauritz. *A Multi-Objective Evolutionary Algorithm Approach for Optimizing Part Quality Aware Assembly Job Shop Scheduling Problems*. Tech. rep. Kansas City Plant (KCP), Kansas City, MO (United States), 2021.

[43] Nguyen Thi Hien et al. “Genetic programming for storm surge forecasting”. In: *Ocean Engineering* 215 (2020), p. 107812.

[44] Yassine Boudouaoui et al. “Solving differential equations with artificial bee colony programming”. In: *Soft Computing* 24 (2020), pp. 17991–18007.

[45] Arvind Kumar et al. “Clinical risk assessment of chronic kidney disease patients using genetic programming”. In: *Computer Methods in Biomechanics and Biomedical Engineering* (2021), pp. 1–9.

[46] Conor Ryan and R. Muhammad Atif Azad. “Sensible Initialisation in Grammatical Evolution”. In: *GECCO 2003: Proceedings of the Bird of a*

Feather Workshops, Genetic and Evolutionary Computation Conference. Ed. by Alwyn M. Barry. Chigaco: AAAI, July 2003, pp. 142–145.

[47] Félix-Antoine Fortin et al. "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research* 13 (July 2012), pp. 2171–2175.

[48] Trevor Stephens. *Genetic Programming in Python, with a scikit-learn inspired API: gplearn*. https://gplearn.readthedocs.io/en/stable/intro.html, Last accessed on 2021-12-17. 2018.

[49] M. Sipper. *Tiny Genetic Programming in Python*. https://github.com/moshesipper/tiny_gp. 2019.

[50] Conor Ryan and R. Muhammad Atif Azad. "Sensible Initialisation in Chorus". In: *Genetic Programming, Proceedings of EuroGP'2003*. Ed. by Conor Ryan et al. Vol. 2610. LNCS. Essex: Springer-Verlag, Apr. 2003, pp. 394–403. ISBN: 3-540-00971-X. DOI: doi:10.1007/3-540-36599-0_37.

[51] Huayang Xie. "An analysis of selection in genetic programming". In: (2009).

[52] Tobias Blickle and Lothar Thiele. "A Comparison of Selection Schemes used in Evolutionary Algorithms". In: *Evolutionary Computation* 4.4 (Dec. 1996), pp. 361–394. ISSN: 1063-6560. DOI: doi:10.1162/evco.1996.4.4.361. URL: http://www.handshake.de/user/blickle/publications/ECfinal.ps.

[53] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms". In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 69–93.

[54] Hari Mohan Pandey, Ankit Chaudhary, and Deepti Mehrotra. "A comparative review of approaches to prevent premature convergence in GA". In: *Applied Soft Computing* 24 (2014), pp. 1047–1077.

[55] James Edward Baker. “Adaptive selection methods for genetic algorithms”. In: *Proceedings of an International Conference on Genetic Algorithms and their applications*. Vol. 1. Hillsdale, New Jersey. 1985.

[56] Huayang Xie, Mengjie Zhang, and Peter Andreae. “An analysis of constructive crossover and selection pressure in genetic programming”. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 2007, pp. 1739–1748.

[57] Yongsheng Fang and Jun Li. “A review of tournament selection in genetic programming”. In: *International Symposium on Intelligence Computation and Applications*. Springer. 2010, pp. 181–192.

[58] Michael O’Neill. *Riccardo Poli, William B. Langdon, Nicholas F. McPhee: a field guide to genetic programming*. 2009.

[59] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* University of Michigan, 1975.

[60] Una-May O’Reilly and Franz Oppacher. *Using Building Block Functions to Investigate a Building Block Hypothesis for Genetic Programming*. Working Paper 94-02-029. 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA: Santa Fe Institute, Apr. 1994. URL: http://www.santafe.edu/media/workingpapers/94-04-020.pdf.

[61] Stephanie Forrest and Melanie Mitchell. “Relative building-block fitness and the building-block hypothesis”. In: *Foundations of genetic algorithms*. Vol. 2. Elsevier, 1993, pp. 109–126.

[62] Peter J Angeline. “Comparing subtree crossover with macromutation”. In: *International Conference on Evolutionary Programming*. Springer. 1997, pp. 101–111.

[63] Peter J. Angeline. “Subtree Crossover: Building Block Engine or Macromutation?” In: *Genetic Programming 1997: Proceedings of the Second Annual Conference*. Ed. by John R. Koza et al. Stanford University, CA, USA: Morgan Kaufmann, July 1997, pp. 9–17. URL: http://ncra.ucd.ie/COMP41190/SubtreeXoverBuildingBlockorMacromutation_angeline_gp97.ps.

[64] Riccardo Poli and Nicholas Freitag McPhee. *Exact GP Schema Theory for Headless Chicken Crossover and Subtree Mutation*. Tech. rep. CSRP-00-23. University of Birmingham, School of Computer Science, Dec. 2000. URL: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2000/CSRP-00-23.ps.gz.

[65] Richard A. Watson, Gregory S. Hornby, and Jordan B. Pollack. “Modeling Building-Block Interdependency”. In: *Late Breaking Papers at the Genetic Programming 1998 Conference*. Ed. by John R. Koza. University of Wisconsin, Wisconsin, USA: Stanford University Bookstore, July 1998, pp. 234–240.

[66] Conor Ryan, Hammad Majeed, and Atif Azad. “A Competitive Building Block Hypothesis”. In: *Genetic and Evolutionary Computation – GECCO-2004, Part II*. Ed. by Kalyanmoy Deb et al. Vol. 3103. Lecture Notes in Computer Science. Seattle, WA, USA: Springer-Verlag, June 2004, pp. 654–665. ISBN: 3-540-22343-6. DOI: doi:10.1007/978-3-540-24855-2_73.

[67] Zahra Zojaji and Mohammad Mehdi Ebadzadeh. “An improved semantic schema modeling for genetic programming”. In: *Soft Computing* 22.10 (May 2018), pp. 3237–3260. ISSN: 1433-7479. DOI: doi:10.1007/s00500-017-2781-6. URL: https://doi.org/10.1007/s00500-017-2781-6.

[68] Riccardo Poli and William B. Langdon. “On the Search Properties of Different Crossover Operators in Genetic Programming”. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Ed. by John R.

Koza et al. University of Wisconsin, USA: Morgan Kaufmann, July 1998, pp. 293–301. ISBN: 1-55860-548-7. URL: `http://www.cs.essex.ac.uk/staff/poli/papers/Poli-GP1998.pdf`.

[69] Riccardo Poli and William B. Langdon. “Schema Theory for Genetic Programming with One-point Crossover and Point Mutation”. In: *Evolutionary Computation* 6.3 (1998), pp. 231–252. DOI: `doi:10.1162/evco.1998.6.3.231`. URL: `http://cswww.essex.ac.uk/staff/poli/papers/Poli-ECJ1998.pdf`.

[70] William B Langdon. “Size fair and homologous tree genetic programming crossovers”. In: *Genetic programming and evolvable machines* 1.1/2 (2000), pp. 95–119.

[71] Hammad Majeed and Conor Ryan. “A Less Destructive, Context-aware Crossover Operator for GP”. In: *Proceedings of the 9th European Conference on Genetic Programming*. Ed. by Pierre Collet et al. Vol. 3905. Lecture Notes in Computer Science. Budapest, Hungary: Springer, Apr. 2006, pp. 36–48. ISBN: 3-540-33143-3. DOI: `doi:10.1007/11729976_4`.

[72] Nuno M Rodrigues, João E Batista, and Sara Silva. “Ensemble genetic programming”. In: *arXiv preprint arXiv:2001.07553* (2020).

[73] Luis Muñoz et al. “Evolving multidimensional transformations for symbolic regression with M3GP”. In: *Memetic Computing* 11.2 (2019), pp. 111–126.

[74] Amir H Gandomi and David Roke. “A Multi-Objective Evolutionary Framework for Formulation of Nonlinear Structural Systems”. In: *IEEE Transactions on Industrial Informatics* (2021).

[75] Lino Rodriguez-Coayahuitl et al. “Cooperative co-evolutionary genetic programming for high dimensional problems”. In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2020, pp. 48–62.

[76] Lawrence Beadle and Colin G Johnson. "Semantically Driven Mutation in Genetic Programming". In: *2009 IEEE Congress on Evolutionary Computation*. Ed. by Andy Tyrrell. IEEE Computational Intelligence Society. Trondheim, Norway: IEEE Press, May 2009, pp. 1336–1342. DOI: `doi:10.1109/CEC.2009.4983099`.

[77] Gilbert Syswerda. "A study of reproduction in generational and steady-state genetic algorithms". In: *Foundations of genetic algorithms*. Vol. 1. Amsterdam: Elsevier, 1991, pp. 94–101.

[78] Vic Ciesielski and Dylan Mawhinney. "Prevention of Early Convergence in Genetic Programming by Replacement of Similar Programs". In: *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*. Ed. by David B. Fogel et al. IEEE Press, May 2002, pp. 67–72. ISBN: 0-7803-7278-6. DOI: `doi:10.1109/CEC.2002.1006211`.

[79] Jim Smith and Frank Vavak. "Replacement strategies in steady state genetic algorithms: Static environments". In: *Foundations of genetic algorithms* 5 (1999), pp. 219–233.

[80] Kalyanmoy Deb and Himanshu Jain. "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints". In: *IEEE transactions on evolutionary computation* 18.4 (2013), pp. 577–601.

[81] W. B. Langdon. "The Evolution of Size in Variable Length Representations". In: *1998 IEEE International Conference on Evolutionary Computation*. Anchorage, Alaska, USA: IEEE Press, May 1998, pp. 633–638. ISBN: 0-7803-4869-9. DOI: `doi:10.1109/ICEC.1998.700102`. URL: `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.wcci98_bloat.pdf`.

[82] W. B. Langdon and R. Poli. *Fitness Causes Bloat*. Tech. rep. CSRP-97-09. Birmingham, B15 2TT, UK: University of Birmingham, School of Computer Science, Feb. 1997. URL: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-09.ps.gz.

[83] William B. Langdon et al. "The Evolution of Size and Shape". In: *Advances in Genetic Programming 3*. Ed. by Lee Spector et al. Cambridge, MA, USA: MIT Press, June 1999. Chap. 8, pp. 163–190. ISBN: 0-262-19423-6. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch08.pdf.

[84] W. B. Langdon and R. Poli. "Fitness causes bloat: Mutation". In: *Late Breaking Papers at the GP-97 Conference*. Ed. by John Koza. Stanford, CA: Stanford Bookstore, 1997, pp. 132–140.

[85] Francisco Fernandez de Vega et al. "Control of bloat in Genetic Programming by means of the Island Model". In: *Parallel Problem Solving from Nature - PPSN VIII*. Ed. by Xin Yao et al. Vol. 3242. LNCS. Birmingham, UK: Springer-Verlag, Sept. 2004, pp. 263–271. ISBN: 3-540-23092-0. DOI: doi:10.1007/b100601.

[86] Anuradha Purohit et al. "Removing code bloating in crossover operation in Genetic Programming". In: *International Conference on Recent Trends in Information Technology (ICRTIT 2011)*. Anna University, Chennai, June 2011, pp. 1126–1130. DOI: doi:10.1109/ICRTIT.2011.5972430.

[87] Lawrence Beadle and Colin Johnson. "Semantically Driven Crossover in Genetic Programming". In: *Proceedings of the IEEE World Congress on Computational Intelligence*. Ed. by Jun Wang. IEEE Computational Intelligence Society. Hong Kong: IEEE Press, June 2008, pp. 111–116. DOI: doi:10.1109/CEC.2008.4630784. URL: http://results.ref.ac.uk/Submissions/Output/1423275.

[88] Mark Willis et al. “Genetic Programming: An Introduction and Survey of Applications”. In: *Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*. Ed. by Ali Zalzala. University of Strathclyde, Glasgow, UK: Institution of Electrical Engineers, Sept. 1997, pp. 314–319. ISBN: 0-85296-693-8. DOI: doi:10.1049/cp:19971199. URL: http://www.staff.ncl.ac.uk/d.p.searson/docs/galesia97surveyofGP.pdf.

[89] Milad Taleby Ahvanooey et al. “A survey of genetic programming and its applications”. In: *KSII Transactions on Internet and Information Systems (TIIS)* 13.4 (2019), pp. 1765–1794.

[90] John R. Koza. “Human-competitive machine invention by means of genetic programming”. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22.3 (2008), pp. 185–193.

[91] Douglas Adriano Augusto and Helio JC Barbosa. “Symbolic regression via genetic programming”. In: *Proceedings. Vol. 1. Sixth Brazilian Symposium on Neural Networks*. IEEE. 2000, pp. 173–178.

[92] Stefan Sette and Luc Boullart. “Genetic programming: principles and applications”. In: *Engineering applications of artificial intelligence* 14.6 (2001), pp. 727–736.

[93] Panitnat Yimyam and Adrian F. Clark. “Agricultural produce grading by computer vision using Genetic Programming”. In: *ROBIO*. IEEE, 2012, pp. 458–463. ISBN: 978-1-4673-2125-9.

[94] Leonardo Trujillo and Gustavo Olague. “Synthesis of Interest Point Detectors through Genetic Programming”. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’06. New York, NY, USA: Association for Computing Machinery, 2006, 887–894.

ISBN: 1595931864. DOI: 10.1145/1143997.1144151. URL: https://doi.org/10.1145/1143997.1144151.

[95] Adil Raja et al. "A Methodology for Deriving VoIP Equipment Impairment Factors for a Mixed NB/WB Context". In: *IEEE Trans. Multim* 10.6 (2008), pp. 1046–1058.

[96] Pedro G. Espejo, Sebastian Ventura, and Francisco Herrera. "A Survey on the Application of Genetic Programming to Classification". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 40.2 (Mar. 2010), pp. 121–144. ISSN: 1094-6977. DOI: doi:10.1109/TSMCC.2009.2033566.

[97] R. Muhammad Atif Azad and Conor Ryan. "Abstract functions and lifetime learning in genetic programming for symbolic regression". In: *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*. Ed. by Martin Pelikan and Jürgen Branke. ACM, 2010, pp. 893–900. ISBN: 978-1-4503-0072-8.

[98] Douglas M Hawkins. "The problem of overfitting". In: *Journal of chemical information and computer sciences* 44.1 (2004), pp. 1–12.

[99] Gopinath Chennupati, R. Muhammad Atif Azad, and Conor Ryan. "Multicore GE: automatic evolution of CPU based multi-core parallel programs". In: *GECCO 2014 student workshop*. Ed. by Tea Tusar and Boris Naujoks. Vancouver, BC, Canada: ACM, July 2014, pp. 1041–1044. DOI: doi:10.1145/2598394.2605670. URL: http://doi.acm.org/10.1145/2598394.2605670.

[100] John R. Koza and James P. Rice. "Automatic programming of robots using genetic programming". In: *Proceedings of Tenth National Conference on Artificial Intelligence*. AAAI Press/MIT Press, 1992, pp. 194–201. URL: http://www.genetic-programming.com/jkpdf/aaai1992.pdf.

[101] Wolfgang Banzhaf et al. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., 1998.

[102] William B Langdon and Mark Harman. “Optimizing existing software with genetic programming”. In: *IEEE Transactions on Evolutionary Computation* 19.1 (2014), pp. 118–135.

[103] A.P. Shanthi, L.K. Singaram, and R. Parthasarathi. “Evolution of asynchronous sequential circuits”. In: *2005 NASA/DoD Conference on Evolvable Hardware (EH’05)*. 2005, pp. 93–96. DOI: 10.1109/EH.2005.23.

[104] Dario Floreano, Peter Dürr, and Claudio Mattiussi. “Neuroevolution: from architectures to learning”. In: *Evolutionary intelligence* 1.1 (2008), pp. 47–62. URL: https://doi.org/10.1007/s12065-007-0002-4.

[105] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. “A Genetic Programming Approach to Designing Convolutional Neural Network Architectures”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. Ed. by Jerome Lang. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 5369–5373. DOI: doi:10.24963/ijcai.2018/755. URL: https://doi.org/10.24963/ijcai.2018/755.

[106] Steven M Manson. “Simplifying complexity: a review of complexity theory”. In: *Geoforum* 32.3 (2001), pp. 405–414.

[107] James Ladyman, James Lambert, and Karoline Wiesner. “What is a complex system?” In: *European Journal for Philosophy of Science* 3.1 (2013), pp. 33–67.

[108] J Gribbin. “Science a history 1543-2001, McPherson’s Printing Group, Maryborough”. In: *Victoria* (2002).

[109] Steven Johnson. “Emergence: The Connected Lives of Ants, Brains”. In: *Cities, and Software. New York: Scribner* (2001).

[110] Carlos Espinosa-Soto, Pablo Padilla-Longoria, and Elena R Alvarez-Buylla. “A gene regulatory network model for cell-fate determination during Arabidopsis thaliana flower development that is robust and recovers experimental gene expression profiles”. In: *The Plant Cell* 16.11 (2004), pp. 2923–2939.

[111] David E Featherstone and Kendal Broadie. “Wrestling with pleiotropy: genomic and topological analysis of the yeast gene expression network”. In: *Bioessays* 24.3 (2002), pp. 267–274.

[112] Curt Stern and Chiyoko Tokunaga. “Autonomous pleiotropy in Drosophilia.” In: *Proceedings of the National Academy of Sciences of the United States of America* 60.4 (1968), p. 1252.

[113] Michael Grossman and Robert Katz. *Non-Newtonian Calculus: A Self-contained, Elementary Exposition of the Authors' Investigations...* Non-Newtonian Calculus, 1972.

[114] Alan V Oppenheim. “Alan S. Willsky with S. Hamid Nawab, Signals & Systems”. In: *ed: Prentice-Hall, Upper Saddle River, New Jersey* (1997).

[115] John G Proakis. *Digital signal processing: principles algorithms and applications*. Pearson Education India, 2001.

[116] David Malkin. “The Evolutionary Impact of Gradual Complexification on Complex Systems”. In: *Engineering Doctorate Thesis, University College London* (2009).

[117] Michael C Whitlock et al. “Multiple fitness peaks and epistasis”. In: *Annual review of ecology and systematics* 26.1 (1995), pp. 601–629.

[118] Lee Altenberg. "Genome growth and the evolution of the genotype-phenotype map". In: *Evolution and biocomputation*. Springer, 1995, pp. 205–259.

[119] Gary J. Gray et al. "Nonlinear model structure identification using genetic programming". In: *Control Engineering Practice* 6.11 (1998), pp. 1341–1352. URL: http://www.sciencedirect.com/science/article/B6V2H-3W1GPR8-4/1/047d9c74e28a6a1a117a3ed9a6d6c409.

[120] Gary J Gray et al. "Issues in nonlinear model structure identification using genetic programming". In: *Second International Conference On Genetic Algorithms In Engineering Systems: Innovations And Applications*. IET. 1997, pp. 308–313.

[121] Janos Madar, Janos Abonyi, and Ferenc Szeifert. "Genetic Programming for the Identification of Nonlinear Input Output Models". In: *Industrial & Engineering Chemistry Research* 44.9 (2005), pp. 3178–3186. DOI: 10.1021/ie049626e. URL: https://doi.org/10.1021/ie049626e.

[122] X. Hong et al. "Model selection approaches for non-linear system identification: a review". In: *International Journal of Systems Science* 39.10 (2008), pp. 925–946. DOI: 10.1080/00207720802083018. URL: https://doi.org/10.1080/00207720802083018.

[123] Aniko Ekart and S. Z. Nemeth. "Selection Based on the Pareto Non-domination Criterion for Controlling Code Growth in Genetic Programming". In: *Genetic Programming and Evolvable Machines* 2.1 (Mar. 2001), pp. 61–73. ISSN: 1389-2576. DOI: 10.1023/A:1010070616149.

[124] Bill Worzel and Rick L. Riolo. "Genetic Programming Theory and Practice". In: *Genetic Programming Theory and Practice*. Ed. by Rick L. Riolo and Bill Worzel. Kluwer, 2003. Chap. 1, pp. 1–10. ISBN: 1-4020-7581-2.

DOI: doi:10.1007/978-1-4419-8983-3_1. URL: http://www.springer.com/computer/ai/book/978-1-4020-7581-0.

[125] Walter Alden Tackett. "Recombination, Selection, and the Genetic Construction of Computer Programs". PhD thesis. USA: University of Southern California, Department of Electrical Engineering Systems, Apr. 1994.

[126] Tobias Blickle and Lothar Thiele. "Genetic Programming and Redundancy". In: *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*. Ed. by J. Hopf. Im Stadtwald, Germany: Max-Planck-Institut für Informatik (MPI-I-94-241), 1994, pp. 33–38. URL: http://www.tik.ee.ethz.ch/~tec/publications/bt94/GPandRedundancy.ps.gz.

[127] K. Harries and P. W. H. Smith. *Code Growth, Explicitly Defined Introns and Alternative Selection Schemes*. www. Earlier version of Evolutionary Computation 6 (4), 336-360, 1998. 1998. URL: http://www.soi.city.ac.uk/homes/peters/pub/Introns6.ps.

[128] Lee Altenberg. "The Evolution of Evolvability in Genetic Programming". In: *Advances in Genetic Programming*. Ed. by Kenneth E. Kinnear, Jr. MIT Press, 1994. Chap. 3, pp. 47–74. URL: http://dynamics.org/Altenberg/FILES/LeeEEGP.pdf.

[129] Peter Nordin and Wolfgang Banzhaf. "Complexity Compression and Evolution". In: *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*. Ed. by Larry J. Eshelman. Pittsburgh, PA, USA: Morgan Kaufmann, July 1995, pp. 310–317. ISBN: 1-55860-370-0. URL: ftp://lumpi.informatik.uni-dortmund.de/pub/biocomp/papers/icga95-1.ps.gz.

[130] Nicholas Freitag McPhee and Justin Darwin Miller. "Accurate Replication in Genetic Programming". In: *Genetic Algorithms: Proceedings of the*

Sixth International Conference (ICGA95). Ed. by Larry J. Eshelman. Pittsburgh, PA, USA: Morgan Kaufmann, July 1995, pp. 303–309. ISBN: 1-55860-370-0. URL: http://citeseer.ist.psu.edu/mcphee95accurate.html.

[131] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. "Explicitly Defined Introns and Destructive Crossover in Genetic Programming". In: *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*. Ed. by Justinian P. Rosca. Tahoe City, California, USA, July 1995, pp. 6–22. URL: http://web.cs.mun.ca/~banzhaf/papers/ML95.pdf.

[132] Terence Soule and James A. Foster. "Removal Bias: a New Cause of Code Growth in Tree Based Evolutionary Programming". In: *1998 IEEE International Conference on Evolutionary Computation*. Anchorage, Alaska, USA: IEEE Press, May 1998, pp. 781–786. ISBN: 0-7803-4869-9. DOI: doi:10.1109/ICEC.1998.700151. URL: http://citeseer.ist.psu.edu/313655.html.

[133] Terence Soule. "Code Growth in Genetic Programming". PhD thesis. Moscow, Idaho, USA: University of Idaho, May 1998. URL: http://www.cs.uidaho.edu/~tsoule/research/the3.ps.

[134] Sean Luke. "Modification Point Depth and Genome Growth in Genetic Programming". In: *Evol. Comput* 11.1 (2003), pp. 67–106.

[135] Sean Luke. "Code Growth is Not Caused by Introns". In: *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. Ed. by Darrell Whitley. Las Vegas, Nevada, USA, July 2000, pp. 228–235. URL: http://www.cs.gmu.edu/~sean/papers/intronpaper.pdf.

[136] Sean Luke. "Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat". PhD thesis. College Park, MD

20742, USA: Department of Computer Science, University of Maryland, 2000. URL: http://www.cs.gmu.edu/~sean/papers/thesis2p.pdf.

[137] Terence Soule and Robert B. Heckendorn. "An Analysis of the Causes of Code Growth in Genetic Programming". In: *Genet. Program. Evolvable Mach* 3.3 (2002), pp. 283–309.

[138] W. B. Langdon and R. Poli. *Fitness causes bloat*. Tech. rep. CSRP-97-09. Birmingham, UK: University of Birmingham, School of Computer Science, 1997.

[139] R. Poli, W. B. Langdon, and Stephen Dignum. *On the Limiting Distribution of Program Sizes in Tree-based Genetic Programming*. Technical Report CSM-464. Department of Computer Science, University of Essex, Dec. 2006. URL: http://cswww.essex.ac.uk/technical-reports/2006/csm464.pdf.

[140] Stephen Dignum and Riccardo Poli. "Crossover, Sampling, Bloat and the Harmful Effects of Size Limits". In: *Genetic Programming, 11th European Conference, EuroGP 2008, Naples, Italy, March 26-28, 2008. Proceedings*. Ed. by Michael O'Neill 0001 et al. Vol. 4971. Lecture Notes in Computer Science. Springer, 2008, pp. 158–169. ISBN: 978-3-540-78670-2.

[141] Riccardo Poli, Nicholas Freitag McPhee, and Leonardo Vanneschi. "The impact of population size on code growth in GP: analysis and empirical validation". In: *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*. Ed. by Conor Ryan and Maarten Keijzer. ACM, 2008, pp. 1275–1282. ISBN: 978-1-60558-130-9.

[142] Stephen Dignum and Riccardo Poli. “Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat”. In: *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*. Ed. by Hod Lipson. ACM, 2007, pp. 1588–1595. ISBN: 978-1-59593-697-4.

[143] Riccardo Poli, William B. Langdon, and Stephen Dignum. “On the Limiting Distribution of Program Sizes in Tree-Based Genetic Programming”. In: *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings*. Ed. by Marc Ebner et al. Vol. 4445. Lecture Notes in Computer Science. Springer, 2007, pp. 193–204. ISBN: 978-3-540-71602-0.

[144] K. G. Janardan. “Weighted Lagrange Distributions and Their Characterizations”. In: *SIAM Journal on Applied Mathematics* 47.2 (Apr. 1987), pp. 411–415. ISSN: 0036-1399 (print), 1095-712X (electronic).

[145] Konanur G. Janardan and B. Raja Rao. “Lagrange Distributions of the Second Kind and Weighted Distributions”. In: *SIAM Journal on Applied Mathematics* 43.2 (Apr. 1983), pp. 302–313. ISSN: 0036-1399 (print), 1095-712X (electronic).

[146] F Fernandez et al. “Efficient use of computational resources in genetic programming: controlling the bloat phenomenon by means of the island model”. In: *IEEE 2002 28th Annual Conference of the Industrial Electronics Society. IECON 02*. Vol. 3. IEEE. 2002, pp. 2520–2524.

[147] Maumita Bhattacharya and Baikunth Nath. “Genetic Programming: A Review of Some Concerns”. In: *Proceedings of International Conference Computational Science Part II - ICCS 2001*. Ed. by V. N. Alexandrov et al. Vol. 2074. Lecture Notes in Computer Science. Late Submissions. San Francisco, CA, USA: Springer, May 2001, pp. 1031–1040. DOI: doi:10.1007/3-540-45718-6_109.

[148] P. W. H. Smith. "Controlling Code Growth in Genetic Programming". In: *Advances in Soft Computing*. Ed. by Robert John and Ralph Birkenhead. De Montfort University, Leicester, UK: Physica-Verlag, 2000, pp. 166–171. ISBN: 3-7908-1257-9. URL: http://www.springer-ny.com/detail.tpl?ISBN=3790812579.

[149] Anuradha Purohit, Narendra S. Choudhari, and Aruna Tiwari. "Code Bloat Problem in Genetic Programming". en. In: *International Journal of Scientific and Research Publications* 3.4 (Apr. 2013), p. 1612. ISSN: 2250-3153. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.301.158.

[150] Leonardo Vanneschi. "Theory and Practice for Efficient Genetic Programming". PhD thesis. Switzerland: Faculty of Sciences, University of Lausanne, 2004. URL: http://old.disco.unimib.it/Vanneschi/thesis_vanneschi.pdf.

[151] Sara Silva et al. *Evolutionary and Complex Systems Group*. en. Mar. 2012. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.216.4283;http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1673.pdf.

[152] Sean Luke and Liviu Panait. "A Comparison of Bloat Control Methods for Genetic Programming". In: *Evolutionary Computation* 14.3 (Sept. 2006), pp. 309–344. ISSN: 1063-6560. DOI: 10.1162/evco.2006.14.3.309. URL: http://cognet.mit.edu/system/cogfiles/journalpdfs/evco.2006.14.3.309.pdf.

[153] Stephen Dignum and Riccardo Poli. "Crossover, sampling, bloat and the harmful effects of size limits". In: *European Conference on Genetic Programming*. Springer. 2008, pp. 158–169.

[154] Nicholas Freitag McPhee, Alex Jarvis, and Ellery Fussell Crane. “On the Strength of Size Limits in Linear Genetic Programming”. In: *Genetic and Evolutionary Computation – GECCO 2004*. Ed. by Kalyanmoy Deb. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 593–604. ISBN: 978-3-540-24855-2.

[155] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. “Operator equalisation for bloat free genetic programming and a survey of bloat control methods”. In: *Genetic Programming and Evolvable Machines* 13.2 (June 2012), pp. 197–238. ISSN: 1389-2576. DOI: 10.1007/s10710-011-9150-5.

[156] Andrei N Kolmogorov. “Three approaches to the quantitative definition ofinformation’”. In: *Problems of information transmission* 1.1 (1965), pp. 1–7.

[157] TM Cover and JA Thomas. “Joint Entropy and Conditional Entropy”. In: *Elements of Information Theory, 2nd ed.; John Wiley & Sons: Hoboken, NJ, USA* (2006), p. 16.

[158] Paul Vitányi. “How incomputable is Kolmogorov complexity?” In: *Entropy* 22.4 (2020), p. 408.

[159] Alexander K Zvonkin and Leonid A Levin. “The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms”. In: *Russian Mathematical Surveys* 25.6 (1970), p. 83.

[160] Jorma Rissanen. “Modeling by shortest data description”. In: *Automatica* 14.5 (1978), pp. 465–471.

[161] Volker Nannen. “A short introduction to model selection, Kolmogorov complexity and Minimum Description Length (MDL)”. In: *arXiv preprint arXiv:1005.2364* (2010).

[162] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. “Genetic Programming Using a Minimum Description Length Principle”. In: *Advances in Genetic Programming*. Ed. by Kenneth E. Kinnear, Jr. Cambridge, MA, USA: MIT Press, 1994. Chap. 12, pp. 265–284. URL: http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap12.pdf.

[163] Nicol N Schraudolph and John J Grefenstette. “A user’s guide to GAucsd 1.4”. In: *Computer Science & Engineering Department, University of California, San Diego–1991* (1992).

[164] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. “Operator equalisation for bloat free genetic programming and a survey of bloat control methods”. In: *Genetic Programming and Evolvable Machines* 13.2 (2012), pp. 197–238. DOI: 10.1007/s10710-011-9150-5.

[165] Ekaterina J. Vladislavleva, Guido F. Smits, and Dick den Hertog. “Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming”. In: *IEEE Transactions on Evolutionary Computation* 13.2 (Apr. 2009), pp. 333–349. ISSN: 1089-778X. DOI: 10.1109/TEVC.2008.926486.

[166] T.J. Rivlin. *The Chebyshev Polynomials*. A Wiley-Interscience publication. Wiley, 1974. ISBN: 9780471724704.

[167] Mauro Castelli et al. “A Quantitative Study of Learning and Generalization in Genetic Programming”. In: *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*. Ed. by Sara Silva et al. Vol. 6621. LNCS. Turin, Italy: Springer Verlag, Apr. 2011, pp. 25–36. DOI: 10.1007/978-3-642-20407-4_3.

[168] Sanjeev R Kulkarni and Gilbert Harman. “Statistical learning theory: a tutorial”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 3.6 (2011), pp. 543–556.

[169] Vladimir Naumovich Vapnik. *Statistical learning theory*. Adaptive and learning systems for signal processing, communications, and control. OCLC: 845016043. New York: Wiley, 1998. 736 pp. ISBN: 978-0-471-03003-4.

[170] V. Vapnik. *The Nature of Statistical Learning Theory*. Information Science and Statistics. Springer New York, 2013. ISBN: 9781475732641.

[171] Vladimir Vapnik. *Statistical learning theory. 1998*. Vol. 3. Wiley, New York, 1998.

[172] Qi Chen, Mengjie Zhang, and Bing Xue. “Structural Risk Minimisation-Driven Genetic Programming for Enhancing Generalisation in Symbolic Regression”. In: *IEEE Transactions on Evolutionary Computation* 23.4 (Aug. 2019), pp. 703–717. ISSN: 1089-778X. DOI: 10.1109/TEVC.2018.2881392.

[173] Qi Chen, Mengjie Zhang, and Bing Cue. “Improving Generalisation of Genetic Programming for Symbolic Regression with Structural Risk Minimisation”. In: *GECCO ’16: Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation*. Ed. by Tobias Friedrich. Denver, USA: ACM, July 2016, pp. 709–716. DOI: 10.1145/2908812.2908842.

[174] Christian Raymond et al. “Genetic Programming with Rademacher Complexity for Symbolic Regression”. In: *2019 IEEE Congress on Evolutionary Computation, CEC 2019*. Ed. by Carlos A. Coello Coello. IEEE Computational Intelligence Society. Wellington, New Zealand: IEEE Press, June 2019, pp. 2657–2664. DOI: 10.1109/CEC.2019.8790341.

[175] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[176] R. Muhammad Atif Azad and Conor Ryan. "Variance based selection to improve test set performance in genetic programming". In: *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Ed. by Natalio Krasnogor et al. Dublin, Ireland: ACM, July 2011, pp. 1315–1322. DOI: `doi:10.1145/2001576.2001754`.

