Genetic Programming Bias

with

Software Performance Analysis

Brendan Cody-Kenny

A Dissertation submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

January 2015

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Brendan Cody-Kenny

Dated: January 29, 2015

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this Dissertation upon request.

Brendan Cody-Kenny

Dated: January 29, 2015

Acknowledgements

This thesis would not have been possible without the support and guidance of many people.

I would like to thank my supervisor, Stephen, for his help. His support and encouragement has helped me get this work to where it is. My research outlook has been also shaped by everyone who has been through the lab over the years. Stefan, Edgar, Michael, Ciaran, Dave, Ricardo, Serena, Colm, Paul, Steve, Sandra, Andriana and Paul D. have all been a great help both technically and personally during this work. A whole host of others have shaped the experience over the years within TCD, too many to name here. I would like to thank the SBSE community who have shaped this thesis greatly through conferences and papers. Needless to say, I am in support of their cause and thank them all for their efforts.

To Lorna whose determination has spurred my own. To my family, especially my brother and sister, thanks for always being there. Finally, to my parents a special thank you, for this thesis and so much more would not have been possible.

Brendan Cody-Kenny

University of Dublin, Trinity College

January 2015

Abstract

The complexities of modern software systems make their engineering costly and time consuming. This thesis explores and develops techniques to improve software by automating re-design. Source code can be randomly modified and subsequently tested for correctness to search for improvements in existing software. By iteratively selecting useful programs for modification a randomised search of program variants can be guided toward improved programs. Genetic Programming (GP) is a search algorithm which crucially relies on selection to guide the evolution of programs. Applying GP to software improvement represents a scalability challenge given the number of possible modification locations in even the smallest of programs.

The problem addressed in this thesis is locating performance improvements within programs. By randomly modifying a location within a program and measuring the change in performance and functionality we determine the probability of finding a performance improvement at that location under further modification.

Locating performance improvements can be performed during GP as GP relies on mutation. A probabilistic overlay of bias values for modification emerges as GP progresses and the software evolves. Measuring different aspects of program change can fine-tune the GP algorithm to focus on code which is particularly relevant to the measured aspect. Measuring execution cost reduction can indicate where an improvement is likely to exist and increase the chances of finding an improvement during GP.

Contents

Acknowledgements **iv**

Abstract **v**

List of Tables **viii**

List of Figures **ix**

Chapter 1 Introduction **1**

1.1 Challenge 2
1.2 Approach 2
1.3 Problem 3
1.4 Hypothesis 4
1.5 Self-Focusing Genetic Programming 5
1.6 Research Question 7
1.7 Evaluation 7
1.8 Findings 8
1.9 Contributions 10
1.10 Reader's Guide 10

Chapter 2 Related Work **12**

2.1 Overview 12
2.2 Improving Performance 14

2.2.1 Configuration 15
2.2.2 Compilers & Interpreters 16
2.2.3 Trading Functionality for Performance 17
2.3 Genetic Improvement 19
2.3.1 Representation 20
2.3.2 Modification 21
2.3.3 Fitness Functions 22
2.3.4 Code Reuse 25
2.3.5 Operators 27
2.3.6 Search Space 27
2.4 Guiding GP 28
2.4.1 Self-Adaption 29
2.4.2 Location Learning 33
2.5 Dynamic Individual-level Node Selection Bias in GP 35
2.5.1 Individual Level Adaption 36
2.5.2 Gaussian Bias 36
2.5.3 Node Selection Bias 37
2.5.4 Fitness Guided Node Selection 38
2.5.5 Summary 38
2.6 Analysing Software 38
2.6.1 Profilers 39
2.6.2 Performance Root-Cause Diagnosis 42
2.6.3 Sensitivity Analysis 44
2.6.4 Program Spectra 44
2.6.5 Mutation Analysis 45
2.7 Summary 47

Chapter 3 Self-Focusing GP for Software Performance Improvement 49

3.1 Design Rationale 50

3.2 Genetic Programming Configuration 53
3.2.1 Representation 54
3.2.2 Node Selection 55
3.2.3 Operators 55
3.2.4 Fitness Function 56
3.2.5 Selection 60
3.2.6 Elitism 60
3.3 Self-focusing Genetic Programming 61
3.3.1 Design Methodology 62
3.3.2 Bias Values 62
3.3.3 Value Initialisation 62
3.3.4 Modification 62
3.3.5 Updating Node Bias During GP 64
3.4 Gaussian Bias 65
3.5 Summary 65

Chapter 4 Evaluation **66**
4.1 Overview 66
4.2 Improving Programs with GP 68
4.2.1 Problem Set 74
4.3 Statistical Comparison 76
4.4 Static Bias 77
4.4.1 Hand-Made Bias 77
4.4.2 Profiler-Derived Bias 80
4.4.3 Mutation-Derived Bias 84
4.5 Dynamic Bias 88
4.5.1 Static VS Dynamic 90
4.5.2 Deceptive Problem 92
4.5.3 Generality of Dynamic Bias 93

4.5.4 Trend Analysis . . . 95
4.5.5 Gaussian Bias . . . 102
4.6 Mutation Spectrum Analysis . . . 105
4.6.1 Exhaustive Modification . . . 105
4.6.2 Fitness Analysis . . . 106
4.6.3 Results . . . 107
4.7 Threats to validity . . . 110
4.8 Summary . . . 111

Chapter 5 Conclusion **112**
5.1 Review of the Research Question . . . 113
5.2 Overview of the Evidence . . . 114
5.3 Review of Contributions . . . 116
5.4 Future Work . . . 116
5.4.1 Analysis . . . 117
5.4.2 Bias Update Rules . . . 117
5.4.3 Problem Set . . . 118
5.4.4 Broader Work . . . 119
5.4.5 Open Questions . . . 119
5.5 Closing Remarks . . . 120

Bibliography **121**

Appendix A Implementation: locoGP **139**

Appendix B Code Listings **142**
B.1 Bubble Sort . . . 142
B.2 Shell Sort . . . 143
B.3 Selection Sort . . . 144
B.4 Selection Sort 2 . . . 145
B.5 Radix Sort . . . 145

B.6 Quick Sort 146
B.7 Merge Sort 148
B.8 Deceptive Bubble Sort 150
B.9 Insertion Sort 151
B.10 Heap Sort 151
B.11 Cocktail Sort 154
B.12 Huffman Codebook Generator 155

Appendix C Bias Rules **160**

C.0.1 Bias Allocation Rules 160

Appendix D Absolute Values for Dynamic Bias **165**

D.1 Insertion Sort 166
D.2 Heap Sort 167
D.3 Merge Sort 168
D.4 Radix Sort 169
D.5 Selection Sort 170
D.6 Selection Sort 2 171
D.7 Shell Sort 172
D.8 Quick Sort 173
D.9 Cocktail Sort 174
D.10 Huffman Codebook 175

List of Tables

3.1 Baseline GP Configuration Summary 54

4.1 Problem improvement overview 75
4.2 Profiler-Derived Bias Values for Bubble Sort 81
4.3 Mutation-derived Node Rankings for Bubble Sort 86
4.4 Size of Program and Minimum Change Required for Improvement 96
4.5 Average Ranking for Improvement Nodes when Modified 108

C.1 Rulesets and the Movement of Top Ranked Nodes 164

List of Figures

3.1 Fitness Function 57
3.2 Functionality Score 59

4.1 Fitness Scatter and Average for GP on Bubble Sort 71
4.2 Best individual programs for GP on Bubble Sort 72
4.3 GP with Hand-Made Outer-Loop Bias on Bubble Sort 78
4.4 GP with Hand-made Inner-Loop Bias on Bubble Sort 79
4.5 GP with Profiler-Derived Bias on Bubble Sort 81
4.6 Profiler-Derived Bias Difference on Bubble Sort 82
4.7 GP with Profiler-Derived Bias Difference on Deceptive Bubble Sort 83
4.8 GP with Mutation-Derived Bias on Deceptive Bubble Sort 87
4.9 Profiler and Mutation-Derived Bias Difference on Deceptive Bubble Sort . 88
4.10 GP with Dynamic Bias Allocation on Bubble Sort 90
4.11 Dynamic and Static Derived Bias Difference on Bubble Sort 91
4.12 GP with Dynamic Bias Allocation on Deceptive Bubble Sort 92
4.13 Dynamic Mutation-Derived Bias Across All Generations 94
4.14 Dynamic Mutation-Derived Bias Ordered by Problem Size 98
4.15 Dynamic Mutation-Derived Bias Ordered by Improvement Size 99
4.16 Dynamic Mutation-Derived Bias Ordered by Improvement Size over Problem Size 100
4.17 Dynamic Mutation-Derived Bias Ordered by Programs Discarded 101

4.18 Canonical GP against Gaussian Bias 103
4.19 Gaussian Bias against Dynamic Mutation-Based Bias 104

D.1 GP with Dynamic mutation-derived bias on Insertion Sort 166
D.2 GP with Dynamic mutation-derived bias on Heap Sort 167
D.3 GP with Dynamic mutation-derived bias on Merge Sort 168
D.4 GP with Dynamic mutation-derived bias on Radix Sort 169
D.5 GP with Dynamic mutation-derived bias on Selection Sort 170
D.6 GP with Dynamic mutation-derived bias on Selection Sort 2 171
D.7 GP with Dynamic mutation-derived bias on Shell Sort 172
D.8 GP with Dynamic mutation-derived bias on Quick Sort 173
D.9 GP with Dynamic mutation-derived bias on Cocktail Sort 174
D.10 GP with Dynamic mutation-derived bias on Huffman Prefix Codebook . . 175

Chapter 1

Introduction

People, as software's chief maintainers currently, have a difficult task in ensuring software meets changing requirements. The complexity of contemporary software systems has limited the effectiveness of traditional engineering processes when applied to large software projects motivating the emergence of more flexible development models. To support the development process, many automated tools exist. As developer time and expertise are costly, further automation in this area would benefit the software engineering discipline and industry.

Randomised search algorithms have been proposed for their potential to provide solutions in the area of Software Engineering. Randomised algorithms are particularly interesting for program improvement as they are not restricted by the type of program modifications performed. While random modification can generate programs which do not compile, evaluating a wide range of program variants increases the chances that an improved program will be found.

Search algorithms have been used for various SE tasks. In software testing search has been used for the generation of test data [88]. Searching through the large number of possible compiler options to find a configuration that produces a program of reduced size [28] and to find a program with reduced execution cost [51, 137] has also been demonstrated.

The use of a particular search algorithm, Genetic Programming (GP) [106], has been proposed for [26, 44] and demonstrated on [108] software improvement tasks such as

bug-fixing [140] and performance improvement [73, 146].

GP is an algorithm which relies on *random modification* to search through many variants of a program. Repeated random modification produces program variants ranging in performance and functionality. Even though programs are created by random, and sometimes destructive modifications, the algorithm is not considered random search. Program evaluation and selection is crucial to the progress of the algorithm toward more improved programs. Provided programs can be evaluated and measured for fitness for purpose, the best program variants can be selected to produce further program variants. How programs are evaluated determines the direction of the overall search process during GP. Over many generations of randomly modified programs, selection provides guidance as to which programs are considered more improved relative to others. Over time, the algorithm is expected to increase the chances of finding a more improved program variant.

Performance improvement of programs is the main concern of this work, and so the number of operations executed [42] is used as performance measure.

1.1 Challenge

Scaling GP to larger programs poses a challenge. Larger programs provide an increased number of possible places which can be modified. As more modification points are possible in larger programs the search process is closer to random search due to the increased number of possible variant programs [85]

1.2 Approach

The goal in much GP work is to navigate the program search space more efficiently by techniques designed to generate only programs which are more likely to be improved. It is appropriate to think of scaling a problem down or “shaping” a problem so that progress can be made in finding a solution using a GP algorithm. To shape a problem we seek to focus code modifications on locations within a program likely to produce improved variants.

Although program evaluation and selection provide guidance as to what programs are modified in canonical GP, there is no guidance for *where* within a program should be modified.

In canonical GP all nodes in a program are equally likely to be selected for modification. To provide a more directed search, modification can be focused on certain locations in a program which are deemed interesting. Modification can be focused by controlling the probability of each location being selected. By attaching values to each node in the program we can vary the probability of each location being selected [6].

Bias values can be changed *dynamically* during GP, or *statically* set before GP is run. During GP, bias can be changed entirely at random [6], in response to program evaluation [49] or derived from program measures such as size [53].

Program analysis techniques [29, 57] have been used to set bias statically before the application of GP. Fault localisation techniques have been used to highlight the location of bug fixes in software [140] and profiling techniques have been used to guide performance improvement [73].

One example of the use of profiling is to vary the size of program input to observe how the execution frequency of lines of code varies accordingly [73]. However, if a 100 line program starts by setting a variable which is subsequently read during a loop at line 90 to determine how many iterations are performed, the cause of execution cost of the loop is not at the same location as the loop itself. Although this example is simplistic, it illustrates that applying change at the loop may not alleviate the problem. The cause of an execution bottleneck in code may be due to complex code with many interdependencies. Finding what code is most likely relevant (or contributes the most) to execution cost is therefore not trivial.

1.3 Problem

The problem addressed in this thesis is locating performance improvements. We wish to find the locations where modification is likely to produce a program variant with reduced

execution cost.

Our problem can be posed as: How can the cause of execution be attributed to locations in code. The cause of execution cost is difficult to attribute to an exact location (or multiple locations) in a program as all code executed contributes to the execution cost of a program. Source code elements can embody interdependencies within a program making it difficult to definitively attribute execution cost directly to a code element.

1.4 Hypothesis

The central hypothesis in this work is that repeated modification can indicate how likely a performance improvement is to be found at each location in a program.

As GP makes many random modifications to generate variant programs, there is a learning opportunity in how modification effects a programs behaviour. By making a modification, we can associate the measurable change in program functionality and performance with the location of that modification, ignoring what the code modification was. We propose that these measurable changes can be used to highlight the location of performance improvements.

Our hypothesis is based on the assumption that random modification to a program at a specific location will produce a variety of programs with characteristics which are specific to that location in the program. If this is true, then modification points in a program should be uniquely distinguishable under modification.

If the assumptions and hypothesis are correct, then repeated modification to a program should highlight the locations of performance improvements. We can perform repeated modification to allocate bias values to different locations in a program. Once bias values are found, they can be used statically to observe how they influence the GP process. We should see improved program variants appearing earlier in the GP process if the bias values highlight improvements.

As this approach relies on the same iterative modification and evaluation mechanism as GP, it should also be possible to learn about locations dynamically and have a

beneficial impact on solution finding during the GP algorithm itself.

1.5 Self-Focusing Genetic Programming

Given an existing program, code elements within the program can be modified at the language level allowing variables, operators, expressions and statements to be modified. A modification can be a deletion, clone or replacement from the set of elements in the program. Compilable programs that result from this type of modification can be tested or “evaluated” for correctness and execution cost. Our proposed solution builds on the iterative modification and evaluation as performed during canonical GP.

The result of program modification is used to infer information about the modification location in our solution. Based on how functionality and performance change, we can increase or decrease the bias value at the location of code modification which produced the variant program.

Repeating this process eventually changes bias values for all nodes in a program. This culminates in a learning process that gathers information from measuring the effect of repeated random modification of a program. The information gathered highlights to which parts of a program more random modification should be applied. Although there is a random element to the selection of nodes for modification, the random selection is biased per the node values.

Initially, bias values are set to the maximum allowed value of one for each modifiable node in a program. To produce a variant “child” program, a clone of the parent program is modified. When the parent is cloned, all bias values are cloned to the child. When the child program is modified bias values propagated depending on the type of modification performed. If code is replaced, then values for the replacement code are initialised to one. If code is deleted, then the bias values are lost.

If modification is performed at a location with value one, it is assumed that the location has not been modified before, and the value is reduced by default to less than one. This means that bias values have the effect of forcing GP to explore at least one

modification for each location in a program.

After the cloned child program is modified, the program is evaluated to ascertain performance and correctness measures. Bias values for the location of modification are updated depending on how the performance and correctness differ for the child program in comparison to the parent program. Bias values are updated in both the parent and child program as both have a chance of being modified again in the evolutionary process.

We update bias values in a number of ways in response to the different results attainable when a program is evaluated. Bias is increased and decreased so that after the initial modification, it is always between zero and one. The bias value is increased by no more than half the difference between the existing value and one. $V = V + (1-V)/2$ For a decrease, the change is similarly, $V = V - (V/2)$.

If a program variant does not compile, the value at the location of modification is reduced. If the program compiles, and the child program evaluates with a lower functionality measure than the parent, the bias value at the location of modification in the parent is increased, and decreased in the child. The rational behind this update strategy is that if we modify the program and it compiles then it is worth modifying there again in the parent. If the functionality increases, then the chances of modifying that location are reduced in the parent, and increased in the child.

The bias update rules are designed to focus change on exploring the functionality variants of the program which have reduced functionality. Destructive modifications are favoured, with a focus on exploration of variants. Modification tends to focus where code has broken before.

To achieve this, functionality decreases lead to bias value increase in the parent program, and bias decrease in the child program. When the functionality measure increases between parent and child, the bias is decreased in the parent and increased in the child. The concept here, is to guide change toward exploration of the program in an attempt to escape local optima, as provided by the original program.

1.6 Research Question

Our research question is posed as follows:

- Can differences in functionality and performance measures caused by arbitrary program modifications indicate the likelihood that further modification will create a program with reduced execution cost?

1.7 Evaluation

In this thesis we are interested in answering the research question to understand how GP can be guided. The GP process is conducted with and without bias to observe the effect bias has on the chances of finding improved programs. We also seek to inspect what change in fitness measures are particularly good indicators for improvements. Bias allocation is also inspected in a static and dynamic context.

Static bias refers to bias values which are attached to a program before GP is started. The values do not change during GP. Profiling techniques have been used to find this bias. In this thesis we find a static bias by repeated mutation on a program which is to be improved. Once this bias is found, it can be attached the seed and GP can be applied. If the bias accurately highlights locations where an improvement exists, then GP should find an improved version of the program more quickly. If mutation-derived bias can find improvements in a program, then we should see GP runs finding these improvements more quickly. Where improvements are known to exist in a program we can observe the bias values (and thus likelihood of a node being modified) that is produced under various bias schemes such as profiling and random bias updates.

During GP, modifications are performed on many variant programs which change from one generation to the next with bias being inherited from parent programs. *Dynamic bias* refers to repeat mutation and crossover on a number of programs which change over time. Thus, dynamic bias is derived during the GP algorithm where bias is inherited from parent to child programs. When used during GP, the range of programs in which

bias is updated is more varied than when dealing with a single program. Although bias is inherited, the number of modifications to any one program is less than when applied to a single program. For this reason, we expect dynamic bias during GP to be less specific to any one program and be somewhat general as it is updated in a number of program variants, each providing a different context for changing the bias. Although this environment is noisy, we want to see if bias can be beneficial to GP when allocated as a by-product of mutation and crossover during GP. Although modifications are performed on many different programs during GP, we wish to see if there is still enough information summarised in bias values to increase the chances of finding improved program variants.

1.8 Findings

We apply GP to a number of sort algorithms and a Huffman codebook algorithm. The test programs are relatively small, with the largest containing 115 lines of code, and have a very specific definition of functionality. We can characterise them as having a high level of interdependencies between source code elements. Although these programs are small they require a reasonable effort to evolve and improve given that any single change is likely to degrade the program. Regardless of this search space GP can find improvements in all of these programs

We repeatedly mutate a Bubble Sort algorithm to derive bias, and then apply GP using this bias for modification location selection. The static bias guides GP to an improved version of this program more quickly than canonical GP which selects locations for modification with equal probability across all locations in a program.

We use a profiler to derive bias on the same problem and compare to the use of static bias. Profiler derived bias increases the chances of finding improved variants over the use of mutation-derived bias on Bubble Sort.

We further compare profiler and mutation derived bias on a hand-modified deceptive version of Bubble Sort which introduces a redundant iteration over the whole Bubble Sort algorithm. The idea in using this problem is to separate the location of a potential

improvement from code which is executed most frequently when the program is run. The outer loop is only executed twice, while the swap function is executed many times during execution. On this problem a profiler reduces the chances that GP will find both possible improvements. Mutation-derived bias is not deceived in this case and improves on canonical GP as well as the use of a profiling technique.

Dynamic bias is compared against canonical GP across a range of sort problems and a Huffman codebook generator problem to inspect generalisability of our approach. We find that, while dynamic bias does not increase GP's chances on Bubble Sort, dynamic bias increases the chances of finding program improvements over canonical GP on 7 out of 12 problems through a large number of generations, but is eventually overtaken by canonical GP on 2 of these problems. This shows that there is a distinction between static and dynamic derivation of bias, and also that the bias rule we have presented does not generalise across all problems.

We compare dynamic mutation-derived bias with randomly modified bias and find that dynamic bias outperforms random bias on half of the problems. This provides evidence that explicit credit assignment can be beneficial during GP [6].

To understand why dynamic bias does not generalise across problems, an exhaustive modification of each of our test problems is performed. For each location in a program, all possible single modifications are performed where modification involves attempting to replace each location with all other code elements in the program. This produces a large number of program variants, of the order n^2 - n, where n is the number of modification points in the program. Analysing the evaluation results for each location can indicate the range of values likely to be encountered as locations are modified. We find that the evaluation results across many modifications are distinct for each location. Furthermore, we find that summarising compilation count, functionality and performance change (as opposed to using a single one of these measures) may be additionally useful in determining the location of performance improvements.

1.9 Contributions

In summary, the contributions of this thesis are:

a Propose repeated program modification and evaluation as a program analysis technique for the location of performance improvements.

b Demonstrate that mutation derived bias can improve the chances that GP will find performance improvements.

c Demonstrate that improvements can be found during GP.

d Demonstrate that profiler-derived bias can be deceived as to the location of performance improvements on a hand-crafted deceptive problem.

e Demonstrate that mutation-derived bias (or explicit credit assignment) can improve GP more quickly than randomly changing bias.

In terms of Software Engineering, our contribution could be stated as execution cause localisation through the analysis of program output over many program variants. When applied to GP, our contribution culminates in a learning mechanism during GP for performance improvement of programs.

1.10 Reader's Guide

The next chapter begins with background in software improvement and moves towards stochastic modification of programs, in particular the use of GP. Attention turns to how GP has been guided and subsequently to software analysis for locating performance improvement. The theme throughout revolves around finding *where* performance can be improved. The chapter concludes with a brief summary and problem overview.

Chapter 3, "Design", details the configuration of our GP algorithm and the operation of our bias mechanism. This chapter provides justification for specific design decisions such as individual node bias and bias allocation rules.

Chapter 4, “Results”, describes the methodology by which we answer our research questions and provides experimental results in support of these answers.

Chapter 5, “Conclusion”, summarises the findings of this thesis. The chapter looks at what work can be explored immediately as a direct consequence of this thesis, as well as broader concepts of interest. Chapter 5 and this thesis conclude with speculation on the future of Software Engineering.

Chapter 2

Related Work

This related work chapter begins with background on general performance improvement techniques. We then introduce GP for improvement. We cover methods for guiding GP, which have been proposed to scale the algorithm to larger programs. From general methods of guiding GP, we look at program analysis techniques specifically for locating performance improvements in code. Software analysis techniques for attributing execution cost to locations in a program have been used to guide GP for improvements including performance and are particularly relevant.

2.1 Overview

We focus on performance improvement in the context of the reduction of execution cost of a program. Execution cost can be measured in time taken to execute a program or the number of operations performed in executing a program. Performance improvement is difficult to design for and is not recommended as a primary concern when building software [63]. Program performance improvement requires a deep understanding of a program, and much trial and error can be expended to understanding where an improvement opportunity exists. Performance is implicit in program code, it is difficult to understand by simply looking at code. As GP is an algorithm which evaluates a large number of possible program variants, it appears a good approach to exploring the broad range of

subtle performance nuances in a program.

Much GP work addresses the creation (or "growing") of programs in their entirety.Programs are initially randomly generated from primitives which can be described as domain specific. For example, to evolve a sorting algorithm, a swap function may be specified as a primitive [61]. In these cases evolution may not operate over a Turing complete language.

Work in GP tends to address the generation of program functionality, as opposed to any measure of execution performance. Functionality is problem specific and generally measured as the correctness of the result returned after program execution. A specific metric is usually designed to measure functionality on a graduated scale. For example, the functionality of a sort algorithm could be measured by the number of items that have been moved to their correct ordering.

GP is a randomised algorithm with little restriction on the code modifications that can be made. Generating a wide variety of program variants means a wide range of program improvements are possible. There is an opportunity to improve performance where code is large and difficult to understand.

Program improvement using GP is somewhat distinct in the GP literature as the algorithm begins with an existing program. Much GP work starts from randomly generated programs of very low utility. Improving existing programs means GP is "seeded" with a significant portion of the solution [75]. Program improvement can be thought of as improving a single aspect of an already mostly acceptable program. Attention is focused on program improvement as a large amount of software is in existence and growing large programs appears beyond the current ability of GP.

The terms "guided GP" and "software analysis" have emerged from different research origins, the field of GP (or more generally Evolutionary Computation), and the field of SE. Though these areas have different origins and thrusts, they have conceptual similarities and differences which are mutually applicable. Both utilise the concept of program inspection, potentially at runtime, for the purpose of generating information about particular locations in source code.

Approaches to guidance from the GP literature speed up the generation of programs

from scratch [65]. Many approaches to guiding GP use general program characteristics, such as code size, to guide the algorithm. In much of the literature the emphasis is on the GP algorithm itself as opposed to being specific to any particular class of problems or improvement types [130]. Guidance information can be generated as an integral part of the GP algorithm, with modification bias for programs emerging during the evolutionary process [71] or "dynamically".

Software analysis techniques from the Software Engineering literature have been used to guide GP for specific instances of program improvement, namely bug fixing [142] and performance improvement [73]. Where software analysis techniques have been used to guide GP, they generally have been used prior to the application of GP [73, 141] or "statically". There is nothing preventing these analysis techniques being used continually during a GP run, but it appears that this has been as yet unexplored.

This related work chapter tends toward a central theme of determining *where* to make a modification in a program to improve the chances of generating a program with a reduced number of execution operations. Specifying where to make a change tends toward looser, more probabilistic methods as the discussion progresses. For example, pattern matching is a deterministic method for finding where to replace code in a program. Conversely, locations for modification can be selected entirely at random even disregarding typing information.

The next section introduces background to performance improvement and the application of GP to this task.

2.2 Improving Performance

Performance improvement approaches seek to reduce the execution time or resource usage of a program. The reduction of execution generally requires the modification of a program to perform the same tasks or produce the same results using less or more efficient operations. When automating the improvement of software, ensuring improved programs behave as expected puts a restriction on the program modifications or transforms that

are allowed.

Performance improvement is traditionally done with a restricted set of code transforms which are deemed safe or behaviour preserving in some sense. As performance improvement requires modification in some aspect of program behaviour, it is difficult to formally ensure a program transform will always produce an equivalent program capable of returning the same values [16]. To preserve the correctness of programs in compiler optimisation for example, great attention has been paid to the development of transforms which are considered correct or safe to some degree [138].

If transforms are defined less rigorously, a wider range of program behaviour can be generated [38]. Relaxing the restrictions on transformation type allowed means that we increase the chances of producing defective programs on average. The benfit of unrestricted transformation is that it enables the generation of program variants which are unlikely to be considered otherwise and have some small chance of being greatly improved. Though searching through many program variants is expensive, especially where unrestricted transformations are performed, there is less developer effort needed to rigorously define program transforms.