[177] Ji Ni and Peter Rockett. "Tikhonov regularization as a complexity measure in multiobjective genetic programming". In: *IEEE Transactions on Evolutionary Computation* 19.2 (2014), pp. 157–166.

[178] Daniel Pierre Bovet, Pierluigi Crescenzi, and D Bovet. *Introduction to the Theory of Complexity*. Vol. 7. Prentice Hall London, 1994.

[179] Michael Sipser. "Introduction to the Theory of Computation". In: *ACM Sigact News* 27.1 (1996), pp. 27–29.

[180] Ian Chivers and Jane Sleightholme. "An introduction to Algorithms and the Big O Notation". In: *Introduction to programming with Fortran*. Springer, 2015, pp. 359–364.

[181] Abdiansah Abdiansah and Retantyo Wardoyo. "Time complexity analysis of support vector machines (SVM) in LibSVM". In: *International journal computer and application* 128.3 (2015), pp. 28–34.

[182] Adam Meyerson and Ryan Williams. "On the complexity of optimal k-anonymity". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2004, pp. 223–228.

[183] Mikkel T Jensen. "Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms". In: *IEEE Transactions on Evolutionary Computation* 7.5 (2003), pp. 503–515.

[184] Aliyu Sani Sambo et al. "Leveraging Asynchronous Parallel Computing to Produce Simple Genetic Programming Computational Models". In:

The 35th ACM/SIGAPP Symposium On Applied Computing. Ed. by Federico Divina and Miguel Garcia Torres. Brno, Czech Republic: ACM, Mar. 2020, pp. 521–528. DOI: 10.1145/3341105.3373921.

[185] Francisco Fernández de Vega et al. "Time and Individual Duration in Genetic Programming". In: *IEEE Access* 8 (2020), pp. 38692–38713.

[186] W. B. Langdon. "Genetic Improvement of Genetic Programming". In: *2020 IEEE Congress on Evolutionary Computation (CEC)*. 2020, pp. 1–8. DOI: 10.1109/CEC48606.2020.9185771.

[187] Cameron Simpson et al. *PEP 418: Add monotonic time, performance counter, and process time functions*. Python.org. Website. URL: https://www.python.org/dev/peps/pep-0418/ (visited on 10/15/2019).

[188] Sean Luke and Liviu Panait. "A Comparison of Bloat Control Methods for Genetic Programming". In: *Evolutionary Computation* 14.3 (Nov. 2006), pp. 309–344. ISSN: 1063-6560. DOI: 10.1162/evco.2006.14.3.309.

[189] Sean Luke and Liviu Panait. "Fighting Bloat with Nonparametric Parsimony Pressure". In: *Parallel Problem Solving from Nature - PPSN VII*. Ed. by Juan J. Merelo-Guervos et al. Lecture Notes in Computer Science, LNCS 2439. Granada, Spain: Springer-Verlag, Sept. 2002, pp. 411–421. ISBN: 3-540-44139-5. DOI: 10.1007/3-540-45712-7_40.

[190] Stephen Dignum and Riccardo Poli. "Operator Equalisation and Bloat Free GP". In: *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*. Ed. by Michael O'Neill et al. Vol. 4971. Lecture Notes in Computer Science. Naples: Springer, Mar. 2008, pp. 110–121. DOI: 10.1007/978-3-540-78671-9_10.

[191] Riccardo Poli. "A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming". In: *Genetic Programming, Proceedings of*

EuroGP'2003. Ed. by Conor Ryan et al. Vol. 2610. LNCS. Essex: Springer-Verlag, Apr. 2003, pp. 204–217. ISBN: 3-540-00971-X. DOI: 10.1007/3-540-36599-0_19.

[192] James McDermott et al. "Genetic programming needs better benchmarks". In: *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*. Ed. by Terry Soule et al. Philadelphia, Pennsylvania, USA: ACM, July 2012, pp. 791–798. DOI: doi:10.1145/2330163.2330273.

[193] David R. White et al. "Better GP benchmarks: community survey results and proposals". In: *Genetic Programming and Evolvable Machines* 14.1 (Mar. 2013), pp. 3–29. ISSN: 1389-2576. DOI: 10.1007/s10710-012-9177-2.

[194] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[195] Steven Gustafson, Edmund K Burke, and Natalio Krasnogor. "On Improving Genetic Programming for Symbolic Regression". In: *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*. Ed. by David Corne et al. Vol. 1. Edinburgh, Scotland, UK: IEEE Press, Sept. 2005, pp. 912–919. ISBN: 0-7803-9363-5.

[196] Maarten Keijzer. "Improving symbolic regression with interval arithmetic and linear scaling". In: *European Conference on Genetic Programming*. EuroGP. Essex, UK: Springer, 2003, pp. 70–82.

[197] John R. Koza and David Andre. *Parallel Genetic Programming on a Network of Transputers*. Technical Report CS-TR-95-1542. Stanford University, Department of Computer Science, Jan. 1995. URL: http://www.genetic-programming.com/jkpdf/tr1542parallelsuversion.pdf.

[198] Eric O. Scott and Kenneth A. De Jong. "Evaluation-Time Bias in Asynchronous Evolutionary Algorithms". In: *GECCO'15 Student Workshop*. Ed. by Tea Tusar and Boris Naujoks. Madrid, Spain: ACM, July 2015, pp. 1209–1212. DOI: 10.1145/2739482.2768482.

[199] Jinhan Kim, Junhwi Kim, and Shin Yoo. "GPGPGPU: Evaluation of Parallelisation of Genetic Programming using GPGPU". In: *Proceedings of the 9th International Symposium on Search Based Software Engineering, SSBSE 2017*. Ed. by Tim Menzies and Justyna Petke. Vol. 10452. LNCS. Paderborn, Germany: Springer, Sept. 2017, pp. 137–142. DOI: 10.1007/978-3-319-66299-2_11.

[200] Mouloud Oussaidène et al. "Parallel Genetic Programming and its application to trading model induction". In: *Parallel Computing* 23.8 (Aug. 1997), pp. 1183–1198. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(97)00045-8.

[201] Eric O. Scott and Kenneth A. De Jong. "Evaluation-Time Bias in Quasi-Generational and Steady-State Asynchronous Evolutionary Algorithms". In: *GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. Ed. by Tobias Friedrich et al. Denver, USA: ACM, July 2016, pp. 845–852. DOI: 10.1145/2908812.2908934.

[202] Erick Cantú-Paz. "A survey of parallel genetic algorithms". In: *Calculateurs paralleles, reseaux et systems repartis* 10.2 (1998), pp. 141–171.

[203] David Power, Conor Ryan, and Raja Muhammad Atif Azad. "Promoting diversity using migration strategies in distributed genetic algorithms". In: *2005 IEEE Congress on Evolutionary Computation*. Vol. 2. Edinburgh, Scotland, UK: IEEE Press, Sept. 2005, 1831–1838 Vol. 2. DOI: 10.1109/CEC.2005.1554910.

[204] Aliyu Sani Sambo et al. "Evolving simple and accurate symbolic regression models via asynchronous parallel computing". In: *Applied Soft Computing* 104 (2021), p. 107198. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2021.107198. URL: https://www.sciencedirect.com/science/article/pii/S1568494621001216.

[205] Aliyu Sani Sambo et al. "Feature Engineering for Improving Robustness of Crossover in Symbolic Regression". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. GECCO '20. internet: Association for Computing Machinery, July 2020, pp. 249–250. DOI: 10.1145/3377929.3390078. URL: https://doi.org/10.1145/3377929.3390078.

[206] Amir Hossein Gandomi and Amir Hossein Alavi. "A new multi-gene genetic programming approach to nonlinear system modeling. Part I: materials and structural engineering problems". In: *Neural Comput. Appl* 21.1 (2012), pp. 171–187.

[207] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. "Multiple regression genetic programming". In: *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*. Ed. by Christian Igel et al. Vancouver, BC, Canada: ACM, July 2014, pp. 879–886. DOI: 10.1145/2576768.2598291. URL: http://doi.acm.org/10.1145/2576768.2598291.

[208] Mihai Oltean and Crina Grosan. "A Comparison of Several Linear Genetic Programming Techniques". In: *Complex Systems* 14.4 (2004), pp. 285–313. ISSN: 0891-2513. URL: http://www.cs.ubbcluj.ro/~cgrosan/030409_edited.pdf.

[209] Shu-Heng Chen and Tzu-Wen Kuo. "Overfitting or Poor Learning: A Critique of Current Financial Applications of GP". In: *Genetic Programming, Proceedings of EuroGP'2003*. Ed. by Conor Ryan et al. Vol. 2610.

LNCS. Essex: Springer-Verlag, Apr. 2003, pp. 34–46. ISBN: 3-540-00971-X. DOI: 10.1007/3-540-36599-0_4. URL: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2610&spage=34.

[210] R. Muhammad Atif Azad and Conor Ryan. "The Best Things Don't Always Come in Small Packages: Constant Creation in Grammatical Evolution". In: *17th European Conference on Genetic Programming*. Ed. by Miguel Nicolau et al. Vol. 8599. LNCS. Granada, Spain: Springer, Apr. 2014, pp. 186–197. DOI: 10.1007/978-3-662-44303-3_16.

[211] Maarten Keijzer. "Improving Symbolic Regression with Interval Arithmetic and Linear Scaling". In: *Genetic Programming, Proceedings of EuroGP'2003*. Ed. by Conor Ryan et al. Vol. 2610. LNCS. Essex: Springer-Verlag, Apr. 2003, pp. 70–82. ISBN: 3-540-00971-X. DOI: 10.1007/3-540-36599-0_7. URL: http://www.cs.vu.nl/~mkeijzer/publications/eurogp2003.ps.gz.

[212] Alexandros Agapitos et al. "A Survey of Statistical Machine Learning Elements in Genetic Programming". In: *IEEE Transactions on Evolutionary Computation* 23.6 (Dec. 2019), pp. 1029–1048. ISSN: 1089-778X. DOI: doi:10.1109/TEVC.2019.2900916.

[213] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.

[214] Trent McConaghy. "FFX: Fast, scalable, deterministic symbolic regression technology". In: *Genetic Programming Theory and Practice IX*. Springer, 2011, pp. 235–260.

[215] William La Cava and Jason Moore. "A General Feature Engineering Wrapper for Machine Learning Using epsilon-Lexicase Survival". In: *EuroGP*

2017: Proceedings of the 20th European Conference on Genetic Programming. Ed. by Mauro Castelli, James McDermott, and Lukas Sekanina. Vol. 10196. LNCS. Amsterdam: Springer Verlag, Apr. 2017, pp. 80–95. DOI: doi:10.1007/978-3-319-55696-3_6.

[216] Leonardo Vanneschi et al. "PSXO: Population-wide Semantic Crossover". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '17. Berlin, Germany: ACM, July 2017, pp. 257–258. DOI: doi:10.1145/3067695.3076003. URL: http://doi.acm.org/10.1145/3067695.3076003.

[217] Krzysztof Krawiec. *Behavioral Program Synthesis with Genetic Programming*. Vol. 618. Studies in Computational Intelligence. Springer International Publishing, 2015. DOI: doi:10.1007/978-3-319-27565-9. URL: http://www.springer.com/gp/book/9783319275635.

[218] Krzysztof Krawiec and Una-May O'Reilly. "Behavioral programming: a broader and more detailed take on semantic GP". In: *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*. Ed. by Christian Igel et al. Best paper. Vancouver, BC, Canada: ACM, July 2014, pp. 935–942. DOI: doi:10.1145/2576768.2598288. URL: http://doi.acm.org/10.1145/2576768.2598288.

[219] Wolfgang Banzhaf et al. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. San Francisco, CA, USA: Morgan Kaufmann, Jan. 1998. ISBN: 1-55860-510-X. URL: https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X.

[220] Aliyu Sani Sambo et al. "Time Control or Size Control? Reducing Complexity and Improving Accuracy of Genetic Programming Models". In: *European Conference on Genetic Programming (Part of EvoStar)*. Springer. 2020, pp. 195–210. DOI: 10.1007/978-3-030-44094-7_13.

[221] John R Koza et al. *Genetic programming III: Darwinian invention and problem solving*. Vol. 3. Morgan Kaufmann, 1999.

[222] Jeremy Miles. "R Squared, Adjusted R Squared". In: *Wiley StatsRef: Statistics Reference Online*. John Wiley & Sons, Ltd, 2014. ISBN: 9781118445112. DOI: 10.1002/9781118445112.stat06627. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat06627.

[223] Ping Yin and Xitao Fan. "Estimating R 2 shrinkage in multiple regression: A comparison of different analytical methods". In: *The Journal of Experimental Education* 69.2 (2001), pp. 203–224.

[224] Randal S Olson et al. "PMLB: a large benchmark suite for machine learning evaluation and comparison". In: *BioData mining* 10.1 (2017), pp. 1–13.

[225] John R. Koza. "Human-competitive machine invention by means of genetic programming". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22.3 (2008), pp. 185–193.

[226] John R. Koza et al. "Fourteen instances where genetic programming has produced results that are competitive with results produced by humans". In: *Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'98)*. Ed. by Takeshi Gomi. Kanata, Canada: AAI Press, 1998, pp. 37–76. URL: http://www.genetic-programming.com/jkpdf/er1998.pdf.

[227] Karthik Kannappan et al. "Analyzing a Decade of Human-Competitive ("HUMIE") Winners: What Can We Learn?" In: *Genetic Programming Theory and Practice XII*. Ed. by Rick Riolo, William P. Worzel, and Mark Kotanchek. Genetic and Evolutionary Computation. Ann Arbor, USA: Springer, May 2014, pp. 149–166. DOI: doi:10.1007/978-3-319-16030-6_9.

[228] "A Genetic Programming Approach to Logic Function Synthesis by Means of Multiplexers". In: July 1999. URL: http://computer.org/proceedings/

eh/1999/0256/46;http://csdl.computer.org/comp/proceedings/eh/1999/0256/00/02560046abs.htm.

[229] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, May 1994.

[230] Simon Harding, Julian Francis Miller, and Wolfgang Banzhaf. "Self modifying cartesian genetic programming: Parity". In: *2009 IEEE Congress on Evolutionary Computation*. IEEE. 2009, pp. 285–292.

[231] Hitoshi Iba and Hugo de Garis. "Extending Genetic Programming with Recombinative Guidance". In: *Advances in Genetic Programming 2*. Ed. by Peter J. Angeline and K. E. Kinnear, Jr. Cambridge, MA, USA: MIT Press, 1996. Chap. 4, pp. 69–88. ISBN: 0-262-01158-1. DOI: doi:10.7551/mitpress/1109.003.0008. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6277499.

[232] David G Kleinbaum et al. *Logistic regression*. Springer, 2002.

[233] Sean Luke and Lee Spector. "A comparison of crossover and mutation in genetic programming". In: *Genetic Programming* 97 (1997), pp. 240–248.

[234] Ajith Abraham and Vitorino Ramos. "Web usage mining using artificial ant colony clustering and linear genetic programming". In: *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.* Vol. 2. IEEE. 2003, pp. 1384–1391.

[235] Edmund Burke, Steven Gustafson, and Graham Kendall. "A survey and analysis of diversity measures in genetic programming". In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. 2002, pp. 716–723.

[236] Francisco Fernandez, Marco Tomassini, and Leonardo Vanneschi. "An empirical study of multipopulation genetic programming". In: *Genetic Programming and Evolvable Machines* 4.1 (2003), pp. 21–51.

[237] W. B. Langdon and R. Poli. *Why Ants are Hard*. Tech. rep. CSRP-98-4. Presented at GP-98. University of Birmingham, School of Computer Science, Jan. 1998. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.6500`.

[238] Jason M. Daida et al. "What Makes a Problem GP-Hard? Analysis of a Tunably Difficult Problem in Genetic Programming". In: *Genetic Programming and Evolvable Machines* 2.2 (June 2001), pp. 165–191. ISSN: 1389-2576. DOI: `doi:10.1023/A:1011504414730`.

[239] Ibrahim Kuscu. "Evolution of Learning Rules for Hard Learning Problems". In: *Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming*. Ed. by Lawrence J. Fogel, Peter J. Angeline, and Thomas Baeck. San Diego: MIT Press, Feb. 1996. ISBN: 0-262-06190-2. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.4762`.

[240] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

Appendix A

Code Base

The code and data used in this thesis are available as follows:

1. for the explict time control methods introduced in Chapter 5 at:

   `https://github.com/aliyus/Explicit-Time-Control.git`

   Codes for the bloat control methods and standard GP are included.

2. for the Implicit Time Control code (APGP) introduced in Chapter 6 at: `https://github.com/aliyus/APGP.git`

3. for the hybridisation of GP with multiple linear regression (MLRGP) introduced in Chapter 8 at: `https://github.com/aliyus/MLR-GP.git`

4. for the GP applications used in Chapter 9 at:

   `https://github.com/aliyus/Applications-Time-Control.git`

Appendix B

Publications

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

Evolving simple and accurate symbolic regression models via asynchronous parallel computing

Aliyu Sani Sambo [a,*], R. Muhammad Atif Azad [a], Yevgeniya Kovalchuk [a], Vivek Padmanaabhan Indramohan [b], Hanifa Shah [c]

[a] School of Computing and Digital Technology, Birmingham City University, UK
[b] School of Health, Education and Life Sciences, Birmingham City University, UK
[c] Faculty of Computing, Engineering and the Built Environment, Birmingham City University, UK

ARTICLE INFO

Article history:
Received 9 October 2020
Received in revised form 9 February 2021
Accepted 13 February 2021
Available online 24 February 2021

Keywords:
Genetic programming
Model complexity
Parallel computing
Evaluation time

ABSTRACT

In machine learning, reducing the complexity of a model can help to improve its computational efficiency and avoid overfitting. In genetic programming (GP), the model complexity reduction is often achieved by reducing the size of evolved expressions. However, previous studies have demonstrated that the expression size reduction does not necessarily prevent model overfitting. Therefore, this paper uses the *evaluation time* – the computational time required to evaluate a GP model on data – as the estimate of model complexity. The evaluation time depends not only on the size of evolved expressions but also their *composition*, thus acting as a more nuanced measure of model complexity than the expression size alone. To discourage complexity, this study employs a novel method called asynchronous parallel GP (APGP) that introduces a race condition in the evolutionary process of GP; the race offers an evolutionary advantage to the simple solutions when their accuracy is competitive. To evaluate the proposed method, it is compared to the standard GP (GP) and GP with bloat control (GP+BC) methods on six challenging symbolic regression problems. APGP produced models that are significantly more accurate (on 6/6 problems) than those produced by both GP and GP+BC. In terms of complexity control, APGP prevailed over GP but not over GP+BC; however, GP+BC produced simpler solutions at the cost of test-set accuracy. Moreover, APGP took a significantly lower number of evaluations than both GP and GP+BC to meet a target training fitness in all tests. Our analysis of the proposed APGP also involved: (1) an ablation study that separated the proposed measure of complexity from the race condition in APGP and (2) the study of an initialisation scheme that encourages functional diversity in the initial population that improved the results for all the GP methods. These results question the overall benefits of bloat control and endorse the employment of both the evaluation time as an estimate of model complexity and the proposed APGP method for controlling it.

1. Introduction

The key challenges in managing the complexity of machine learning (ML) models include defining what complexity is and constructing a mechanism to control it; however, because the motivations behind existing definitions vary significantly, these definitions fail to transfer across various applications or algorithms. For example, a common reason for managing the complexity of ML models is attaining models that are just complex enough to explain the phenomenon generating the given data but not too complex to reflect noise in the training data. This way, the predictions on unseen data will be accurate [1]. Often in standard regression methods, the complexity is constrained by penalising the increase in the magnitudes of the coefficients of a fixed model. However, in methods such as Genetic Programming [2] the model is not fixed and hence such a penalty does not make sense. Another reason for controlling complexity is interpretability because simple models can be more interpretable [3]; interpretability of ML models is now a legal requirement due to frameworks such as the EU General Data Protection Regulation (GDPR).[1] Although very useful, such research produces post-hoc methods to explain pre-trained models on each training instance [4]. Therefore, this interpretability does not concern accuracy on test data. A yet

* Corresponding author.
E-mail address: aliyu.sambo@mail.bcu.ac.uk (A.S. Sambo).

[1] The EU General Data Protection Regulation includes a *right to explanation* in situations, where ML algorithms are applied to make a decision affecting a person.

https://doi.org/10.1016/j.asoc.2021.107198

another incentive for managing complexity is computational constraints. For example, *Internet of Things* (IoT) devices constrain the evaluation time of an acceptable model even if this compromises the model's accuracy [5]. However, a question arises as to whether constraining evaluation times can effectively decrease complexity and improve accuracy. In summary, the reasons for controlling model complexity vary and so does the notion of complexity [6].

This study focuses on the complexity of models produced by Genetic Programming (GP) [7]. In GP, the concern is that models may grow too complex and render ineffective evolutionary search [8]. Bloat control is thus the most common way of controlling model complexity in GP, and it limits the sizes of the evolved expressions. However, previous studies have demonstrated that bloat control alone does not always overcome the model overfitting problem (that is, a good performance on the training/seen data but a poor performance on the test/unseen data) [9,10].

To address such ineffective control of model complexity in GP, this paper presents a novel method called *Asynchronous Parallel Genetic Programming* (APGP). Instead of model size, APGP employs the *evaluation time* – the computational time required to evaluate a GP model on data – as a notion of its complexity. This notion is based on the observation that a model made up of computationally expensive building blocks or having large structures takes a long time to be evaluated, and hence is computationally complex. Therefore, the evaluation time control discourages both the structural as well as functional complexity.

The next question is how to control the evaluation times. Instead of subjectively penalising the slow evaluations, APGP takes a simple view: induce a *race* among competing models that allows a model to join the breeding population as soon as it has finished evaluating. Hence, the faster models can (fitness permitting) join the breeding population before their slower counterparts and gain an evolutionary advantage. This advantage arises because the competing models compete in terms of not only their accuracy but also their evaluation times due to the race; this is quite unlike in standard GP where each evaluation (or a batch of evaluations, as in generational replacement) is allowed to finish before the next evaluation (or a batch of evaluations) can start. Note, however, that selection is solely based on accuracy; therefore, APGP facilitates a dynamic interplay between accuracy and simplicity. To induce this race, APGP evaluates multiple models simultaneously across multiple asynchronous threads.

To evaluate the effectiveness of the proposed APGP method, this work first compares APGP with standard GP and GP with a very effective bloat control mechanism (GP+BC). The results indicate that APGP is capable of breeding models that are simpler than those produced with GP, yet more accurate on both training and test data than the models produced by the other two methods. Moreover, APGP takes a lower number of evaluations to match the training accuracy of GP compared to GP+BC.

This work then further analyses the proposed APGP in two ways. The first is an ablation study that analyses containing complexity *explicitly* with evaluation time instead of using APGP, which controls complexity *implicitly* with a race. To do that we employ four effective bloat-control techniques but to control evaluation time instead of expression sizes. The results indicate that while evaluation time control performs better than size control, the APGP still produces more accurate models (both training and test) than explicit control of evaluation time but while allowing greater model complexity. This suggests that the complexity control in APGP balances the accuracy-simplicity dilemma better than its counterparts.

This work also explored an initialisation scheme where the initial population comprises of identically sized individuals. This is because when models in a population are sized identically, the evaluation times are determined primarily by functional complexity; therefore, an evaluation time control can discourage functional complexity more than it can in a size-varying population. The results indicate that this initialisation scheme benefited all the methods, even the standard bloat control methods that do not employ time control.

The rest of this paper is organised as follows. Section 2 provides some background on GP and some notions of complexity related to it. Also in Section 2, the concept of evaluation time is introduced and the challenge of measuring it reliably is discussed. Section 3 presents the proposed APGP method. The experimental setup is outlined in Section 4. The results of the experiments are reported and discussed in Section 5. Sections 6 and 7 provide further analysis of the proposed method. Finally, Section 8 concludes the paper and outlines some directions for further work.

2. Background

2.1. Genetic Programming (GP)

GP enables computers to program themselves and build models automatically. It is also known as an evolutionary algorithm (EA) because it loosely imitates the Darwinian principle of natural selection to automatically search the space of possible models without any prior knowledge [2]. The process involves probabilistically choosing better performing individuals (models) from a given population, and producing offspring via simulated genetic operators of crossover and mutation. GP has been used to produce solutions to many practical problems. A detailed account of practical and innovative results produced by GP can be found in [11].

A variable tree-styled structure is the traditional and most popular representation of GP individuals [12], although other variations are also common [13–18]. These representations produce models of different sizes; in fact, these sizes may grow without improving accuracy (also known as code bloat) [19]. To address this issue, a lot of research efforts have been dedicated to developing methods for measuring and managing the complexity of GP models. The existing notions of complexity in GP and ways of its control are discussed below in turn.

2.2. Complexity in genetic programming

Some motivations for managing the complexity of computational models were discussed in Section 1. Here, we consider how some of these motivations are being addressed in GP. This involves examining the most widely used measures of complexity and mechanisms employed to control them.

2.2.1. Structural complexity

Traditionally, controlling complexity in GP means controlling the size of evolved structures, which includes limiting the number of nodes, encapsulated sub-trees and layers making up the evolved expressions, while ignoring the underlying functional or computational complexity.

The size is a simple and easy-to-calculate measure of complexity [20]. Early in the development of GP, it was observed that the size of GP models can grow unnecessarily (a phenomenon termed as bloat), severely constraining the computational resources [8]. The study of bloat control in GP has produced a large body of literature that cannot be covered extensively here. Instead, we provide a quick summary of some of the popular bloat control methods.

Many bloat-control techniques either set an arbitrary size (or depth) limit for models, or penalise large models [21]. However, such setups also encourage growing up to these size limits because that guarantees survival of such individuals after crossover [22,23]. Other bloat-control techniques either limit the search space or reduce the likelihood of producing large models through customisation of the evolutionary process [21]. Another approach is to use multi-objective genetic programming (MOGP) [20] that optimises the twin objectives of fitness and size to obtain Pareto optimal solutions. MOGP is sometimes used in combination with other measures of complexity.

Dynamic Operator equalisation (DynOpEQ) [24], a recent and advanced bloat-control technique, dynamically changes the distribution of size in a population to admit more individuals in those size ranges that are producing fitter models. First, the individuals of the population are segmented into bins according to their sizes. When a new generation is produced, the number of individuals allowed in each bin is determined by the average fitness score of the corresponding bin in the parent generation; bins with higher average fitness scores are allowed more individuals. However, an offspring will always be admitted if its fitness score is higher than the best of the bin it belongs to, even if the bin's quota has been exhausted. Further, if an offspring is the new best and its size does not fit into any existing bin, then a new bin is created for it. In effect, DynOpEQ does not guarantee that bloated individuals will not exist in the population; however, the population is controlled in such a way that growth in size is somewhat contained unless fitness is improving. DynOpEQ is inefficient as it discards a large portion of evaluated candidates; an evaluated offspring is likely to be discarded if the bin it belongs to is full. To alleviate this problem, Mutation Operator Equalisation [24] was introduced; it mutates candidates (in small steps) to fit into the nearest bin with available space. However, the required changes may be exceedingly destructive to the model's fitness when the distance between the individual's original size and the size it needs to be is too large.

Some studies have explored the concept of Kolmogorov complexity [25,26] to manage complexity in GP. This concept is adopted from algorithmic information theory and relates to specifying an object such as a binary string or a sequence of numbers. The Kolmogorov complexity of such an object is the length of the shortest computer program that can produce the object and nothing more. An abstract coding language (universal Turing machine) is used as a reference. However, this measure is uncomputable [27,28] because the output of every program cannot be computed to definitively determine the shortest. Hence, the minimum description length (MDL) [29,30], a computable form of the Kolmogorov complexity, has been applied in GP instead [31]. The study [31] calculated the MDL value of an individual by summing the length of code required to encode it and the length of code required to encode its classification errors. Thereafter, a scaling technique [32] was used to transform the MDL value to a windowed MDL value; the windowed MDL value is relative to the maximum MDL value of the individuals produced up to that point of the GP run. The MDL-based fitness function, which was setup to minimise the windowed MDL value, controlled the growth of the individuals. However, the implementation only works under two conditions: (1) where performance of individuals improves with the growth of the trees, and (2) where the fitness of the substructures of the trees are well-defined.

Restricting the structural representation of GP solutions is not an effective way of managing their complexity. In symbolic regression, for example, the model size does not represent the underlying functional and computational complexity of the model. For instance, restricting size would mean penalising the large yet linear expression $9x + 6x + 3x + 2x + x$, which is computationally less complex than the smaller expression $sin(x)$ [10], which is computationally equivalent to the implementation of its Taylor series expansion $\sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$. Moreover, the response surface of $sin(x)$ is more complex than that of the linear function that is larger in size. Further, two expressions with the same size may have different response surfaces; the response of the function $sin(9x)$ has greater fluctuations than $sin(2x)$. Thus, model complexity in GP is more than their representation.

2.2.2. Functional complexity

Approaches based on functional complexity focus on the functionality of models rather than their representation. To elicit functional complexity, one method approximates the evolved expressions with Chebyshev polynomials [33] such that functionally complex expressions are approximated by polynomials of a higher degree. The degree of approximating polynomials is termed as the *order of non-linearity* of the corresponding GP expressions [34], which should be minimised. However, the minimisation procedure requires the evolved expressions to be twice differentiable, a property that is not always guaranteed. The study [33] aimed to optimise three objectives: accuracy, the order of non-linearity and expressional complexity (size). A trade-off between accuracy and the order-of-non-linearity to guide selection led to improved generalisation over bloat control. Further, the study introduced a system that alternates between the optimisation of two sets of objectives during the evolution: (1) the optimisation of accuracy and the order of non-linearity, and (2) the optimisation of accuracy with size. This 2-D optimisation framework showed improvements in both managing the size of evolved models and their generalisation ability.

To avoid the constraint of twice-differentiability in [33] Vanneschi et al. [9] defined a less rigorous measure of functional complexity, whereby the slope of an expression along each feature dimension is approximated with a simpler but error-prone measure that approximates second order partial derivatives with a finite difference method that uses unequal intervals; however, the eventual measure of complexity that the work proposes is mathematically questionable because it simply averages these approximations to partial derivatives across all the feature dimensions to get an average complexity. To what extent that average diverges from a Hessian is not discussed. To avoid computational expense, this measure was only computed for the best individual in each generation. Crucially, however, the paper reported that a decrease in this measure did not necessarily decrease overfitting.

Castelli et al. [35] proposed other complexity measures that relate to curvature. The authors quantified the *degree of curvature* by examining the output of the pairs of close training points for a given model. The number of pairs with very different outputs were used to reflect the curvature. This formed the basis of two measures: (1) *graph-based complexity* to measure functional complexity and (2) *graph-based learning ability* to quantify the ability to learn difficult training points. The result of the experiments showed some generalisation gains over standard GP.

2.2.3. Complexity based on statistical learning

Methods for managing model complexity based on statistical learning theory, including generalisation error-bound Vapnik–Chervonenkis (VC) theory and Rademacher complexity theory [36, 37], are designed to find a balance between model complexity and its generalisation capability; while models should be complex enough to find patterns in training data, those that are too complex do not generalise well. The VC dimension is a general measure of the capacity or complexity of a learning machine [38, 39]. According to the original definition proposed for a set of indicator functions, the VC dimension is the maximum number of vectors that can be separated (shattered) into two classes in all possible ways by a set of functions [38]. The VC dimension is used to provide various estimations of generalisation errors.

Structural risk minimisation (SRM) is a framework that uses the VC dimension to assess the generalisation ability of a learning machine [40]. The assessment involves predicting the distance between the training and test errors. To do this, SRM defines the upper bound of the generalisation error based on the empirical risk (training error) and confidence interval. The confidence interval measures the difference between the empirical risk (training error) and the expected risk (generalisation error); it depends on (1) the size of the training set and (2) VC dimension. When the size of the training data is fixed, the generalisation bound is indicated by the VC dimension only; therefore, it is also called the VC generalisation bound or VC bound. SRM is used to produce an optimal model that finds a balance between minimising the upper bound of the generalisation error and minimising the training error. SRM was also used to manage the complexity of evolved models in [41]. While the proposed method outperformed standard GP by producing smaller sized models with better generalisation, the authors acknowledged the expensive computational cost of the method and lack of exploration of the parameters used in measuring the VC dimension.

Rademacher complexity extends the VC dimension to handle real-valued functions; therefore, it can be applied to both classification and regression problems. This measure of complexity has a tighter bound on the generalisation error than the VC dimension, and unlike the VC dimension, it depends on the data distribution. Rademacher complexity was used in [42] to drive the evolutionary process, which led to models with significantly smaller sizes and better generalisation ability compared to those generated by standard GP. However, the implementation has a lengthy process of tuning the parameter used to increase the pressure when overfitting occurs. Similar to the VC dimension, this is not a trivial task to compute.

Azad and Ryan [43] used the variance of the output values of evolving expressions to infer their complexity. They combined the variance and the fitness as twin objectives to optimise. Although variance differs with mathematical smoothness (a straight line can have a greater variance than a sinusoid), its combination with fitness meant that expressions within similar functional space may normally be compared in later generations with greater genetic convergence; however, this needs further verification. The method improved generalisation. Moreover, since the method does not require specialised multi-objective optimisation methods, it is simple to implement and computationally inexpensive.

Ni and Rockett [44] coined a measure of complexity that incorporates Tikhonov regularisation (as a functional complexity indicator) and size (a structural complexity measure). Tikhonov regularisation (also known as ridge regression) is a simple and common L^2 parameter regularisation strategy. This work was motivated by the understanding that addressing functional complexity does not necessarily address structural complexity and vice versa; also both are important in GP. Therefore, they used a two-dimensional vector that consisted of the Tikhonov regulariser (an indicator of the smoothness of response of the function) and the size of the model to represent its complexity; the Pareto optimal individual is considered the simpler individual. The two-dimensional vector was then used as an objective in conjunction with accuracy in a traditional multi-objective optimisation approach. This method led to generalisation gains over GP with size control. It also attained higher accuracy over GP with Tikhonov regularisation.