By way of introducing GP, we review how human-designed improvements are automatically reused. We then move discussion toward methods where program modification or transformations are less rigorously defined. When using less restricted program transforms, the effect of a transform is largely unknown until the transform has been applied and the variant program has been tested for correctness.

2.2.1 Configuration

A program can be designed in numerous ways to produce the same results. Multiple implementations can be included in a program and selected between at runtime. A program's operation may also be modified by changing values outside of those which are considered the program's input data. These "tunable" values can be hard coded in a program's source code, or read in at runtime enabling the tuning of a program after design and before runtime. In both these cases, it is expected that the program must

be specifically designed to allow such post-design and pre-execution configuration of a program [136]. The location of program change is therefore manually defined.

The assumption is that configuration options are in some sense “safe” for use in that they have been previously designed to allow tuning of the program. Automated approaches have used search to find the best configuration options to use for performance [99]. The approach assumes that configuration options can be modularised and exposed succinctly external to a program.

Program configuration can be automated through the application of previously useful configuration changes to other observed programs. Replaying configuration changes [125] can improve the performance of a program. The source code of a program remains the same in these cases, where improvement is achieved through modifying the input data to a program in the form of configuration changes. This does not constitute a re-design of a program’s code.

A change which fixed one machine is recorded and applied for similar problems on different machines [125]. The location of change which fixed a bug previously, can be used to match against similar contexts and the fix can be replayed. This is also a good example of automated code reuse and illustrates how the location of change can be found using pattern matching.

The approach is interesting as it shows the requirement for semantics-preserving transforms is relaxed. It is possible to produce programs with different code through configuration but depends on these code variants being provided manually. Configuration also manually determines what program code is executed.

2.2.2 Compilers & Interpreters

Arguably the most widely used example of automated performance improvement can be found in compilers and interpreters. The term “optimisation” is used in association with the improvements performed by a compiler. While a broad review of optimisations in these areas is out of scope for this thesis, we briefly classify the type of optimisations performed as behaviour preserving. To broadly classify the optimisation types used

in both cases, we may say that optimisations are achieved by code transforms which are required to only affect a narrow aspect of performance, and strive to have little or no effect on program functionality. The application of optimisations is less specifically defined within compilers and interpreters as they are applied through the use of various heuristics [123]. While the type of code transform performed is rigorously defined, the location is less so.

Due to the requirement that compiler optimisations maintain the behaviour of a program, optimisation in this context fall short of program redesign [147]. Compiler optimisations, whether guided or not, can be described as "program tuning" as opposed to "algorithm tuning" [19]. While the difference between compiler optimisation and algorithm redesign is not rigorously defined, we are interested in broadening the type of program improvements which can be automated.

2.2.3 Trading Functionality for Performance

For some program use cases we can trade functionality or allow the degradation of a program for the sake of performance.

One way to improve program performance is to reduce the number of iterations in a loop. While this approach is nearly guaranteed to improve performance, it is equally likely to degrade functionality. "Loop Perforation" trades program functionality for execution speed [118]. This is a direct approach to performance improvement where loops are targeted in a program and modified in such a way as to reduce the number of loop iterations. Transforms are specifically targeted at loop constructs within code.

Variance in accuracy can also be tolerated for pixel shaders [101, 120]. Shader algorithms adjust colour across an image. Shader simplification [101] attempts to find many fine grained program variants which differ by a small amount of performance. Code can be modified to explore the range and extent of functionality and performance trade-offs possible by applying predefined program modifications. The goal of this work is to produce a range of program variants each one having a slightly different performance as evenly distributed across a scale as possible. Variant implementations which differ

slightly in terms of performance can be selected at runtime depending on system load and execution context requirements.

Determining where best to make a a modification in a program can be determined by applying many changes at a location in the program, then selecting which one is the best [101]. No prediction is made as to what modification at what location is best before the program is evaluated. This approach evaluates many programs to find what modification is best similar to the use of a hill-climber algorithm [90] or brood-selection in GP [126]. The location of modification is found by matching the location of loops. The changes made to loops are predefined from a relatively small set of regex-like replacement patterns which target loop modifications. The location and content of the change is pre-written manually. The approach is manually guided but is not semantics-preserving in its mutation of a program.

Relaxing restrictions on the transformation type further, shader simplification can be achieved by mutating a program using the existing lines of code within the program [120]. This automates the specification of code which is used to modify a program essentially leaving the specification of transforms up to the original developer of the program. The code which is used to improve the program was designed as part of the original program and was not specifically created for the purposes of optimising the program. It appears that performance improvements can be found by reordering and recombination of existing code within a program. The location of reordering is random and the code modification type is specified by the lines of code created when the program was written. This work relaxes the specification of what a code modification consists of by reusing code from within the existing program under modification. The transform type is determined only in passing by the code which was design to fulfill functional requirements.

While performance improvement is possible if functionality can be sacrificed, ideally we would like to improve performance without loss of functionality. It is possible, though rare, that randomly generated program variants have expected functionality.

2.3 Genetic Improvement

Search algorithms have been applied to improve program performance by searching through program compositions [79] and compiler configuration options [124]. As these works deal with searching through relatively safe program transforms which are expected to be at least functionally correct, we turn attention toward search methods which produce a wider range of program variants. During the search process many program variants do not compile.

The advantage of using randomised search algorithms to search through program variants is that a wide range of program transforms can be considered. Using unrestricted transforms means that programs may vary more in behaviour. Although randomly modifying and evaluating programs is computationally expensive, there is no restriction on the improvement type which can be found. The likelihood of a degraded program being created is high, but the possibility of finding an improved program is also enabled.

This section gives an overview of a particular search algorithm, Genetic Programming (GP) [66,106], which is used to improve programs by directly modifying a program [105]. A number of important topics in GP are discussed here to give an intuition of how GP works and what makes software difficult to improve using this technique. As such, it provides an introduction to terminology and context needed to frame our approach and problem.

GP operates by iteratively creating variant programs using random modification [106]. Variant programs are produced in batches referred to as "generations". The programs in a generation are selected and modified to produce variant programs for the next generation. Programs have different chances of being selected for modification. Typically programs are measured for a certain characteristic and this ultimately decides each program's chances of being selected from a generation. The higher a program measures on this scale in comparison to other programs in the generation the more likely it is to be selected. How programs are measured is vitally important to the operation of GP as a measurement scale provides a mechanism by which programs can be differentiated. The

algorithm uses this measurement scale to favour programs to modify on the assumption that they are likely to produce even better programs. The measurement scale is referred to as the "fitness function" in GP literature.

GP has been used to improve execution performance of existing program code [8, 72, 105, 146]. A general issue with the application of GP, is its ability to find improved programs in large search spaces. In the remainder of this section we cover how the various choices in applying GP to software improvement affect the range of program variants produced and the chances of finding performance improved program variants.

2.3.1 Representation

GP has been used on existing code at a range of representations from fine grained low level representations to high level representations. Bytecode [98] and assembler [114] level modifications have been explored as well as higher level modularisations of code such as line of code [140] and statement [73]. Even higher level changes have been made using design patterns [25, 95, 119]. The most relevant representation to our discussion is modifications which can be made at the language level, in particular for Java source code [8, 146].

Work which seeks to uncover better understanding of the GP algorithm itself typically evolves programs in functional languages leaving the evolution of imperative languages less explored [18]. It is less likely that valid imperative programs will be produced during GP than valid functional programs due to syntactic constraints [122]. It is interesting in itself to understand how different language paradigms make problems harder or easier for the GP process [122, 150]. The evolution of programs in imperative languages is also of practical importance due to the widespread use of imperative languages throughout industrial settings.

The more places there are to modify in a program, the "finer" we can call the unit of modification. We differentiate representation from granularity as representation refers to what form the program is in, such as source code, byte-code or design pattern. We work under the assumption that the representation is fixed, but we can still choose how fine

the unit of modification is during GP.

An alternative to directly modifying a program is to evolve a program modification. Program patches, as opposed to programs, are the subject of evolution. Each evolved patch is applied to the original program and the resulting patched program is then evaluated. This approach appears to have been initially developed as a solution to memory issues related to modifying very large programs [72,77]. When modifying existing programs, a modification can be expected to be relatively small. The small size of a patch can make it easily compared and guaranteed unique [72]. Evolving patches allow two working patches to be merged in a single operation, whereas merging the same code changes in tree-based GP would require a growing number of edits as the patch size grows. If the resulting program from the merged patch does not compile, then the contents of the two working patches may be dissected and recombined to see if there is a combination of the two which will compile. As this is a relatively new approach, little is known about how evolving patches influences GP search or compares with traditional tree-based evolution.

2.3.2 Modification

Modification of programs during GP is performed under two schemes referred to as mutation and crossover [149]. A node within a program is chosen for modification which yields an "offspring" or "child" program. When nodes are modified, it can be assumed all nodes in the subtree are included in the modification.

Under subtree mutation, a node (and its subtree) can be replaced by a node (or subtree) of a similar type, deleted or cloned. The initial generation is created by repeatedly applying mutation to the seed program.

Crossover operates with two programs and replaces a subtree in one program with a subtree from the other.

2.3.2.1 Reachability

If the representation chosen is a Turing complete language, then the sequence of language elements which can be accepted as valid is governed by the syntax and semantics of that language. If a non-Turing-complete representation is used, such as lines of code, then the programs generated will be made of only some combination of allowed code lines. Provided the lines allowed have the right structure, presumably requiring large numbers of diverse lines, it is conceivable that evolving lines of code could be considered Turing complete. We mention this because if the GP configuration does not allow the generation of programs in a Turing complete way then we are restricted in the programs that can be generated. This intuitively means our GP system is searching through a restricted subset of possible valid programs. We are unsure whether an improved program exists in this set and if our GP configuration is too limited then we may have entirely removed any chance of finding an improved program where an improved program is not "reachable" given a particular GP configuration.

The structure of software makes modification difficult where increasing interdependencies makes modification less likely to result in valid programs [15, 31]. As some programs are invalid, and thus can not be evaluated, they receive low fitness values. If invalid programs must be modified to reach a more improved program then it is unlikely that those improved programs are *practically reachable*.

A single modification to a program may produce an uncompilable program which may be discarded during GP. If this discarded program is required to produce programs which do compile then these programs may not be reachable. Where only single modifications are applied before evaluation (and discarding of programs) the programs subsequent to a non-compiling program may not be reachable. Such a problem has been labeled a "coding trap" in a more general context [41].

This echoes the restrictiveness of behaviour-preserving transforms utilised in compilers (as discussed in subsection 2.2.2). If the code modifications allowed are restricted then it may be difficult or impossible to create certain subsets of program variants.

2.3.3 Fitness Functions

There are a wide range of program metrics from the SE literature [43], even for more subjective measures such as "elegance" [119]. Software characteristics can be measured in terms of functional or non-functional metrics. Metrics can be combined and used as a fitness function. Similarly in the GP literature there are a wide range of measures which are used as fitness functions.

In GP, it is important for the program measures to capture the characteristics of a desired solution to ensure evolutionary pressure is towards the desired goals. Designing a fitness function which gives a scale for how good a program is can be non-trivial particularly when the scale relates to functionality [61]. The issue is how to ensure that aspects of a programs operation that are important to a solution put that program at the right place on the scale in comparison to other programs. In a sorting example, a program which contains the code for swapping two values in a list to produce a less sorted list would still be considered better than a program which doesn't change the list at all. Additionally if an empty program is compared with a large program but a swap is never performed then both programs may be given the same fitness even though one contains code which makes it a better candidate for selection and mutation. The point can be made that if a desired trait cannot be measured, or receives a value lower than it should, then GP will not promote its use in the population.

2.3.3.1 Performance Measurement

The notion of program performance can be measured as wall clock time, number of CPU cycles taken, number of method calls, number of lines executed or even as the energy consumption of the program. These measures are clearly related though wall-clock time or energy consumption may not correlate with discrete summation measures such as CPU cycles, method calls or lines of code executed. Measuring time or energy of a program may include measurements that can be attributed to the program as well as the execution environment. A complex execution environment, where scheduling or arbitrary network costs may exist, can introduce variance between individual measurement samples. An

average across multiple samples can reduce the effect of such variance but increase the evaluation cost of a program.

Each measure is particularly relevant for a different purpose. By measuring performance as part of a fitness function a parsimony pressure is put on the search which means smaller, or less costly programs to execute will be favoured mitigating the issue of bloat somewhat [121].

A typical measure of performance is wall-clock time. Wall clock time gives an overall measure of the time the program took to run as well as how quickly the environment can execute the program. Time may be useful when specialising an implementation to a particular environment. Accurate execution time is difficult to measure due to high variability in execution environment conditions and complexity [70]. Measuring wall clock time is subject to large amounts of variability. Although GP can tolerate a certain amount of "noise" in the fitness function in terms of how deterministic it ranks different programs GP, noise may slow the evolutionary process. To get a more deterministic measure of performance, the number of clock cycles, lines of code or instructions executed can be counted.

Low level measures of program performance, such as time or hardware counters show how the program interacts with it's execution environment. Estimates of execution cost can also be used to evolve programs with respect to power consumption [148]. Performance measures which count higher level operators executed ignore how code actually performs in a particular environment.

These two measurement approaches provide a choice for what type of software improvement is sought. Meausurements which include environment time or energy are particularly suited to specialisation of a program to the environment, where any resulting improved programs may perform well in the specific case, but are less likely to perform as well in different system configurations [24]. Where only high-level measurements are taken ignoring environment specific interations only more general improvements will be detected.

This distinction and separation of what we are measuring is useful to know as we may

wish to choose what aspect of the code is to be improved. If we want to improve the code at some general level, where it can reasonably be expected to perform better no matter what environment it is run in then we can ignore measurements which are environment specific in our fitness function. This would focus the evolutionary pressure on improving the general design of the code without specialising the code to an environment.

2.3.3.2 Test Cases

When modifying code using random modification a concern is maintaining functional correctness. When improving performance of a program through random change, the functionality will frequently deteriorate. To maintain correctness, test cases can be used [8] as part of the fitness function. In SE, test cases are widely used to check the functionality of a program. Test cases are usually made of test input data and a known correct response to this input data. A collection of these test cases forms a test suite which can be used to check the functionality of many parts of a program. When improving the functionality of programs using GP, the fitness of programs can determined by how many test cases they pass [77]. The number of tests passed is summed and used as a fitness function. As each test case returns a boolean value, a coarse fitness gradient may result especially where a small number of test cases are used.

The choice of input data used in test cases influences evolutionary pressure. Where a general algorithmic improvement is needed, a general representative set of input data is desirable. This is similar to the generation of adequate test cases for general software development. If we were to use input of a particular distribution, we should expect solutions to be more specialised for that distribution [9].

Software is said to have high mutational robustness if random mutations to a program show little change in test cases passed. A large percentage of the programs created during random modification may not compile or cannot be evaluated due to infinite loops [115]. When programs do not compile, they are typically discarded [105, 139].

2.3.4 Code Reuse

A prominent characteristic of program improvement is the extensive reuse of existing code. The extent of code reuse is a main distinguishing point in GP for source improvement. Reusing code which already exists in the program prior to the application of GP may seem a somewhat arbitrary choice of code, though it has been shown that reordering and recombining code allows the knowledge and functionality embodied in existing code to be made more widely applicable through small edits [76, 120].

For GP to be successfully "grow" solutions outright, primitives which can be considered domain specific are commonly used. For example, the primitives used to evolve sort algorithms could include a swap function [61]. It is questionable whether the resulting set of primitives are Turing complete or are more in line with a Domain Specific Language for the problem at hand. As more of the solution is provided to GP it becomes more likely GP will find improved programs.

Part of the solution other than useful primitives can be manually provided, such as program architecture [1]. Manual can also be used to refine the fitness function [119]. When existing code is reused the primitives and architecture are manually defined when the original code is written. Though existing code is not written specifically with the intent of aiding GP there exists useful material for improvement nonetheless.

When considering larger sized programs it appears difficult to grow programs outright and so the emphasis is placed on improving code [44]. If evolving existing working code then we can expect it be already at least functionally correct to a high degree. Provided the code is relevant, reusing existing code relieves the GP practitioner of handcrafting domain specific primitives for the problem. The larger the program, the higher the chance that the required code already exists in some other part of the program. This has been demonstrated very successfully for bug-fixing [76] as well as performance improvement [104].

The concept of code reuse has also been applied during GP [83] and for the automatic definition of functions [62, 65]. Reusing code aids GP by providing partial solutions when improving a program, but it doesn't specify where this code should be applied within a

program. We distinguish the "with what" from the "where" for code modifications within a program.

2.3.5 Operators

The way in which code is modified can affect the solutions which are searched through. For example, an operator may remove a FOR loop from a program, but leave the body of the loop intact. An operator would have to be designed to do this in one edit to a program. If using only basic tree edits such as delete, replace and clone, then it would take two of these operations to replace a FOR loop with the body of the loop. In these cases, domain specific operators may be of use [11]. In SE, code transforms such as those classed as refactoring [25,119] can be used for improvement.

2.3.6 Search Space

The previous sections cover GP configuration options which determine the search space of programs created and evaluated during search. The chances of finding an improvement is influenced by the abstractions used as primitives in GP.

The size of the programs evolved affects the chances of finding a solution.

In GP, a number of issues affect the probability of solving a programming problem. The size of the required program, or program under improvement affects the search space. Where GP is allowed produce programs of arbitrary size, there is a tendency for programs to grow large, containing much superfluous code, referred to as "bloat" [85]. Approaches to alleviate this include simply bounding the size of programs allowed or restricting code modifications to those that will not grow the program size [53, 86]. If the unit of change in GP is particularly coarse, say only functions can be exchanged, and the fitness function only allows compilable problems then the practical size of the search space may be restricted. The size of a program can be measured in the number of places where a modification is allowed. The larger the program that is required to solve a problem, and even if GP is seeded with a partial solution, the less likely better solutions are to be found due to the size of the search space [81]. The size of a program

can be offset by using a coarse granularity of change but runs the risk of making some solutions unlikely to be generated and evaluated as discussed in subsubsection 2.3.2.1.

A large program with few dependencies between code elements, may be difficult for GP to improve simply because of the large number of combinations of elements. Conversely, a small program which has high dependence between code elements, means any single change is highly likely to produce an uncompilable program. This problem is difficult in that many programs can not be evaluated and do not register on the fitness gradient scale. We are essentially performing random search where all program variants do not compile as there is no measurement gradient for guidance.

The issue of a large search space has been recognised [9,85].To tackle the problem, we can try to scale down the problem by recommending what programs are worth evaluating by making a more top-down choice as to what programs are likely to be relevant [137].

2.4 Guiding GP

Previous sections have shown how performance improvement can be automated. This section turns attention toward how code modification can be guided in GP to increase changes of finding an improved program. Although program selection provides guidance during canonical GP, once a parent program is selected the choice of how a subsequent offspring program is generated is entirely random by design.

All nodes in a program are equally likely to be selected in canonical GP. Changing the probability of modification at these nodes is an opportunity to introduce search bias [145]. By attaching values to each node in a tree, node selection can be biased [6]. Bias values can be set *before* the application of GP and has been used to great effect for bug fixing [36] as well as performance improvement [74]. Biasing the selection of nodes before applying GP has the advantage of introducing little overhead during the GP run. As the population evolves, the bias values remain the same on the seed program. Biasing modification to specific locations in a program means that a certain section of the search space will be more thoroughly sampled. The chances of finding an improved program

variant are increased if a solution lies within the chosen subset of the code under search.

Many different approaches to updating bias values *during* GP have been proposed. Furthermore, bias can be applied “globally” where bias applies the same to all programs, or can be applied “individually”

When bias values are allowed to change during GP, we can divide mechanisms into two approaches, which we term “global” [155] and “individual” based bias mechanisms [4, 80]. The labels global and individual refer to how bias is summarised. Global mechanisms summarise information which is used to guide program modification across many programs, and assumes information found is generalisable to some degree across these programs. Individual bias refers to bias information which is more specific to a single program. Individual bias learned with respect to each individual program is only transferred between descendants or individuals which have some genetic interaction. As each unique program is likely to be derived through a different lineage [6], so too does individual bias. We focus on individual bias in this thesis, but discuss global oriented bias for comparison mechanisms of inferring bias.

In the next section we discuss general self-adaption techniques in GP. Many self-adaption techniques attempt to find *how* or with *what* a change is made. In the subsequent section we discuss how locations in programs can be highlighted, and for what purpose they are used during the GP algorithm. Biasing locations can be thought of as finding *where* to make a change.

2.4.1 Self-Adaption

Self-adaptive GP refers to modifying or tuning parameters, or other parts of the GP algorithm itself as the algorithm runs. In this section we note self adaption techniques in GP and their purposepaying particular attention to how the adaption is driven.

2.4.1.1 Operator Adaption

A number of works have attempted to evolve GP operators. In the canonical case, operators usually allow deletion, cloning or replacement of genetic material. Crossover

operators allow entire subtrees or a node to be replaced. An evolved operator can be expected to make more complicated operations as a single modification to a program, equivalent to repeated applications of simpler operators. The approach is intended to discover any problem specific transforms, with the intention that these operators will be able to more effectively navigate the search space. Evolved operators may be able to make decisions about the structure of the code they are modifying [34,60,128,151].

This approach is considered global, as en evolved operator can be applied to any program in the population. The selection of what operator to apply can also be adapted [117] which may make operator application more local to the context which has been selected for modification. The selection of where to make a change is still random but the decision for how to modify is biased.

2.4.1.2 Reusing Code

Many approaches to self-adaption have attempted to encapsulate and reuse code which was evolved and found useful during GP [46,64,83,110]. The motivation for this approach stems from the "building block hypothesis" that has been the subject of much discussion in GP literature [7].

As nodes are selected for crossover, sub-trees are exchanged between programs. To bias the use of sub-trees, the usefulness of each sub-tree can be stored in a central location and updated as it occurs in new individuals [71]. The usefulness value can then be used to bias the selection of sub-trees when generating new individuals. This approach updates a single value for a sub-tree for each new individual it occurs in and takes into account the resulting fitness of the new individual. This is a sub-tree centric approach which recommends with what a change should be made at the global level. As a subtree-centric approach, the emphasis is on what a code modification is, as opposed to where the modification is applied. This work shows that code can be reused in other places and information about a piece of code is relevant to some degree in other places supporting the building block hypothesis. The information and code can be considered for transfer to any other program or on a *global* basis.

By assigning fitness to subtrees evolutionary search can be focused on subtrees which have a low fitness measure [56]. Tracing the lineage of programs can be used to find useful genetic material [112] and fitness reward propagation back through previous program lineages [113] can achieve similar measures of code relevance.

Although not considered genetic, neural programming deserves a mention [129]. The contribution that each node makes to the overall output is used to determine what nodes are contributing the most and the least. Proposed in this work is internal reinforcement with a credit-blame map for programs represented as graphs. In this way the nodes are separated into "good" and "bad" nodes. Detailed tracking of internal state information is attributed to each node in a neural program. The goal is to grow programs outright with an appropriate goal to maximise the most important nodes and maintain beneficial code as "building blocks".

Measuring the frequency of terminals in successful programs can be used to bias the distribution of these terminals in subsequent programs [39]. This approach is concerned with removing terminals superfluous to the problem at hand. It is unclear how this would help in scenarios where the initial population is seeded with programs which presumably contain mostly relevant terminals. Gene dominance is a similar approach which counts the frequency which subtrees exist in programs [154].

2.4.1.3 Evolving Representation

Bias can be introduced at the grammar level to influence how new code segments are generated [144]. Modifying the grammar used during GP changes typing of different program elements and what code is valid in certain contexts. The purpose of this work is to influence the frequency that certain code constructs appear in programs. GP is described as unsuited to developing recursive structures. The approach can evolve the language itself so that it may be better suited to the problem at hand [145]. The approach appears global as changes to the grammar affect all modifications in all programs.

2.4.1.4 Learning Structure

A wide range of mechanisms which are extensions to GP or differ so much as to be named differently exist which reuse global information almost exclusively, instead of using anything that could be termed "genetic". One such approach evolves a single tree is by repeatedly sampling from the tree. At each node in the tree there are a number of options as to what code can be chosen. Through successive sampling and evaluation the options and associated probabilities are updated. The probability value for each code section are modified influencing the chances of particular code being picked during subsequent samplings. Essentially a probabilsitic model of a solution program is expected to emerge. Estimation of Distribution (EDA) algorithms are of this type [22, 82, 107, 152, 152] as is Probability Based Incremental Learning (PBIL) [96, 116] and Linkage Learning [20]. Linkage learning builds a probabilistic model of a solution [21] to discover building blocks that are associated with each other through a grouping or clustering mechanisms [20].

Individual and global models are considered in this work [20] but no evidence as to whether a distributed or centralised model seems to be better. The survey hypothesises that a distributed model might have better scalability, in terms of at least processing across many machines. This work makes no direct reference as to how global and distributed models differ in terms of search space traversed by algorithms. The use of "fitness only" measures to find linkages is non-optimal.

As a follow-on from internal reinforcement in neural programming, Reinforced Genetic Programming (RGP) was introduced [33] with the goal of supporting Lamarckian and Baldwinian evolution. Lamarckian evolution refers to characteristics learned during a programs execution being passed on to offspring via genetic material. Baldwinian evolution refers to how information learned during a program execution affects the fitness of that program without changing the genetic material. RGP adds nodes to a GP tree which can be fine-tuned over a number of evaluations of the tree. The tree can thus improve its fitness over subsequent evaluations as local search is conducted at these special nodes. RGP is used on maze navigation problems where the individual is a lisp tree. Trees are evaluated at every step movement in the maze. Values are reinforced positively

or negatively in response to good or bad movements in the maze. Hitting a wall attributes a negative value whereas reaching a "goal" segment of the maze receives a large positive value. The most successful node behaviour is then favoured during subsequent evaluations. The rule used is to update the bias when the fitness increases. This assumes the population starts from a low fitness, and any improvement in fitness is a good thing.

2.4.1.5 Cultural Algorithms

Cultural algorithms are relevant as they perform a type of reinforcement learning in the form of a belief system. This belief system captures where, when and how a change should be performed [109, 155]. As such, Cultural Algorithms appear useful as a framework for capturing general knowledge produced during a GP run for reuse.

The way beliefs are generated in cultural algorithms is based only on what is involved in generating the best programs and can be used to determine a crossover point [155].

Particularly relevant here is information at the micro-evolutionary level of cultural algorithms [27]. Information gathered consists of "behaviour traits" which pass between generations using "socially motivated operators". The micro-level links in with the global macro-evolutionary layer or "belief space" which is accessible to all individuals. Cultural additions affect the tournament selection of individuals only.

Cultural algorithms make use of information above what is used in GP in a number of different classes [102]. Normative knowledge covers the range of values which have been found to be beneficial. This assumes that the improvement of these values is single peaked and contains a smooth gradient to the global optima.

Cultural algorithms appear to be a framework within which knowledge can be stored and utilised to influence further evolution. The rules by which knowledge is captured and reused can be created for the task at hand. The approach is global in that all programs can reuse knowledge from a shared belief space. Information is included in this belief space from programs of high fitness only.

2.4.2 Location Learning

This section discusses self-adaption techniques which highlight locations in a program where modification should be applied.