In summary, the popular techniques based on structural complexity tend to reduce the model sizes but not overfitting; likewise, the techniques based on functional complexity show some generalisation gains but often need to be married up with structural complexity and require parameter tuning overhead; also, these techniques are often tailored to specific tasks such as regression whereas automatic programming in GP extends beyond that. In contrast, the present study aims to induce non-complex models naturally regardless of the application domain by using the model evaluation time as a measure of its complexity.

The use of the evaluation time as an indicator of complexity is relatively new in GP such as in the proof of concept of this work [45]. At around the time of publication of [45] another study [46] also used evaluation times to control bloat, where the correlation between the size and the evaluation time was employed to manage bloat in a variety of problems. Although that second study corroborates our proposal that the evaluation time can be used to control model complexity, this work goes beyond bloat control to assess how evaluation time reflects both size and functional complexity, and proposes a new method, called APGP, for managing model complexity by enabling a race condition among evolved models.

2.3. Time is not size

Since Section 2.2.1 explains that model size does not necessarily represent functional complexity, this section empirically demonstrates how, instead, evaluation time can discriminate between different functional complexities. This demonstration is important because it verifies whether considerable time differences exist between different functional complexities of identically sized expressions so that a GP system can exploit these differences to promote functional simplicity. After all, evaluation time is also a function of model size, and if functional differences of identically sized expressions do not show up in evaluation times, then measuring time is just another way of measuring expression sizes. Clearly, this is undesirable.

To this end, four different *function sets* were used to generate symbolic regression models of different complexities. Functional complexity of these sets decreases in the following order: $\{cos, sin\}$, $\{cos, sin, +, -\}$, $\{\times, \div, +, -\}$ and $\{+, -\}$. Then, differently sized expressions (10, 20, 30, …, 300) were generated for each function set, and 30 random expressions were generated for each size. All models (expressions) were then evaluated 50 times, each with the same set of data. The four plots in Fig. 1 represent the average evaluation times of the individuals according to their size and complexity.

Two trends are clearly visible in Fig. 1: (1) for a given size, the higher the functional complexity the greater are the evaluation times; and (2) evaluation times are strongly correlated with the expression sizes, as expected. Hence, the evaluation time indeed discriminates between different functional complexities. However, if a simple function is represented inefficiently by an excessively large expression, it evaluates slowly. Therefore, evaluation time control impacts conditionally: it curbs functional complexity when individual sizes are similar and allows greater sizes for functionally simpler models up to a certain tolerance (or range); however, in the presence of very different sizes (such as during early generations) it curbs growth in size (controls bloat). To estimate the tolerance for greater sizes of simpler models, compare different curves at identical values along y-axis of Fig. 1: for example, at evaluation time = 300 ms, ADD-SUB models can be more than twice as large as COS-SIN models.

The above findings also reveal the limiting behaviour of evaluation time control in GP. In a functionally diverse but size-converged population – where bloat control is impotent – evaluation times discriminate between functional complexities, whereas in a functionally converged but size-diverse population, evaluation times discriminate between sizes.

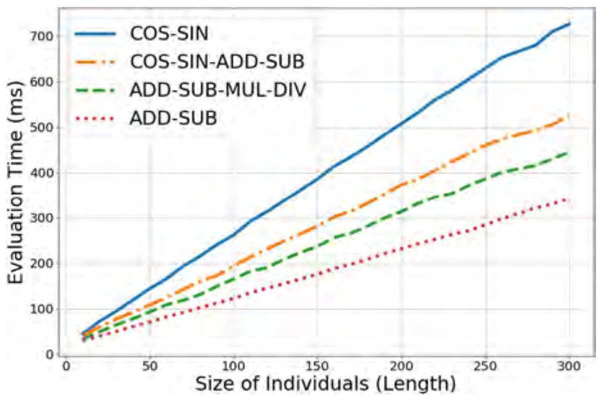


Fig. 1. Relationship between the evaluation time, size and composition of models. Individuals made up of COS and SIN operators have higher average evaluation times than same-sized individuals from other function sets. Furthermore, the size correlates with the evaluation time.

2.4. Measuring evaluation time reliably

Consistency in the measurement of the evaluation times is critical to estimate the complexity of models reliably. However, evaluation times vary across multiple executions, and if this variability is high, the reliability of the complexity estimate is low. However, this variability cannot be completely eliminated because it results from CPU scheduling decisions that are beyond our control. Still, we managed to significantly minimise this variation across evaluations.

In particular, we found that certain CPU management options can help minimise the evaluation time variation. These options include: stopping all background services; locking the CPU speed to prevent the operating system power management from interfering; executing the experiments on dedicated processors; and assigning the experimental tasks a high priority. Fig. 2 illustrates the impact of these options. In the figure, each box-plot represents multiple evaluation times for an individual of a given size. Fig. 2(a) shows that when no CPU management options were applied the variation in the evaluation times was high. In contrast, Fig. 2(b) shows that when the CPU management options were applied the variation clearly decreased. Thus, we were able to use a single evaluation to measure the evaluation time in the later reported experiments.

The time measurements were made using a CPU-time-based function that employs CPU performance counters [47]. This function is available in Python 3.3 and above. The function offers high resolution (in nanoseconds) across platforms, while the returned values are in fractional seconds.

The techniques used to improve time readings may not work in all situations. For example, stabilising time measurements becomes more challenging when the models being evaluated are exceptionally large, like those used in [48], where trees with millions of nodes were evolved. In this case, CPU memory caching comes into play. This is when the processor, while executing a task, has to access CPU memory caches (L1, L2 and L3) having different speeds. Moving from one type of cache to another may introduce delays and inconsistencies in the time measurements. In any case, the achieved improved stability enabled us to explore the proposed concepts and systems.

2.5. Parallel genetic programming

Because we leverage parallel computing for our proposed system, we first review its use in existing GP literature and contrast it with our objectives.

Although the use of parallel computing is not new in GP [49–51], it is typically to improve the run times [52] of GP, as opposed to reducing the complexity of individual models, which is the target of this study. As GP runs can take a long time to complete, parallelising these runs reduces the overall run time. Most commonly, generational replacement schemes parallelise the evaluation of offspring populations. However, the generational replacement requires the *entire* offspring population to be ready before its members can start breeding, which means that all evaluation threads join at a single *point of synchronisation* before the evolution proceeds further. Hence, this parallelisation confers no advantage to simpler individuals and is inefficient in terms of resource utilisation: while a complex individual is taking an excessive amount of time to be evaluated on one CPU, the remaining CPUs stay idle after other, less complex, individuals have been evaluated.

Some EAs have employed asynchronous parallel computing in non-GP setups to alleviate this problem of idle time. For example, an evaluation time bias favouring smaller evaluation times can be observed in [50,53]; however, these studies are concerned with real parameter optimisation using fixed-length chromosomes and do not study the impact of asynchronous parallelism on the functional complexity of variable length structures in GP.

EAs also use parallel computing in the so called *island model* [54,55], where the evolved populations are divided into multiple distinct islands, and the *sub-population* in each island can be evolved in a separate parallel thread. Regardless of parallelisation, the island model offers advantages such as greater diversity in the overall population because each island remains oblivious to other islands except at discrete intervals when selected individuals are sent to other islands. Parallelisation in this model just speeds up the run times.

3. Asynchronous Parallel Genetic Programming (APGP)

The APGP method evaluates GP individuals asynchronously to allow simple individuals that are evaluated faster an opportunity to get into the breeding population prior to their more complex (and still evaluating) counterparts. This mechanism is also natural; after all, natural populations do not simply halt breeding while one of their members is being tested against the environment. Yet, this is precisely what happens in traditional GP: while an individual of any complexity is being evaluated, the evolution stops regardless of how long that evaluation takes. Thus traditional GP confers no advantage to fast evaluations (and hence potentially low complexity). However, APGP aims to leverage the advantage of fast evaluations to breed simple yet accurate models.

APGP is based on *Steady State GP* [56], a replacement scheme in which the offspring immediately competes for a place in the parent population after being evaluated. In APGP steady state replacement allows multiple breeding operations and fitness evaluations to be executed in parallel and asynchronously. For example, a maximum of 50 of such breed operations (followed immediately by the corresponding fitness evaluation; note our setup produces one offspring per breed operation) can be allowed to run at the same time; as soon as one operation finishes, another one starts. However, these operations may finish at different times due to the varying times taken to evaluate different models. This difference is due only to the different make up of models because all models are evaluated on exactly the same dataset. Thus, a race condition develops such that less complex individuals taking less time to be evaluated can *apply* for a place into the population before their slower counterparts; applying for a place means checking if the new model is fitter than the

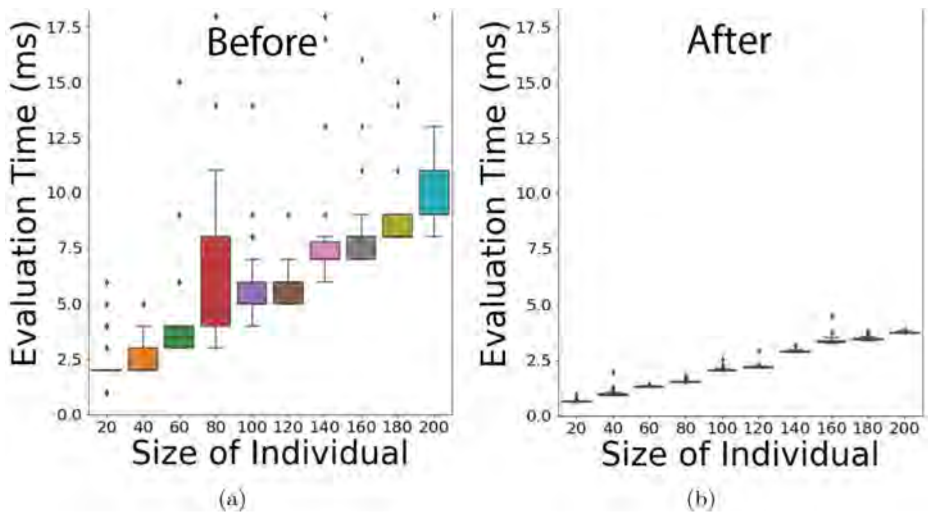


Fig. 2. Measuring evaluation time reliably: applying CPU management options leads to more consistent measurements.

"winner"[2] of an inverse tournament, and if so replacing that winner with the new model. Hence, the fast evaluating individuals may reproduce earlier than the more complex individuals taking longer to be evaluated.

Algorithm 1 lists the pseudo code of the APGP algorithm. The algorithm begins by initialising the number of allowed concurrent evaluations, population size and total number of offspring (total number of fitness evaluations in the run). This is followed by creating the initial population and evaluating it. The parallel breeding then begins by initiating multiple breed operations up to the allowed limit. These breed operations evaluate offspring in independent threads. As soon as an offspring completes evaluating, it replaces the winner of an inverse tournament in the current population if it is fitter than the winner, and the corresponding thread is released to allow another breed operation to commence. As these parallel operations work over the same population, a temporary lock is set on the position of the individual being replaced to avoid clashes.

As discussed above, the decision on whether an offspring finds a place in the population is made based on its accuracy only. As such, speed becomes an advantage only when it is accompanied by high accuracy. Where complex candidates are more accurate, they will eventually get in, get selected and propagate. Thus, complex models are not excluded, and the simplicity of good models is constrained by the possibilities within the specific problem.

Algorithm 1: Asynchronous Parallel Genetic Programming (APGP) Algorithm

```
/* Initialise                                                  */
$N \leftarrow$ set total number of offspring to produce;
$threadpool \leftarrow$ set number of concurrent operations allowed;
$popsize \leftarrow$ set population size;
/* Generate and evaluate initial population                    */
$population \leftarrow$ generate initial population of size $popsize$;
Evaluate($population$);
/* Generate and evaluate offspring in parallel                 */
$count = 0$;
while $count < N$ do
    if $threadpool > 0$ then
        Thread($threadpool \leftarrow threadpool - 1$ &&
        $offspring \leftarrow$ Breed_and_evaluate() );
        $count \leftarrow count + 1$;
        if $offspring\_evaluated$ then
            $replace \leftarrow$ To_Replace($population$);
            if $fitness(offspring) > fitness(population[replace])$ then
                Lock $population[replace]$ memory position;
                $population[replace] \leftarrow offspring$;
                Release locked position;
            else
                Discard $offspring$;
            Release thread: $threadpool \leftarrow threadpool + 1$;
    else
        Wait
end
```

4. Experiments

The performance of APGP was evaluated against standard GP and GP with an effective bloat control mechanism (GP+BC) on a suite of symbolic regression problems. Identical parameters were adopted across the three methods except: the race condition, which was present only in APGP; and the bloat control mechanism, which was present only in GP+BC.

4.1. Test problems

We considered the recommendations provided in [57] when choosing the test problems. Five multi-dimensional problems and one bi-dimensional problem were used. The datasets for Problems 1–5 are described in [58]; Problem 6 is a bi-variate version of the function used in [59]. The datasets are summarised in Table 1. As the results reported in Section 5 show, all but Dow are hard for GP (the accuracy scores are less than 41%). Hence, these problems require GP to run long and thus present a good test bed for evaluating complexity control because complexity in GP grows with the increase in the duration of runs.

4.2. Configuration and parameters

The basic parameters for all the methods are summarised in Table 2. Other key experimental decisions included the following.

[2] The winner of an inverse tournament is the worst member of the tournament; we use an inverse tournament of size 5.

Table 1
Overview of the test problems.

ID	Label	Test problem	No. of variables	No. of instances
1	*Airfoil*	Airfoil self-noise	5	1503
2	*Boston*	Boston housing	13	506
3	*Concrete*	Concrete strength	8	1030
4	*Dow*	Dow chemical	57	1066
5	*Energy*	Energy efficiency	8	768
6	*Y2X6*	$y^2x^6 - 2.13y^4x^4 + y^6x^2$	2	250 (x = min:-0.3, step: 0.012; $y = x + 0.03$)

- *Bloat Control for GP+BC*: *Double tournament* [60] was employed to control bloat. This is a method that has been very successful on a variety of benchmark problems [60,61], and the results of our experiments reported in Section 5 also show that this method indeed limits sizes aggressively. The best problem-independent settings for this method were used as recommended in [60]: in the first round, run n probabilistic tournaments, each with a tournament of size 2, to select a set of n individuals; then, in the second round, select the fittest out of the n individuals. The tournaments in the first round choose the smallest individual with a probability of 0.7. We also considered using Operator Equalisation (OpEq), a more recent bloat control method [62,63]; however, unlike APGP, which uses steady-state replacement, OpEq requires generational replacement.
- *Degree of Concurrency:* The size of the thread pool (number of parallel threads available) can vary in APGP. We tested pool sizes 5, 25, 50, 75 and 100. While size 50 produced the best or competitive results generally, some problem-specific improvement may be possible with other thread sizes. Future work can further investigate this.
- *System Configuration:* The experiments were run on Windows 10 (64-bit) with 32GB RAM, and Intel Core i7-6700 CPU @ 3.40 GHz (quad-core).
- *Additional Configuration and Parameters*: The other key experimental decisions were the following. First, the individuals with divide-by-zero errors were assigned the worst fitness; as discussed in [64], the protected operators commonly used in GP lead to poor generalisation. Second, the datasets were randomly split into training and test sets using the 80:20 ratio. Finally, the fitness was calculated as $\frac{1}{1+\frac{1}{n}\Sigma_{i=1}^{n}(y_i-\hat{y}_i)^2}$ such that y_i and $\hat{y}_i$ represent ideal and model's outputs respectively; the fitness value stays between 0 and 1, where a higher value is better.

5. Experimental results

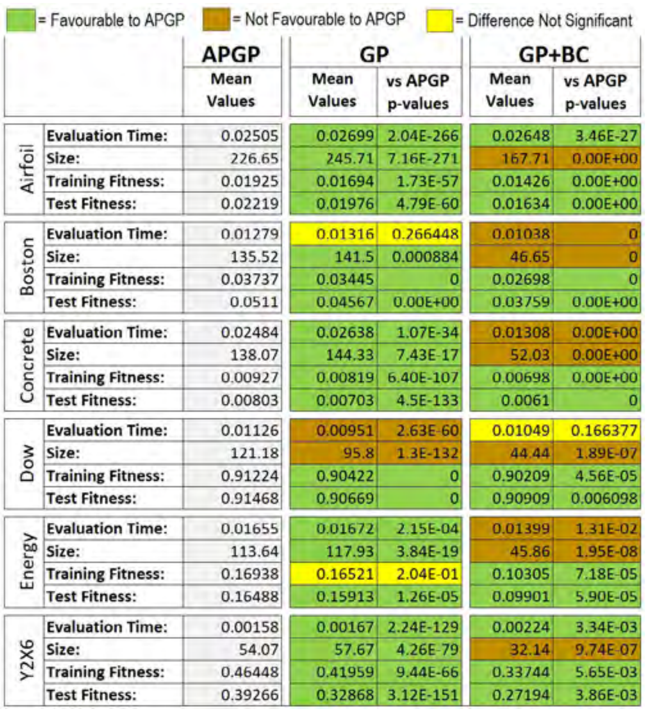
To evaluate the performance of the proposed APGP method, we compared the accuracy (fitness) scores of its evolved models over the training and test sets against those of models evolved by standard GP and GP+BC. In addition, we noted the average number of evaluations it took APGP and GP+BC to reach the average training accuracy of GP (target accuracy); this shows the efficiency of the other methods relative to GP.

5.1. Accuracy of the tested methods

As fitness in our experiments was set up as a maximisation function (where higher fitness scores are better), we use the term fitness and accuracy interchangeably. Fig. 3 shows the colour-coded results of the Mann–Whitney U statistical test on the final populations (of models) that were output by APGP, standard GP and GP+BC for all test problems. The results include the evaluation time, size, training and test fitness of the models. The

Table 2
Summary of parameters used in the tested methods.

Parameter	Setting
Number of runs	50
Population size	500
Run terminates	After 35,000 evaluations ($\equiv$ 70 generations)
Random tree/ subtree generation	Ramped half-and-half (depth *min* = 1, *max* = 4)
Tree depth limit	17
Operators & probabilities	One point crossover = 0.9 Point mutation = 0.1
Function set	$+, -, *, /$, sin, cos, neg
Ephemeral random constants (ERC)	$\lvert ERC \rvert = 100$ (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament size = 3
Replacement	steady state, inverse tournament size = 5


🟩 = Favourable to APGP 🟧 = Not Favourable to APGP 🟨 = Difference Not Significant

		APGP	GP		GP+BC	
		Mean Values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values
Airfoil	Evaluation Time:	0.02505	🟩 0.02699	🟩 2.04E-266	🟩 0.02648	🟩 3.46E-27
	Size:	226.65	🟩 245.71	🟩 7.16E-271	🟧 167.71	🟧 0.00E+00
	Training Fitness:	0.01925	🟩 0.01694	🟩 1.73E-57	🟩 0.01426	🟩 0.00E+00
	Test Fitness:	0.02219	🟩 0.01976	🟩 4.79E-60	🟩 0.01634	🟩 0.00E+00
Boston	Evaluation Time:	0.01279	🟨 0.01316	🟨 0.266448	🟧 0.01038	🟧 0
	Size:	135.52	🟩 141.5	🟩 0.000884	🟧 46.65	🟧 0
	Training Fitness:	0.03737	🟩 0.03445	🟩 0	🟩 0.02698	🟩 0
	Test Fitness:	0.0511	🟩 0.04567	🟩 0.00E+00	🟩 0.03759	🟩 0.00E+00
Concrete	Evaluation Time:	0.02484	🟩 0.02638	🟩 1.07E-34	🟧 0.01308	🟧 0.00E+00
	Size:	138.07	🟩 144.33	🟩 7.43E-17	🟧 52.03	🟧 0.00E+00
	Training Fitness:	0.00927	🟩 0.00819	🟩 6.40E-107	🟩 0.00698	🟩 0.00E+00
	Test Fitness:	0.00803	🟩 0.00703	🟩 4.5E-133	🟩 0.0061	🟩 0
Dow	Evaluation Time:	0.01126	🟧 0.00951	🟧 2.63E-60	🟨 0.01049	🟨 0.166377
	Size:	121.18	🟧 95.8	🟧 1.3E-132	🟧 44.44	🟧 1.89E-07
	Training Fitness:	0.91224	🟩 0.90422	🟩 0	🟩 0.90209	🟩 4.56E-05
	Test Fitness:	0.91468	🟩 0.90669	🟩 0	🟩 0.90909	🟩 0.006098
Energy	Evaluation Time:	0.01655	🟩 0.01672	🟩 2.15E-04	🟧 0.01399	🟧 1.31E-02
	Size:	113.64	🟩 117.93	🟩 3.84E-19	🟧 45.86	🟧 1.95E-08
	Training Fitness:	0.16938	🟨 0.16521	🟨 2.04E-01	🟩 0.10305	🟩 7.18E-05
	Test Fitness:	0.16488	🟩 0.15913	🟩 1.26E-05	🟩 0.09901	🟩 5.90E-05
Y2X6	Evaluation Time:	0.00158	🟩 0.00167	🟩 2.24E-129	🟩 0.00224	🟩 3.34E-03
	Size:	54.07	🟩 57.67	🟩 4.26E-79	🟧 32.14	🟧 9.74E-07
	Training Fitness:	0.46448	🟩 0.41959	🟩 9.44E-66	🟩 0.33744	🟩 5.65E-03
	Test Fitness:	0.39266	🟩 0.32868	🟩 3.12E-151	🟩 0.27194	🟩 3.86E-03

Fig. 3. Results of Mann–Whitney U tests on the final populations (APGP vs GP; APGP vs GP+BC). The results show that APGP produced significantly more accurate models (in terms of their training and test fitness) on all datasets compared to both GP and GP+BC. While APGP produced significantly simpler models than GP in five out of six test, GP+BC produced significantly simpler models than APGP at the price of accuracy.

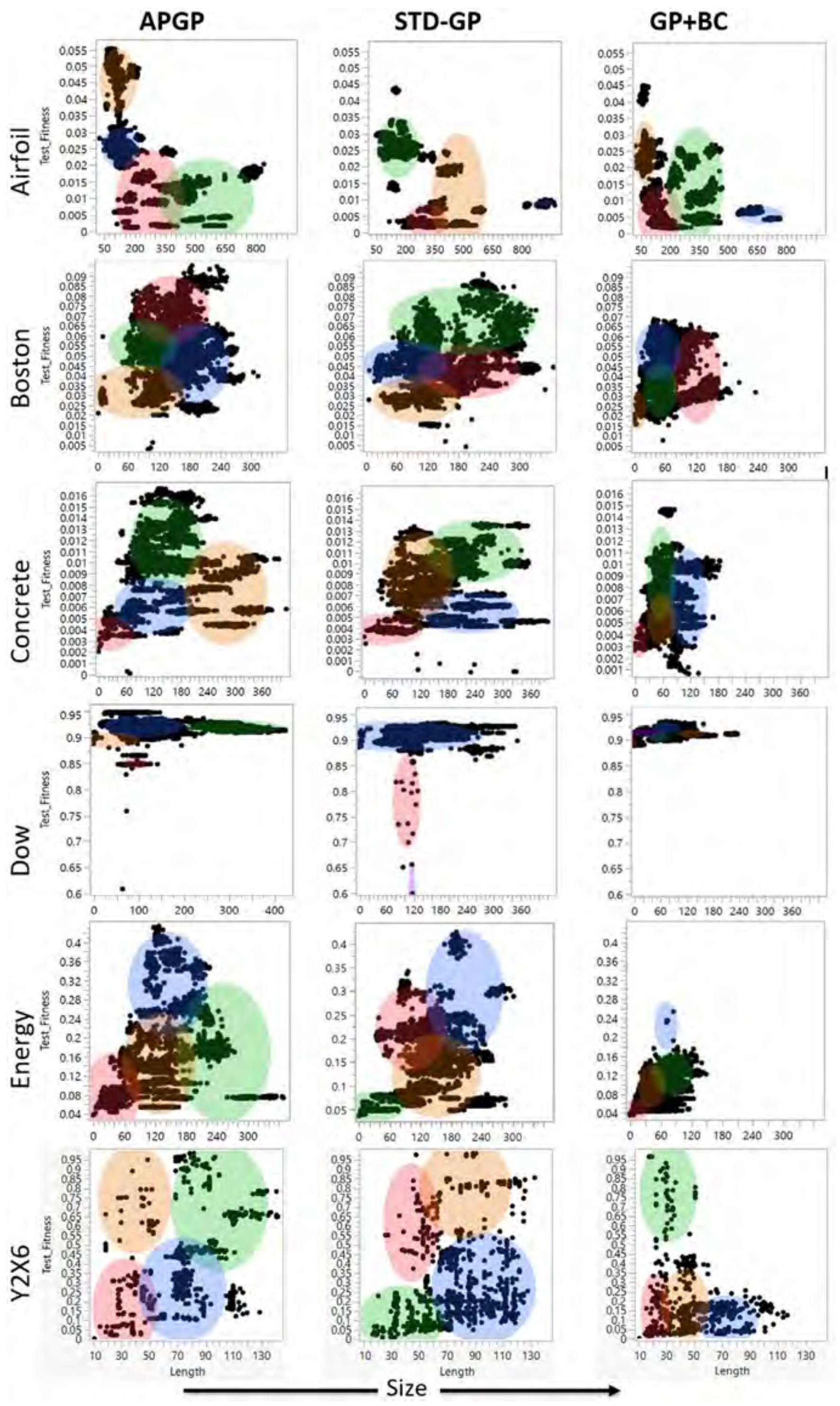


Fig. 4. Scatter-plot Test-fitness by Size of final populations of APGP, GP and GP+BC. The preferred individuals are in the top left corners of the charts (individuals with high test fitness accuracy and small sizes). The clusters discovered by K-means clustering are indicated.

p-values shown in Fig. 3 correspond to the statistical difference between APGP and GP, as well as APGP and GP+BC. The rows coloured in green represent the cases when APGP is significantly better (i.e. outputs less complex models or achieves higher accuracy) than either GP or GP+BC, red when it is significantly worse, and yellow when it is not significantly different.

The dominating green colour in columns 3 and 4 of Fig. 3 indicate that APGP significantly outperforms GP in the vast majority of the tests. In particular, the final population output by APGP is simpler than that output by GP for all the problems except Dow; this is indicated by significantly smaller mean evaluation times and sizes. Furthermore, the training and test accuracy scores (fitness) of APGP are significantly better than those of GP for all the problems except for the training fitness of Energy, where the improvement is not significant. Finally, the APGP test fitness values, which is the major concern when evaluating models, are significantly better than those of GP for all the problems. This indicates that the models output by APGP tend to generalise

Table 3
Comparison of the best individuals that were output by the three considered methods. Positive/negative numbers correspond to the increase/decrease in the corresponding metric values of APGP relative to GP or GP+BC. APGP produced more accurate models than GP and GP+BC for all the problems, however, sometimes, at the expense of longer evaluation times and/or larger model sizes.

Test ID	Test fitness gain	Eval. time increase	Size increase
Relative performance of APGP compared to GP			
Airfoil	+24.99%	−43.26%	−47.17%
Boston	+6.64%	+12.31%	−0.896%
Concrete	+22.43%	−36.99%	−44.96%
Dow	+2.04%	−78.13%	−85.71%
Energy	+3.13%	−22.92%	−18.38%
Y2X6	+0.11%	+42.57%	−11.11%
Relative performance of APGP compared to GP+BC			
Airfoil	+19.93%	+18.00%	+15.0%
Boston	+43.24%	+95.17%	+452.5%
Concrete	+11.56%	+72.70%	+100.0%
Dow	+2.43%	−78.56%	−65.82%
Energy	+69.10%	−34.51%	+40.91%
Y2X6	+1.71%	+49.45%	+695.0%

better on unseen data compared to the models output by GP. It can be noticed that the reported p-values are very small, which indicates that the differences between the results of the two methods are very significant.

While APGP produced more complex individuals than GP for Dow, it demonstrated better training and test fitness values. The relative increase in complexity (size and evaluation time) in this one instance comes with an associated gain in fitness; therefore, it is not necessarily bloat — bloat is growth in size without an associated gain in accuracy.

When comparing APGP with GP+BC, as captured in columns 5 and 6 of Fig. 3, GP+BC produced significantly smaller models for all the problems since it aggressively and solely targets the model size. However, these simpler models achieved significantly lower training and test accuracy values. In other words, simplicity in GP+BC came at the expense of accuracy.

While APGP outperformed GP and GP+BC *on average*, we were also interested in identifying the method that produced the better individuals; GP applications are often interested in the best solutions and not averages. We used the test fitness scores as a criteria for identifying the best of each method. A comparison is summarised in Table 3; the values show how APGP differs from the other compared methods in percentages. It can be noticed from the table that APGP produced individuals with the overall best test accuracy (fitness) values for all the six problems. As noted before, the best models that were output by APGP were also smaller than those output by GP but sometimes larger than GP+BC.

5.2. *Cost of producing accurate models*

Using the average training accuracy of GP as a benchmark, we compared the average evaluations taken by each method to match that training accuracy. Where a particular run did not achieve that target, it was assigned the maximum number of budgeted evaluations. As summarised in Table 4, APGP used 10% to 40% fewer evaluations than GP, and 15% to 84% fewer than GP+BC. Thus, APGP is the fastest to train, whereas GP+BC despite producing smaller individuals is the slowest; note, as in Fig. 3, both GP and GP+BC are also consistently less accurate than APGP on both training and test sets.

5.3. *Constitution of the populations*

The proposed APGP is based on the idea that simple models that are competitive will be allowed to thrive and that complex models that are accurate will not be excluded. In this section we examine the makeup of the final populations in terms of the accuracy and simplicity of the individual models.

Scatter-plots that plot test fitness against model-size in the final populations are presented in Fig. 4. Each method has a separate plot for each problem. Columns 1, 2 and 3 are plots for APGP, GP and GP+BC respectively. The plots also have clusters indicated (by ovals) that were identified using K-means algorithm [65].

The GP+BC plots (column 3) show more convergence towards the origin than the other methods. In other words, the aggressive restriction of size by GP+BC also restricts test fitness accuracy. On the other hand, GP allows size to grow with gains in accuracy. Whereas, APGP produced the highest test fitness scores while gently managing complexity.

These trends are similar to those observed in the population in Section 5.1, where it was established that the difference in the populations is statistically significant.

These observations suggest that the best test fitness values are not associated with either extreme of size restriction. They also suggest that APGP normally offers the best trade-off in terms of accuracy and simplicity.

As all the methods show variation within the populations and an appropriate size is problem specific, finding ideal solutions is more a matter of investigation than persistently controlling size. APGP aims to control complexity in an organic approach, where simple solutions can thrive when their training accuracy is better than their more complex counterparts.

From the above results the merit of the ideas behind the APGP system are apparent. However, a questions arises: if we use *time-control* methods that explicitly minimise evaluation time instead of sizes, can we reproduce the previous results without using the race condition that APGP uses to induce a more *implicit* time control? The next section explores this question.

6. Explicit time control

The proposed APGP controls complexity by implicitly controlling evaluation time. Therefore, it offers novelty in two fronts: (1) the idea that evaluation time can serve as a measure of complexity and (2) the race condition that APGP employs to control this indicator of complexity. Here we decouple the two to examine how explicit control of evaluation time compares with APGP. This will also enable us to explore the behaviour of evaluation time.

As in [66] that introduced explicit time control, we use four well-known techniques for bloat-control to now instead control evaluation time; hence, the techniques effect an explicit time-control. However, unlike in [66] that compared time-control with only standard GP methods, here we compare the explicit time-control against a relatively implicit time-control in APGP.

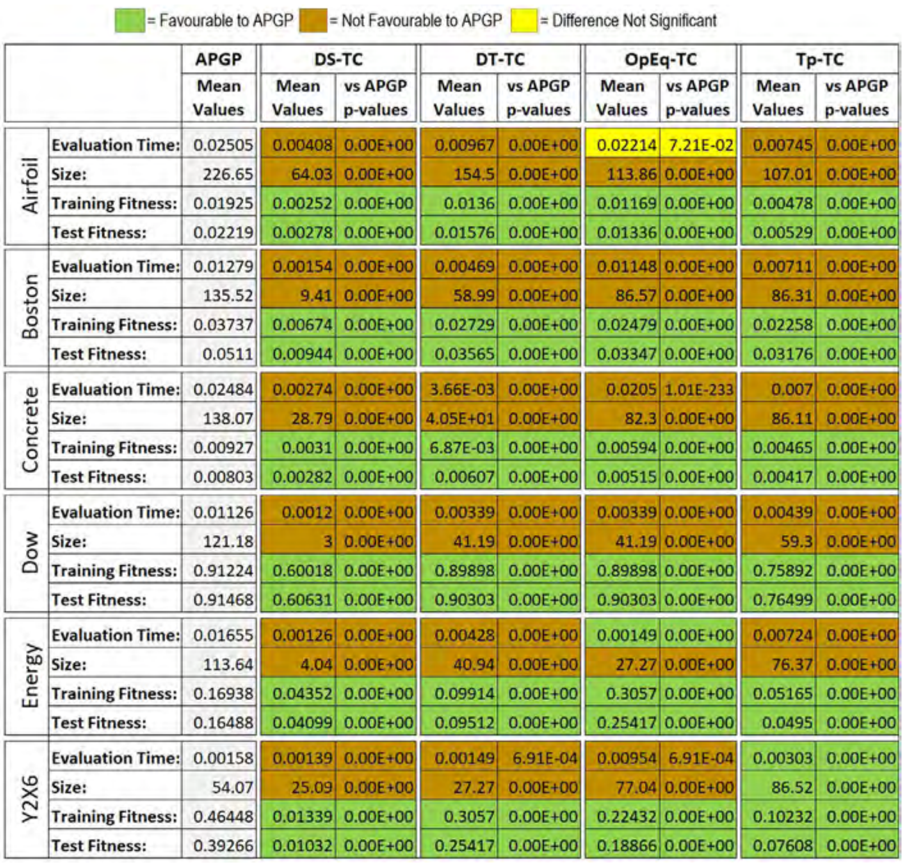
6.1. *Techniques for explicit time control*

The bloat-control techniques that were adapted to control evaluation time are as follows:

1. *Death by Size* (DS) [60] increases the probability of replacing the larger individuals from the present population. To replace an individual, DS randomly selects two individuals and replaces the larger with a given probability (typically 0.7; we use the same).