In canonical GP, all nodes in a program are equally likely to be selected for modification. Code is selected at random for modification but it has been noted that some places in code should not be modified due to destructive effects [87]. As discussed in the previous section, mechanisms have been used to maintain blocks of code which appear useful as a unit. These works reuse code as exact blocks of code but a specific block of code may not always be valid when placed in certain contexts [97]. By reusing functions the generality for reuse is improved [111] as any variable of the same type can be used (as opposed to a specific variable as referenced in a specific piece of code).

2.4.2.1 Local Search

One straightforward way to find where to modify a program is to make a range of possible changes and discard all but the fittest generated program [87,120,126]. Another similar approach is to prune sub-trees from a program and evaluate the resulting effect on fitness [49].

Local search falls short of being a learning technique as information gathered during the process is not retained or summarised for subsequent use. There is no attempt to predict where to change before modification. This approach is performed per individual program as opposed to on a global basis. An additional re-evaluation of the program is needed to find the impact of each sub-tree pruned and provides an aggressive selection mechanism which will only accept the best mutation. Adding local search to GP can increase the computational cost of the algorithm due to the large number of re-evaluations performed.

2.4.2.2 Program Size

Modification can be restricted based on a program measure such as size [47]. The purpose of this approach is to mitigate the effects of bloat [30,86] or reduce the number of inviable

programs produced [54]. The size of genetic material or tree depth can be taken into account when considering where code is to be exchanged between programs [87] with the emphasis on reducing destructive code modifications. This can be considered a global approach as the depth at which modification is performed is tuned for all programs.

Although size is used to select location, the location of code is randomly selected in one program and the size or depth of this subtree is then used to select a location in the second program. This approach still relies on random node selection in the first case and then selects what to exchange this node with based on size [87].

Locations for crossover can be chosen at the same places in both parent programs [37]. The initial selection of a node is random but the choice of node is at the same location in both programs under homologous crossover.

2.4.2.3 Semantics

The internal operation or behaviour of a program can be used to guide search and is referred to as the use of "semantics" [134]. By considering the data that is processed by a particular subtree, similar locations in another program can be chosen. This means that subtrees are selected (or locations in a program are selected) based on the similarity of their behaviour [89,133]. Assuming sub-trees in code can be partially evaluated code can be matched based on the return values produced [89]. Another approach is to perform multiple trials to select crossover points which produce valid programs [133]. Crossover points are initially selected randomly and the similarity of the subtree's result is used to constrain what code is used to replace other code. Subtrees can be selected for exchange if they return the same values. This work is appropriate when dealing with S-expression from functional languages and may need to be technically redesigned to be applied to Abstract Syntax Trees (AST).

2.5 Dynamic Individual-level Node Selection Bias in GP

We have discussed self-adaptive GP techniques in subsection 2.4.1 and more specifically techniques which can find program modification locations in subsection 2.4.2. This following section discusses mechanisms which bias the location of modification during GP specific to individual programs.

2.5.1 Individual Level Adaption

The fitness of a program as quotient of the average fitness of the entire generation can be used to tune crossover and mutation rates for individual programs [35]. The purpose is to balance exploration (with mutation) and exploitation (with crossover) at different stages of the evolutionary process. Mutation is applied more frequently on more fit individuals, with the more coarse modification performed by crossover being applied to programs which are less fit with respect to the total population. The approach appears to refine fit programs and provide large changes to unfit programs. Nodes are selected at random from each program in this approach.

2.5.2 Gaussian Bias

One of the earliest descriptions of node selection bias proposed the use of values for each modification point in a program tree [6].

In this work, node values are updated every time an offspring program is created. The values are updated by randomly applying Gaussian noise. Updating bias is achieved by the addition of random noise to the parameters at each node. Attaching parameters to every node in a program yields bias which can be unique to each program [6]. The parameter tree is individual to each program in the population. When the parameter tree is created, all values are the same.

The programs with the best fitness are more likely to propagate through the generations, and similarly the bias attached to these programs also propagates. Bias is shaped by the selection of programs based on fitness. The values evolve with each program

meaning a good set of values is maintained in the population if the values can maintain good fitness in individuals. A bias which produces a fit program is more likely to remain in the population. Bias emerges based on it being less destructive and more likely to produce viable and fitter programs. As node probabilities are selected by the overall individual selection process the bias is not explicitly shaped by any explicit rule and instead is shaped by a more implicit evolutionary pressure. Useful bias is thus allowed to emerge through standard evolutionary pressure.

As the GP algorithm progresses through generations the difference in values in the parameter tree should grow more pronounced. When random noise is used to change the values the only cumulative bias in how these values emerge is driven by the individual selection mechanism. Selection favours fitter individuals and therefore their parameter trees will also survive. Parameter trees that survive will likely continue to survive if they continually provide good locations to apply operators. The mechanism they arrive at providing good bias for node selection is through random modification. The value modification mechanism, random noise, is disconnected with how well the existing value has been in previously recommending good locations to apply operators. We regard the use of random noise in this manner as a gentle or light pressure on the emergence of large differences in parameter values, and hence a gentle guiding pressure on the overall GP search algorithm. As the noise applied is essentially random it may take a reasonable amount of time for useful bias to emerge.

As the values are arbitrarily changed it may happen that the values are changed in the wrong direction or in the wrong place for an individual. This could mean that a promising individual is subsequently changed in a less-than-optimal location in a non-optimal direction due to arbitrary allocation of noise within the parameter tree. Where noise is added which is not optimal the GP algorithm could briefly perform worse than a GP system without any node selection bias. As GP can tolerate a certain amount of sampling noise this may have negligible effect on an evolutionary run.

This work attempts to create programs from a set of primitives, and does not start with an existing partial solution. The authors propose that there is no need for explicit

credit assignment for each node, and that evolutionary pressure is enough to allow a useful bias emerge.

2.5.3 Node Selection Bias

A guided technique for choosing where to modify a program is to pick the largest subtree from those in a tournament [48, 52]. The purpose of this approach is to control program size generated and reduce bloat. If larger trees are chosen for modification the average effect on program size is neutral neither increasing or decreasing the average size. This work does not require that subtree size be equivalent as is the case in Homologous crossover [37].

2.5.4 Fitness Guided Node Selection

Node selection bias has been used to control program size during GP [131] by picking the “fittest” node with roulette wheel selection. Node fitness is measured by the amount of fitness each node contributes to the overall fitness. This technique is applicable where the results of parts of a program can be evaluated directly as per the fitness function.

When roulette wheel selection is used node selection is biased towards nodes with higher values but lower value nodes still have a chance of being picked.

This work is distinct, as it does not perform local search to specify what nodes to select in a program and uses fitness information to propose what node is selected.

Subtree fitness per individual is relatively straightforward when evolving functional programming languages on certain problems provided that a subtree can be directly evaluated by the fitness function. Measuring the contribution of partial results as they are computed during program execution may not be applicable where state is stored in temporary variables during program execution. Depending on the program it may not always be possible to inspect partial solutions during the execution of a program.

2.5.5 Summary

The approaches listed here have all addressed issues specific to GP and have not attempted to utilise aspect specific to the domain of improvement.

2.6 Analysing Software

As discussed in the previous section, many techniques for determining what programs to generate have been proposed to guide or navigate GP through the search space. When modifying a program, we can consider where to modify a program and with what to modify a program with. Modularisation and coding styles can make source code more easily navigated by people. Analysis techniques such as profiling can guide developers but can also be used to automatically expose where in code to modify.

In this section methods for analysing program execution are discussed. We can measure programs by their output or by analysing the operations performed in producing the output. To observe variance in these measures we can modify program input or the program code itself. Analysing program output after source code modification is most relevant for this thesis. The program analysis techniques covered here are specific to performance improvement as more general software metrics do not appear to be suitable for guiding GP [17].

One approach is to measure differences in fine grained profiling information such as execution path [57] or count of specific execution operations [45] along with differences in program output to make inferences about locations in a program. Similarly, differences in program behaviour can be observed amongst two slightly different versions of a program [45].

2.6.1 Profilers

This section discusses how program profiling techniques can be used to guide automatically improving program execution cost.

Profiling generally refers to measuring the resources used by a program during exe-

cution [12]. This may be in terms of memory, I/O, system calls or time spent executing. Profiling information can show what parts of a program are using the most resources. We are most interested in profiling for execution cost but other profiling techniques are of interest for their measurement technique.

Profilers which focus on CPU or energy profiling are of particular relevance as they are related to cost of execution [94]. Profilers are widely used to aid program understanding when performance bottlenecks are observed [3].

The main execution profiling approaches can be broken into *sampling*, *timing* and *execution count*. Sampling methods take note of what part of the program is executing sporadically during execution [127]. This is a low cost way of find where in the program the most time is being spent. A relatively small number of samples are needed to find the most time-consuming segment of code. As such, it highlights the most problematic code quickly, as opposed to attempting a measure of exactly how much time is spent in different sections of code.

Timing measures introduce relatively low-overhead also by measuring the time elapsed or "wall-clock" time for various portions of a program. Measures are taken at certain points of interest in the program, such as measuring duration of a functional call. Timing measures can also be useful to understand complex interactions with an execution environment [69, 92]. Timing may be particularly relevant for program specialising or tuning. Accurate timing of code can be difficult to perform especially as computing environments grow more complex, for example where there is some runtime interpretation and improvement performed [67]. Interpreted or semi-interpreted languages allow a larger variance in time as frequently interpreted code can be improved in many ways such as compiling to machine code [13]. The operating system and its configuration play a part in how a runtime environment behaves. Due to this, timing measurements must be repeated to gain a dependable average measure but the variance can be expected to be high.

For a more general understanding of a programs execution profile, an execution or operation count may be used. Frequency of execution can be measured by instrumenting a

program [14,78]. Instrumenting refers to adding information gathering code to a program. An example would be the addition of logging before and after a function call. For execution frequency measures a counter can be updated after every function or line of code executed. For program inspection with a view to improving an algorithms operation code can be instrumented at different granularities [40] such as per statement or per line of code. Instructions executed can be measured to give a deterministic value for execution cost at the bytecode level [68].

Various advantages such as accuracy, repeatability, lack of bias, low overhead and portability have been well described [42]. Counting lines or operations executed ignores environment measures for how long any one operation took to perform. Such measures capture a profile of how many times operations were performed as opposed to how long they took. Measuring frequency of execution is somewhat general in that it doesn't capture how long or how much resources a unit of execution takes up in any particular environment. Thus environmental aspects of the programs execution are mostly ignored and a more general "algorithmic" measurement is made. One potential issue with their use is that instrumentation may affect the programs operation in some cases especially where the program is complex, threaded or has a lot of external interaction (system buffers, io).

Fine-grained approaches to program profiling involve the instrumentation of a program at the level of instructions [59, 103]. At the lowest level, hardware measurements of instructions called by a processor can be taken. A compilation step can be used to instrument a program at the machine-language level [84].

Even at the finest unit of measurement, there still remains the question of how variable the actual time is for each operation performed. Not all bytecodes are the same. There is an arbitrarily large amount of variance in the time it takes to execute any single bytecode. Consider that a number of bytecodes may be executed before a particular system or network operation is performed. The time it takes for these operations is significant and may be attributed to the last bytecode executed before the operation is invoked. This saving one bytecode, may not be as significant as saving some other bytecode. Due to

this, bytecodes can be given different weightings to provide a more accurate measure of cost [13, 67]. An example may be where a small number of bytecode may reserve or release a large amount of memory, triggering work for the garbage collector.

We cannot broadly expect all operations to be similar across many execution environments. Not all operations are equal either. A number of operations may need to be performed before a system call is made, but under the execution frequency measure an earlier operation is counted the same as the final operation which invokes a remote procedure or performs some disk I/O. Saving the execution of some operations may constitute a negligible reduction in a programs runtime cost. Any significant saving in terms of operations is highly likely to yield similar savings in execution time and we can use it as an estimate of performance [70].

Lightweight profiling techniques have been used to guide improvements at runtime, as readily exemplified in prominent Java Virtual Machine (JVM) implementations. Profile-based compiler optimisations rely on measures such as invocation count of functions, branch instructions and basic blocks of code to guide which optimisations should be applied. Profile information improves optimisation in relation to the execution environment and the input data range. Knowing what code is executed most frequently allows the compiler to choose between conflicting optimisations for the likely usage scenario of the program. The optimisations performed are widely depended on to be semantics-preserving.

Profiling information has also been used to select between different variants of code which performs the same task. Where multiple pre-written function implementations are available variouis combinations of these function calls can be evaluated. Profile information is gathered and used as a measure for which code combinations produce the most performant program [23]. How safe the manually written code is can vary and there is little assurance that the code change represents a semantics preserving one. Similarly, a manually written code change is unlikely to be an entirely random change.

2.6.2 Performance Root-Cause Diagnosis

While profiling can attribute execution cost to certain parts of a program to find performance bottlenecks, taint tracking has been used to find the cause of performance issues [10]. This work attempts to move from the detection of performance bottlenecks, that is the symptoms of performance issues, toward the diagnosis of what is causing these bottlenecks. Diagnosing bottleneck cause is described in terms of program configuration and input data. Using this approach, performance issues are attributed to entities outside of the source code of the program. The point can be made that a bottleneck and the cause of that bottleneck may not appear in the same location within code.

The approach taken to determine root-cause of performance issues is to measure the program performance in respect to the execution environment in terms of program instructions and system calls [10]. By tracing program input data through a program it can be seen what system calls are associated. The time it takes to execute the code and perform system calls is then attributed to the different input data. Program input in this context can be software configuration read in a startup, or data passed to the program during normal runtime. Taint tracking is performed by marking input data and capturing a trace of resources used in processing input data. By comparing the cost of many traces across differing input data, input data which causes the most cost can be found.

The purpose of associating execution cost with input data is to help debug configuration issues and understand why certain requests to a program cause radically different performance characteristics. Manually tracing the cause of a performance issue within a program can be time-consuming and difficult to perform especially in large systems. This approach modifies input data to the program as opposed to modifying the program itself and uses the concept of comparing or differencing the change in execution on response to change of input data.

The analysis is heavy-weight and is performed after recording of instrumented program execution. This approach is applicable for performance issues which are caused by problems extraneous to the source code. Access to source code is not necessary when

tracing input data. The approach is particularly useful for configuration of a program without requiring a deep understanding of the internals of a program. Re-designing the actual program code is not the focus of this work, nor is the focus to generate possible configuration fixes for performance issues (though it seems a plausible next step).

The approach assumes that a program is structured in a way which allows configuration values to be modularised in config files external to the program itself. The code has to be written to be tunable, where conflicting program design decisions have already been incorporated into the program. When appropriately designed, it is a matter of configuration to select program behaviour. The approach is designed to operate on binary programs. It does not go as far as recommending where in the source code a solution should be applied and treats a program as a black-box in that regard. It is proposed as a runtime configuration tool as opposed to a design-time tool.

This form of performance cause diagnosis using taint analysis has not been used with GP, although the approach appears that it could be useful in guiding GP.

2.6.3 Sensitivity Analysis

Varying input data has been used to determine bottleneck locations which are likely to become more problematic as input data size increases [29, 73]. The execution frequency of a "basic block" of code can be measured using different input data sizes [29] and can be alternatively described as "asymptotic analysis" [29] or "sensitivity analysis" [73]. For each different input size over a range of sizes the program is profilied to get a line count. For a number of different input sizes each line of code in a program will have a range of execution frequency counts. Observing how execution frequency changes for each line gives it's sensitivity to the input size. For example, some lines may have a quadratic or qubic response to input size [73]. Particularly costly lines may be described as performance bugs which are can manifest as size increases.

Highlighting scalability-critical code extends the ability of execution profiling on input of a fixed size. The approach can highlight code which scales poorly when input size is increased and draws mutation to bottleneck code which contributes the most to program

cost. Removing these bottlenecks improves the programs performance but also may guide improvement of a program's stability under varying loads.

For performance, sensitivity analysis has been used to directly attribute bias to nodes in a program as discussed in subsection 2.6.3 before the application of GP [73]. Statements which are executed exponentially more often as the input size increases are given the highest bias. The bias that is allocated is fixed in magnitude manually with respect to the exponent level to which their execution increases.

2.6.4 Program Spectra

Program Spectra refers to analysing changes in a programs execution [45, 50] using a number of different measures of execution. Analysis can be performed over measures or "spectra" such as execution path, count or program output. A programs execution, and as a consequence its spectra, is varied by providing different input data to the program or across a number of program variants. The approach is interesting for the concept of analysing some measurable spectrum produced by either varying program input or varying the program itself.

From the spectra measures mentioned [45], "output" spectra, or measures of the programs output, were not found to be particularly relevant to fault localisation. This work is very useful in framing the various ways program behaviour can be measured. Although the work inspects two program variants, a correct and buggy version, the variants are used to determine what program metrics highlight a bug most prominently. The input is varied to observe the difference in measured spectra. The spectra that they measure include internal traces of program execution such as which lines are executed when the bug is measured for some input.

Other spectra appear similar in concept to measuring the semantics or behaviour of a program over time. Getting an execution trace of a program is similar to measuring semantics of a program [133].

Work on bug fixing [141] appears an example application of this concept to guide GP, although it does not appear to have been investigated for highlighting potential

performance improvements.

2.6.5 Mutation Analysis

Mutation analysis refers to modification of a program repeatedly and analysing the resulting effect on program behaviour. While mutation analysis is widely used to improve software test suites, modification or mutation of programs has been used to drive analysis for fault localization [91, 100]. Programs are mutated using predefined rules and the output of the program is analysed. Over a number of mutations, the analysis is able to localise faults. A number of test cases, each of which is evaluated as true or false, are used to find faults. The general notion is that faults produced by code mutations are directly attributed to the locations of those code changes.

The localisation of faults has received plenty of attention in the related work, with many advances being made in the area [57]. We include a brief discussion of fault localisation to show that it is possible to accurately infer the location of improvements in a program.

Fault localisation can be performed by executing a program with a range of input data. The correctness of a program is measured with a number of test cases. The input data which causes test cases to fail is likely to have a different execution path than tests which do not fail. The difference in test cases passed and execution paths is analysed to show where a bug is likely to exist. Execution paths, and the associated code are thus separated by the test cases passed. Those associated with positive test cases only are of low "suspiciousness" for being faulty. Those associated with both negative and positive test cases are medium. Most suspicious of all are portions of an execution path traversed only by negative test cases.

The approach has been validated as accurate for bug fixing for a number of bugs across a number of programs [139]. The fault localisation technique has been used to apply probabilities to lines of code before GP is applied. This reduces the problem of dealing with a solution space resulting from 20k lines of source code to one that is orders of magnitude smaller, 34 to 3.8k lines of code in examples. Probabilities are

set for each line of code through the use of multiple positive and negative test cases. Code which is executed under positive test cases is less likely to be the cause of a bug. Code executed by negative test cases, which test for the bug, are likely associated with the bug. Probabilities are used to guide the GP operators to lines of code that have a high chance of relevance to the bug. This work highlights the importance and impact of finding locations within source code. It is also interesting to note the granularity that GP operates at. Lines of code are swapped as opposed to picking nodes at a finer granularity. This improves scalability, but decreases chances of finding a solution given that the number of possible solutions has been reduced through the use of a coarser granularity. This approach is based on the assumption that the lines of code required to fix a bug exist somewhere else within the program. In demostrated examples this appears to be the case for certain bugs.

The core method in this work, named Tarantula [57], ignores the number of times a section of code has been run which is fitting when considering fault localisation.

Ideally we would like a similar mechanism which can accurately find the location of code elements which cause performance problems in programs. Developers typically "backtrack until the fault is found" [57] when using localisation techniques. This highlights the difference between where the fault manifests, occurs or is exposed and where the cause of the fault is.

There does not seem to be any work on the use of software mutation or the analysis of many program variants for attributing runtime cost to source code.

2.7 Summary

Traditionally, GP has been used to evolve functions. Advances in the size of programs generated with this approach have not materialised. As a result, attention has turned to the improvement of existing programs using randomised algorithms such as GP. GP search becomes less effective as more modification points in an existing program are considered.

Even when GP is used on a partial solution which contains most of the required genetic material, there is still an opportunity to further guide GP in searching through program variants.

Approaches to guiding GP have sought to learn information about a problem during the GP algorithm. Learning what building blocks are relevant to a problem is not of upmost importance where the majority of a solution and structure has been provided. When building blocks can be ascertained as relevant for a problem, they can be considered globally relevant for all program variants which are considered as solutions. Where improvement is possible by only a small number of code edits, learning where to make program modifications can be beneficial. Learning where to modify a program has been performed during GP for purposes specific to traits of the GP algorithm, such as bloat. Knowing where best to modify a specific program may not be generally applicable across a wide variety of variant programs. Learning where to modify existing code specifically for performance improvement has not been examined on an individual program basis.

Profiling is beneficial for focusing search where bottlenecks manifest. Taint analysis has been used to attribute the cause of performance issues to input data extraneous to source code. Neither approach has been used during GP. In both cases, program analysis has generally been performed by observing the effect of varying input data to a program.

As a taxonomy of program analysis, program spectra is a good model to distinguish the different ways of analysing measurable aspects of a program at runtime. Spectra is any measurable aspect of a program at runtime. To create a varying spectra at runtime, we can vary either change the program input or the program itself. Changing the input has been used for bug fixing as well as performance analysis of bottlenecks and the cause of bottlenecks in input data. Changing the code itself for indicating the location of performance improvements has not been rigorously evaluated.

Finding the cause of a performance bottleneck is difficult to analyse as the cause and bottleneck are not necessarily at the same location in code. Determining the cause of a performance bottleneck or where a performance is likely to exist is difficult as there are complex interdependencies between code elements in a program and all code elements

executed in a program contribute in some way to the execution cost. The performance of a program is a single summed value of all operations performed during program execution which does not appear to directly expose the cause of performance improvements.

Our problem is guiding GP specifically for the purpose of performance improvement. From our coverage of the related work, and as far as we know, the derivation of individual bias from program mutation, specifically for performance improvement during GP has not been proposed.

Chapter 3

Self-Focusing GP for Software Performance Improvement

To evaluate the effect of bias on GP, we use a reference GP configuration which we refer to as "canonical GP" and its extension with mutation-derived bias. Canonical GP is designed primarily to support inspection of our hypothesis about bias in GP. We use a baseline experimental setup where our GP system was able to find improvements in all problems we considered. Canonical GP is designed using concepts from the related work, but the implementation was developed for this thesis. Although our GP system could be improved by more rigorous typing to prevent wasted evaluations [18], we scope out the consideration of complex GP additions which may distract from the research questions of interest.

Our solution lies at the intersection of dynamically guided GP and performance analysis of software. The core design concept is that bias is allocated to nodes based on measuring the effect code modifications have on program fitness during the evolutionary process. If performance improvements can be highlighted through analysing the effects of random modification then GP should have increased chances of finding those improvements. If the chances of finding improvements increases then improvements should be observed earlier in the GP process.

Improvement analysis through mutation-derived bias allocation is performed by the same mechanisms that GP relies on. The use of mutation-derived bias is affected by the generational nature of GP, where bias is allocated to many programs variants. Bias allocation can alternatively be performed on a single program without being part of GP. Our proposed approach, Self-Focusing GP, specifies biasing node selection for modification during the GP process. The iterative process of measuring change to guide further change is an additional learning mechanism which extends the canonical GP algorithm.

We propose the use of differences in program function measure to increase bias values of nodes which have been modified. Increased values directly increase chances of selection when functionality is reduced through modification. Measuring a reduction of functionality is bounded by the requirement that the program be amenable to evaluation. The program must compile for evaluation to take place. If the program does not compile, bias can not increase.

We experiment with GP and bias design as implemented in a GP system which modifies, compiles and evaluates java code named "locoGP" (more details can be found in Appendix A).

3.1 Design Rationale

We use GP as it is a randomised search algorithm with little restriction on the program modification possible, and subsequently little restriction on the functionality or performance of programs generated. As generating substantial programs from scratch is currently beyond the practical ability of GP, we focus on improvement of existing code.

We seek to improve GP's chances of finding improved programs by adding additional selection mechanisms to GP. As GP is a computationally expensive process which relies on program modification, it would be beneficial if the result of program modification can be reused to instruct subsequent modification. Modification is performed as part of GP and so we seek additional information generated from modification that can be instructive to search.

Adding values to locations in a program allows the algorithm to distinguish between locations in a program when attempting modification. We are generally of the opinion that such mechanisms should have some randomised element. We use tournament selection which allows values to be selected which are not the highest. This avoids the algorithm repeatedly selecting the same node every time in a program.

We choose an individual bias mechanism as opposed to some centralised global store of information. Summarising information in a central location assumes that the information is generally applicable and reusable across many programs. A global store of code appears useful when considering reuse of common code elements. For example, a sort algorithm is likely useful in many programs. It is less clear how information about a particular location in a program may be relevant in different programs. We argue that knowledge specific to the current individual which is distilled through its unique lineage will be more accurate for modifying this specific individual than a general sub-tree fitness mechanism.

A guidance mechanism which updates during the search progresses can evolve with the population of individuals and be more sensitive to each individual in the population [122]. This approach may be useful where there is a large amount of diversity in the population.

We inspect the use of rule-based or 'explicit' credit assignment in comparison to allowing bias to change randomly [6]. The overall approach is still somewhat randomised due to the selection mechanism. A rule-based bias update mechanism allows information gathered during GP be directly used to update bias values.

The performance or execution cost of a program is measured as a single scalar value which increases and decreases as input data size and distribution changes. We assume there is little variance in execution path through a program apart from the frequency of execution. In comparison with bug fixing, functionality is measured as a factor of binary values. By changing input data, binary values in this vector change and can be attributed to differing execution paths taken through a program. Performance on the other hand cannot be readily inspected by observing different execution paths through a program under different input data.

As our fitness function is comprised of performance and functionality measures, we

have a number of ways of constructing a rule for updating bias. A number of design decisions are presented, the fitness, functionality or performance values can move up or down when a child program is produced. We want to see if mutation can indicate performance improvements as there is a fluctuation in evaluation values when a program is modified.

As we are starting from an existing functioning program, we are starting at a local optima which the search must "escape". When compilable programs are created by random modifications at random locations in a program, there is a trend toward degraded functionality and increased performance measures. The canonical example is of an empty program which does not have any function yet has an excellent measure of performance.

We use our GP system to inspect the use of functionality change as a factor in highlighting performance improvements. We choose functionality as it has shown more relevance to performance improvements under initial application to problems. We evaluate how useful functionality is at highlighting performance improvements in chapter 4. As we experiment with changes in functionality to control bias, it may seem counter-intuitive for changes in functionality to be used to infer locations relevant to performance.Additionally, modifications which have not been tried before have no information associated with them and are thus worth attempting to modify at least once [71].

As real-world software may have multiple requirements and only a single one is required to change the portion of the code which needs to be modified can be expected to be small [73,140]. We modify programs represented as Abstract Syntax Trees (AST) which gives the ability to modify a program at a fine-grained level (a Turing complete language) and includes typing and syntax information.