Table 4
APGP used significantly fewer evaluations than both GP and GP+BC to reach the same target accuracy. .

No. of evaluations to reach target accuracy					
Test ID	GP mean	GP+BC mean	APGP mean	Difference APGP /GP	Difference APGP /GP+BC
Airfoil	29407	33041	23941	18.59%	38.01%
Boston	20761	27792	17762	14.44%	56.47%
Concrete	31668	33595	20422	35.51%	64.50%
Dow	17861	19762	10719	39.99%	84.36%
Energy	28852	34842	25658	11.07%	35.79%%
Y2X6	33077	33895	29546	10.68%	14.72%



🟩 = Favourable to APGP 🟧 = Not Favourable to APGP 🟨 = Difference Not Significant

		APGP	DS-TC		DT-TC		OpEq-TC		Tp-TC	
		Mean Values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values
Airfoil	Evaluation Time:	0.02505	🟧 0.00408	0.00E+00	🟧 0.00967	0.00E+00	🟨 0.02214	7.21E-02	🟧 0.00745	0.00E+00
	Size:	226.65	🟧 64.03	0.00E+00	🟧 154.5	0.00E+00	🟧 113.86	0.00E+00	🟧 107.01	0.00E+00
	Training Fitness:	0.01925	🟩 0.00252	0.00E+00	🟩 0.0136	0.00E+00	🟩 0.01169	0.00E+00	🟩 0.00478	0.00E+00
	Test Fitness:	0.02219	🟩 0.00278	0.00E+00	🟩 0.01576	0.00E+00	🟩 0.01336	0.00E+00	🟩 0.00529	0.00E+00
Boston	Evaluation Time:	0.01279	🟧 0.00154	0.00E+00	🟧 0.00469	0.00E+00	🟧 0.01148	0.00E+00	🟧 0.00711	0.00E+00
	Size:	135.52	🟧 9.41	0.00E+00	🟧 58.99	0.00E+00	🟧 86.57	0.00E+00	🟧 86.31	0.00E+00
	Training Fitness:	0.03737	🟩 0.00674	0.00E+00	🟩 0.02729	0.00E+00	🟩 0.02479	0.00E+00	🟩 0.02258	0.00E+00
	Test Fitness:	0.0511	🟩 0.00944	0.00E+00	🟩 0.03565	0.00E+00	🟩 0.03347	0.00E+00	🟩 0.03176	0.00E+00
Concrete	Evaluation Time:	0.02484	🟧 0.00274	0.00E+00	🟧 3.66E-03	0.00E+00	🟧 0.0205	1.01E-233	🟧 0.007	0.00E+00
	Size:	138.07	🟧 28.79	0.00E+00	🟧 4.05E+01	0.00E+00	🟧 82.3	0.00E+00	🟧 86.11	0.00E+00
	Training Fitness:	0.00927	🟩 0.0031	0.00E+00	🟩 6.87E-03	0.00E+00	🟩 0.00594	0.00E+00	🟩 0.00465	0.00E+00
	Test Fitness:	0.00803	🟩 0.00282	0.00E+00	🟩 0.00607	0.00E+00	🟩 0.00515	0.00E+00	🟩 0.00417	0.00E+00
Dow	Evaluation Time:	0.01126	🟧 0.0012	0.00E+00	🟧 0.00339	0.00E+00	🟧 0.00339	0.00E+00	🟧 0.00439	0.00E+00
	Size:	121.18	🟧 3	0.00E+00	🟧 41.19	0.00E+00	🟧 41.19	0.00E+00	🟧 59.3	0.00E+00
	Training Fitness:	0.91224	🟩 0.60018	0.00E+00	🟩 0.89898	0.00E+00	🟩 0.89898	0.00E+00	🟩 0.75892	0.00E+00
	Test Fitness:	0.91468	🟩 0.60631	0.00E+00	🟩 0.90303	0.00E+00	🟩 0.90303	0.00E+00	🟩 0.76499	0.00E+00
Energy	Evaluation Time:	0.01655	🟧 0.00126	0.00E+00	🟧 0.00428	0.00E+00	🟩 0.00149	0.00E+00	🟧 0.00724	0.00E+00
	Size:	113.64	🟧 4.04	0.00E+00	🟧 40.94	0.00E+00	🟧 27.27	0.00E+00	🟧 76.37	0.00E+00
	Training Fitness:	0.16938	🟩 0.04352	0.00E+00	🟩 0.09914	0.00E+00	🟩 0.3057	0.00E+00	🟩 0.05165	0.00E+00
	Test Fitness:	0.16488	🟩 0.04099	0.00E+00	🟩 0.09512	0.00E+00	🟩 0.25417	0.00E+00	🟩 0.0495	0.00E+00
Y2X6	Evaluation Time:	0.00158	🟧 0.00139	0.00E+00	🟧 0.00149	6.91E-04	🟧 0.00954	6.91E-04	🟩 0.00303	0.00E+00
	Size:	54.07	🟧 25.09	0.00E+00	🟧 27.27	0.00E+00	🟧 77.04	0.00E+00	🟩 86.52	0.00E+00
	Training Fitness:	0.46448	🟩 0.01339	0.00E+00	🟩 0.3057	0.00E+00	🟩 0.22432	0.00E+00	🟩 0.10232	0.00E+00
	Test Fitness:	0.39266	🟩 0.01032	0.00E+00	🟩 0.25417	0.00E+00	🟩 0.18866	0.00E+00	🟩 0.07608	0.00E+00

Fig. 5. APGP vs explicit time-control methods. Detailed here are the results of the test for significance of difference in the final populations of APGP against those of the explicit time control methods. The results show that APGP produced significantly more accurate (training and test fitness) models on all tests against and against all time-control methods. The time-control methods produced simpler models at the cost of accuracy.

2. *Double Tournament* (DT) [60,61] encourages the reproduction of small offspring by increasing the probability of choosing smaller individuals as parents. This is achieved with two rounds of tournaments. In the first round, it runs n probabilistic tournaments each with a tournament of size 2 to select a set of n individuals. Each of these tournaments selects the smaller individual with a probability of 0.7. Then, in the second round, DT selects the fittest out of the n individuals.
3. *Operator Equalisation* (OpEq) [62,63] allows the sizes of individuals to grow only when fitness is improving. It controls the distribution of size in the population by employing two core functions. The first determines the target distribution (by size) of individuals in the next generation; the second ensures that the next generation matches the target distribution. To define the target distribution, OpEq puts the current individuals into bins according to their sizes and calculates the average fitness score of each bin. This average score is then used to calculate the number of individuals to be allowed in a corresponding bin in the next generation (target distribution). Thus, from one generation to the next the target distribution changes to favour the sizes that produce fitter individuals. In our experiments we used Dynamic OpEq, which is the variant that produces higher accuracy [63].
   To adapt OpEq to control evaluation time, we had to estimate the time equivalent of the bin width at the beginning of the run; we used *bin width = 5* in our experiments.
4. *The Tarpeian* (TP) [19] discourages growth in size by penalising larger individuals in the population and making them noncompetitive. This is effected by calculating the average size of the population at every generation and then assigning the worst fitness to a fraction W of the individuals that have above-average size (recommended $W = 0.3$; we use the same).

Similar to APGP, Death by Size and Double Tournament use Steady-state replacement strategy. However, although Operator

Equalisation and Tarpeian use the generational replacement strategy that differs from the steady state in APGP, we still report their results as benchmarks due to their popularity as bloat control methods.

6.2. Results of comparing APGP with explicit time control

APGP produced more accurate models (both training and test fitness) on all tests against all time-control methods. Fig. 5 details the results of the tests for significance of difference in the final populations. The improvement in accuracy by APGP was significant. Also, similar to the result of comparing APGP with bloat control in Section 5, the control of complexity by APGP was not as aggressive as the explicit time-control methods. The explicit time-control techniques produced significantly simpler models but lost out on accuracy.

Although [66] reported that the explicit control of evaluation time instead of size outperforms the standard GP methods with and without bloat control, these results show that APGP is still more accurate. Thus, the asynchronous parallel breeding in APGP manages complexity in such a way that allows greater accuracy and while keeping complexity less than that in standard GP. In the next section, we explore how evaluation time can be manoeuvred to enhance APGP.

7. APGP for functionally diverse populations

In Section 2.3 we demonstrated that the evaluation time can reflect both the size and functional complexity of models. This finding suggests that when the individuals are identically sized but functionally diverse, then differences in evaluation times will largely reflect the differences in functional or computational complexity of the functions that make up the models. Therefore, restricting evaluation time in this environment will restrict the complexity of the functions that make up the individuals. This in turn may lead to simpler functional behaviour of the models and better generalisation. Therefore, in [66] we proposed a *Fixed-Length Initialisation (FLI)* scheme that can be used to start the evolution with a size converged and functionally diverse population. The results demonstrated that using FLI significantly improves the test fitness on a variety of GP techniques both with bloat-control and time-control. Also, the results indicated that time-control with FLI produces simpler functions.

However, FLI was not tested with APGP in [66]; therefore, we test the impact of FLI on APGP in this paper.

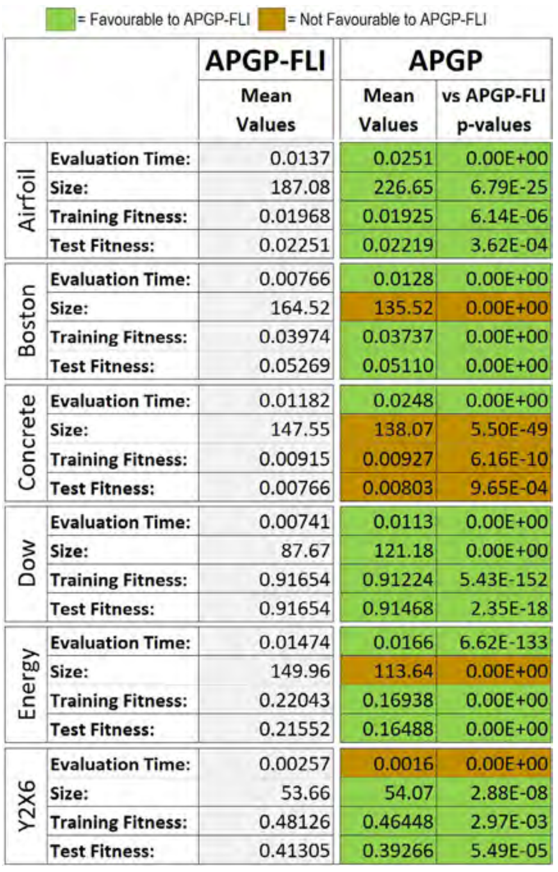
7.1. Fixed Length Initialisation scheme (FLI)

We use FLI to produce an initial population of unique individuals each having the same length (or size) of 10 nodes. Given the functions set size of 7 a length of 10 can easily produce populations of a few hundred unique individuals. Later in Section 7.4, we explored the impact of varying the lengths.

To encourage functional diversity, two individuals that only differ by numeric constants are considered the same.

7.2. Impact of fixed length initialisation on APGP

APGP with FLI (APGP-FLI) produced significantly more accurate models (on both training and test data) than plain APGP on five out of the six test problems; see Fig. 6. In terms of sizes the results were mixed (though APGP-FLI is better 4 out 6 times); however, in terms of evaluation times, APGP-FLI was significantly better than APGP on 5 out of six problems.



🟩 = Favourable to APGP-FLI 🟧 = Not Favourable to APGP-FLI

		APGP-FLI	APGP	
		Mean Values	Mean Values	vs APGP-FLI p-values
Airfoil	Evaluation Time:	0.0137	🟩 0.0251	🟩 0.00E+00
	Size:	187.08	🟩 226.65	🟩 6.79E-25
	Training Fitness:	0.01968	🟩 0.01925	🟩 6.14E-06
	Test Fitness:	0.02251	🟩 0.02219	🟩 3.62E-04
Boston	Evaluation Time:	0.00766	🟩 0.0128	🟩 0.00E+00
	Size:	164.52	🟧 135.52	🟧 0.00E+00
	Training Fitness:	0.03974	🟩 0.03737	🟩 0.00E+00
	Test Fitness:	0.05269	🟩 0.05110	🟩 0.00E+00
Concrete	Evaluation Time:	0.01182	🟩 0.0248	🟩 0.00E+00
	Size:	147.55	🟧 138.07	🟧 5.50E-49
	Training Fitness:	0.00915	🟧 0.00927	🟧 6.16E-10
	Test Fitness:	0.00766	🟧 0.00803	🟧 9.65E-04
Dow	Evaluation Time:	0.00741	🟩 0.0113	🟩 0.00E+00
	Size:	87.67	🟩 121.18	🟩 0.00E+00
	Training Fitness:	0.91654	🟩 0.91224	🟩 5.43E-152
	Test Fitness:	0.91654	🟩 0.91468	🟩 2.35E-18
Energy	Evaluation Time:	0.01474	🟩 0.0166	🟩 6.62E-133
	Size:	149.96	🟧 113.64	🟧 0.00E+00
	Training Fitness:	0.22043	🟩 0.16938	🟩 0.00E+00
	Test Fitness:	0.21552	🟩 0.16488	🟩 0.00E+00
Y2X6	Evaluation Time:	0.00257	🟧 0.0016	🟧 0.00E+00
	Size:	53.66	🟩 54.07	🟩 2.88E-08
	Training Fitness:	0.48126	🟩 0.46448	🟩 2.97E-03
	Test Fitness:	0.41305	🟩 0.39266	🟩 5.49E-05

Fig. 6. Impact of FLI on APGP. Statistical tests showed that apply FLI to APGP improved the mean accuracy values (both training and test) of the population in 5 out of 6 problems. Also there was a reduction in size and evaluation times in 4 out of 6 and 5 out of 6 respectively.

7.3. APGP with fixed length initialisation vs. time control with fixed length initialisation

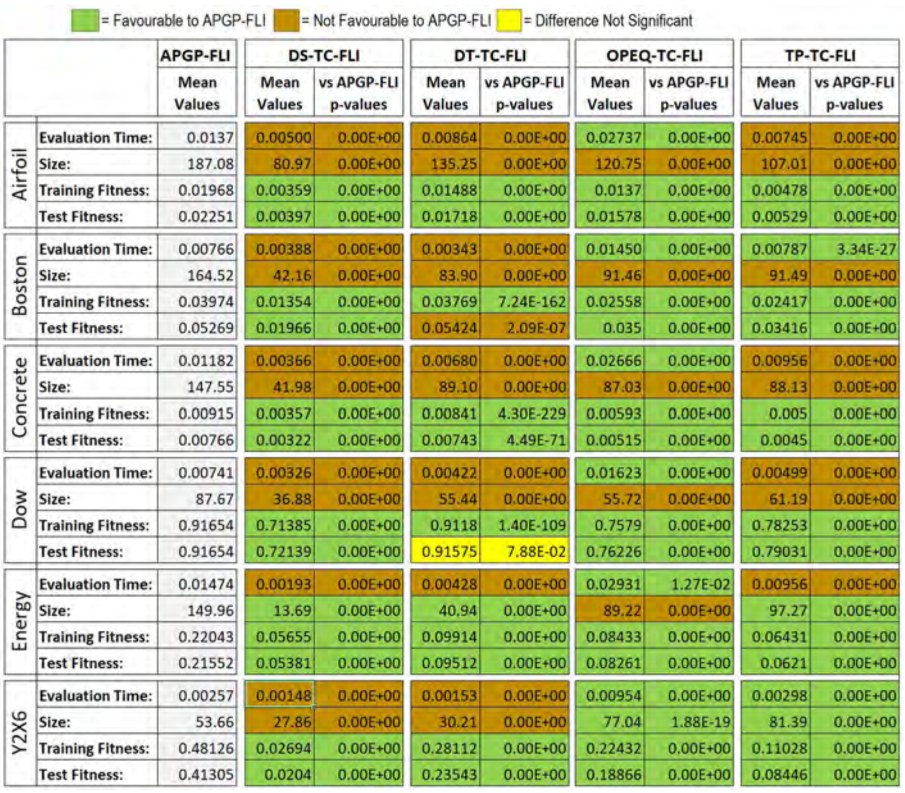
Next, we compare APGP-FLI against the time-control methods with FLI. For this, we re-ran time-control methods with FLI and tested for a significance in difference in the final populations against APGP-FLI; the results of the Mann-U Whitney tests are captured in Fig. 7.

The results show that APGP-FLI produced significantly more accurate individuals than all the compared time-control techniques with FLI. This is both in terms of training and test fitness accuracy. Out of the twenty-four comparisons of accuracy with APGP-FLI, only once another method outperformed APGP-FLI. Also, the accuracy difference was not significant in one out of twenty-four tests.

As with earlier comparisons with time-control methods, mostly APGP-FLI produced more complex individuals; however, again this complexity also brings higher accuracy on the test data.

7.4. Varying the lengths in the fixed length initialisation

In all the previous FLI experiments all the individuals in the initial generation were made up of ten nodes. Here we explore the impact of varying the initial size on APGP. Therefore, we re-ran the APGP-FLI with ten different sizes from five to fifty (FLI-5 to FLI-50, in steps of 5). Fig. 8 details the results. For each problem, box-plots of the average test fitness values and average lengths of the final populations are shown; the *X*-axis show the different FLI sizes.


🟩 = Favourable to APGP-FLI 🟧 = Not Favourable to APGP-FLI 🟨 = Difference Not Significant

		APGP-FLI	DS-TC-FLI		DT-TC-FLI		OPEQ-TC-FLI		TP-TC-FLI	
		Mean Values	Mean Values	vs APGP-FLI p-values	Mean Values	vs APGP-FLI p-values	Mean Values	vs APGP-FLI p-values	Mean Values	vs APGP-FLI p-values
Airfoil	Evaluation Time:	0.0137	🟧 0.00500	🟧 0.00E+00	🟧 0.00864	🟧 0.00E+00	🟩 0.02737	🟩 0.00E+00	🟧 0.00745	🟧 0.00E+00
	Size:	187.08	🟧 80.97	🟧 0.00E+00	🟧 135.25	🟧 0.00E+00	🟧 120.75	🟧 0.00E+00	🟧 107.01	🟧 0.00E+00
	Training Fitness:	0.01968	🟩 0.00359	🟩 0.00E+00	🟩 0.01488	🟩 0.00E+00	🟩 0.0137	🟩 0.00E+00	🟩 0.00478	🟩 0.00E+00
	Test Fitness:	0.02251	🟩 0.00397	🟩 0.00E+00	🟩 0.01718	🟩 0.00E+00	🟩 0.01578	🟩 0.00E+00	🟩 0.00529	🟩 0.00E+00
Boston	Evaluation Time:	0.00766	🟧 0.00388	🟧 0.00E+00	🟧 0.00343	🟧 0.00E+00	🟩 0.01450	🟩 0.00E+00	🟩 0.00787	🟩 3.34E-27
	Size:	164.52	🟧 42.16	🟧 0.00E+00	🟧 83.90	🟧 0.00E+00	🟧 91.46	🟧 0.00E+00	🟧 91.49	🟧 0.00E+00
	Training Fitness:	0.03974	🟩 0.01354	🟩 0.00E+00	🟩 0.03769	🟩 7.24E-162	🟩 0.02558	🟩 0.00E+00	🟩 0.02417	🟩 0.00E+00
	Test Fitness:	0.05269	🟩 0.01966	🟩 0.00E+00	🟧 0.05424	🟧 2.09E-07	🟩 0.035	🟩 0.00E+00	🟩 0.03416	🟩 0.00E+00
Concrete	Evaluation Time:	0.01182	🟧 0.00366	🟧 0.00E+00	🟧 0.00680	🟧 0.00E+00	🟩 0.02666	🟩 0.00E+00	🟧 0.00956	🟧 0.00E+00
	Size:	147.55	🟧 41.98	🟧 0.00E+00	🟧 89.10	🟧 0.00E+00	🟧 87.03	🟧 0.00E+00	🟧 88.13	🟧 0.00E+00
	Training Fitness:	0.00915	🟩 0.00357	🟩 0.00E+00	🟩 0.00841	🟩 4.30E-229	🟩 0.00593	🟩 0.00E+00	🟩 0.005	🟩 0.00E+00
	Test Fitness:	0.00766	🟩 0.00322	🟩 0.00E+00	🟩 0.00743	🟩 4.49E-71	🟩 0.00515	🟩 0.00E+00	🟩 0.0045	🟩 0.00E+00
Dow	Evaluation Time:	0.00741	🟧 0.00326	🟧 0.00E+00	🟧 0.00422	🟧 0.00E+00	🟩 0.01623	🟩 0.00E+00	🟧 0.00499	🟧 0.00E+00
	Size:	87.67	🟧 36.88	🟧 0.00E+00	🟧 55.44	🟧 0.00E+00	🟧 55.72	🟧 0.00E+00	🟧 61.19	🟧 0.00E+00
	Training Fitness:	0.91654	🟩 0.71385	🟩 0.00E+00	🟩 0.9118	🟩 1.40E-109	🟩 0.7579	🟩 0.00E+00	🟩 0.78253	🟩 0.00E+00
	Test Fitness:	0.91654	🟩 0.72139	🟩 0.00E+00	🟨 0.91575	🟨 7.88E-02	🟩 0.76226	🟩 0.00E+00	🟩 0.79031	🟩 0.00E+00
Energy	Evaluation Time:	0.01474	🟧 0.00193	🟧 0.00E+00	🟧 0.00428	🟧 0.00E+00	🟩 0.02931	🟩 1.27E-02	🟧 0.00956	🟧 0.00E+00
	Size:	149.96	🟩 13.69	🟩 0.00E+00	🟩 40.94	🟩 0.00E+00	🟧 89.22	🟧 0.00E+00	🟩 97.27	🟩 0.00E+00
	Training Fitness:	0.22043	🟩 0.05655	🟩 0.00E+00	🟩 0.09914	🟩 0.00E+00	🟩 0.08433	🟩 0.00E+00	🟩 0.06431	🟩 0.00E+00
	Test Fitness:	0.21552	🟩 0.05381	🟩 0.00E+00	🟩 0.09512	🟩 0.00E+00	🟩 0.08261	🟩 0.00E+00	🟩 0.0621	🟩 0.00E+00
Y2X6	Evaluation Time:	0.00257	🟧 0.00148	🟧 0.00E+00	🟧 0.00153	🟧 0.00E+00	🟩 0.00954	🟩 0.00E+00	🟩 0.00298	🟩 0.00E+00
	Size:	53.66	🟧 27.86	🟧 0.00E+00	🟧 30.21	🟧 0.00E+00	🟩 77.04	🟩 1.88E-19	🟩 81.39	🟩 0.00E+00
	Training Fitness:	0.48126	🟩 0.02694	🟩 0.00E+00	🟩 0.28112	🟩 0.00E+00	🟩 0.22432	🟩 0.00E+00	🟩 0.11028	🟩 0.00E+00
	Test Fitness:	0.41305	🟩 0.0204	🟩 0.00E+00	🟩 0.23543	🟩 0.00E+00	🟩 0.18866	🟩 0.00E+00	🟩 0.08446	🟩 0.00E+00

Fig. 7. Results of Mann–Whitney U test for significance in the differences between the final populations of APGP with FLI and explicit time-control with FLI. APGP produced more accurate solutions (both training and test) in all cases. Explicit time-control produced simpler (smaller sizes and evaluation times) models than APGP in xx out of 24 tests.

The lengths of individuals in the final populations are not affected significantly despite varying FLI lengths. However, there is some activity in test-fitness: with the exception of Boston, the results show that starting with individuals of size five (5 nodes each) appears disadvantageous. The lengths that produced the best results are problem specific. However, it is reassuring to note that relatively lower lengths in FLI-10, FLI-15 and FLI-20 are at least competitive with the greater lengths.

7.5. Summary of APGP in a functionally diverse population

The results in this section show that FLI significantly improved APGP. Similarly, our related work [66] showed that FLI improved the performance of time-control methods. However, APGP retained its leading position in the environment that FLI creates.

The analysis of time in Section 2.3 indicated that in a functionally diverse and size converged environment, controlling time will distinguish complexity based more on composition than on size; FLI creates such an environment at the beginning of the evolution but does not control the remaining generations. Future work will study if encouraging such an environment in the remaining evolution will further intensify the effect of time-control.

8. Conclusions and future work

This paper exemplifies redefining complexity in GP with the evaluation time. The evaluation time is both a function of the size, as well as functional and computational effort of a model. Although we only applied the measure here to regression problems, it is also applicable to other domains.

A criticism of the proposed evaluation time measure is the variability in its repeated measurements. To address this issue, the paper demonstrated a way to minimise this variability, so that the evaluation time could be measured reliably.

Leveraging evaluation times, this paper further presented a novel method, called Asynchronous Parallel GP (APGP), which uses asynchronous parallel computing to induce a race between concurrent executions of multiple models to discourage complexity. According to this race condition, models are allowed to join a steady-state population as soon as they are evaluated rather than waiting for other models to be evaluated. This condition provides an evolutionary advantage to simpler models, that are also competitive in terms of accuracy. APGP thus challenges the conventional, but ultimately unnatural, way of evaluating individuals one after the other or in a lock-step.

Unlike aggressive bloat control techniques that penalise sizes at the expense of achieving lower accuracy and hence increasing training times, APGP produces models that are simpler than those from standard GP and are more accurate than those from GP both with or without complexity control; complexity was controlled with standard bloat control methods or their time variants that we defined as *time-control* methods. Against standard GP, APGP improved test fitness accuracy on 6/6 test; the solutions were also significantly simpler in 5/6 tests, both in terms of size and evaluation times. Against GP with bloat control (GP+BC), APGP improved test fitness accuracy on 6/6 test; but the solutions were not simpler than BC (in 3/6 for evaluation times and 6/6 for size). Thus, instead of aggressively going after complexity control, the APGP decreases complexity but not to the extent that it starts hurting accuracy on test data.

The APGP trained faster than GP and GP+BC on all 6 problems; it reached the average training scores of GP with 11% to 40%

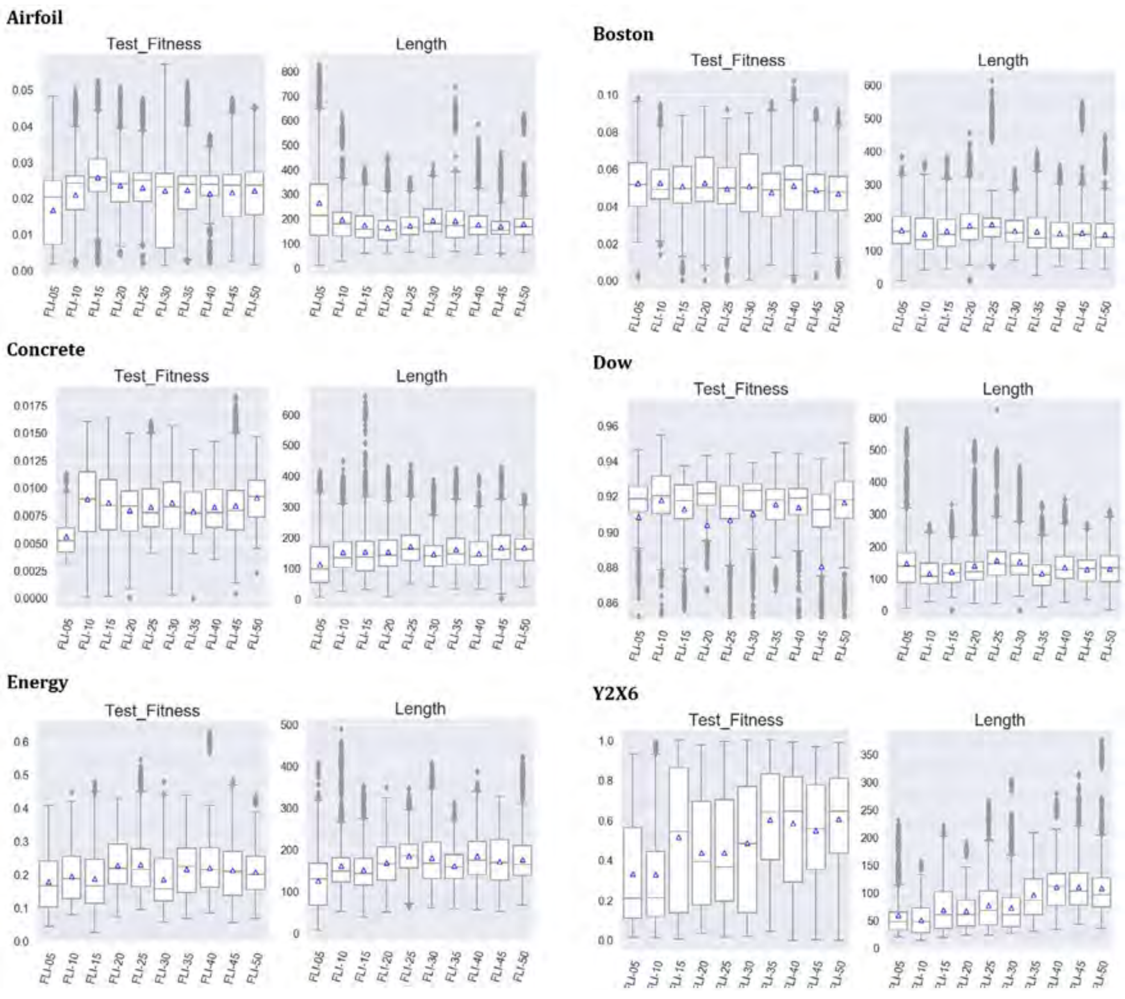


Fig. 8. Impact of changing the sizes of individuals for FLI. Despite varying FLI lengths, the lengths of individuals in the final populations has remained fairly stable. However, some changes were observed with test-fitness accuracy.

fewer evaluations than GP and 15% to 84% fewer evaluations than GB+BC.

When APGP was compared against the explicit control of evaluation time using four well-known and effective bloat control techniques, APGP produced significantly more accurate solutions on all tests (24/24). The explicit control methods controlled complexity more aggressively; they produced significantly simpler models in all tests but 1/24 for size and 3/24 for evaluation times. Again, this shows the merit of APGP's approach of not directly targeting and penalising the complexity of models but rather encouraging simplicity in competitive models.

Motivated by the idea that the evaluation time can distinguish functional complexity in a functionally diverse and size-converged environment, we tested APGP with a new and effective initialisation scheme called the Fixed Length Initialisation (FLI). The FLI improved APGP even further: The test set accuracy increased significantly in 5/6 tests; and the complexity decreased significantly in 8/12 tests (3/6 for size and 5/6 for evaluation times).

We also tested the sensitivity of the FLI to different initial lengths and we found that the performance of APGP remains largely robust against the variation. However, usefully, the relatively smaller lengths of 10, 15, and 20 produced at least as good performance as the greater lengths. While FLI allows functional complexity to be detected in the earliest generations, future work can explore similar possibilities throughout the evolution.

In summary, the results of the experiments presented in this paper demonstrated that APGP has the potential for generating simple and accurate models fast. While this study focused on regression problems only, in principle, the evaluation time can represent complexity in other domains as well. In the future, we plan to test the proposed method on classification problem with discrete fitness values and non-machine-learning problems such as automatic programming and design.

CRediT authorship contribution statement

Aliyu Sani Sambo: Conceptualization, Software, Investigation, Methodology, Writing - original draft. **R. Muhammad Atif Azad:** Conceptualization, Methodology, Supervision, Writing - review & editing. **Yevgeniya Kovalchuk:** Supervision, Writing - review & editing. **Vivek Padmanaabhan Indramohan:** Supervision. **Hanifa Shah:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] G. Paris, D. Robilliard, C. Fonlupt, Exploring overfitting in genetic programming, in: P. Liardet, P. Collet, C. Fonlupt, E. Lutton, M. Schoenauer (Eds.), Evolution Artificielle, 6th International Conference, in: Lecture Notes in Computer Science, 2936, Springer, Marseilles, France, 2003, pp. 267–277, http://dx.doi.org/10.1007/b96080, Revised Selected Papers.

[2] J.R. Koza, Genetic programming: On the programming of computers by means of natural selection, MIT Press, Cambridge, MA, USA, 1992, http://mitpress.mit.edu/books/genetic-programming.

[3] Z.C. Lipton, The mythos of model interpretability, Commun. ACM 61 (10) (2018) 36–43, http://dx.doi.org/10.1145/3233231, http://doi.acm.org/10.1145/3233231.

[4] J. Hatwell, M.M. Gaber, R.M.A. Azad, CHIRPS: Explaining random forest classification, Artif. Intell. Rev. (2020) 1–42, http://dx.doi.org/10.1007/s10462-020-09833-6, https://doi.org/10.1007/s10462-020-09833-6.

[5] A. Kumar, S. Goyal, M. Varma, Resource-efficient machine learning in 2 KB RAM for the internet of things, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, 70, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 1935–1944.

[6] M. Couture, Complexity and Chaos-State-Of-The-Art; Formulations and Measures of Complexity, Tech. rep., Defence research and development Canada Valcartier (QUEBEC), 2007.

[7] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA, 1992, http://mitpress.mit.edu/books/genetic-programming.

[8] T. Soule, J.A. Foster, J. Dickinson, Code growth in genetic programming, in: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, 1996, pp. 215–223, http://cognet.mit.edu/sites/default/files/books/9780262315876/pdfs/9780262315876_chap26.pdf.

[9] L. Vanneschi, M. Castelli, S. Silva, Measuring bloat, overfitting and functional complexity in genetic programming, in: J.B. et al (Ed.), GECCO '10: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, ACM, Portland, Oregon, USA, 2010, pp. 877–884, http://dx.doi.org/10.1145/1830483.1830643.

[10] R.M.A. Azad, C. Ryan, A simple approach to lifetime learning in genetic programming based symbolic regression, Evol. Comput. 22 (2) (2014) 287–317, http://dx.doi.org/10.1162/EVCO_a_00111, http://www.mitpressjournals.org/doi/abs/10.1162/EVCO_a_00111.

[11] J.R. Koza, Human-competitive machine invention by means of genetic programming, Artif. Intell. Eng. Des. Anal. Manuf. 22 (3) (2008) 185–193, http://dx.doi.org/10.1017/S0890060408000127.

[12] N.X. Hoai, R.I.B. McKay, D. Essam, Representation and structural difficulty in genetic programming, IEEE Trans. Evol. Comput. 10 (2) (2006) 157–166, http://dx.doi.org/10.1109/TEVC.2006.871252, http://sc.snu.ac.kr/courses/2006/fall/pg/aai/GP/nguyen/Structdiff.pdf.

[13] C. Ryan, M. O'Neill, J.J. Collins (Eds.), Handbook of Grammatical Evolution, Springer, 2018, http://dx.doi.org/10.1007/978-3-319-78717-6.

[14] R.M.A. Azad, A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System (Ph.D. thesis), University of Limerick, Ireland, 2003, http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/azad_thesis.ps.gz.

[15] G. Chennupati, R.M.A. Azad, C. Ryan, Performance optimization of multicore grammatical evolution generated parallel recursive programs, in: S.S. et al (Ed.), GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, Madrid, Spain, 2015, pp. 1007–1014, http://dx.doi.org/10.1145/2739480.2754746, http://doi.acm.org/10.1145/2739480.2754746.

[16] L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the push programming language, Genet. Program. Evol. Mach. 3 (1) (2002) 7–40, http://dx.doi.org/10.1023/A:1014538503543, http://hampshire.edu/lspector/pubs/push-gpem-final.pdf.

[17] T. Hu, J. Payne, W. Banzhaf, J. Moore, Evolutionary dynamics on multiple scales: a quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming, Genet. Program. Evol. Mach. 13 (3) (2012) 305–337, http://dx.doi.org/10.1007/s10710-012-9159-4, Special issue on selected papers from the 2011 European conference on genetic programming.

[18] J.A. Walker, J.F. Miller, The automatic acquisition, evolution and reuse of modules in cartesian genetic programming, IEEE Trans. Evol. Comput. 12 (4) (2008) 397–417, http://dx.doi.org/10.1109/TEVC.2007.903549, http://results.ref.ac.uk/Submissions/Output/3354578.

[19] R. Poli, A simple but theoretically-motivated method to control bloat in genetic programming, in: C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, E. Costa (Eds.), Genetic Programming, Proceedings of EuroGP'2003, in: LNCS, 2610, Springer-Verlag, Essex, 2003, pp. 204–217, http://dx.doi.org/10.1007/3-540-36599-0_19.

[20] A. Ekart, S.Z. Nemeth, Selection based on the Pareto nondomination criterion for controlling code growth in genetic programming, Genet. Programm. Evol. Mach. 2 (1) (2001) 61–73, http://dx.doi.org/10.1023/A:1010070616149.

[21] S. Luke, L. Panait, A comparison of bloat control methods for genetic programming, Evol. Comput. 14 (3) (2006) 309–344, http://dx.doi.org/10.1162/evco.2006.14.3.309, http://cognet.mit.edu/system/cogfiles/journalpdfs/evco.2006.14.3.309.pdf.

[22] S. Dignum, R. Poli, Crossover, sampling, bloat and the harmful effects of size limits, in: European Conference on Genetic Programming, Springer, 2008, pp. 158–169.

[23] N.F. McPhee, A. Jarvis, E.F. Crane, On the strength of size limits in linear genetic programming, in: K. Deb (Ed.), Genetic and Evolutionary Computation – GECCO 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 593–604.

[24] S. Silva, S. Dignum, L. Vanneschi, Operator equalisation for bloat free genetic programming and a survey of bloat control methods, Genetic Program. Evol. Mach. 13 (2) (2012) 197–238, http://dx.doi.org/10.1007/s10710-011-9150-5.

[25] A.N. Kolmogorov, Three approaches to the quantitative definition ofinformation', Probl. Inf. Transm. 1 (1) (1965) 1–7.

[26] T.M. Cover, J. Thomas, Joint entropy and conditional entropy, in: Elements of Information Theory, second ed., John Wiley & Sons: Hoboken, NJ, USA, 2006, p. 16.

[27] P. Vitányi, How incomputable is Kolmogorov complexity?, Entropy 22 (4) (2020) 408.

[28] A.K. Zvonkin, L.A. Levin, The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms, Russian Math. Surveys 25 (6) (1970) 83.

[29] J. Rissanen, Modeling by shortest data description, Automatica 14 (5) (1978) 465–471.

[30] V. Nannen, A short introduction to model selection, Kolmogorov complexity and minimum description length (MDL), 2010, arXiv preprint arXiv:1005.2364.

[31] H. Iba, H. de Garis, T. Sato, Genetic programming using a minimum description length principle, in: K.E. Kinnear, Jr. (Ed.), Advances in Genetic Programming, MIT Press, Cambridge, MA, USA, 1994, pp. 265–284, http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap12.pdf.

[32] N.N. Schraudolph, J.J. Grefenstette, A user's guide to GAucsd 1.4, in: Computer Science & Engineering Department, University of California, San Diego, 1992, p. 1991.

[33] E.J. Vladislavleva, G.F. Smits, D. den Hertog, Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming, IEEE Trans. Evol. Comput. 13 (2) (2009) 333–349, http://dx.doi.org/10.1109/TEVC.2008.926486.

[34] T. Rivlin, The Chebyshev Polynomials, in: A Wiley-Interscience publication, Wiley, 1974.

[35] M. Castelli, L. Manzoni, S. Silva, L. Vanneschi, A quantitative study of learning and generalization in genetic programming, in: S. Silva, J.A. Foster, M. Nicolau, M. Giacobini, P. Machado (Eds.), Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011, in: LNCS, 6621, Springer Verlag, Turin, Italy, 2011, pp. 25–36, http://dx.doi.org/10.1007/978-3-642-20407-4_3.

[36] S.R. Kulkarni, G. Harman, Statistical learning theory: a tutorial, Wiley Interdiscip. Rev. Comput. Stat. 3 (6) (2011) 543–556.

[37] V.N. Vapnik, Statistical learning theory, in: Adaptive and learning systems for signal processing, communications, and control, Wiley, New York, NY, 1998, OCLC: 845016043.

[38] V. Vapnik, The Nature of Statistical Learning Theory, in: Information Science and Statistics, Springer New York, 2013.

[39] V. Vapnik, Statistical learning theory. 1998, Vol. 3, Wiley, New York, 1998.

[40] Q. Chen, M. Zhang, B. Xue, Structural risk minimisation-driven genetic programming for enhancing generalisation in symbolic regression, IEEE Trans. Evol. Comput. 23 (4) (2019) 703–717, http://dx.doi.org/10.1109/TEVC.2018.2881392.

[41] Q. Chen, M. Zhang, B. Cue, Improving generalisation of genetic programming for symbolic regression with structural risk minimisation, in: T. Friedrich (Ed.), GECCO '16: Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation, ACM, Denver, USA, 2016, pp. 709–716, http://dx.doi.org/10.1145/2908812.2908842.

[42] C. Raymond, Q. Chen, B. Xue, M. Zhang, Genetic programming with rademacher complexity for symbolic regression, in: C.A.C. Coello (Ed.), 2019 IEEE Congress on Evolutionary Computation, CEC 2019, IEEE Press, Wellington, New Zealand, 2019, pp. 2657–2664, http://dx.doi.org/10.1109/CEC.2019.8790341.

[43] R.M.A. Azad, C. Ryan, Variance based selection to improve test set performance in genetic programming, in: N. Krasnogor, P.L. Lanzi, A. Engelbrecht, D. Pelta, C. Gershenson, G. Squillero, A. Freitas, M. Ritchie, M. Preuss, C. Gagne, Y.S. Ong, G. Raidl, M. Gallager, J. Lozano, C. Coello-Coello,

D.L. Silva, N. Hansen, S. Meyer-Nieberg, J. Smith, G. Eiben, E. Bernado-Mansilla, W. Browne, L. Spector, T. Yu, J. Clune, G. Hornby, M.-L. Wong, P. Collet, S. Gustafson, J.-P. Watson, M. Sipper, S. Poulding, G. Ochoa, M. Schoenauer, C. Witt, A. Auger (Eds.), GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, ACM, Dublin, Ireland, 2011, pp. 1315–1322, http://dx.doi.org/10.1145/2001576.2001754.

[44] J. Ni, P. Rockett, Tikhonov regularization as a complexity measure in multiobjective genetic programming, IEEE Trans. Evol. Comput. 19 (2) (2014) 157–166.

[45] A.S. Sambo, R.M.A. Azad, Y. Kovalchuk, V.P. Indramohan, H. Shah, Leveraging asynchronous parallel computing to produce simple genetic programming computational models, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, in: SAC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 521–528, http://dx.doi.org/10.1145/3341105.3373921, https://doi.org/10.1145/3341105.3373921.

[46] F.F. de Vega, G. Olague, D. Lanza, W. Banzhaf, E. Goodman, J. Menendez-Clavijo, A. Martinez, et al., Time and individual duration in genetic programming, IEEE Access 8 (2020) 38692–38713.

[47] C. Simpson, J. Jewett, S. Turnbull, V. Stinner, PEP 418: Add monotonic time, performance counter, and process time functions, Website, https://www.python.org/dev/peps/pep-0418/.

[48] W.B. Langdon, Genetic Improvement of Genetic Programming, in: 2020 IEEE Congress on Evolutionary Computation (CEC), 2020, pp. 1–8, doi:10.1109/CEC48606.2020.9185771.

[49] J.R. Koza, D. Andre, Parallel Genetic Programming on a Network of Transputers, Technical Report CS-TR-95-1542, Stanford University, Department of Computer Science, 1995, http://www.genetic-programming.com/jkpdf/tr1542parallelsuversion.pdf.

[50] E.O. Scott, K.A. De Jong, Evaluation-time bias in asynchronous evolutionary algorithms, in: T. Tusar, B. Naujoks (Eds.), GECCO'15 Student Workshop, ACM, Madrid, Spain, 2015, pp. 1209–1212, http://dx.doi.org/10.1145/2739482.2768482.

[51] J. Kim, J. Kim, S. Yoo, GPGPGPU: Evaluation of parallelisation of genetic programming using GPGPU, in: T. Menzies, J. Petke (Eds.), Proceedings of the 9th International Symposium on Search Based Software Engineering, SSBSE 2017, in: LNCS, 10452, Springer, Paderborn, Germany, 2017, pp. 137–142, http://dx.doi.org/10.1007/978-3-319-66299-2_11.

[52] M. Oussaidène, B. Chopard, O.V. Pictet, M. Tomassini, Parallel genetic programming and its application to trading model induction, Parallel Comput. 23 (8) (1997) 1183–1198, http://dx.doi.org/10.1016/S0167-8191(97)00045-8.

[53] E.O. Scott, K.A. De Jong, Evaluation-time bias in quasi-generational and steady-state asynchronous evolutionary algorithms, in: T.F. et al (Ed.), GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, ACM, Denver, USA, 2016, pp. 845–852, http://dx.doi.org/10.1145/2908812.2908934.

[54] E. Cantú-Paz, A survey of parallel genetic algorithms, Calc. Paralleles Res. Syst. Repar. 10 (2) (1998) 141–171.

[55] D. Power, C. Ryan, R.M.A. Azad, Promoting diversity using migration strategies in distributed genetic algorithms, in: 2005 IEEE Congress on Evolutionary Computation, 2, IEEE Press, Edinburgh, Scotland, UK, 2005, pp. 1831–1838, http://dx.doi.org/10.1109/CEC.2005.1554910.

[56] G. Syswerda, A study of reproduction in generational and steady-state genetic algorithms, in: Foundations of Genetic Algorithms, 1, Elsevier, Amsterdam, 1991, pp. 94–101.

[57] D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaskowski, U.-M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals, Genet. Program. Evol. Mach. 14 (1) (2013) 3–29, http://dx.doi.org/10.1007/s10710-012-9177-2.

[58] D. Dua, E. Karra Taniskidou, UCI Machine Learning Repository, University of California, Irvine, School of Information and Computer Sciences, 2017, http://archive.ics.uci.edu/ml.

[59] S. Gustafson, E.K. Burke, N. Krasnogor, On improving genetic programming for symbolic regression, in: D.C. et al (Ed.), Proceedings of the 2005 IEEE Congress on Evolutionary Computation, 1, IEEE Press, Edinburgh, Scotland, UK, 2005, pp. 912–919.

[60] S. Luke, L. Panait, A comparison of bloat control methods for genetic programming, Evol. Comput. 14 (3) (2006) 309–344, http://dx.doi.org/10.1162/evco.2006.14.3.309.

[61] S. Luke, L. Panait, Fighting bloat with nonparametric parsimony pressure, in: J.J. Merelo-Guervos, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature - PPSN VII, in: Lecture Notes in Computer Science, LNCS, (2439) Springer-Verlag, Granada, Spain, 2002, pp. 411–421, http://dx.doi.org/10.1007/3-540-45712-7_40.

[62] S. Dignum, R. Poli, Operator equalisation and bloat free GP, in: M.O. et al (Ed.), Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, in: Lecture Notes in Computer Science, 4971, Springer, Naples, 2008, pp. 110–121, http://dx.doi.org/10.1007/978-3-540-78671-9_10.

[63] S. Silva, S. Dignum, L. Vanneschi, Operator equalisation for bloat free genetic programming and a survey of bloat control methods, Genet. Program. Evol. Mach. 13 (2) (2012) 197–238, http://dx.doi.org/10.1007/s10710-011-9150-5.

[64] M. Keijzer, Improving symbolic regression with interval arithmetic and linear scaling, in: European Conference on Genetic Programming, Springer, EuroGP, Essex, UK, 2003, pp. 70–82.

[65] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, A.Y. Wu, An efficient k-means clustering algorithm: analysis and implementation, IEEE Trans. Pattern Anal. Mach. Intell. 24 (7) (2002) 881–892.

[66] A.S. Sambo, R.M.A. Azad, Y. Kovalchuk, V.P. Indramohan, H. Shah, Time control or size control? Reducing complexity and improving accuracy of genetic programming models, in: European Conference on Genetic Programming (Part of EvoStar), Springer, 2020, pp. 195–210, http://dx.doi.org/10.1007/978-3-030-44094-7_13.

Time Control or Size Control? Reducing Complexity and Improving Accuracy of Genetic Programming Models

Aliyu Sani Sambo[1], R. Muhammad Atif Azad[1], Yevgeniya Kovalchuk[1], Vivek Padmanaabhan Indramohan[2], and Hanifa Shah[3]

[1] School of Computing and Digital Technology, Birmingham City University, Birmingham, UK
`aliyu.sambo@mail.bcu.ac.uk`,
`{atif.azad,yevgeniya.kovalchuk}@bcu.ac.uk`
[2] School of Health Science, Birmingham City University, Birmingham, UK
`vivek.indramohan@bcu.ac.uk`
[3] Faculty of Computing, Engineering and the Built Environment, Birmingham City University, Birmingham, UK
`hanifa.shah@bcu.ac.uk`

Abstract. Complexity of evolving models in genetic programming (GP) can impact both the quality of the models and the evolutionary search. While previous studies have proposed several notions of GP model complexity, the size of a GP model is by far the most researched measure of model complexity. However, previous studies have also shown that controlling the size does not automatically improve the accuracy of GP models, especially the accuracy on out of sample (test) data. Furthermore, size does not represent the functional composition of a model, which is often related to its accuracy on test data. In this study, we explore the *evaluation time* of GP models as a measure of their complexity; we define the evaluation time as the time taken to evaluate a model over some data. We demonstrate that the evaluation time reflects both a model's size and its composition; also, we show how to measure the evaluation time reliably. To validate our proposal, we leverage four well-known methods to size-control but to control evaluation times instead of the tree sizes; we thus compare size-control with *time-control*. The results show that time-control with a nuanced notion of complexity produces more accurate models on 17 out of 20 problem scenarios. Even when the models have slightly greater times and sizes, time-control counterbalances via superior accuracy on both training and test data. The paper also argues that time-control can differentiate functional complexity even better in an identically-sized population. To facilitate this, the paper proposes Fixed Length Initialisation (FLI) that creates an identically-sized but functionally-diverse population. The results show that while FLI particularly suits time-control, it also generally improves the performance of size-control. Overall, the paper poses evaluation-time as a viable alternative to tree sizes to measure complexity in GP.

T. Hu et al. (Eds.): EuroGP 2020, LNCS 12101, pp. 195–210, 2020.
https://doi.org/10.1007/978-3-030-44094-7_13

Keywords: Genetic Programming · Complexity · Evaluation time

1 Introduction

Motivations for controlling the complexity of machine learning (ML) models vary and so does the notion of complexity [3]. One reason for managing the complexity of ML models is to attain models that are only complex enough to explain the phenomenon generating the given data but not too complex to reflect noise in the data. Doing so means that the predictions produced by the models on previously unseen data are accurate [18]; in other words, the model *generalises* well. However, the challenge in this goal is determining when the complexity is just enough. Another incentive for managing complexity is the requirement for models to use computational resources efficiently. For example, some computational environments such as the *Internet of Things* (IoT) devices constrain the evaluation time of an acceptable model even if this compromises its accuracy [13]. In Genetic Programming (GP), preventing the models from growing too complex is also necessary to prevent the evolutionary search from becoming ineffective [11]. A further motivation for managing complexity is the demand for interpretable models: simple models can be more interpretable [14], and the interpretability of ML models is now important. For example, the EU General Data Protection Regulation (GDPR) stipulates a *right to explanation* where ML algorithms are applied to make a decision affecting a person.

The challenge of defining a notion of complexity is compounded in the context of Genetic programming (GP). For example, while ridge regression penalises the growth in the magnitude of numeric coefficients in an otherwise fixed regression model, this penalty does not necessarily work in GP because GP evolves the model itself. Moreover, GP is a versatile tool that can also evolve compilable programs; therefore, minimising the coefficients does not automatically make sense. Also, since during evolution the GP models grow in size, controlling this growth (bloat control) has dominated the landscape of complexity control in GP. However, some previous work [21] shows that controlling the size alone does not automatically produce models that generalise as might have been expected. Moreover, [2] shows that size does not indicate functional composition (or complexity): after all, a very large GP tree may compose a simple linear function; likewise, a small GP tree can compose a highly non-linear function. Together the above challenges show that universally defining complexity is difficult.

This paper uses the *evaluation time* – the computational time required to evaluate a model on the given data – to indicate its complexity. Due to different functional and syntactic compositions of models in the evolving populations, the evaluation time of the models varies. For example, the models made up of computationally expensive functions or exceptionally large syntactic structures take long to evaluate. Unlike size, the evaluation time thus indicates both the syntactic and functional complexity; Sect. 2.2 expands further on that. However, since evaluation times vary from one measurement to another, Sect. 2.3 shows how to measure them reliably.

To control evaluation times, we use four well-known techniques for bloat-control to control evaluation time. However, instead of controlling size, we control evaluation times using the same mechanisms; the techniques thus effect *time-control*. We then compare the effect of time-control with that of size (or bloat) control on the composition, size and accuracy of the evolving models.

The results of our experiments suggest that time-control with a nuanced notion of complexity outperforms size-control in model-accuracy on 17 out of 20 problem scenarios. Even when time-control produces models with slightly greater times and sizes, it counterbalances via superior accuracy on both training and test data. The paper also shows that time-control can differentiate functional complexity even better in an identically-sized population. To facilitate this, the paper proposes Fixed Length Initialisation (FLI) that creates an identically-sized but functionally-diverse population. The results show that while FLI particularly suits time-control, it also generally improves the performance of size-control.

Following this introductory part, Sect. 2 of this paper provides some background; Sect. 3 details the experiments; Sect. 4 presents the results; and Sect. 5 covers future works and concludes the paper.

2 Background

2.1 Complexity in Genetic Programming

Traditionally, controlling complexity in GP means controlling structural complexity such as the size (bloat control) of the evolved expressions, or the number of encapsulated sub-trees and layers, while ignoring the underlying functional or computational complexity [6,7,9,17,21]. For example, bloat control penalises a large yet linear expression $4x+8x+2x+x+x$, which is functionally and computationally less complex than a smaller expression $sin(x)$ [2], which is equivalent to its Taylor series expansion $\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{(2n+1)!}$. Clearly, the smaller expression $sin(x)$ needs more computational resources than its linear counterpart. Thus, complexity in GP is more than merely the expression size.

Approaches based on functional complexity recognise that small structures may be more complex than larger ones and hence focus on the functionality of structures. To elicit functional complexity, one approach approximates the evolving expressions by polynomials [23]; complex expressions are approximated by polynomials of a high degree owing to large oscillations in the response of the function. This degree of approximating polynomials is thus minimised in [23]. However, the minimisation requires the evolving expressions to be twice differentiable, a property that is not always guaranteed. To alleviate this constraint, Vanneschi et al. [21] defined a less rigorous measure of functional complexity, whereby the slope of an expression is approximated by a simpler but error prone measure. As such Vanneschi et al. did not control the complexity; instead, they only measured the complexity of evolving expressions. Another approach [1] used the variance of the outputs of the evolving expressions to measure the functional complexity; this approach explicitly minimised the variance and maximised accuracy using a multi-objective optimisation approach. Note, however, that slope of the evolving functions can not indicate complexity when evolving compilable programs for tasks such as robot navigation.

Similarly, statistical learning theory measures the complexity of a space of functions that can be learned using statistical classification. The main techniques include generalisation error bound VC theory and VC dimensions [12,22].

As the discussion highlights, the above techniques are either specialised to various domains or challenging to implement. In contrast, the present study simply measures the complexity of a model with its evaluation time.

2.2 Evaluating Time Is More Than Measuring Size

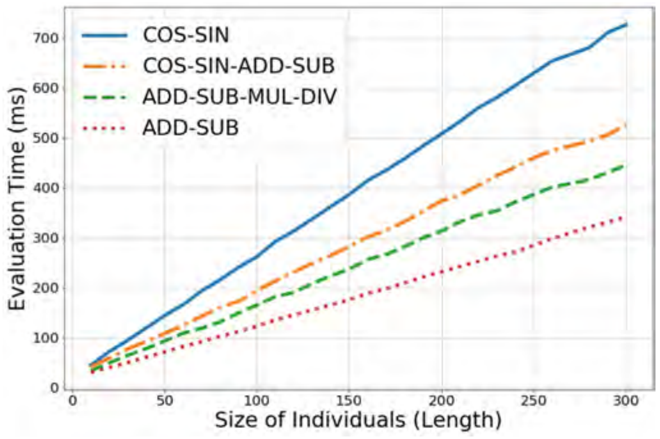


Fig. 1. Relationship between evaluation time, size and the composition of models is shown. Individuals made up of COS and SIN operators have higher average evaluation times than the same-sized individuals from other functions sets. Also, note that size correlates with evaluation time.

While the previous section argues why measuring size is fundamentally different to evaluating time, it is also important to empirically verify that. After all, the evaluation time also increases when the expression size increases; however, we must also ascertain if the evaluation time also practically increases with the functional complexity. Otherwise, measuring time becomes simply a proxy for measuring size. Clearly, that is undesirable.

To this end, we used four different *functions sets* to generate symbolic regression models of different complexities; Fig. 1 details the functions sets. For each functions set, we generated differently sized individuals (10, 20, 30, ..., 300), and in turn for each size we generated 30 random expressions. All the models were then evaluated 50 times, each with the same data. Figure 1 presents the average evaluation times of individuals according to their size and complexity.

Two trends are clearly visible in Fig. 1: (1) given the same size, the evaluation times of functionally complex individuals are consistently higher than those for their counterparts; and (2) evaluation times are also strongly correlated with the expression sizes, as expected. Hence, the evaluation times indeed differentiate between functional complexities; however, if a simple function is inefficiently coded as an excessively large expression, it evaluates slower. Therefore, evaluation time control impacts conditionally: it prefers functional simplicity if the sizes of a competing set of individuals are within a certain tolerance (or range); otherwise, it prefers smaller sizes. Note, this tolerance increases as the size of individuals increases. For example, the evaluation time of size 75 with functions set COS-SIN is the same as that for size 175 with the functions set ADD-SUB.

The above findings also predict the limiting behaviour of evaluation time control in GP. In a functionally diverse but a size-converged population – where bloat control is impotent – evaluation times discriminate between functional complexities, whereas in a functionally converged but a size-diverse population, evaluation times discriminate between sizes.

The idea that time control discriminates between functional complexities when sizes have converged prompted us to try a new initialisation scheme. The new scheme starts with a population of identically sized but functionally diverse individuals. We tested the impact of this new initialisation on all methods before applying it to our experiments. Section 3.4 details this scheme and its impact.

2.3 Stabilising Evaluation Time Measurements

A problem with measuring evaluation times is that they vary across multiple executions, and if this variability is high, one cannot reliably estimate the complexity of a given model from a single evaluation. Since this variation results from CPU scheduling that is under the control of the operating system, we can not eliminate this variation totally. However, we found ways to significantly minimise this variation across evaluations.

We found that CPU management options can help minimise this variation. These options include: (1) stopping all background services, (2) locking the CPU speed to prevent the operating system power management from interfering, (3) executing the experiments on dedicated processors and (4) assigning the experimental tasks a high priority. Figure 2 illustrates the impact of these changes. Each box-plot represents multiple evaluation times for an individual of a given size. Clearly the variation decreases significantly after CPU management. Thus, we were able to use a single evaluation to measure the evaluation time.

3 Experiments

We used four existing bloat control techniques to compare size-control with time-control. When controlling time, the evaluation time replaces size in each of the bloat control techniques.

3.1 Bloat Control Techniques

(a) Death by Size (DS) [16] is a steady state replacement method that replaces the larger individuals from the present population with a given probability

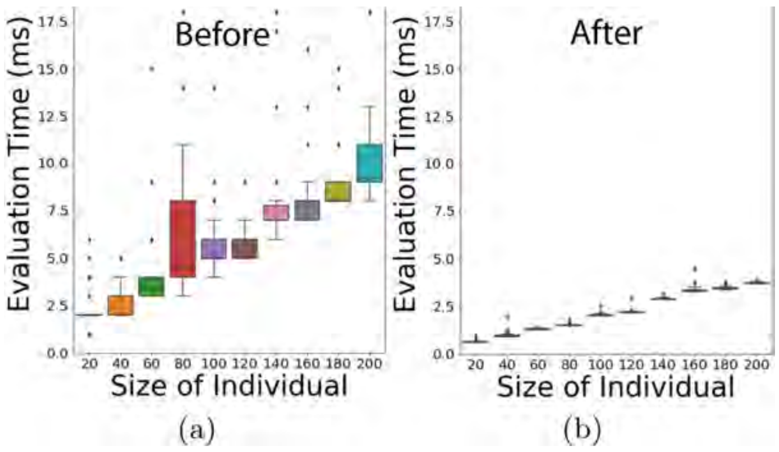


Fig. 2. Using CPU management options decreases variability in evaluation times.

(typically 0.7; we use the same). To replace an individual, DS selects two individuals randomly and replaces the larger one probabilistically. By necessity, DS uses steady-state GP.

(b) Double Tournament (DT) [15,16] increases the probability of choosing smaller individuals as parents to encourage the reproduction of similarly small offspring. DT runs two rounds of tournaments. In the first round, it runs n probabilistic tournaments each with a tournament of size 2 to select a set of n individuals. Each of these tournaments selects the smaller individual with a probability of 0.7. Then, in the second round, DT selects the fittest out of the n individuals. We implemented the DT experiments using steady-state GP.

(c) Operator Equalisation (OpEq) [4,20] allows the sizes of individuals to grow only when fitness is improving. It controls the distribution of the population by employing two core functions. The first determines the target distribution (by size) of individuals in the next generation; the second ensures that the next generation matches the target distribution. To define the target distribution, OpEq puts the current individuals into bins according to their sizes and calculates the average fitness score of each bin. This average score is then used to calculate the number of individuals to be allowed in a corresponding bin in the next generation (target distribution). Thus, from one generation to the next the target distribution changes to favour the sizes that produce fit individuals. The width of the bins can vary and thus is a parameter. Bin width of 1 to 10 has been successfully used previously [20]. In our experiments we used the better performing Dynamic OpEq variant [20] and used *bin width = 5*. Note, OpEq uses generational replacement.

To adapt OpEq to control evaluation time, we had to estimate the time equivalent of the bin width. This value is then used to create bins to classify individuals according their evaluation times in the same way as bin width size is used to create bins to classify individuals by their sizes. To get a reliable estimate we used multiple samples, evaluated multiple times and used the median of several estimates. This is done only once at the beginning of the GP run.

(d) The Tarpeian (TP) [19] method controls size-growth by assigning the worst fitness to a fraction W (recommended $W = 0.3$; we use the same) of the individuals that have above-average size. TP uses generational replacement and calculates the average size of the population at every generation.

To adapt this method to control evaluation time we simply replaced the average size with the average evaluation time.

3.2 Test Problems

We use five tough problems to compare the results in this paper. The problems are tough because the results in Sect. 4 show that the accuracy scores are low (less than 41%). Hence, these problems require GP to run long and thus present a good test bed for complexity control because at least the size-complexity in GP grows with long runs. Four of these problems are multi-dimensional (with five or more input variables). The data set for problems 1–4 are available at [5]; Problem 5 is a bi-variate version of the function used in [8]. A summary of the data sets is available in Table 1.

Table 1. Overview of test problems

ID	Problem label	No. of variables	No. of instances
1	Airfoil	5	1503
2	Boston housing	13	506
3	Concrete strength	8	1030
4	Energy efficiency	8	768
5	$y^2x^6 - 2.13y^4x^4 + y^6x^2$	2	250 (x = min: −0.3, step: 0.012; y = x + 0.03)

3.3 Configuration and Parameters

The basic parameters for all the methods are summarised in Table 2. The other key experimental decisions are as follows. First, the individuals with divide-by-zero errors were assigned the worst fitness; as discussed in [10], the protected operators commonly used in GP lead to poor generalisation. Next, the data-sets were randomly split (without replacement) into 80% for training and 20% testing. Finally, the fitness was computed as the normalised mean squared error (MSE) and maximised as follows: $\frac{1}{1+\frac{1}{n}\Sigma_{i=1}^{n}(y_i-\hat{y_i})^2}$.

The experiments were run on Windows 10 (64-bit) with 32 GB RAM, and Intel Core i7-6700 CPU @ 3.40 GHz (Quad-Core).

3.4 Initialising the Population

Section 2.2 motivated the need for an initialisation scheme that produces functionally diverse but identically sized individuals; such a scheme can increase

Table 2. Summary of Parameters

Parameter	Setting
Number of runs	50
Population size	500
Run terminates	After 35,000 evaluations ($\equiv$ 70 generations)
Random tree/subtree generation	Ramped half-and-half(1 =< *depth* =< 4); and Fixed Length Initialisation (see Sect. 3.4)
Operators & probabilities	One point crossover = 0.9; Point mutation = 0.1
Depth Limit	17
Function set	$+, -, *, /, \sin, \cos$, neg
Constants (ERC)	$\lvert ERC \rvert$ = 100 (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament size = 3
Replacement	steady state/generational as per each method

the focus of the time-control on differentiating functional complexity. Therefore, we created a *Fixed-Length Initialisation scheme (FLI)* for these experiments. Henceforth, we call the Ramped-Half-and-Half initialisation the Variable Length Initialisation (VLI).

For the present study, we used the FLI to produce an initial population of unique individuals each having the same length (or size) of 10 nodes. Given the functions set size, a fixed length of 10 can easily produce populations of a few hundred unique individuals; we leave studying the impact of varying the lengths to future work. To encourage functional diversity, we do not consider two individuals different if they only differ by numeric constants.

Before applying FLI to our experiments, we examined its impact on all the methods. The charts in Fig. 3 show the mean test fitness accuracy by generation for all the methods and problems. The significance of the differences of the final populations as established by the Mann-Whitney U test are captured in Fig. 4. The figure is colour coded so that green indicates where the accuracy of the final populations produced by FLI are significantly higher, brown where VLI is higher, and yellow where the difference is not significant. FLI produced better results in 16 out of 20 for Time-control and 11 out of 20 for size-control.

We observed that when using OpEq, size-control with VLI was better than size-control with FLI on all the problems. Therefore, for OpEq we compare time-control with the result of size-control with VLI (the better result). For all other methods we used the proposed FLI.