Contrary to the goals of much of the GP literature, we are less interested in preserving building blocks but finding modifications to existing blocks. Because of this guiding random change suits our purposes better than mechanisms for preserving or cultivating building blocks. To improve software, you need to break it first. In terms of the schema or building block hypothesis we ignore whether we think these can be used as blocks and instead focusing on change. We argue that there is no one perfect modularisation and

that attempting to reuse rigorously defined chunks of code may be counter-productive for improvement. Generally the larger and more general a block of code is the more opportunity exists within it for specialising it to be part of a larger composition of code. Our philosophy, in the context of GP for program improvement, is that we should shape the application of modification as opposed to modularising functionality. In biological systems, modification is focused at the boundary of natures building blocks but modification also occurs throughout genetic code albeit at a reduced rate [93]. As there is some contention about the importance of building blocks we may leave the definition and reuse of a building block to evolutionary pressure as its explicit encapsulation may hinder search.

3.2 Genetic Programming Configuration

The settings and operation of our baseline reference GP configuration are summarised in Table 3.1. The representation and GP parameters are chosen to support the modification of code in the Java language. The elitism we use is the most non-standard part of our canonical GP algorithm. Technical and implementation details can be found in Appendix A.

The initial population of individuals is generated by taking the program to be evolved as the seed individual and mutating it. Code elements that exist within the seed program are taken as the primitives used by the GP system. New generations of individuals are created by the application of mutation and crossover operators.

In more theoretical work dealing with the understanding of GP as an algorithm the initial generation is populated with randomly created programs. As we are interested in the improvement of existing programs the initial generation consists of variants of the original program [9]. When starting GP from a correct seed program fitness improvements are rare. Most changes to the seed will degrade the fitness and few will increase fitness. It is highly likely that any single modification to the program will degrade the program's functionality. It is likely that the seed can be considered a local optima which must be

Table 3.1: Baseline GP Configuration Summary

Parameter	*Value*
Representation	Java AST
Node Selection	Random with AST typing
Operators	Crossover, Mutation
Crossover Rate	0.9
Mutation Rate	0.3
Fitness function	$F = \frac{ExectionCount_{\text{individual}}}{ExecutionCount_{\text{seed}}} + 100 \cdot \frac{Error_{\max} - Error_{\text{individual}}}{Error_{\max}}$
Individual Selection	Tournament (2)
Initialisation Method	Mutated Seed
Max Operator Applications	100
Population Size	250
Generations	100
Elitism	30% of unique fittest

escaped. Using single modifications means the fitness must first get worse before it can get better.

When creating a new individual a parent program is cloned and then modified as per the operators. For each crossover or mutation operator application only one offspring individual program is generated.

3.2.1 Representation

We convert source code to an Abstract Syntax Tree (AST) for modification. This gives us a tree devoid of source code artifacts such as parentheses and line terminators. It provides us with a tree which can nonetheless be modified to produce semantically incorrect programs which do not compile. The number of discarded programs created due to compilation errors is relatively high. The representation used is for the most part defined by the operation of the java AST library used [132].

Operators are not leaf nodes in the AST representation. We workaround this by

selecting leaf nodes when their containing expression is selected for modification. When an expression is selected, a further choice is made as to what part of the expression is modified giving operators the chance of being selected. Such idiosyncrasies are common when dealing with the Java AST library [132].

3.2.2 Node Selection

Our node selection mechanism is based on the typing provided by the AST representation. The probability of node selection is uniform with no distinction between internal or leaf nodes in the tree (as is sometimes contended in the GP literature [5]). Such a distinction may be an arbitrary focus on certain elements as the representation is an AST.

3.2.2.1 Node Typing

The typing used to restrict node compatibility is based directly on the java language syntax. This is also supported by the AST library we use. Typing and object inheritance determine how elements can be changed. When a node such as an expression is chosen it can be be replaced with another expression. Where a statement is selected, it may be replaced by another statement, or statement subtype such as an IF or WHILE statement.

While the use of typing prevents some invalid code replacement, such as replacing an operator with a variable, it does not prevent all syntax errors and programs can be generated which do not compile.

3.2.3 Operators

Once a node is selected, it and the whole AST subtree is subject to replacement, deletion or cloning depending on the node type. A replacement node can be a whole subtree, e.g. in the case of a statement replacement. In this case, the node which is the root of the subtree is of concern. Deletion is only applied to statements which can include blocks of code as sub-trees, for example the body of a loop. If we pick a block, we clone some other random line of code into the block.

A new distinct child program is produced by the application of mutation and/or crossover. Crossover is applied at a rate of 30% and mutation is applied at a rate of 90%. It is also possible for two code edits to be made in generating a single child program where crossover and mutation happen to be selected together.

3.2.3.1 Crossover

Crossover is applied on a pair of programs. A node is selected in both programs, and one is replaced with the other to produce a new program.

Crossover is limited to the exchange of a subtree in one program with a cloned subtree from another. Leaf nodes may not be involved in this process. The idea behind crossover is the exchange of useful code sequences between programs. The modification of a single element in a program would be classified as mutation. As such, our crossover operator only allows the exchange of statement and expression node types.

Crossover is repeatedly attempted until a compilable program is produced, or a maximum of 100 attempts have been made.

3.2.3.2 Mutation

Mutation involves applying change to a single individual program in the form of cloning, deletion or replacement of nodes. It is used exclusively to generate the initial population of programs and in conjunction with crossover for subsequent generations.

How mutation is performed is dependent on the node that is selected within an individual. A node can be deleted, replaced or cloned. If the node is a block statement, then a statement is inserted, with no choice for deletion or modification of the contents of the block. If the node is a particular expression, such as infix or postfix, then the operator is changed for a different one. If the node is a statement, it's contents may be modified or the statement can be deleted. If the node is of another type, such as variable, then it can only be replaced and deletion is not an option due to syntax constraints.

3.2.4 Fitness Function

A performance measure is used to put evolutionary pressure toward improving performance. A measure of functionality is also used to maintain the evolution of correct implementations. While we can use a general measure for performance, functionality measures are currently problem specific.

Programs that compile and finish executing without error within a specific time can be evaluated for their fitness. Programs that do not compile are discarded. Programs which exhibit runtime errors or do not finish within a certain timeframe are given the worst fitness values possible.

The fitness function is a weighted sum of a performance and (100 times the) functionality error as shown in Figure 3.1. Measures of performance (individual instruction count over seed instruction count) and functionality score (test case error over max test case score) are normalised against measures for the seed program. If a program is considered correct, evaluation will give a value of zero for functionality error. Two correct programs (or programs which score the same functionality error) are then comparable by performance only. A program variant which returns the same values as the (assumed "correct") seed and executes the same number of bytecodes will receive a fitness value of one. The weighting of 100 for functionality errors means that semi-functional programs are "binned" or grouped roughly by their functionality and programs with the same functionality are differentiated only on performance measures. In our experiments we are trying to minimise these values and so programs with a value less than one can be considered improved with regard to execution cost [1].

$$F = \frac{ExectionCount_{\text{individual}}}{ExecutionCount_{\text{seed}}} + 100 \cdot \frac{Error_{\max} - Error_{\text{individual}}}{Error_{\max}} \tag{3.1}$$

Fig. 3.1: Fitness Function

[1] We do not consider the possibility of program variants which are functionally better than the seed for our experimental problems as they are very unlikely to occur. This may become an issue as improvement is applied to further problem programs.

3.2.4.1 Performance

The type of improvement that can be expected is related to the measures of software used during GP. For improving performance different measures of time or operation execution can be used to search for different types of performance improvement.

We search for program improvements which reduce the number of operations performed during program execution regardless of the platform-specific costs associated with executing of each operation.

We measure code operations executed specifically by counting bytecodes executed in Java [68]. Measuring bytecode provides an evolutionary pressure towards programs which cause less operations to be executed and so there is an in-built parsimony pressure toward smaller programs.

Our measure of performance is not solely an implementation issue. The evaluation mechanism and how performance is measured affects the type of program improvement that is likely to be found. Measuring operations performed ignores platform specific differences in performance, and means the GP algorithm differentiates programs only on their ability to reduce execution count. This is expected to promote search for general code improvements. Our intent is to improve programs at a high-level, which can be considered program re-design at a general level.

Counting bytecodes executed gives a measure which does not vary between runs and can be expected to be portable [68].

3.2.4.2 Functionality

Measuring functionality at a detailed level is not trivial and requires careful consideration to get the ordering of functionality improvements correct [61]. For example, a sort algorithm which swaps two identical numbers in a list is still considered better than a program which does nothing to the list at all. Such edge cases are not captured by a general count of correct ordering of values and have to be specifically assigned a value on the functionality gradient.

The way functionality is measured during GP can be thought of as a “variable” test

case. In SE work a test case usually returns a binary true or false value. The test cases we use attempt to provide a measure of how much of the functionality is provided. This gives the GP algorithm a way of distinguishing functional programs and a gradient along which improvement can be biased toward.

We provide a number of input test values which are passed to variant programs. The results of which are compared with known correct answers derived from the seed program which is considered our "oracle" implementation [36].

The error count returned for all test cases is summed. This value is subtracted from the error count value for the seed program and divided by the seed error count as can be seen in Figure 3.2. Functionality is calculated as the difference between new program error and the seed error scores divided by the seed error score. If the program error count is the same as the seed, the functionality score is zero.

$$FunctionalityScore = \frac{Error_{seed} - Error_{individual}}{Error_{seed}} \tag{3.2}$$

Fig. 3.2: Functionality Score

For a sort algorithm, we construct a measure for how "sorted" a list of numbers are by counting how far each number is away from where it should be in the correctly sorted list. This is calculated by summing up "errors". If a value is in the right location, the error count is incremented by one. If the value is not in its correct place but also not in the place it originally was, then it has been moved and error count is incremented by two. If the value has not moved at all then error count is incremented by three. This approach measures when a value does not move and allocates the highest error value to these events. At least when a value has been moved, albeit to the wrong place, the program is exhibiting functionality which is more desirable than not moving the value at all. A number of fixed test arrays are used to inspect variant sort programs. The sort algorithm in the Java library is used as an oracle to check correct answers.

We also experiment with a Huffman codebook problem and have written a function which checks if the result returned is a valid set of prefix codes. A valid set of prefix

codes is a set of codewords where no codeword forms a prefix or the initial part of any other codeword.The functionality error is a sum of a number of problems encountered in prefix codebooks.

A prefix codebook should have the same number of codes as there are different characters in the input data. We count the number of extra or missing codes and add them to the error.

We count the number of prefix violations and add them to the error count. A prefix violation occurs when a codeword forms the prefix of any other codeword.

As the goal of prefix code algorithms is to reduce the overall length of valid codewords for a particular input, we sum the length of each codeword and add this to the error.

If either a codebook length or prefix violation error is found the error count is further penalised by the addition of the length of all codewords in the oracle codebook answer to the error count. As codebook length and prefix violations may be small we must ensure that a program which has either of these errors does not get a better score than the oracle answer. If a single prefix error is detected the value of one may be added to the error even though the codeword length is smaller than the oracle answer. Without penalising such cases a program with prefix errors would appear better than a correct oracle answer.

3.2.5 Selection

The lower the fitness value for a program, the "better" it is considered, and the more likely the program will be selected to create a new program for the next generation. A program is selected to be modified by initially picking two (or more) programs at random, with the fittest program being selected. This is referred to as "tournament" selection in the GP literature [106].

3.2.6 Elitism

GP is a randomised algorithm which can be destructive when modifying programs, generations can be created consisting of less fit individuals than their ancestors. To ensure the

best fitness in each generation does not drop, *elitism* promotes the best programs from the parents generation to the children's. Typically the worst programs in the children's generation are replaced by the best of their parents.

As program modification is highly destructive elitism rates used for program improvement tend to be high. Some approaches do not replace low fitness individuals at all and simply delete the lower 50% of programs ranked by fitness [36]. Existing programs are assumed to have relatively high fitness to begin with. In this scenario elitism is useful for removing defective programs which have poor fitness as opposed to propagating high-fitness programs.

We use a form of elitism at the rate of 30% which we term "diverse elitism". When gathering the elite programs from the previous generation the decimal part of fitness value is ignored. We then take a program for the top 30% *unique* fitness values. Ignoring the decimal place means we are largely ignoring performance and only distinguishing on functionality. For each whole number fitness value we select only one program. The effect of this is that we do not select similar programs which have the same functionality. This approach prevents any one program from dominating the generation. The least fit 30% of the new generation by fitness value is replaced by these diverse elite programs.

3.3 Self-focusing Genetic Programming

When a program modification is performed a change in the fitness value is likely to be produced. The change in fitness is used to attribute information to the location of the modification. As more modifications are performed using the same program to produce more variants further information is attributed to locations within the program. Over time repeated modifications differentiate the different locations in a program. Furthermore, we specifically inspect the use of the functionality measure as a key factor in highlighting the location of performance improvements in code.

The canonical GP configuration as described in the previous section is extended in this section. Our solution takes the form of a number of additions to the GP algorithm.

3.3.1 Design Methodology

Our design methodology used known optimisations in Bubble Sort to test whether our bias allocation mechanism was highlighting the best locations. A variety of bias allocation rules were investigated and the most promising is used for our evaluation. The experimental work which was used to derive the evaluated bias allocation rule can be found in Appendix C.

3.3.2 Bias Values

To allow nodes to be biased each node in a program AST has a value attached. The bias value determines the likelyhood that a node is selected to be modified by an operator [6].

3.3.3 Value Initialisation

Initially all nodes are given a value of one, the highest allowed value, which promotes *exploration*. All nodes are modified at least once [71]. There is significance to the value of one as it can be taken to mean that the node has not been changed before. If the node is changed and the bias updated then the bias value will never be set to one again due to the way bias is updated. Any node which has been modified will have a value less than one. If the location of that change is found to have a value of one after a code change the bias value is immediately set to 0.9 before bias update is applied.

3.3.4 Modification

When a program is modified an offspring or child program is produced. Bias values at the location of modification are changed in the parent as well as the child program. The amount that bias value are changed by is dependant on the fitness change.

The assumption within our approach is that bias values will be changed in the parent program as it is likely to be selected again in the generation of a new program. For every time the program is modified bias values are updated and also passed along to, or "inherited" by, the offspring program. Bias is updated in different ways depending on the

offspring program's fitness.