4 Results

We compare the accuracy, complexity and compositions of the models produced by each method to controlling size and time. For accuracy, although our key measure is test fitness (accuracy on out-of-sample data), we also report training fitness; the higher the value the better. For complexity we report both the size and evaluation times of the models; the lower the values the better. Finally, to give further insight into the complexity of the evolved models, we report the composition of final populations as to what percentage of the genetic material comprised of more or less complex mathematical functions.

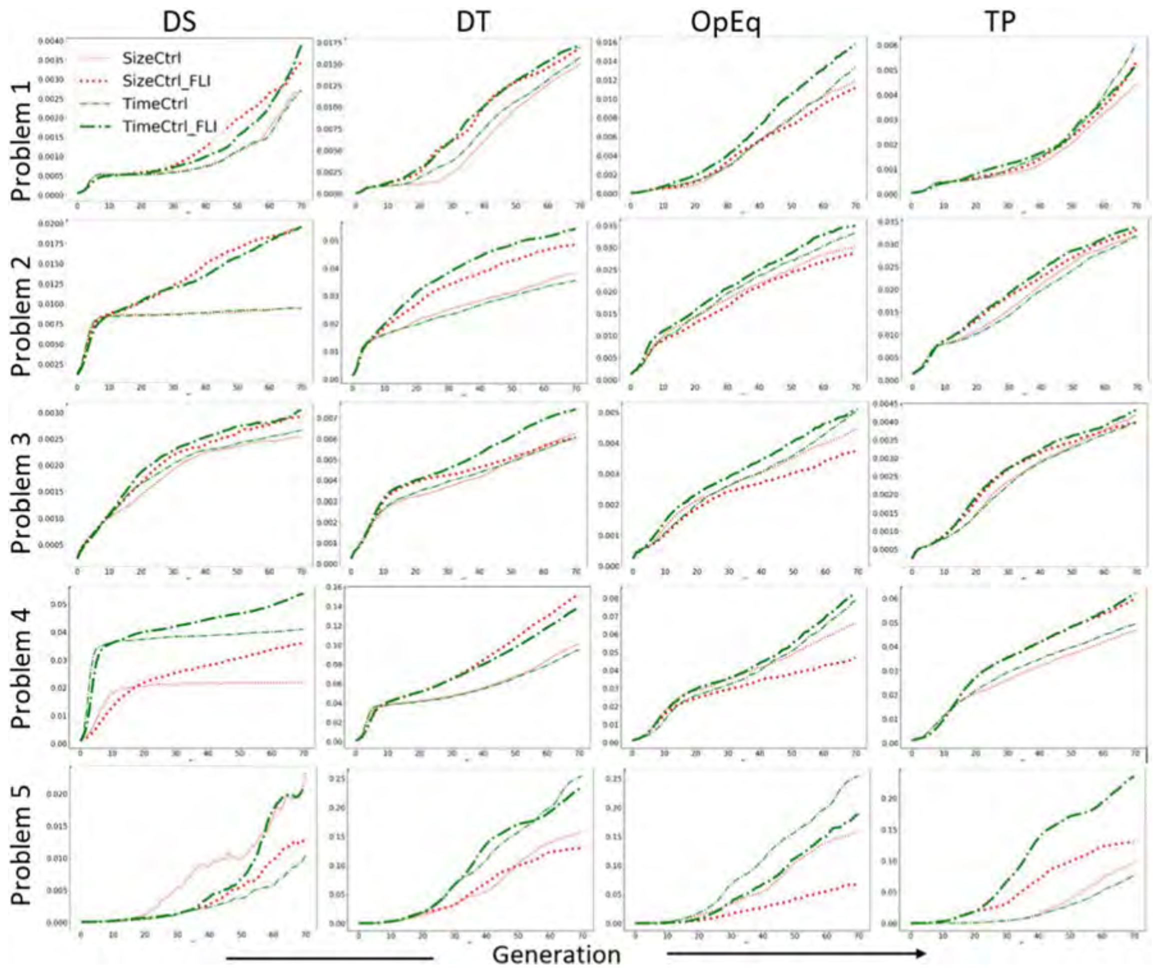
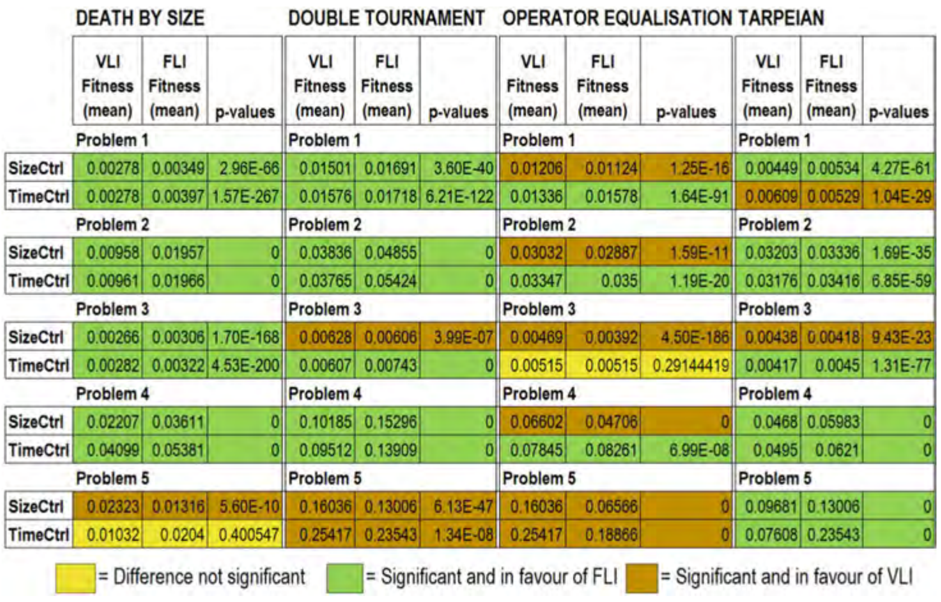


Fig. 3. Comparing the test fitness of initialisation schemes, VLI and FLI. The mean test fitness values are plotted by generation. The thick lines represent FLI and the thin VLI; the green and red lines represent time-control and size-control respectively. (Color figure online)

Figures 6, 7, 8 and 9 show how the test set accuracy, size and evaluation times of both time-control and size-control evolve with each of DS, DT, OpEq and TP. The figures show that for all the methods the values of all the measures increase continuously through to the final generations. Therefore, we evaluate the statistical significance of the differences in the performances in the final generations and report it in Fig. 5. Also, unless stated otherwise, henceforth, the discussion of results concerns Fig. 5.

Statistical Significance: Figure 5 shows the colour-coded results of the Mann-Whitney U statistical test comparing the final populations of time-control and size-control. The table contains results for all the test problems and techniques. The attributes tested include the evaluation time, size, training and test fitness (accuracy on out-of-sample data). The p-values included in Fig. 5 statistically compare the metrics of time-control against those of size-control. The rows are green when time-control is significantly better (more for accuracy, and less for both size and evaluation time), brown when it is significantly worse, and yellow when the difference is not significant.



	DEATH BY SIZE			DOUBLE TOURNAMENT			OPERATOR EQUALISATION			TARPEIAN		
	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values	VLI Fitness (mean)	FLI Fitness (mean)	p-values
	Problem 1			Problem 1			Problem 1			Problem 1		
SizeCtrl	0.00278	0.00349	2.96E-66	0.01501	0.01691	3.60E-40	0.01206	0.01124	1.25E-16	0.00449	0.00534	4.27E-61
TimeCtrl	0.00278	0.00397	1.57E-267	0.01576	0.01718	6.21E-122	0.01336	0.01578	1.64E-91	0.00609	0.00529	1.04E-29
	Problem 2			Problem 2			Problem 2			Problem 2		
SizeCtrl	0.00958	0.01957	0	0.03836	0.04855	0	0.03032	0.02887	1.59E-11	0.03203	0.03336	1.69E-35
TimeCtrl	0.00961	0.01966	0	0.03765	0.05424	0	0.03347	0.035	1.19E-20	0.03176	0.03416	6.85E-59
	Problem 3			Problem 3			Problem 3			Problem 3		
SizeCtrl	0.00266	0.00306	1.70E-168	0.00628	0.00606	3.99E-07	0.00469	0.00392	4.50E-186	0.00438	0.00418	9.43E-23
TimeCtrl	0.00282	0.00322	4.53E-200	0.00607	0.00743	0	0.00515	0.00515	0.29144419	0.00417	0.0045	1.31E-77
	Problem 4			Problem 4			Problem 4			Problem 4		
SizeCtrl	0.02207	0.03611	0	0.10185	0.15296	0	0.06602	0.04706	0	0.0468	0.05983	0
TimeCtrl	0.04099	0.05381	0	0.09512	0.13909	0	0.07845	0.08261	6.99E-08	0.0495	0.0621	0
	Problem 5			Problem 5			Problem 5			Problem 5		
SizeCtrl	0.02323	0.01316	5.60E-10	0.16036	0.13006	6.13E-47	0.16036	0.06566	0	0.09681	0.13006	0
TimeCtrl	0.01032	0.0204	0.400547	0.25417	0.23543	1.34E-08	0.25417	0.18866	0	0.07608	0.23543	0

🟨 = Difference not significant 🟩 = Significant and in favour of FLI 🟧 = Significant and in favour of VLI

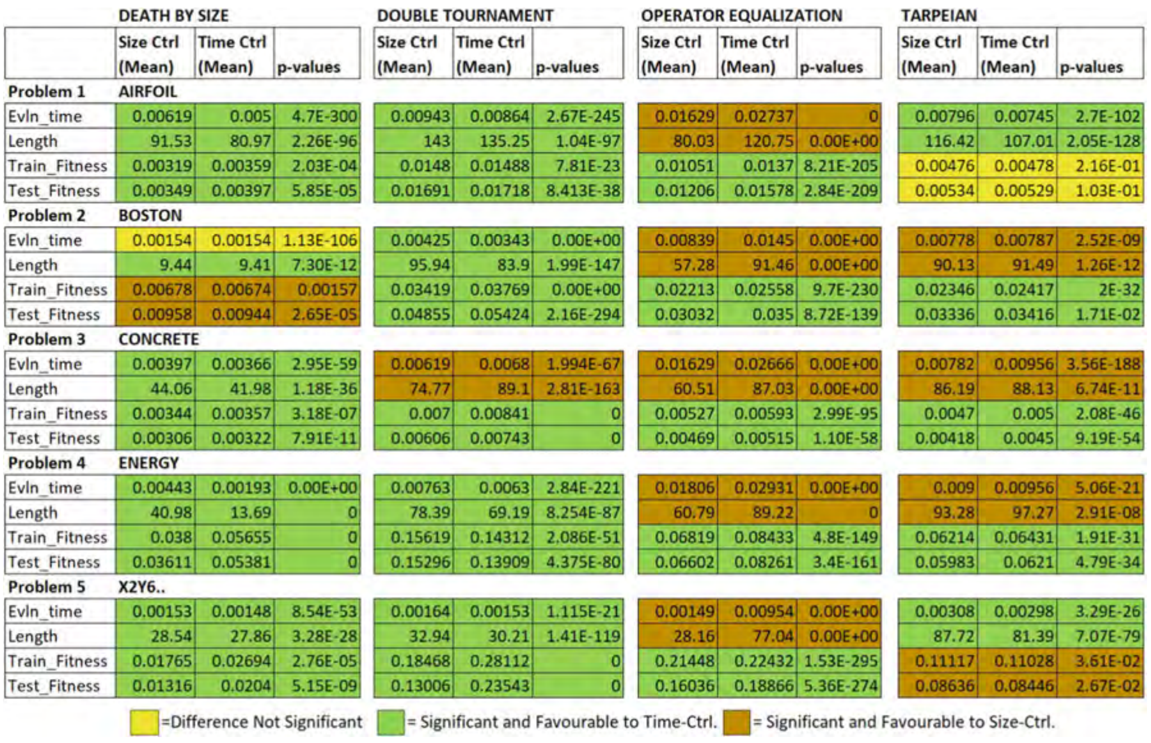
Fig. 4. Testing the significance of the impact of the new FLI initialisation scheme. In the final populations, FLI test fitness accuracy improved 16 of 20 for time-control and 11 of 20 for size-control. (Color figure online)

Accuracy of Models: Time-control produced significantly more accurate models (on both training and test data) across all problems and all control techniques except on three occasions. The exceptions are problem 1 on TP (the difference is not significant), and problem 2 on DS and problem 5 on TP where size-control outperformed time-control. Overall, time-control outperformed size-control on training and test accuracy on 17 of the 20 occasions and matched size-control on one occasion.

Complexity of Models: Time-control produced less complex (evaluation time and size) models with 2 out of the 4 control techniques; the techniques are DS and DT. As seen in Fig. 5, DS produced simpler models on all the problems except on problem 2 where the difference in evaluation times is not significant.

Likewise, DT produced simpler models on 4 out of 5 problems, the exception being problem 3.

Composition of Models: Table 3 counts and differentiates the nature of nodes constituting the GP trees in the final populations to understand the composition of the genetic material therein. Consistent with the results on evaluation times and sizes, time-control with DS and DT used smaller percentages of complex mathematical functions: the percentages of tree nodes containing SIN and COS with time-control are smaller than the respective figures for size-control. Likewise, OpEq and TP – much like their results on evaluation times and sizes – use greater percentages of SIN and COS.



	DEATH BY SIZE			DOUBLE TOURNAMENT			OPERATOR EQUALIZATION			TARPEIAN		
	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values	Size Ctrl (Mean)	Time Ctrl (Mean)	p-values
Problem 1	**AIRFOIL**											
Evln_time	0.00619	0.005	4.7E-300	0.00943	0.00864	2.67E-245	0.01629	0.02737	0	0.00796	0.00745	2.7E-102
Length	91.53	80.97	2.26E-96	143	135.25	1.04E-97	80.03	120.75	0.00E+00	116.42	107.01	2.05E-128
Train_Fitness	0.00319	0.00359	2.03E-04	0.0148	0.01488	7.81E-23	0.01051	0.0137	8.21E-205	0.00476	0.00478	2.16E-01
Test_Fitness	0.00349	0.00397	5.85E-05	0.01691	0.01718	8.413E-38	0.01206	0.01578	2.84E-209	0.00534	0.00529	1.03E-01
Problem 2	**BOSTON**											
Evln_time	0.00154	0.00154	1.13E-106	0.00425	0.00343	0.00E+00	0.00839	0.0145	0.00E+00	0.00778	0.00787	2.52E-09
Length	9.44	9.41	7.30E-12	95.94	83.9	1.99E-147	57.28	91.46	0.00E+00	90.13	91.49	1.26E-12
Train_Fitness	0.00678	0.00674	0.00157	0.03419	0.03769	0.00E+00	0.02213	0.02558	9.7E-230	0.02346	0.02417	2E-32
Test_Fitness	0.00958	0.00944	2.65E-05	0.04855	0.05424	2.16E-294	0.03032	0.035	8.72E-139	0.03336	0.03416	1.71E-02
Problem 3	**CONCRETE**											
Evln_time	0.00397	0.00366	2.95E-59	0.00619	0.0068	1.994E-67	0.01629	0.02666	0.00E+00	0.00782	0.00956	3.56E-188
Length	44.06	41.98	1.18E-36	74.77	89.1	2.81E-163	60.51	87.03	0.00E+00	86.19	88.13	6.74E-11
Train_Fitness	0.00344	0.00357	3.18E-07	0.007	0.00841	0	0.00527	0.00593	2.99E-95	0.0047	0.005	2.08E-46
Test_Fitness	0.00306	0.00322	7.91E-11	0.00606	0.00743	0	0.00469	0.00515	1.10E-58	0.00418	0.0045	9.19E-54
Problem 4	**ENERGY**											
Evln_time	0.00443	0.00193	0.00E+00	0.00763	0.0063	2.84E-221	0.01806	0.02931	0.00E+00	0.009	0.00956	5.06E-21
Length	40.98	13.69	0	78.39	69.19	8.254E-87	60.79	89.22	0	93.28	97.27	2.91E-08
Train_Fitness	0.038	0.05655	0	0.15619	0.14312	2.086E-51	0.06819	0.08433	4.8E-149	0.06214	0.06431	1.91E-31
Test_Fitness	0.03611	0.05381	0	0.15296	0.13909	4.375E-80	0.06602	0.08261	3.4E-161	0.05983	0.0621	4.79E-34
Problem 5	**X2Y6..**											
Evln_time	0.00153	0.00148	8.54E-53	0.00164	0.00153	1.115E-21	0.00149	0.00954	0.00E+00	0.00308	0.00298	3.29E-26
Length	28.54	27.86	3.28E-28	32.94	30.21	1.41E-119	28.16	77.04	0.00E+00	87.72	81.39	7.07E-79
Train_Fitness	0.01765	0.02694	2.76E-05	0.18468	0.28112	0	0.21448	0.22432	1.53E-295	0.11117	0.11028	3.61E-02
Test_Fitness	0.01316	0.0204	5.15E-09	0.13006	0.23543	0	0.16036	0.18866	5.36E-274	0.08636	0.08446	2.67E-02

🟨 =Difference Not Significant 🟩 = Significant and Favourable to Time-Ctrl. 🟧 = Significant and Favourable to Size-Ctrl.

Fig. 5. Results of Mann-WhitneyU test for significance in the differences between the final populations of time-control and size-control. Time-control produced more accurate training and test scores in 17 out of 20 tests. While time-control with the steady-state methods (DS and DT) produced simpler (smaller sizes and evaluation times) models than size-control in 9 out of 10 tests, time-control with the generational methods (OpEq and TP) produced more complex models in 8 out of 10 tests. (Color figure online)

4.1 Discussion

Section 1 argued that sensible management of complexity should produce models that are only complex enough to explain the phenomenon generating the given data but not too complex. The results show that time control almost consistently delivers superior accuracy despite splitting results on complexity measures. Even so, the increased complexity with time-control with OpEq and TP is not off the scale as is typically the case with the standard, unrestrained GP.

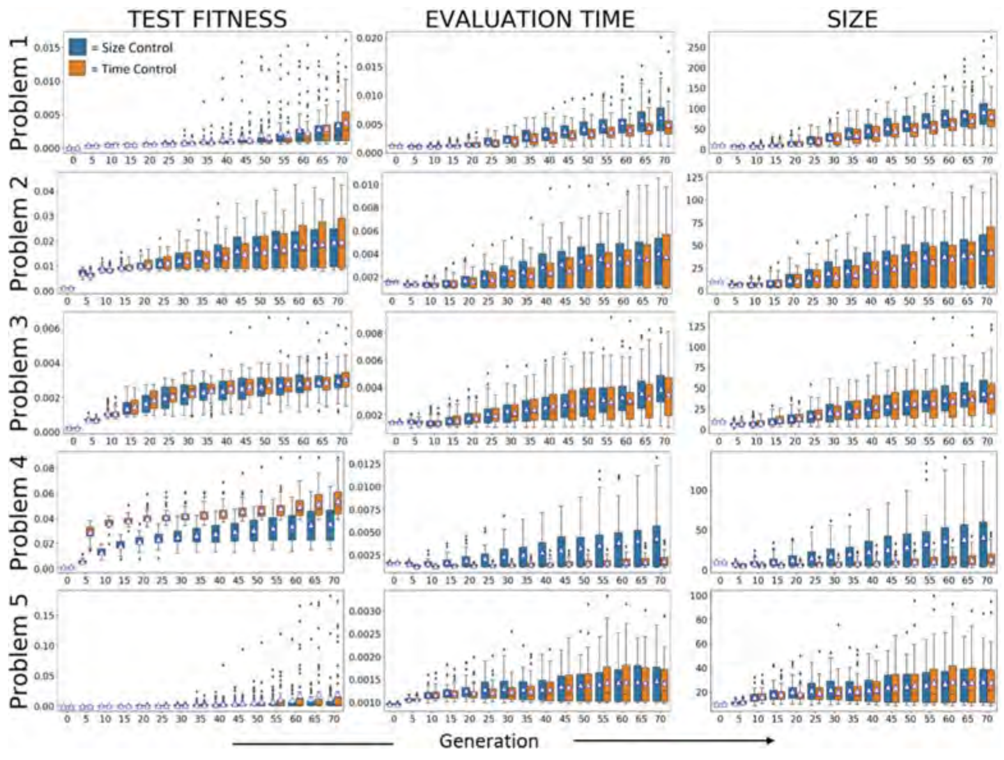


Fig. 6. Death By Size: Comparing changes in metrics by generation between time-control and size-control using DS.

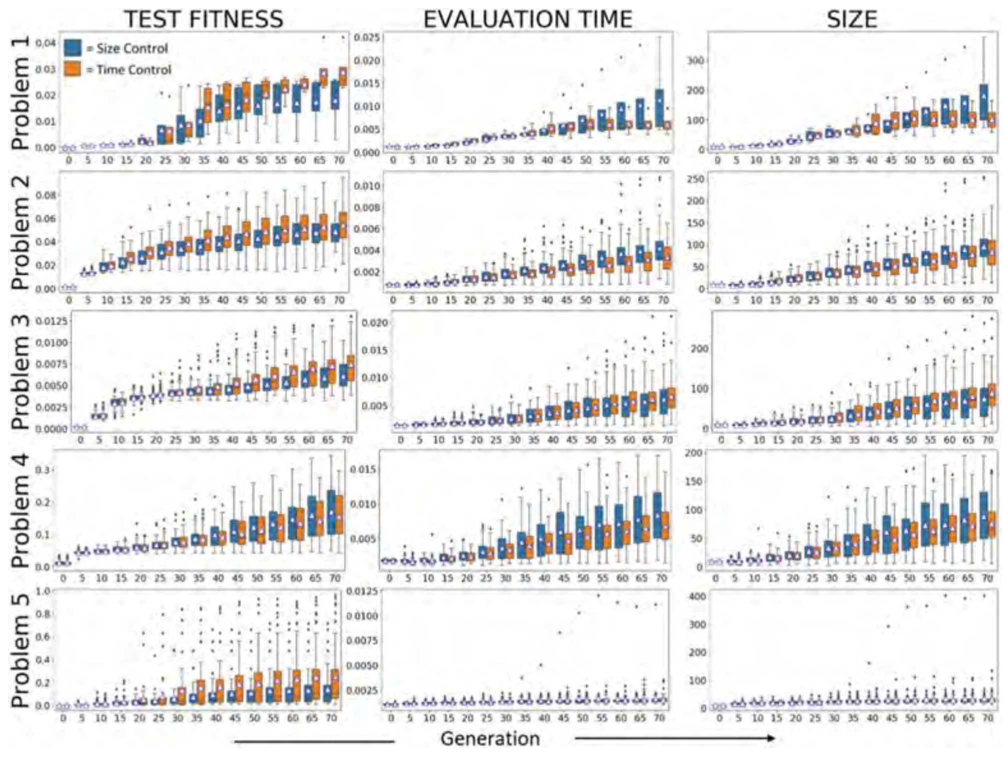


Fig. 7. Double Tournament: Comparing changes in metrics by generation between time-control and size-control using DT.

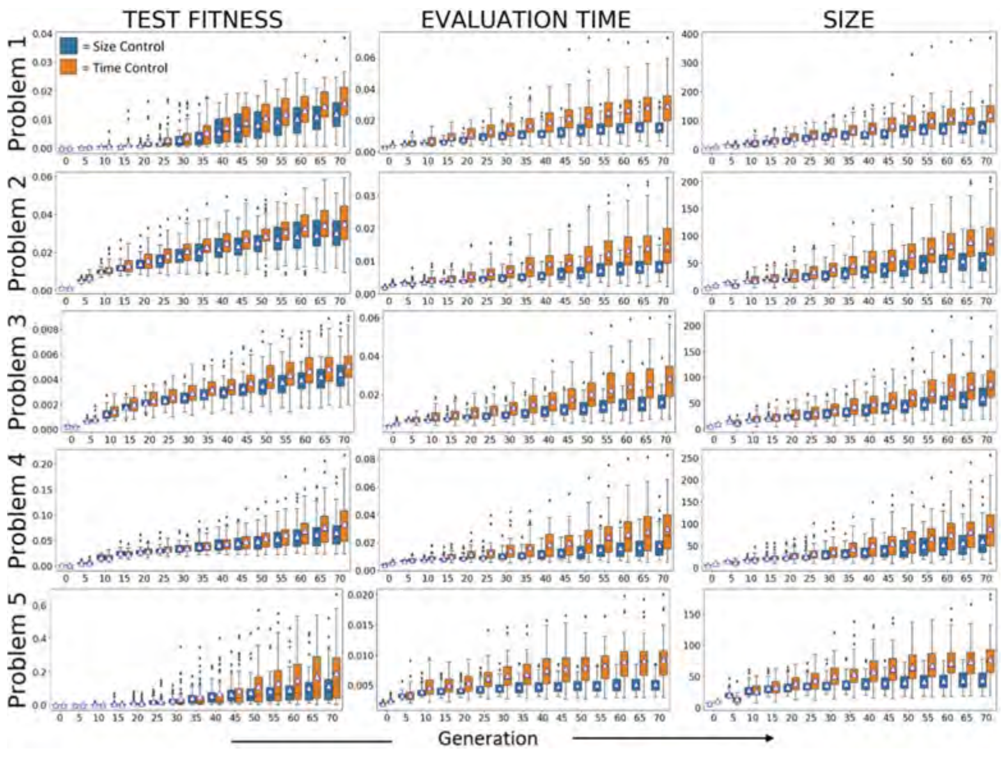


Fig. 8. Operator Equalisation: Comparing changes in metrics by generation between time-control and size-control using OpEq.

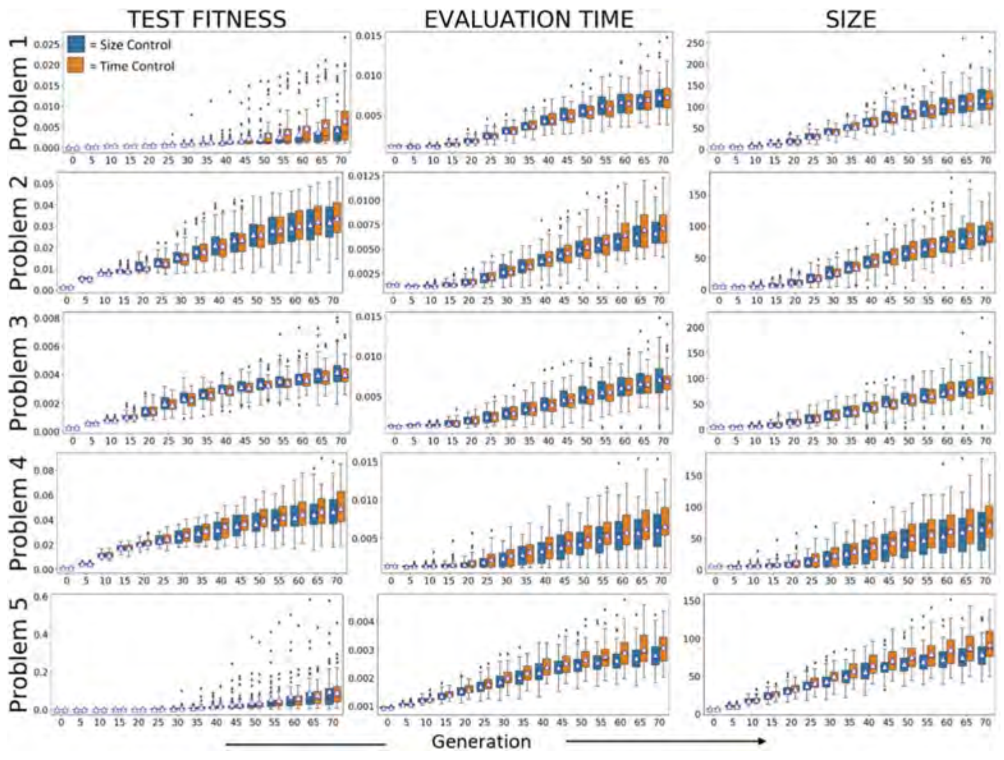
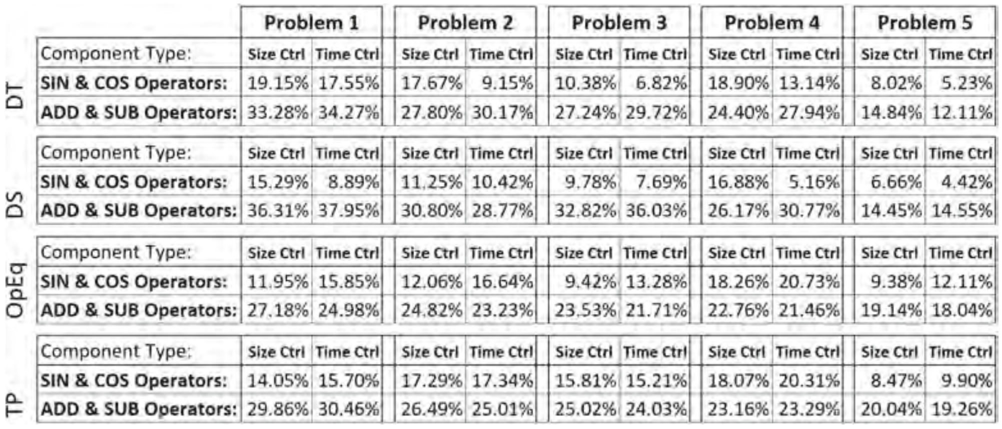


Fig. 9. Tarpeian: Comparing changes in metrics by generation between time-control and size-control using TP.

Table 3. Composition of the final populations.



		Problem 1		Problem 2		Problem 3		Problem 4		Problem 5	
DT	Component Type:	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS Operators:	19.15%	17.55%	17.67%	9.15%	10.38%	6.82%	18.90%	13.14%	8.02%	5.23%
	ADD & SUB Operators:	33.28%	34.27%	27.80%	30.17%	27.24%	29.72%	24.40%	27.94%	14.84%	12.11%
DS	Component Type:	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS Operators:	15.29%	8.89%	11.25%	10.42%	9.78%	7.69%	16.88%	5.16%	6.66%	4.42%
	ADD & SUB Operators:	36.31%	37.95%	30.80%	28.77%	32.82%	36.03%	26.17%	30.77%	14.45%	14.55%
OpEq	Component Type:	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS Operators:	11.95%	15.85%	12.06%	16.64%	9.42%	13.28%	18.26%	20.73%	9.38%	12.11%
	ADD & SUB Operators:	27.18%	24.98%	24.82%	23.23%	23.53%	21.71%	22.76%	21.46%	19.14%	18.04%
TP	Component Type:	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl	Size Ctrl	Time Ctrl
	SIN & COS Operators:	14.05%	15.70%	17.29%	17.34%	15.81%	15.21%	18.07%	20.31%	8.47%	9.90%
	ADD & SUB Operators:	29.86%	30.46%	26.49%	25.01%	25.02%	24.03%	23.16%	23.29%	20.04%	19.26%

As to why time-control with OpEq and TP produces greater complexity is not exactly clear at present; however, it is worth noting that these two methods require generational replacement where the size (or time) distributions of the entire generations must be computed before allowing new individuals in. In contrast, DT and DS are steady state methods where a new individual replaces the loser of a tournament.

Interestingly, Fixed Length Initialisation (FLI) improved the results with not only time control but more often than not even with size control. The results encourage further investigation into this initialisation technique. FLI is designed to promote compositional (functional) diversity and thus allow time-control to distinguish complexity based more on composition than on size. However, FLI can not enforce size similarity beyond the initial generation; therefore, further work must investigate the effects of promoting size similarity in the remaining evolution and see if that further intensifies the effect of time-control.

5 Conclusions and Future Work

This paper asks the question - why not use time instead of size to measure complexity in GP? Unlike model size, evaluation time is a function of both syntactic and computational characteristics of a model. This measure is broadly applicable, and although this paper studies regression problems, in principle, evaluation time can represent complexity in other domains as well.

A criticism of evaluation time is the variability in its repeated measurements; therefore, this paper shows how to minimise this variability.

The results indicate that the nuanced notion of complexity in time-control almost consistently produces superior accuracy on both training and test data. Even when time-control produces slightly greater sizes or times, the correspondingly superior accuracy counter-weighs these increases. After all, the complexity-control is not the end-goal alone; instead, it should also accompany better accuracy. Even so, the increase in complexity is not off the scale as is typically the case with unrestrained GP.

The paper also shows that time-control can differentiate functional complexity especially when the population has identically-sized individuals. To facilitate this, the paper proposes Fixed Length Initialisation (FLI) that creates an identically-sized but functionally-diverse population. The results show that while FLI particularly suits time-control, it also generally improves the performance of size-control.

Overall, the paper poses evaluation time as a promising alternative to counting nodes in GP.

References

1. Azad, R.M.A., Ryan, C.: Variance based selection to improve test set performance in genetic programming. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, pp. 1315–1322. ACM, Dublin (2011). http://dl.acm.org/citation.cfm?id=2001754
2. Azad, R.M.A., Ryan, C.: A simple approach to lifetime learning in genetic programming based symbolic regression. Evol. Comput. **22**(2), 287–317 (2014). https://doi.org/10.1162/EVCO_a_00111. http://www.mitpressjournals.org/doi/abs/10.1162/EVCOa00111
3. Couture, M.: Complexity and chaos-state-of-the-art; formulations and measures of complexity. Technical report, Defence research and development Canada Valcartier, Quebec (2007)
4. Dignum, S., Poli, R.: Operator equalisation and bloat free GP. In: O'Neill, M., et al. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 110–121. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78671-9_10
5. Dua, D., Karra Taniskidou, E.: UCI machine learning repository (2017). http://archive.ics.uci.edu/ml
6. Falco, I.D., Iazzetta, A., Tarantino, E., Cioppa, A.D., Trautteur, G.: A kolmogorov complexity-based genetic programming tool for string compression. In: Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation, pp. 427–434. Morgan Kaufmann Publishers Inc., Las Vegas (2000)
7. Griinwald, P.: Introducing the minimum description length principle. Adv. Minimum Description Length: Theory Appl. **3**, 3–22 (2005)
8. Gustafson, S., Burke, E.K., Krasnogor, N.: On improving genetic programming for symbolic regression. In: Corne, D., et al. (eds.) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, vol. 1, pp. 912–919. IEEE Press, Edinburgh, 2–5 September 2005. http://ieeexplore.ieee.org/servlet/opac?punumber=10417&isvol=1
9. Iba, H., de Garis, H., Sato, T.: Genetic programming using a minimum description length principle. In: Kinnear, Jr., K.E. (ed.) Advances in Genetic Programming, chap. 12, pp. 265–284. MIT Press, Cambridge, MA, USA (1994). http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap12.pdf
10. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 70–82. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36599-0_7
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992). http://mitpress.mit.edu/books/genetic-programming

12. Kulkarni, S.R., Harman, G.: Statistical learning theory: a tutorial. Wiley Interdisc. Rev.: Comput. Stat. **3**(6), 543–556 (2011). https://doi.org/10.1002/wics.179
13. Kumar, A., Goyal, S., Varma, M.: Resource-efficient machine learning in 2 KB RAM for the internet of things. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning. Proceedings of Machine Learning Research, PMLR, International Convention Centre, vol. 70, pp. 1935–1944. Sydney, 06–11 August 2017
14. Lipton, Z.C.: The mythos of model interpretability. Commun. ACM **61**(10), 36–43 (2018). https://doi.org/10.1145/3233231
15. Luke, S., Panait, L.: Fighting bloat with nonparametric parsimony pressure. In: Guervós, J.J.M., Adamidis, P., Beyer, H.-G., Schwefel, H.-P., Fernández-Villacañas, J.-L. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 411–421. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45712-7_40. http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2439&spage=411
16. Luke, S., Panait, L.: A comparison of bloat control methods for genetic programming. Evol. Comput. **14**(3), 309–344 (2006). https://doi.org/10.1162/evco.2006.14.3.309. http://cognet.mit.edu/system/cogfiles/journalpdfs/evco.2006.14.3.309.pdf
17. Mei, Y., Nguyen, S., Zhang, M.: Evolving time-invariant dispatching rules in job shop scheduling with genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 147–163. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_10
18. Paris, G., Robilliard, D., Fonlupt, C.: Exploring overfitting in genetic programming. In: Liardet, P., Collet, P., Fonlupt, C., Lutton, E., Schoenauer, M. (eds.) EA 2003. LNCS, vol. 2936, pp. 267–277. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24621-3_22
19. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 204–217. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36599-0_19. http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2610&spage=204
20. Silva, S., Dignum, S., Vanneschi, L.: Operator equalisation for bloat free genetic programming and a survey of bloat control methods. Genet. Program Evolvable Mach. **13**(2), 197–238 (2012). https://doi.org/10.1007/s10710-011-9150-5
21. Vanneschi, L., Castelli, M., Silva, S.: Measuring bloat, overfitting and functional complexity in genetic programming. In: GECCO 2010: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 877–884. ACM, Portland, 7–11 July 2010. https://doi.org/10.1145/1830483.1830643
22. Vapnik, V.N.: Statistical Learning Theory. Adaptive and Learning Systems for Signal Processing, Communications, and Control. Wiley, New York (1998). OCLC: 845016043
23. 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 Trans. Evol. Comput. **13**(2), 333–349 (2009). https://doi.org/10.1109/TEVC.2008.926486. http://ieeexplore.ieee.org/document/4632147/

Leveraging Asynchronous Parallel Computing to Produce Simple Genetic Programming Computational Models

Aliyu Sani Sambo
School of Computing & Digital Technology
Birmingham City University, UK
aliyu.sambo@mail.bcu.ac.uk

R. Muhammad Atif Azad
School of Computing & Digital Technology
Birmingham City University, UK
atif.azad@bcu.ac.uk

Yevgeniya Kovalchuk
School of Computing & Digital Technology
Birmingham City University, UK
Yevgeniya.Kovalchuk@bcu.ac.uk

Vivek Padmanaabhan Indramohan
Health, Education & Life Sciences
Birmingham City University, UK
vivek.indramohan@bcu.ac.uk

Hanifa Shah
Computing Engineering & the Built Environment
Birmingham City University, UK
hanifa.shah@bcu.ac.uk

ABSTRACT

Traditionally, reducing complexity in Machine Learning promises benefits such as less overfitting. However, complexity control in Genetic Programming (GP) often means reducing the sizes of the evolving expressions, and past literature shows that size reduction does not necessarily reduce overfitting. In fact, whether size consistently represents complexity is itself debatable. Therefore, this paper proposes *evaluation time* of an evolving model – the computational time required to evaluate a model on data – as the estimate of its complexity. Evaluation time depends upon the size, but crucially also on the *composition* of an evolving model, and can thus distil its underlying complexity. To discourage complexity, this paper takes an innovative approach that **asynchronously** evaluates multiple models concurrently. These models *race* to their completion; thus, those models that finish earlier, join the population earlier to breed further in a steady-state fashion. Thus, the computationally simpler models, even if less accurate, get further chances to evolve before the more accurate yet expensive models join the population. Crucially, since evaluation times vary from one execution to another, this paper also shows how to *significantly* minimise this variation.

The paper compares the proposed method on six challenging symbolic regression problems with both standard GP and GP with an effective bloat control method. The results demonstrated that the proposed asynchronous parallel GP (APGP) indeed produces individuals that are smaller, faster and more accurate than those in standard GP. While GP with bloat control (GP+BC) produced smaller individuals, it did so at the cost of lower accuracy than APGP both on training and test data, thus questioning the overall benefits of bloat control. Also, while APGP took the fewest evaluations to match the training accuracy of GP, GP+BC took the most.

These results, and the portability of evaluation time as an estimate of complexity encourage further experimentation and fine-tuning of this hitherto unexplored style of GP.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; *Machine learning*; *Genetic programming*; Parallel computing methodologies;

KEYWORDS

genetic programming, model complexity, parallel computing

ACM Reference Format:
Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Leveraging Asynchronous Parallel Computing to Produce Simple Genetic Programming Computational Models. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3373921

1 INTRODUCTION

The key challenges of managing the complexity of machine learning (ML) models include defining what complexity is and constructing a mechanism to control it. Not only do these definitions fail to transfer from one ML algorithm to another, but the motivations for this endeavour vary. A common reason for managing the complexity of ML models is attaining models that are just complex enough to explain the phenomenon generating the given data and not too complex to reflect noise in the data. This way, the predictions on unseen data will be accurate [28]. A challenge related to this goal is determining when the complexity is just enough. Another incentive is a requirement for models to use computational resources efficiently; for example, some computational environments such as in the *Internet of Things* (IoT) devices [22] constrain the evaluation time of an acceptable model even if this compromises its accuracy. Moreover, a concern specifically in Genetic Programming (GP) [17] is that models grow too complex and render the evolutionary search ineffective. A further motivation is the demand for interpretable models: simple models can be more interpretable [23], and interpretability of ML models is now important, especially due to legal frameworks such as the EU General Data Protection Regulation (GDPR)[1]. To summarise, the reasons for controlling model complexity vary and so does the notion of complexity [6].

Genetic programming (GP), the algorithm of our interest, is a versatile tool used in ML, automatic programming and design, and as a general problem-solving instrument. The notions of complexity in these diverse applications are not always interchangeable. For example, in data modelling, managing the size and structure of a model is useful but some other notions of complexity are also desired. Some of these notions are briefly discussed in section 2.1.2. These notions are highly contextual and application specific; therefore, universally defining a notion of complexity is difficult. Furthermore, implementing different complexity controls can be non-trivial.

This paper proposes a novel method called *Asynchronous Parallel Genetic Programming* (APGP) for managing the complexity of GP models based on a broad notion of complexity. This notion is the *evaluation time*, namely, the computational time required for evaluating a model on data. The accuracy of different models being produced during the evolutionary process must be evaluated using the same data-set but the time required to evaluate a model varies from another according to its makeup. For example, models made up of computationally expensive building blocks or having exceptionally large structures take longer to evaluate. Section 2.2 discusses the correlation between the evaluation time and some notions of complexity.

If complex models take longer to evaluate than the simpler ones, that allows an opportunity to naturally control complexity while still encouraging accuracy. Let multiple models *race* in parallel to finish evaluating *asynchronously* such that the simpler models who finish earlier can potentially join the breeding population in a steady-state fashion earlier than the complex ones that are still evaluating. Thus, the simpler models can thrive but only until a more accurate yet complex model arrives; that is because selection solely considers accuracy. Thus, a dynamic interplay between accuracy and simplicity occurs during evolution.

Note, unlike common practice of parallel GP, in the APGP the breeding for the next generation does not wait for the *entire* current set of evaluations to finish. Due to asynchronous parallelism, as soon as an individual finishes evaluation and joins the parent population, it can be selected to breed while some others are still evaluating. This is natural as well; after all, breeding in natural populations is not synchronised. Section 3 further details APGP.

Despite the absence of a universal notion of complexity, we must still assess the complexity of the models produced by APGP. In this study, we note both the model expressions size (an easily quantifiable and popular notion) and the evaluation time. To then assess as to whether APGP controls complexity *effectively*, we test accuracy on test sets (generalisation); this is because some previous work [3, 37] shows that bloat control alone does not automatically produce models that generalise. To this end, we compare APGP with standard GP, and GP with a very effective bloat control (GP+BC).

The results indicate that the APGP breeds simple and the most accurate (both on the training and test data) models; also, it trains the fastest in that it takes the fewest evaluations to match the training accuracy of GP, whereas GP+BC takes the most.

Section 2 of this paper provides some background; section 4 details the experiments and the associated challenges; section 5 presents the results; and section 6 concludes the paper.

2 BACKGROUND

2.1 Genetic Programming

GP enables computers to program themselves and build models automatically. It is also known as an evolutionary algorithm (EA) due to the way it loosely simulates the Darwinian principle of natural selection to automatically search a space of possible models without any prior knowledge [18]. This involves a continuous use of operators that probabilistically choose the better performing individuals (models) from the current population, and then cross and mutate them to produce new ones. A variable tree-based structure is the traditional and most popular representation of GP individuals [12], although other variations are also common [1, 5, 13, 31, 35, 40]. As a result, the sizes of GP individuals may vary, which introduces challenges such as uncontrolled growth in the size of evolving code, also termed code bloat [29].

GP has been used to produce solutions to many practical problems. A detailed account of practical and innovative results produced by GP is provided in [19].

2.1.1 **Parallel Genetic Programming.** The use of parallel computing is not new in GP [16, 20, 32]. However, the aim of applying parallel computing has been typically to improve the run times [27] of GP, as opposed to reducing the complexity of individual models, which is the target of this study. As GP runs can take a long time to complete, parallelising these runs reduces the overall run time. Most commonly, generational replacement schemes parallelise the evaluation of offspring populations. However, the generational replacement requires the *entire* offspring population to be ready before its members can start breeding, which means that all evaluation threads join at a single *point of synchronisation* before the evolution proceeds further. Hence, this parallelisation confers no advantage to simpler individuals and is disadvantageous in terms of resource utilisation: while an individual is taking an excessive amount of time to evaluate on one CPU, the remaining CPUs stay idle because other individuals have completed their evaluations. Some recent work has employed asynchronous parallel computing in non-GP EAs to alleviate this problem of idle time.

An evaluation time bias, that favours smaller evaluation times, has been observed in [32][33], However, they were concerned with real parameter optimisation using fixed-length chromosomes; therefore, the impact of asynchronous parallelism on the variable length structures in GP and associated challenges in terms of both the complexity and accuracy have not been studied.

Another candidate for parallelisation in EAs is the so called *island model* [4, 30], where the evolving populations are divided into multiple distinct islands, and the *subpopulation* in each island can be evolved in a separate parallel thread. Regardless of parallelisation, the island model offers advantages such as greater diversity in the overall population because each island is shielded from other islands except at discrete intervals when selected individuals are

[1]The EU General Data Protection Regulation includes a *right to explanation* in situations, where ML algorithms are applied to make a decision affecting a person.

exchanged across islands. The role of parallelisation in this model is still to just speed up the run times.

2.1.2 **Complexity in Genetic Programming.** Deciding what complexity is and implementing the functionality to manage it are often non-trivial tasks [37]. Therefore, existing techniques for controlling complexity in GP vary in their spirit.

Traditionally, controlling complexity in GP means controlling structural complexity such as the size (bloat control) of the evolved expressions, or the number of encapsulated sub-trees and layers, while ignoring the underlying functional or computational complexity. Representative studies in this area proposed bloat control measures [37], techniques based on the minimum description length principle [10][14], Kolmogorov-based approaches [9] and invariance theorem [26]. For example, bloat control penalises a large yet linear expression $4x + 8x + 2x + x + x$, which is functionally and computationally less complex than a smaller expression $sin(x)$ [3], which is computationally equivalent to its Taylor series expansion $\sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$. In this case, the smaller expression $sin(x)$ requires a lot more resources to execute than its linear counterpart. Thus, complexity in GP is more than merely the expression size.

Approaches based on functional complexity recognise that small structures may be more complex than larger ones and hence focus on the functionality of structures. To elicit functional complexity, one approach approximates the evolving expressions by polynomials [39]; complex expressions are approximated by polynomials of a high degree owing to large oscillations in the response of the function. This degree of approximating polynomials is termed the *order of non-linearity* of the corresponding GP expressions, and it should be minimised. However, the minimisation requires evolving expressions to be twice differentiable, a property that is not always guaranteed. To alleviate this constraint, Vanneschi et al. [37] defined a less rigorous measure of functional complexity, whereby the slope of an expression is approximated by a simpler but error prone measure. As such Vanneschi et al. did not control the complexity; instead, they only measured the complexity of evolving expressions. Note, in fact, if the evolving code is not made of mathematical expressions, neither of the above approaches apply. In another related approach, Azad and Ryan [2] used the variance of the outputs of evolving expressions as a measure of the functional complexity. They explicitly minimised the variance and maximised accuracy using a multi-objective optimisation approach.

Finally, statistical learning theory-based approaches measure the complexity of a space of functions that can be learned using statistical classification. Main techniques include generalisation error bound VC theory and VC dimensions [21] [38].

Implementing these techniques for controlling complexity is not always trivial and can be different from one application of GP to another. In contrast, the present study aims to induce non-complex models naturally, where the complexity of a model is simply its evaluation time.

2.2 Time is not Size

While the previous section theoretically exemplifies why controlling size is not the same as controlling evaluation time or computational complexity, it is also important to empirically verify that. Without that verification we can not be certain as to whether we can practically differentiate between functional complexities of equally sized expressions. Moreover, obviously, evaluation time also increases when the expression size increases; however, if the evaluation time does not practically increase with the functional complexity, then measuring this time is just a fancy way of counting nodes in a GP expression. Clearly, that is undesirable.

To this end, we used four different *functions sets* to generate symbolic regression models of different complexities; Figure 1 details the functions sets. For each functions set, we generated differently sized individuals (10, 20, 30, ..., 300), and in turn for each size we generated 30 random expressions. All models were then evaluated 50 times, each with the same set of data. The four plots in Figure 1 represent the average evaluation times of individuals according to their size and complexity.

Two trends are clearly visible in Figure 1: (1) given the same size, the evaluation times of functionally complex individuals are consistently higher than those for their counterparts; and (2) evaluation times are also strongly correlated with the expression sizes, as expected. Hence, evaluation times indeed discriminate between different functional complexities; however, if a simple function is represented inefficiently by an excessively large expression, it will evaluate slower. Therefore, evaluation time control impacts conditionally: it curbs functional complexity when the sizes of a given set of individuals are within a certain tolerance (or range); otherwise, it curbs the growth in size (controls bloat). Note, this tolerance increases as the size of individuals increases. For example, the evaluation time of size 75 with functions set COS-SIN is the same as that for size 175 with the functions set ADD-SUB.

The above findings also predict the limiting behaviour of evaluation time control in GP. In a functionally diverse but a size-converged population – where bloat control is impotent – evaluation times discriminate between functional complexities, whereas in a functionally converged but a size-diverse population, evaluation times discriminate between sizes.

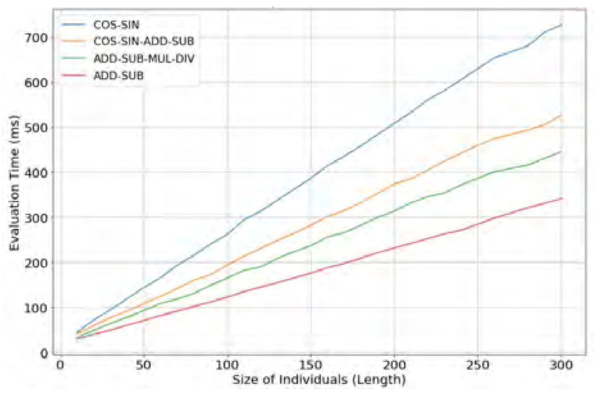


Figure 1: Relationship between evaluation time, size and the composition of models is shown. Individuals made up of COS and SIN operators have higher average evaluation times than same-sized individuals from other functions sets. Also, note size correlates with evaluation time.

Algorithm 1: The APGP Algorithm

```
/* Initialise */
n ← set total number of offspring to produce;
threadpool ← set no. of concurrent operations allowed;
popsize ←set population size;
/* Generate and evaluate initial population */
for $j$ ← 1 to $popsize$ do
    new ← generate individual;
    Evaluate new ;
    population + new;
end
/* Generate and evaluate offspring in parallel */
for $i$ ← 1 to $n$ do
    if $threadpool > 0$ then
        Create thread: threadpool ← threadpool − 1;
        Select parents;
        Produce offspring;
        Evaluate offspring;
        i ← Selected individual to be replaced;
        if $Offspring\ fitness > i\ fitness$ then
            Lock i memory position;
            i ← Offspring;
            Release locked position;
        else
            Drop offspring;
        Release thread: threadpool ← threadpool + 1;
    else
        Wait;
end
```

3 ASYNCHRONOUS PARALLEL GENETIC PROGRAMMING

The proposed Asynchronous Parallel GP (APGP) method evaluates GP individuals asynchronously to potentially let the simple, fast-evaluating individuals get into the breeding population prior to their more complex counterparts. This is also natural; after all, natural populations do not simply halt while one of their members is tested against the environment. Yet, this is precisely what happens in traditional GP: while an individual of any complexity is being evaluated, the evolution stops regardless of how long that evaluation takes. As a result, in classical GP, there is no advantage for being evaluated quickly and thereby being less complex. However, the APGP aims to leverage that performance advantage to breed simple yet accurate models.

APGP is based on *Steady State GP* [36], a technique that can be used to produce a single offspring that immediately competes for a place in the population after being evaluated. This method allows multiple independent breeding operations, working on the same population, to be executed in parallel and asynchronously. For example, a maximum of 50 of such breed operations can be allowed to run at the same time; and as soon as one operation finishes, another one starts. However, these operations may finish at different times due to the varying times it takes to evaluate different models. This difference is due to different make up of models because all models are evaluated on exactly the same data-set. Thus, a race condition develops such that less complex individuals that take less time to be evaluated and are good enough to find a place in the population may reproduce earlier than the more complex individuals that evaluate slower.

Algorithm 1 lists the pseudo code of the APGP algorithm, which is initialised by setting the number of allowed concurrent evaluations, population size and total number of offspring (total number of fitness evaluations in the run). This is followed by creating the initial population and evaluating it. The parallel breeding then begins by initiating multiple breed operations up to the allowed limit. These breed operations evaluate offspring in independent threads. As soon as the evaluation of an offspring is complete, if it is more accurate than a randomly selected individual in the current population, it replaces that individual and then releases the computing resources to enable another breed operation to commence. As these parallel operations work over the same population, a temporary lock is set on the individual being replaced to avoid clashes.

As discussed above, only the accuracy of an offspring decides whether it finds a place in the population. As such, speed becomes an advantage only when it is accompanied by high accuracy. Where complex candidates are more accurate, they will eventually succeed and propagate. Thus, complex models are not excluded, and the simplicity of good models will be constrained by the possibilities within the specific problem.

4 EXPERIMENTS

We compared the performance of APGP against standard GP and against GP with a bloat control mechanism (GP+BC) on a suite of symbolic regression problems. Identical parameters were adopted except for the race condition, which was present only in APGP, and the bloat control mechanism in GP+BC.

4.1 Test Problems

We considered the findings in [41] when choosing the test problems. Five high-dimensional problems (with five or more input variables) and one low-dimensional problem were used. The data set for problems 1–5 are available at [8]; Problem 6 is a bi-variate version of the function used in [11]. A summary of the data sets is available in Table 1. As the results in section 5 show, all but Problem 4 are hard for GP (the accuracy scores are less than 41%). Hence, these problems require GP to run long and thus present a good test bed for complexity control because complexity in GP grows with long runs.

4.2 Configuration and Parameters

The basic parameters for all the methods are summarised in Table 2. Other key experimental decisions are as follows:
- ***Bloat Control for GP+BC***: To control bloat, we used the *Double tournament* [25], a non parametric method that neither requires finding the right penalty for greater sizes, nor does it assume an appropriate size. This method has been very successful on a variety of benchmark problems [25][24] and the results of our experiments later in Section 5 also show that this method indeed limits sizes

ID	Problem label	No. of Variables	No. of instances
1	Airfoil	5	1503
2	Boston Housing	13	506
3	Concrete Strength	8	1030
4	Dow Chemical	57	1066
5	Energy Efficiency	8	768
6	$y^2x^6 - 2.13y^4x^4 - +y^6x^2$	2	250 (x=min:-0.3, step: 0.012; y=x + 0.03)

Table 1: Overview of Test Problems

aggressively. As recommended [25], we used the best problem-independent settings for this method as follows: in the first round, run n probabilistic tournaments, each with a tournament of size 2, to select a set of n individuals; then, in the second round, select the fittest out of the n individuals. The tournaments in the first round choose the smallest individual with a probability of 0.7. We also considered using Operator Equalisation (OpEq), a more recent bloat control method [7] [34]; however, unlike APGP, which uses steady state replacement, OpEq requires generational replacement.

- ***Division by zero***: Individuals with zero division errors were assigned the worst fitness. As discussed in [15], protected operators commonly used in GP lead to poor generalisation.
- ***Data split***: The data-sets were randomly split (without replacement) into 80% for training and 20% testing.
- ***Fitness function***: The following minimisation function based on the normalised mean squared error was used: $\frac{1}{1-\sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(y_i-\hat{y}_i)^2}}$.

Parameter	Setting
Number of runs	50
Population size	500
Run terminates	After 35,000 evaluations (≡ 70 generations)
Random tree/ subtree generation	Ramped half-and-half (depth $min = 1, max = 4$)
Tree depth limit	17
Operators & probabilities	One point crossover = 0.9 ; Point mutation = 0.1
Function set	$+, -, *, /$, sin, cos, neg
Constants (ERC)	|ERC| = 100 (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament size = 3
Replacement	steady state, inverse tournament size = 5

Table 2: Summary of Parameters

4.3 Key Implementation Challenges

The key challenges associated with the proposed implementation include the following.

4.3.1 **Measuring Evaluation Time Consistently.** A problem with measuring evaluation times is that they vary across multiple executions, and if this variability is high, one cannot reliably estimate the complexity of a given model from a single evaluation. Unfortunately, since CPU scheduling is the prerogative of the operating system kernel, we can not eliminate this variation totally. However, we found ways to significantly minimise this variation across evaluations.

We found that CPU management options can help minimise this variation. These options include: (1) stopping all background services, (2) locking the CPU speed to prevent the operating system power management from interfering, (3) executing the experiments on dedicated processors and (4) assigning the experimental tasks a high priority. Figure 2 illustrates the impact of these changes. Each box-plot represents multiple evaluation times for an individual of a given size. Figure 2a represents the case when no CPU management was applied; clearly the variation in evaluation times is high. In contrast, Figure 2b shows that after applying the above-mentioned CPU management options, the variation clearly decreased.

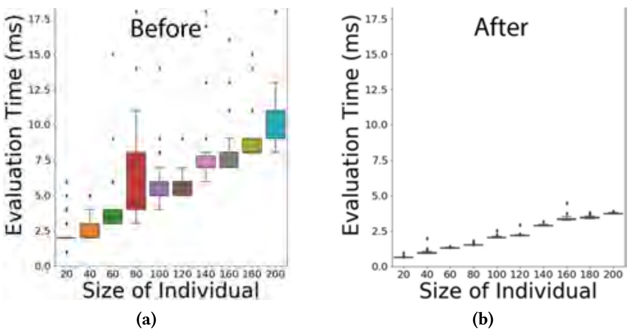


Figure 2: Measuring evaluation time consistently. Applying CPU management options lead to more consistency.

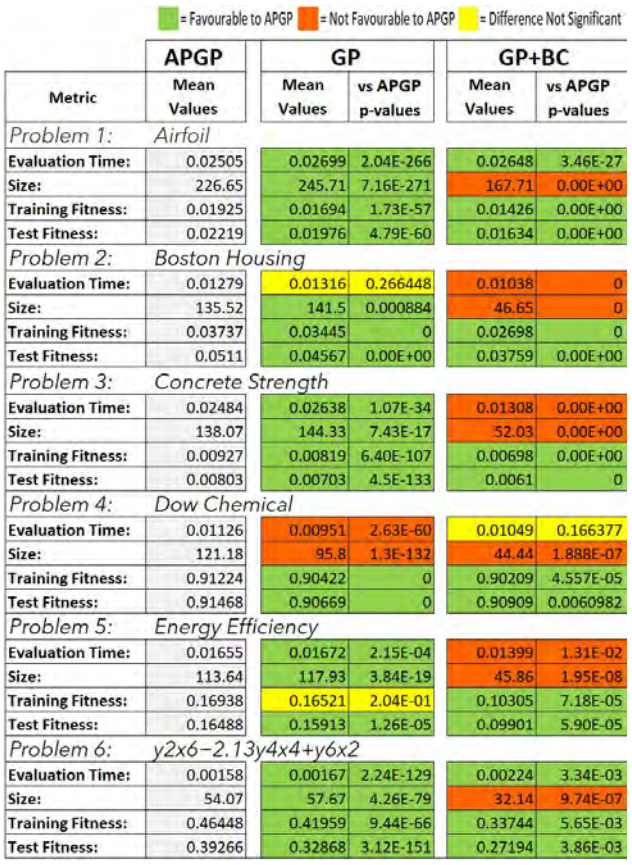
4.3.2 **Degree of Concurrency.** The size of the thread pool (number of parallel threads available) can vary in APGP. We tested pool sizes 5, 25, 50, 75 and 100. Although, size 50 produced the best or competitive results generally, some problem specific improvement may be possible with other thread sizes. Future work can further investigate this.

5 RESULTS

To evaluate the performance of the proposed APGP we compared the metrics of the final populations and of the best individuals (on test data) against standard GP and GP with bloat-control (GP+BC). In addition, we also noted the average number of evaluations to reach the average training accuracy of GP (target accuracy); this shows the efficiency of the other methods relative to GP.

5.1 APGP is the Most Accurate

Figure 3 shows the colour-coded results of the Mann-Whitney U statistical test on the final populations for all test problems. The attributes tested include the evaluation time, size, training and test fitness (accuracy on out-of-sample data). The p-values included in Figure 3 statistically compare GP and GP+BC with APGP only. The rows are green when the APGP is significantly better (more for accuracy and less for complexity), red when it is significantly worse, and yellow when it is not significantly different.



🟩 = Favourable to APGP 🟥 = Not Favourable to APGP 🟨 = Difference Not Significant

	APGP	GP		GP+BC	
Metric	Mean Values	Mean Values	vs APGP p-values	Mean Values	vs APGP p-values
Problem 1:	Airfoil				
Evaluation Time:	0.02505	0.02699	2.04E-266	0.02648	3.46E-27
Size:	226.65	245.71	7.16E-271	167.71	0.00E+00
Training Fitness:	0.01925	0.01694	1.73E-57	0.01426	0.00E+00
Test Fitness:	0.02219	0.01976	4.79E-60	0.01634	0.00E+00
Problem 2:	Boston Housing				
Evaluation Time:	0.01279	0.01316	0.266448	0.01038	0
Size:	135.52	141.5	0.000884	46.65	0
Training Fitness:	0.03737	0.03445	0	0.02698	0
Test Fitness:	0.0511	0.04567	0.00E+00	0.03759	0.00E+00
Problem 3:	Concrete Strength				
Evaluation Time:	0.02484	0.02638	1.07E-34	0.01308	0.00E+00
Size:	138.07	144.33	7.43E-17	52.03	0.00E+00
Training Fitness:	0.00927	0.00819	6.40E-107	0.00698	0.00E+00
Test Fitness:	0.00803	0.00703	4.5E-133	0.0061	0
Problem 4:	Dow Chemical				
Evaluation Time:	0.01126	0.00951	2.63E-60	0.01049	0.166377
Size:	121.18	95.8	1.3E-132	44.44	1.888E-07
Training Fitness:	0.91224	0.90422	0	0.90209	4.557E-05
Test Fitness:	0.91468	0.90669	0	0.90909	0.0060982
Problem 5:	Energy Efficiency				
Evaluation Time:	0.01655	0.01672	2.15E-04	0.01399	1.31E-02
Size:	113.64	117.93	3.84E-19	45.86	1.95E-08
Training Fitness:	0.16938	0.16521	2.04E-01	0.10305	7.18E-05
Test Fitness:	0.16488	0.15913	1.26E-05	0.09901	5.90E-05
Problem 6:	y2x6−2.13y4x4+y6x2				
Evaluation Time:	0.00158	0.00167	2.24E-129	0.00224	3.34E-03
Size:	54.07	57.67	4.26E-79	32.14	9.74E-07
Training Fitness:	0.46448	0.41959	9.44E-66	0.33744	5.65E-03
Test Fitness:	0.39266	0.32868	3.12E-151	0.27194	3.86E-03

Figure 3: Results of Mann-Whitney U tests on the final populations (APGP vs GP; APGP vs GP+BC). The results show that APGP produced significantly more accurate (training and test fitness) models on all tests against both GP and GP+BC. While APGP produced significantly simpler models than GP in 5 out of the 6 test, GP+BC produced significantly simpler models than APGP at the price of accuracy.

Columns 3 and 4 in Figure 3 show that the APGP significantly outperforms GP in the majority of the tests, as indicated by the green colour (colour code for results favouring APGP). Also, the final population of APGP is simpler than GP for all problems except Problem 4; this is indicated by significantly smaller mean evaluation times and sizes. Furthermore, the training and test accuracy scores of APGP are significantly better than those of GP in all problems except for the training fitness of Problem 5 where the improvement is not significant. Finally, the APGP test fitness values, which is the major concern when evaluating models, are significantly better than those of GP in all problems. This indicates that the APGP models tend to generalise better on unseen data. Notice also that the reported p-values are very small; this indicates that the differences between the results of the two methods are very significant.

In Problem 4, where the APGP produced more complex individuals, the training and test fitness values are better. This is not the traditional bloat, where size grows without an associated increase in fitness. In our case, the increase in the size comes with an increase in fitness. Also, note that Problem 4 is the easiest of all the problems: accuracy scores are more than 90%.

When comparing the APGP with GP with bloat control (GP+BC), as captured in columns 5 and 6 of Figure 3, GP+BC has produced significantly smaller models in all problems. However, these simpler models have significantly lower training and test accuracy values. Unlike APGP, the simplicity produced by GP+BC comes at the expense of accuracy.

That GP+BC produces significantly smaller models is not surprising given that it aggressively targets sizes. In contrast, the complexity control in APGP is gentler as in it simply offers a potential advantage of getting a place in the population due to quicker evaluations; parent selection solely prefers training accuracy. Even so, the APGP outperforms GP+BC on training **and** test set accuracy, and GP on all accounts. Note, much like GP+BC, APGP can also leverage the bloat control techniques to select individuals; we leave that investigation for future work.

While the APGP outperforms GP and GP+BC *on average*, next we compare the best individuals produced by each algorithm in terms of test set accuracy, which is a principal motivator behind complexity control in Machine Learning.

Test ID	Best Method	Test Fitness Gain	Eval. Time Reduction	Size Reduction
GP and APGP Compared				
Problem 1	APGP	24.99 %	43.26 %	47.17 %
Problem 2	APGP	6.64 %	-12.31 %	0.896 %
Problem 3	APGP	22.43 %	36.99 %	44.96 %
Problem 4	APGP	2.04 %	78.13 %	85.71 %
Problem 5	APGP	3.13 %	22.92 %	18.38 %
Problem 6	APGP	0.11 %	-42.57 %	11.11%
GP+BC and APGP Compared				
Problem 1	APGP	19.93 %	-18.00 %	-15.0 %
Problem 2	APGP	43.24 %	-95.17 %	-452.5 %
Problem 3	APGP	11.56 %	-72.70 %	-100.0 %
Problem 4	APGP	2.43 %	78.56 %	65.82 %
Problem 5	APGP	69.10 %	34.51 %	-40.91 %
Problem 6	APGP	1.71 %	-49.45 %	-695.0%

Table 3: Best individuals Compared. On all problems, APGP produced more accurate models than GP and GP+BC. GP+BC produced simpler models at the expense of accuracy.

5.1.1 APGP Tops the Leaderboards. GP is often concerned with finding the best solution. Therefore, we examine the best of each set of runs and summarise the results in Table 3. For all the six problems APGP produced individuals with the overall best test fitness (accuracy) values. As before, the best models produced by the APGP were also smaller than those by GP.

5.2 Cost of Producing Accurate Models

Using the average training accuracy of GP as a benchmark, we compared the average evaluations taken by each method to match that training accuracy. Where a particular run did not achieve that target, it was assigned the maximum number of budgeted evaluations. As summarised in Table 4, APGP used 10% to 40% fewer evaluations than GP, and 15% to 84% fewer than GP+BC. Thus, APGP is the fastest to train, whereas GP+BC despite producing smaller individuals is the slowest; note, as in Figure 3, both GP and GP+BC are also consistently less accurate than APGP on both training and test sets.

No. of Evaluations To Reach Target Accuracy					
Test ID	GP Mean	GP+BC Mean	APGP Mean	GP & APGP Difference	GP+BC & APGP Diff.
Problem 1	29407	33041	23941	18.59 %	38.01 %
Problem 2	20761	27792	17762	14.44 %	56.47 %
Problem 3	31668	33595	20422	35.51 %	64.50 %
Problem 4	17861	19762	10719	39.99 %	84.36 %
Problem 5	28852	34842	25658	11.07 %	35.79%%
Problem 6	33077	33895	29546	10.68 %	14.72 %

Table 4: APGP used significantly fewer evaluations than both GP and GP+BC to reach the same target accuracy.

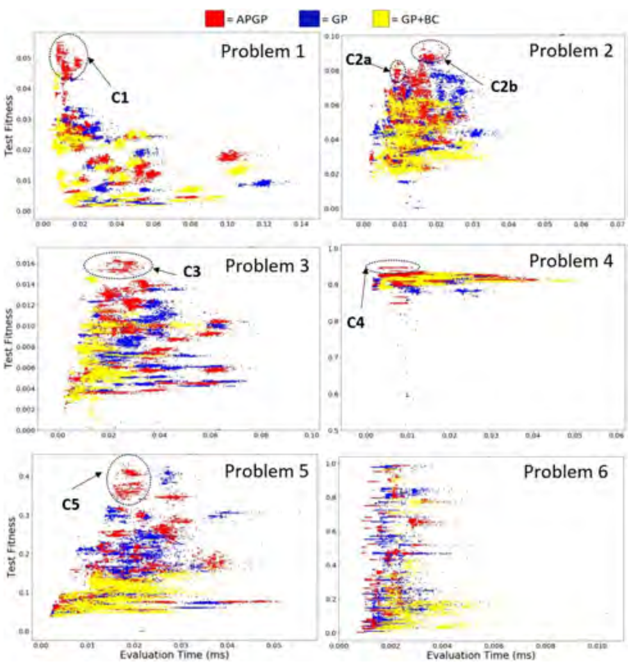


Figure 4: Mapping accuracy and evaluation time of the individuals in the populations. APGP produced clusters with the combined attributes of simplicity and accuracy (C1-C5).

5.3 Standout Multi-Objective Solutions

As the aim of APGP is to produce both simple *and* accurate models, we examine the final populations to see if this is happening. The test fitness (representing accuracy on unseen data) is plotted against the evaluation time (our proposed notion of complexity) to produce Figure 4. The red, blue and yellow dots represents APGP, GP and GP+BC respectively. Desirable models are thus close to the top left corner (low complexity; high accuracy).

We observed that the APGP produced some distinct clusters of models that are both accurate and simple in five out of the six test problems (labelled C1, C2a, C2b, C3, C4a and C5). Cluster C2b represents a situation where the accurate models lie in the relatively more complex individuals.

When the test fitness was plotted against size, we found similar clusters but space constrains do not permit producing them here.

The presence of these breakout clusters indicates that the APGP normally offers the best trade-off in terms of accuracy and simplicity.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes a new measure of complexity, evaluation time; unlike model size, evaluation time is a function of size, functional and computational effort of a model, both theoretically as well as practically. This measure is broadly applicable, and is especially useful where complexity is difficult to define.

A criticism of evaluation time is the variability in its repeated measurements; therefore, this paper shows how to minimise this variability so the evaluation time can be measured reliably.

Instead of explicitly and subjectively penalising evaluation times, the paper proposes asynchronous parallel GP (APGP), which leverages asynchronous parallel computing to induce a race between concurrent executions of multiple models; the models that finish evaluating early can join a steady state population early and thus may get an evolutionary advantage. The APGP thus questions the conventional, but eventually unnatural, way of evaluating individuals only one after the other, or in a lock-step.

Although the evolutionary pressure to breed simpler models in the APGP is potentially gentler than that with an aggressive bloat control technique (because selection in the APGP prefers accuracy on training data), the APGP still produces models that are not only more accurate both on training and out of sample (test) data but also simpler than those produced by standard GP. Although, GP with an effective bloat control (GP+BC) still produced smaller solutions, these solutions were inferior to those from the APGP in terms of both training and test set accuracy and required greater training time (more fitness evaluations); in fact GP+BC took the greatest number of evaluations to match the training accuracy of simple GP.

Often Machine Learning associates low complexity with better performance on test sets. However, the results produced in this paper echo the concerns in some previous literature that decries the link between bloat control and better generalisation in GP. Instead, the results indicate that controlling evaluation time may indeed link complexity control with better generalisation in GP.

Future work can explore, how combining bloat control with the APGP works. Better still, how about *time control* instead of bloat control? Moreover, since evaluation time is a function of expression size, can we develop better initialisation schemes that produce size-identical but time-diverse populations? Perhaps, that can further encourage functional simplicity.

REFERENCES

[1] Raja Muhammad Atif Azad. 2003. *A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System.* Ph.D. Dissertation. University of Limerick, Ireland. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/azad_thesis.ps.gz

[2] Raja Muhammad Atif Azad and Conor Ryan. 2011. Variance based selection to improve test set performance in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation.* ACM, Dublin, Ireland, 1315–1322. http://dl.acm.org/citation.cfm?id=2001754

[3] Raja Muhammad Atif Azad and Conor Ryan. 2014. A Simple Approach to Lifetime Learning in Genetic Programming-Based Symbolic Regression. *Evolutionary Computation* 22, 2 (2014), 287–317. https://doi.org/10.1162/EVCO_a_00111

[4] Erick Cantú-Paz. 1998. A Survey of Parallel Genetic Algorithms. *CALCULATEURS PARALLELES, RESEAUX ET SYSTEMS REPARTIS* 10, 2 (1998), 141–171.

[5] Gopinath Chennupati, Raja Muhammad Atif Azad, and Conor Ryan. 2015. Performance Optimization of Multi-Core Grammatical Evolution Generated Parallel Recursive Programs. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Sara Silva et al (Ed.). Springer, Madrid, Spain, 1007–1014.

[6] Mario Couture. 2007. *Complexity and chaos-state-of-the-art; formulations and measures of complexity.* Technical Report. DEFENCE RESEARCH AND DEVELOPMENT CANADA VALCARTIER (QUEBEC).

[7] Stephen Dignum and Riccardo Poli. 2008. Operator Equalisation and Bloat Free GP. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008 (Lecture Notes in Computer Science)*, Michael O'Neill et al (Ed.), Vol. 4971. Springer, Naples, 110–121. https://doi.org/10.1007/978-3-540-78671-9_10

[8] Dheeru Dua and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[9] I De Falco, Aniello Iazzetta, Ernesto Tarantino, A Delia Cioppa, and Giuseppe Trautteur. 2000. A Kolmogorov Complexity-based Genetic Programming tool for string compression. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation.* Morgan Kaufmann Publishers Inc., Morgan Kaufmann, Las Vegas, Nevada, USA, 427–434.

[10] Peter Griinwald. 2005. Introducing the minimum description length principle. *Advances in minimum description length: Theory and applications* 3 (2005), 3–22.

[11] Steven Gustafson, Edmund K Burke, and Natalio Krasnogor. 2005. On Improving Genetic Programming for Symbolic Regression. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, David Corne et al (Ed.), Vol. 1. IEEE Press, Edinburgh, Scotland, UK, 912–919. http://ieeexplore.ieee.org/servlet/opac?punumber=10417&isvol=1

[12] Nguyen Xuan Hoai, Robert I McKay, and Daryl Essam. 2006. Representation and structural difficulty in genetic programming. *IEEE Transactions on evolutionary computation* 10, 2 (2006), 157–166.

[13] Ting Hu, Joshua Payne, Wolfgang Banzhaf, and Jason Moore. 2012. Evolutionary dynamics on multiple scales: a quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming. *Genetic Programming and Evolvable Machines* 13, 3 (Sept. 2012), 305–337. Special issue on selected papers from the 2011 European conference on genetic programming.

[14] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. 1994. Genetic Programming Using a Minimum Description Length Principle. In *Advances in Genetic Programming*, Kenneth E. Kinnear, Jr. (Ed.). MIT Press, Cambridge, MA, USA, Chapter 12, 265–284. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap12.pdf

[15] Maarten Keijzer. 2003. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming.* EuroGP, Springer, Essex, UK, 70–82.

[16] Jinhan Kim, Junhwi Kim, and Shin Yoo. 2017. GPGPGPU: Evaluation of Parallelisation of Genetic Programming using GPGPU. In *Proceedings of the 9th International Symposium on Search Based Software Engineering, SSBSE 2017 (LNCS)*, Tim Menzies and Justyna Petke (Eds.), Vol. 10452. Springer, Paderborn, Germany, 137–142. https://doi.org/10.1007/978-3-319-66299-2_11

[17] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[18] John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA.

[19] John R. Koza. 2008. Human-competitive machine invention by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22, 3 (2008), 185–193. https://doi.org/10.1017/S0890060408000127

[20] John R. Koza and David Andre. 1995. *Parallel Genetic Programming on a Network of Transputers.* Technical Report CS-TR-95-1542. Stanford University, Department of Computer Science. http://www.genetic-programming.com/jkpdf/tr1542parallelsuversion.pdf

[21] Sanjeev R. Kulkarni and Gilbert Harman. 2011. Statistical learning theory: a tutorial. *Wiley Interdisciplinary Reviews: Computational Statistics* 3, 6 (2011), 543–556.

[22] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1935–1944.

[23] Zachary C. Lipton. 2018. The Mythos of Model Interpretability. *Commun. ACM* 61, 10 (Sept. 2018), 36–43. https://doi.org/10.1145/3233231

[24] Sean Luke and Liviu Panait. 2002. Fighting Bloat with Nonparametric Parsimony Pressure. In *Parallel Problem Solving from Nature - PPSN VII (Lecture Notes in Computer Science, LNCS)*, Juan J. Merelo-Guervos, Panagiotis Adamidis, Hans-Georg Beyer, Jose-Luis Fernandez-Villacanas, and Hans-Paul Schwefel (Eds.). Springer-Verlag, Granada, Spain, 411–421. https://doi.org/10.1007/3-540-45712-7_40

[25] Sean Luke and Liviu Panait. 2006. A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation* 14, 3 (Fall 2006), 309–344. https://doi.org/10.1162/evco.2006.14.3.309

[26] Yi Mei, Su Nguyen, and Mengjie Zhang. 2017. Evolving Time-Invariant Dispatching Rules in Job Shop Scheduling with Genetic Programming. In *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming (LNCS)*, Mauro Castelli, James McDermott, and Lukas Sekanina (Eds.), Vol. 10196. Springer Verlag, Amsterdam, 147–163. https://doi.org/10.1007/978-3-319-55696-3_10

[27] Mouloud Oussaidène, Bastien Chopard, Olivier V. Pictet, and Marco Tomassini. 1997. Parallel Genetic Programming and its application to trading model induction. *Parallel Comput.* 23, 8 (Aug. 1997), 1183–1198.

[28] Gregory Paris, Denis Robilliard, and Cyril Fonlupt. 2003. Exploring Overfitting in Genetic Programming. In *Evolution Artificielle, 6th International Conference (Lecture Notes in Computer Science)*, Pierre Liardet, Pierre Collet, Cyril Fonlupt, Evelyne Lutton, and Marc Schoenauer (Eds.), Vol. 2936. Springer, Marseilles, France, 267–277. https://doi.org/10.1007/b96080 Revised Selected Papers.

[29] Riccardo Poli. 2003. A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming. In *Genetic Programming, Proceedings of EuroGP'2003 (LNCS)*, Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa (Eds.), Vol. 2610. Springer-Verlag, Essex, 204–217. https://doi.org/10.1007/3-540-36599-0_19

[30] David Power, Conor Ryan, and Raja Muhammad Atif Azad. 2005. Promoting diversity using migration strategies in distributed genetic algorithms. In *2005 IEEE Congress on Evolutionary Computation*, Vol. 2. IEEE Press, Edinburgh, Scotland, UK, 1831–1838 Vol. 2. https://doi.org/10.1109/CEC.2005.1554910

[31] Conor Ryan, Michael O'Neill, and J. J. Collins (Eds.). 2018. *Handbook of Grammatical Evolution.* Springer, New York, NY, USA. https://doi.org/10.1007/978-3-319-78717-6

[32] Eric O. Scott and Kenneth A. De Jong. 2015. Evaluation-Time Bias in Asynchronous Evolutionary Algorithms. In *GECCO'15 Student Workshop*, Tea Tusar and Boris Naujoks (Eds.). ACM, Madrid, Spain, 1209–1212. https://doi.org/10.1145/2739482.2768482

[33] Eric O. Scott and Kenneth A. De Jong. 2016. Evaluation-Time Bias in Quasi-Generational and Steady-State Asynchronous Evolutionary Algorithms. In *GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, Tobias Friedrich et al (Ed.). ACM, Denver, USA, 845–852. https://doi.org/10.1145/2908812.2908934

[34] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. 2012. Operator equalisation for bloat free genetic programming and a survey of bloat control methods. *Genetic Programming and Evolvable Machines* 13, 2 (2012), 197–238.

[35] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40.

[36] Gilbert Syswerda. 1991. A study of reproduction in generational and steady-state genetic algorithms. In *Foundations of genetic algorithms.* Vol. 1. Elsevier, Amsterdam, 94–101.

[37] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2010. Measuring bloat, overfitting and functional complexity in genetic programming. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, Juergen Branke et al (Ed.). ACM, Portland, Oregon, USA, 877–884. https://doi.org/10.1145/1830483.1830643

[38] Vladimir Naumovich Vapnik. 1998. *Statistical learning theory.* Wiley, New York, NY. OCLC: 845016043.

[39] Ekaterina J Vladislavleva, Guido F Smits, and Dick Den Hertog. 2009. Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. *IEEE Transactions on Evolutionary Computation* 13, 2 (2009), 333–349. https://doi.org/10.1109/TEVC.2008.926486

[40] James Alfred Walker and Julian Francis Miller. 2008. The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation* 12, 4 (Aug. 2008), 397–417. https://doi.org/10.1109/TEVC.2007.903549

[41] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaskowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines* 14, 1 (March 2013), 3–29. https://doi.org/10.1007/s10710-012-9177-2

Feature Engineering for Improving Robustness of Crossover in Symbolic Regression

Aliyu Sani Sambo
School of Computing & Digital Tech.
Birmingham City University, UK
aliyu.sambo@mail.bcu.ac.uk

R. Muhammad Atif Azad
School of Computing & Digital Tech.
Birmingham City University, UK
atif.azad@bcu.ac.uk

Yevgeniya Kovalchuk
School of Computing & Digital Tech.
Birmingham City University, UK
Yevgeniya.Kovalchuk@bcu.ac.uk

Vivek Padmanaabhan Indramohan
Birmingham City University, UK
vivek.indramohan@bcu.ac.uk

Hanifa Shah
Computing Eng. & Built Environment
Birmingham City University, UK
hanifa.shah@bcu.ac.uk

ABSTRACT

Isolating the fitness-contribution of substructures is typically a difficult task in Genetic Programming (GP). Hence, useful substructures are lost when the overall structure (model) performs poorly. Furthermore, while crossover is heavily used in GP, it typically produces offspring models with significantly lower fitness than that of the parents. In symbolic regression, this degradation also occurs because the coefficients of an evolving model lose utility after crossover. This paper proposes isolating the fitness-contribution of various substructures and reducing the negative impact of crossover by evolving a set of features instead of monolithic models. The method then leverages multiple linear regression (MLR) to optimise the coefficients of these features. Since adding new features cannot degrade the accuracy of an MLR produced model, MLR-aided GP models can bloat. To penalise such additions, we use *Adjusted* R^2 as the fitness function. The paper compares the proposed method with standard GP and GP with linear scaling. Experimental results show that the proposed method matches the accuracy of the competing methods within only 1/10th of the number of generations. Also, the method significantly decreases the rate of post-crossover fitness degradation.

CCS CONCEPTS

• **Computing methodologies → Classification and regression trees**; **Genetic programming**; *Machine learning*; **Artificial intelligence**;

KEYWORDS

genetic programming, regression, feature engineering

ACM Reference Format:
Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Feature Engineering for Improving Robustness of Crossover in Symbolic Regression. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377929.3390078

1 INTRODUCTION

The canonical Genetic Programming (GP) method evaluates the quality of the produced model as a whole and can not isolate the quality of its substructures. These substructures may differ in their contributions to the performance of the entire model. While some substructures may be useful, others may negatively impact the model performance; moreover, the utility of these substructures as building blocks for producing new superstructures (better models) has been a subject of intense debate [2, 7]. Therefore several studies report that subtree crossover, which exchanges these substructures, is often ineffective [7] and largely produces models that are significantly worse than their parents [4].

A further talking point in GP has been the optimisation of constants or coefficients in the evolving expressions [1, 3] . Typically, GP based symbolic regression does not use numerical methods that are often employed in Machine Learning (ML) to tune numerical constants (parameters), and instead relies on evolution to manufacture the constants. This is challenging because each expression requires bespoke constants; however, in standard GP, which does not use some form of *lifetime learning* [1, 3], both the expression and the constants are produced together. Future improvement of these constants may take genetic adaptation over several generations, and requires that the context [6] of the evaluation of these constants does not change due to genetic operators. Therefore, some researchers have even pointed out that while overfitting is a problem for every ML method, GP must first avoid underfitting [5].

Despite these shortcomings GP has thrived as a strand of ML that produces symbolic models that can even outperform competing methods from ML or Statistics. One potential reason behind the success of GP is that it can automatically engineer features as hierarchical substructures; in contrast, many standard ML/statistical methods require the user to pre-specify the features.

This paper acknowledges the relative strengths of the competing methods and instead proposes a collaborative approach whereby GP evolves features for symbolic regression tasks that are then combined via Multiple Linear Regression (MLR) into a useful model. The proposed method, MLR-GP, employs a customised tree representation that designates various subtrees as features, and then MLR optimises the coefficients of these features.

However, neutralising the effect of bad features can encourage code bloat because if these features are not penalised they can still join in. To counter this, this study uses the well known **Adjusted R^2** (AR^2) (regression-related) measure as the fitness function of MLR-GP; AR^2 decreases if an additional feature does not improve the fitness of a model significantly.

2 THE PROPOSED MLR-GP

Representing Expressions: MLR-GP modifies the standard tree representation of SR models to auto designate subtrees as features. Figure 1 illustrate a sample MLR-GP tree. The new type of node labeled *tag* acts as a placeholder that is not evaluated. This placeholder was defined with an arity of two and can either branch out further creating more placeholders or contain a feature directly below it. *Tag* nodes are constrained to have only *tag* nodes as parents.

Crossover can either recombine the entire features (or sets of features) that can act as building blocks, or exchange genetic material within the individual features.

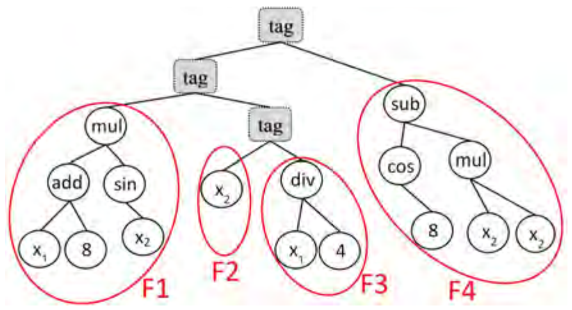


Figure 1: An MLR-GP tree representing a model made up of features. F1 - F4 are subtrees that are treated as features.

Fitness Evaluation Function in MLR-GP: The fitness evaluation in MLR-GP requires identifying the features from a tree. Each identified feature is then evaluated with the training data. The output of the feature evaluation is a design matrix (one column for each feature) that MLR uses to tune the coefficients of each feature (regression parameters) and produce a model. For example, the expression in Figure 1 will return the following final model:

$$Y = \beta_0 + \beta_1 F_1 + \beta_2 F_2 + \beta_3 F_3 + \beta_4 F_4$$

where Y is the output of the model, β_0 is the intercept, β_1 to β_4 are the coefficients of features F_1 to F_4 respectively.

3 RESULTS

Six regression problems were used to compare the proposed MLR-GP method against (1) plain MLR, (2) standard GP (std-GP), and (3) GP with linear scaling (LS-GP). The outcome is as follows:

Accuracy: MLR-GP showed remarkable gains in accuracy on both the training and test data. MLR-GP gained up to 1304% over the best result of the other GP methods and up to 542% over MLR.

Size: MLR-GP methods is producing larger individuals than both std-GP and LS-GP. However, the growth in size is accompanied by growth in training accuracy, whereas bloat is typically viewed as code growth in the absence of any fitness improvement.

Speed: MLR-GP is able to exceed the accuracy of other compared GP methods in early generation of its run; the high accuracy is also achieved with smaller sized models. By the fifth generation, MLR-GP was able to at least match the accuracy of the other compared GP methods at generation 50.

Adjusted R^2 is Controlling Bloat: MLR-GP with AR^2 as fitness measure is suppressing the unwanted growth in the number of features. The average number of features of MLR-GP with AR2 is consistently less than that in MLR-GP with Normalised Mean Square Error.

Crossover Effect: Crossover with MLR-GP produced offspring with fewer negative improvements in fitness values on all tests. The method increases the efficiency of the crossover operator by significantly decreasing the rate of post-crossover fitness degradation. This is a contributing factor for the observed higher training speed in MLR-GP than in the compared GP methods.

4 CONCLUSIONS

This study combines the strengths of two methods, that is, GP and MLR to produce a system that trains faster and better than either of the two systems alone. The study shows that due to its innate innovative nature, GP is effective as a feature engineer when it is aided by numerical methods such as Multiple Linear Regression. Therefore, instead of pitting GP against the other methods, it is more fruitful to collaborate with them.

The combination of methods not only improved the accuracy of the results, but also improved the internal dynamics of a major evolutionary operator – crossover. Normally deemed as disruptive because it often lowers fitness, crossover in the proposed GP-MLR methods became much more robust as the number of fitness degradations decreased substantially. This study thus adds to the literature on the nature of crossover in Genetic Programming.

REFERENCES

[1] Cesar L. Alonso, Jose Luis Montana, and Cruz Enrique Borges. 2009. Evolution Strategies for Constants Optimization in Genetic Programming. In *21st International Conference on Tools with Artificial Intelligence, ICTAI '09*. IEEE, Newark, NJ, USA, 703–707. https://doi.org/10.1109/ICTAI.2009.35

[2] Peter J. Angeline. 1997. Subtree Crossover: Building Block Engine or Macromutation?. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Koza et al (Ed.). Morgan Kaufmann, Stanford University, CA, USA, 9–17. http://ncra.ucd.ie/COMP41190/SubtreeXoverBuildingBlockorMacromutation_angeline_gp97.ps

[3] Raja Muhammad Atif Azad and Conor Ryan. 2014. A Simple Approach to Lifetime Learning in Genetic Programming based Symbolic Regression. *Evolutionary Computation* 22, 2 (jul 2014), 287–317. https://doi.org/10.1162/EVCO_a_00111

[4] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. 1998. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA. https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X

[5] Shu-Heng Chen and Tzu-Wen Kuo. 2003. Overfitting or Poor Learning: A Critique of Current Financial Applications of GP. In *Genetic Programming, Proceedings of EuroGP'2003 (LNCS)*, Ryan et al (Ed.), Vol. 2610. Springer-Verlag, Essex, 34–46. https://doi.org/10.1007/3-540-36599-0_4

[6] Hammad Majeed, Conor Ryan, and R. Muhammad Atif Azad. 2005. Evaluating GP schema in context. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, New York, NY, USA, 1773–1774.

[7] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 2008. Semantic Building Blocks in Genetic Programming. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008 (Lecture Notes in Computer Science)*, O'Neill et al (Ed.), Vol. 4971. Springer, Naples, 134–145. https://doi.org/10.1007/978-3-540-78671-9_12

Improving the Generalisation of Genetic Programming Models with Evaluation Time and Asynchronous Parallel Computing

Aliyu Sani Sambo
School of Computing & Digital Tech.
Birmingham City University, UK
aliyu.sambo@mail.bcu.ac.uk

R. Muhammad Atif Azad
School of Computing & Digital Tech.
Birmingham City University, UK
atif.azad@bcu.ac.uk

Yevgeniya Kovalchuk
School of Computing & Digital Tech.
Birmingham City University, UK
Yevgeniya.Kovalchuk@bcu.ac.uk

Vivek Padmanaabhan Indramohan
Birmingham City University, UK
vivek.indramohan@bcu.ac.uk

Hanifa Shah
Computing Eng. & Built Environment
Birmingham City University, UK
hanifa.shah@bcu.ac.uk

ABSTRACT

In genetic programming (GP), controlling complexity often means reducing the size of evolved expressions. However, previous studies show that size reduction may not avoid model overfitting. Therefore, in this study, we use the evaluation time — the computational time required to evaluate a GP model on data — as the estimate of model complexity. The evaluation time depends not only on the size of evolved expressions but also their composition, thus acting as a more nuanced measure of model complexity than size alone. To constrain complexity using this measure of complexity, we employed an explicit control technique and a method that creates a race condition. We used a hybridisation of GP and multiple linear regression (MLRGP) that discovers useful features to boost training performance in our experiments. The improved training increases the chances of overfitting and facilitates a study of how managing complexity with evaluation time can address overfitting. Also, MLRGP allows us to observe the relationship between evaluation time and the number of features in a model. The results show that constraining evaluation time of MLRGP leads to better generalisation than both plain MLRGP and with an effective bloat-control.

CCS CONCEPTS

• **Computing methodologies → Classification and regression trees**; **Genetic programming**; **Artificial intelligence**.

KEYWORDS

genetic programming, generalisation, regression, complexity

ACM Reference Format:
Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2021. Improving the Generalisation of Genetic Programming Models with Evaluation Time and Asynchronous Parallel Computing. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3449726.3459583

1 INTRODUCTION

It has always been an important challenge in machine learning (ML) to avoid generating models that fit the training data very well but without generalising to the unseen data; this is termed overfitting. Often these overfitting models are overly complex [4]; however, determining how much complexity is just enough is a challenge. Another traditional concern in GP is complexity, which is often manifested by a tendency to grow model sizes to a point that renders the evolutionary search process ineffective. The most popular approach to controlling complexity in GP is bloat control, that is to limit the growth in size of the evolved expressions. However, previous studies have shown that bloat control alone does not always overcome the model overfitting problem [2]. This begs the question: is bloat control really complexity control?

To address the above limitation in bloat control, recent literature [6–8] has proposed alternative approaches to control the computational complexity of models in GP. Instead of using size as a measure of complexity, they use the evaluation time — the computational time it takes to evaluate a GP model on data. The use of evaluation time as a measure of complexity is built on the observation that a model that is made up of computationally expensive building blocks or that has large structures takes a long time to be evaluated, and hence it is computationally complex. The work in [6] empirically shows how the functional and structural complexity are different by plotting the evaluation times of identically sized but functionally diverse GP models, see a reproduction in Figure 1. Therefore, if the evaluation time of evolving models are constrained then the growth in the structural as well as the functional complexity will be discouraged. The same work also recommended various techniques to significantly minimise the noise in measuring evaluation times. This paper adopts the use of all these recommendations to measure complexity of the evolving models in MLRGP.

The next question is how to control the evaluation times. We used two approaches to control evaluation times. First, we use an effective bloat control method to discourage high evaluation times in the same way it discourages large size, named Time-control (TC). The second approach takes a simple view: induce a *race* among competing models, named the *Asynchronous Parallel Genetic Programming* (APGP) [6][8]. With APGP, the faster models can (if their fitness is competitive) join the breeding population before their slower counterparts and gain an evolutionary advantage.

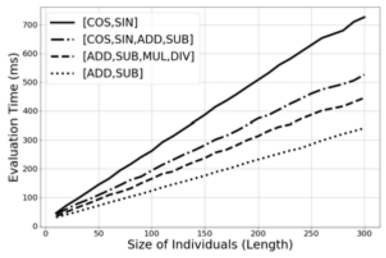


Figure 1: Size and composition affect evaluation times. Higher average evaluation times were returned by individuals made up of COS and SIN operators than same-sized individuals made up of simpler functions sets.

Method	Test Fitness	Size	Evaluation Times	No. of Features
Time-Control Success				
STD	9/10	10/10	10/10	10/10
BC	7/10	9/10	10/10	10/10
APGP Success				
STD	9/10	10/10	10/10	10/10
BC	7/10	0/10	0/10	0/10

Table 1: Summary of the test for significance in difference. The figures show the fraction of the tests where TC and APGP produced significantly better results, respectively.

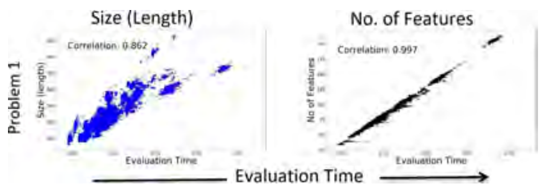


Figure 2: The correlation between evaluation time and the number of features is greater than it is with size.

We used a GP system that is aided by Multiple Linear Regression (MLR) [5] in our experiments. Such *MLRGP* systems [3] have become increasingly popular lately because they improve the training performance significantly; this is because the traditional GP often underfits the data because it can not efficiently generate numeric constants [1]. As this improved training accuracy can result in serious overfitting [5], MLRGP is suitable for studying the effect of controlling complexity with evaluation time on overfitting. With this system, we compare the performance of the two methods that restrict evaluation time with plain MLRGP (STD) and with MLRGP combined with an effective bloat control technique (BC).

2 RESULTS

Ten widely used datasets were selected as test problems. We compared test fitness scores and our three indicators of complexity: size, evaluation times and the number of features. The Mann-Whitney U test was used to determine the significance of the difference in the final populations and the results are summarised in Table 1.

In terms of test-fitness accuracy (generalisation), the evaluation times methods (TC and APGP) prevailed over STD and BC. Also, they had matching results in terms of the number of tests they prevailed; they both produced significantly higher test scores in 9 out of 10 tests against STD and 7 out of 10 against BC. However, the two time control methods differed in how they handled complexity (size, evaluation times and the number of features). TC produced significantly simpler solutions against BC and STD with one exception out of 60 tests, this is despite TC and BC used the same techniques to control time and size, respectively. APGP complexity control was gentler; it produced significantly simpler solutions than STD in all tests but more complex solutions than BC.

In addition to the effective control of the number of features by TC and APGP, the final populations that MLRGP produced showed a stronger correlation between the evaluation times and the number of features (average of 0.996) than between evaluation times and the sizes (average of 0.843), see a representative example in Figure 2.

3 CONCLUSION

We showed that the evaluation time behaves differently from size. We demonstrated that it can discriminate between the size, the complexity of the components, and the number of features of the MLRGP individual. Also, the results asserts that using the evaluation time to mange complexity leads to better generalisation. Thus, this approach promises to be broadly applicable. Overall, this study highlights the feasibility and merits of using the evaluation time.

REFERENCES

[1] R. Muhammad Atif Azad and Conor Ryan. 2014. The Best Things Don't Always Come in Small Packages: Constant Creation in Grammatical Evolution. In *17th European Conference on Genetic Programming (LNCS, Vol. 8599)*, M. Nicolau et al. (Ed.). Springer, Spain, 186–197. https://doi.org/10.1007/978-3-662-44303-3_16

[2] Raja Muhammad Atif Azad and Conor Ryan. 2014. A Simple Approach to Lifetime Learning in Genetic Programming based Symbolic Regression. *Evolutionary Computation* 22, 2 (June 2014), 287–317. https://doi.org/10.1162/EVCO_a_00111

[3] Amir Hossein Gandomi and Amir Hossein Alavi. 2012. A new multi-gene genetic programming approach to nonlinear system modeling. Part I: materials and structural engineering problems. *Neural Comput. Appl* 21, 1 (2012), 171–187.

[4] Gregory Paris, Denis Robilliard, and Cyril Fonlupt. 2003. Exploring Overfitting in Genetic Programming. In *Evolution Artificielle, 6th International Conference (Lecture Notes in Computer Science, Vol. 2936)*, Pierre Liardet et al. (Ed.). Springer, Marseilles, France, 267–277. https://doi.org/10.1007/b96080

[5] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Feature Engineering for Improving Robustness of Crossover in Symbolic Regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (Cancún, Mexico) *(GECCO '20)*. ACM, NY, USA, 249–250. https://doi.org/10.1145/3377929.3390078

[6] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Leveraging Asynchronous Parallel Computing to Produce Simple Genetic Programming Computational Models. In *The 35th ACM/SIGAPP Symposium On Applied Computing*, Federico Divina and Miguel Garcia Torres (Eds.). ACM, Brno, Czech Republic, 521–528. https://doi.org/10.1145/3341105.3373921

[7] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Time Control or Size Control? Reducing Complexity and Improving Accuracy of Genetic Programming Models. In *Genetic Programming - 23rd European Conference, EuroGP 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15-17, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12101)*, Ting Hu et al. (Ed.). Springer, Berlin, Heidelberg, 195–210. https://doi.org/10.1007/978-3-030-44094-7_13

[8] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2021. Evolving simple and accurate symbolic regression models via asynchronous parallel computing. *Applied Soft Computing* 104 (2021), 107198. https://doi.org/10.1016/j.asoc.2021.107198