3.3.4.1 Decay

When programs do not compile or timeout the bias value is decreased or subject to *decay* at the location which produce the degraded program in the parent program. Uncompilable offspring programs are discarded when created. Decay is only applied to parent programs in this scenario.

3.3.4.2 Functionality Change

The rule by which we update or decrease bias values is based on changes in functionality between parent and child program. If the offspring functionality is less than the parent, the parent location of modification is increased and the child location of modification is decreased. If the offspring functionality is more than the parent, then the parent location is decreased and the offspring location increased

3.3.4.3 Bias Update Magnitude

The amount by which bias is updated at each node is fixed. The change does not depend on the magnitude of functionality change. Bias updates are however made with respect to the bias existing value. Updating bias is done as a quotiant of the current value. An increase or decrease in bias is performed so that values may never be smaller than zero or larger than one. If the bias value is 0.6 and we want to increase it, we may only increase it by a an amount which will not set the value to one or above.

Bias values are increased by a quarter of the difference between the existing value and one. If the bias value is 0.6, then an increase would result in a bias value of 0.7.

Decreases can not set the bias value to zero or less. Existing values are decreased by 0.05 times that value. A bias value of 0.6, would decrease to 0.57.

The magnitude chosen must adequately tell apart nodes in a program. What is important is the node ranking produced as opposed to the magnitude of bias of any node.

We have experimented briefly with making the magnitude of bias change related to the change in fitness observed but were unable to yield any interesting results. Rules which allocate magnitude of bias in response to magnitude of fitness change are noted in Appendix C.

3.3.5 Updating Node Bias During GP

The previous description can be used on a single program without being part of GP. We term this "static" bias allocation which is performed to observe the bias produced in a more controlled environment on the same program.

Dynamic bias updates can be performed during GP. As many programs are created during GP and GP is a randomised algorithm it is more difficult to observe bias allocation.

Bias is updated in the parent program which was cloned as well as in the new child program. In the case of crossover, the fitness of the second parent which provided genetic material is also compared to the child and bias is updated in the child a second time.

It is likely that there is no one rule which can find out all improvements in a program. Similar work in fault localisation has recently proved that such a rule does not exist for fault localisation [153].

3.3.5.1 Node Selection

Node selection is performed by tournament selection although any selection mechanism which takes into account bias could be used [48]. The tournament size is set to 20% of the number of nodes in the program.

3.3.5.2 Bias Inheritance

When a new program is generated during GP all the parent values are copied to the offspring program. When nodes are replaced during crossover the replacement node takes the bias value of the original node. The values of the nodes in the rest of the subtree maintain their original values.

Under mutation the root node is given the bias value of the node it replaces. All other nodes in the subtree are initialised to one. If this program is selected for modification subsequently these new nodes will have a high likelyhood of being selected. In this way, we ensure that the bias update mechanism gathers information about all nodes in the program so it can make more informed modifications to the program.

Bias propagates through generations during GP in this manner as well as by moving individuals from one generation to the next during elitism. Inheritance happens before program evaluation and bias update.

3.4 Gaussian Bias

We compare the use of dynamic bias during GP to the use of randomly modified bias [6]. The distribution used for random bias is Gaussian. Bias is randomly changed after every modification but normalised within the bounds of zero and one.

3.5 Summary

We have detailed our baseline GP system and its extension with mutation-derived bias which is performed dynamically during GP. We also identify a technique which randomly modifies bias from the state of the art which will be be used for comparison.

Chapter 4

Evaluation

In this section program modification is evaluated as a mechanism to indicate locations of performance improvements in code.

4.1 Overview

We begin by showing how GP performs on our problem set as a baseline for further experiments and to characterise the operation of GP in the context of program improvement.

We know the improvements that can be found in each of our test problems and what code needs to change to produce these improvements. By setting the bias appropriately for an improvement in one of these problems, Bubble Sort, we can observe the optimal effect that bias can have on GP in a static context. We do this to validate the approach of using bias to affect GP. We then use a program profiling technique and our mutation-based bias allocation technique to similarly apply bias statically and compare.

Hand-made bias for Bubble Sort is effective at increasing GP's chances of finding known improvements. We test profiler-derived bias on Bubble Sort and a variant which includes an extra redundant loop of the whole algorithm. The purpose of this problem is to inspect whether a profiler can be deceived as to the location of a performance improvement. While the use of profiler derived bias increases the chances of finding an

improvement during GP greatly, the profiler technique is far less effective on the deceptive problem and is out-performed by GP without bias in later generations.

We then generate bias by repeatedly modifying the seed program and evaluating the effect this bias has on GP. Mutation-derived bias greatly improves GP on Bubble Sort and outperforms a profiler on the deceptive problem. This shows the ability of mutation-derived bias to find improvements in code. Mutation-derived bias is also not as deceived as the use of a profiler.

Although program improvements can be highlighted statically with mutation-derived bias, the process is computationally expensive to perform. As the derivation of bias is distributed across a wider range of program variants when performed dynamically during GP we inspect the effect of dynamic bias on the GP process. If improvements can still be found then we know mutation-derived bias is possible in a dynamic context.

On our Bubble Sort problem, dynamic bias allocation increases chances of finding an improvement in early generations, but is eventually overtaken by GP without bias. We apply GP with dynamic bias to a range of sort algorithms and a Huffman codebook generation algorithm to observe the generality of this approach. Dynamic bias shows an advantage in improving the chances of finding improvements on 7 out of the 12 problems, but is eventually overtaken by GP without bias on 2 of these problems.

Bias can also be changed entirely at random per a Gaussian distribution. The bias is shaped by the evolutionary pressure of fitness selection in GP. Fitter programs and their associated bias values are propagated through the population. We find that this approach improves GP on 7 out of 12 problems. Gaussian bias is better on 6 of the 12 problems when directly compared with dynamic mutation-derived bias.

While our results so far are somewhat inconclusive as to what method is unanimously better than others, we can say that different approaches are more useful on some problems that other approaches. A profiler can be expected useful where improvement opportunities are co-located with bottlenecks in code. We can explain the ability of Gaussian bias by considering that Gaussian bias is shaped by the evolutionary pressure of GP which is dependent on the selection of fit programs. Programs with high fitness and appropriate

bias which lead to high fitness offspring will propagate. Bias which produces high fitness programs will thus propagate. We can say that Gaussian bias performs well where the sequence of variant programs required are all of high fitness. To validate this hypothesis, further analysis is required which is left to future work. By elimination we can say that mutation-derived bias can find improvements which are not colocated with bottlenecks and have a sequence of lower-fitness variant programs which must be created to reach an improved program. Again, this hypothesis would require further work.

Our final analysis is to perform every valid edit (per AST typing) to a test problem with every possible replacement code element as found in the test problem. This requires n^2 - n evaluations of a program, where n is the number of AST nodes in the program. The purpose of this analysis is to see if there is an overall difference in the fitness values created when modifying nodes which are known to be part of an improvement. If these nodes can be separated from other nodes then we can validate that mutation-derived bias can find these improvements during GP. We find that on 8 out of 12 problems nodes which are required to change for an improvement to be found compiled more often under modification. This analysis suggests that magnitude of performance change may also be used as a factor in highlighting improvement opportunities. This work has the potential to aid further design of bias update rules and is left to future work.

The core finding from this work is that locations, which compile when modified, are where search can be focused to find improvements. Further to this, spectrum analysis suggests that modifications which reduce performance values the least are worth focusing modification on. The results shown support these claims on the majority of programs inspected though do not generalise across all problems tested.

4.2 Improving Programs with GP

The results of our instantiation of "canonical" GP are introduced in this section to provide understanding of the process and characterise the search performed. We begin by showing the results of a successful run on one of our test problems. The only part of this setup

that may not be termed "traditional" is the elitism mechanism which is geared towards maintaining diverse programs as per their functionality score. The purpose of this section is to establish a baseline for how GP is applied to our problem set.

We introduce our basic experiment on a naive form of Bubble Sort 1. Bubble Sort is of little use practically due to it being relatively inefficient when compared with other sort algorithms. Bubble Sort is however an ideal candidate for improvement as a simple algorithm where improvement is easy to understand.

```
1  class Sort1Problem {
2    public static void sort( Integer[] a,  Integer length){
3      for (int i=0; i < length; i++) {
4        for (int j=0; j < length - 1; j++) {
5          if (a[j] > a[j + 1]) {
6            int k=a[j];
7            a[j]=a[j + 1];
8            a[j + 1]=k;
9          }
10       }
11     }
12     return a;
13   }
14 }
```

Listing 1: Naive Bubble Sort implementation [146]

There are a number of ways of producing a functionally equivalent improvement in bytecode execution. We know of one edit which improves the code in Listing 1 by reducing the number of bytecodes executed [146]. The improvement involves replacing "i++" on line 3, with "length−−" as compared in Listing 2.

```
3  for (int i=0; i < length; i++)        for (int i=0; i < length; length--)
```

Listing 2: Bubble Sort Outer-Loop Improvement

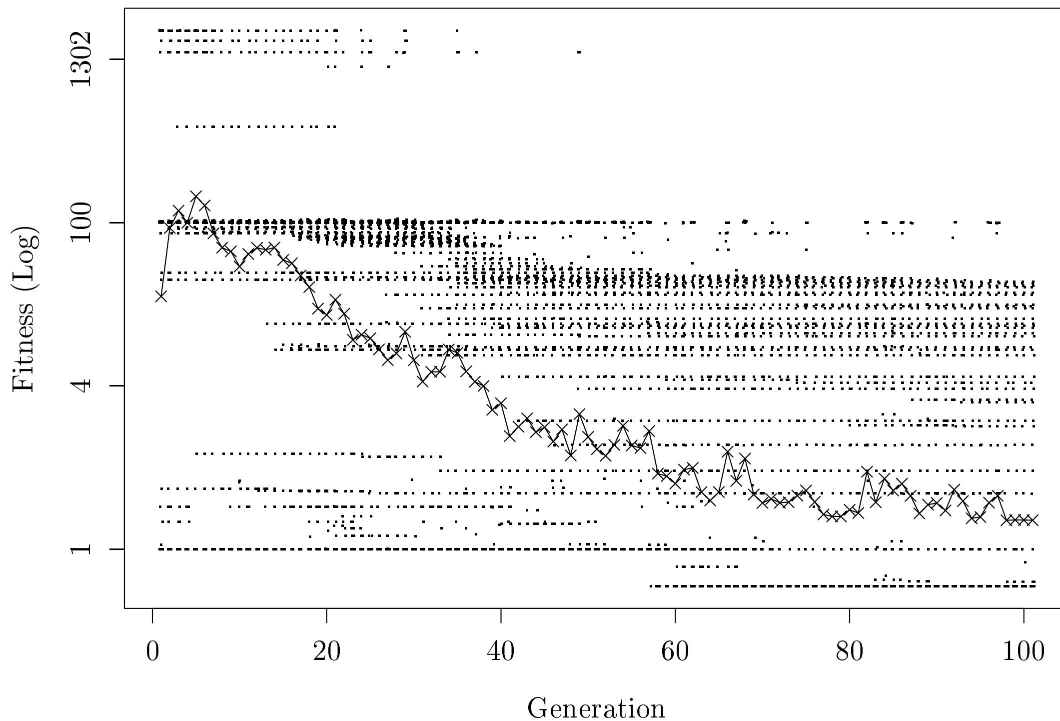
```
4  for (int j=0; j < length - 1; j++)    for (int j=0; j < length - 1 - i; j++)
```

Listing 3: Bubble Sort Inner-Loop Improvement

During the course of our experiments another equivalent improvement was found where “length − 1” on line 4 is changed to “length − 1 − i” as compared in Listing 3. Both these code changes improve the Bubble Sort by skipping iterations over portions of the array which have already been sorted. As Bubble Sort iterates over an array, elements are sorted from the end of the array forward. The number of sorted elements at the end of the array grows with each iteration and does not need to be iterated over again.

Figure 4.1 shows the fitness values for all individuals in each generation and the average fitness per generation of an example GP run on Bubble Sort. Each dot in the graph is the fitness of a single individual program. The line shows the average fitness of each generation. Crossover begins to operate (the first generation is created with mutation only) within the first number of generations. On average, there is a reasonable spread of functionality created during the initial generations. In following generations elitism favours better programs which become more numerous. Many variants of the better programs are created and begin to dominate in subsequent generations. In later generations there is a greater chance that a new individual will have a higher fitness. Of note is the modality of the distribution. Few individuals evaluate to some regions of the fitness gradient. This can be seen in the grouping of individual fitness values into rows in Figure 4.1. This may be because our multiplication of 100 of the functionality provides spacing between functionality levels which the performance variability is not

Fig. 4.1: Fitness Scatter and Average for GP on Bubble Sort



large enough to fill in. The variability of performance is low enough that it does not fill in the gap between functionality "levels". Although the tournament size of 2 we use is a light pressure on the population fitness, the elitism rate of 30% is more aggressive and this can be seen as the algorithm progresses where the worst individuals are being replaced faster than they are being created.

Considering that improvements in fitness are rare it may seemingly appear that improved programs are found more or less at random and the genetic nature of the GP algorithm does not give much advantage over random search for the problem of program improvement. Although there is the temptation to classify this search process as being more "random" than it is "genetic", GP on subsequent problems show that genetic operators are advantageous for delivering even a single fitness improvement. Even though there is no fitness improvement on the seed program, the search process is busy exploring lineages of programs which are necessary to reach an improved variant.

The search starts at a local optima when using a seed program. This optima must be "escaped" before an improvement can be made. In a sense, we need to break the program before it can be improved. This is especially the case when using programming languages which have complicated syntax and semantics.

In improved programs we see both the reuse of existing code and the modification of code at a fine level. Only a small number of edits are required to improve some programs. On others, it is necessary for a "block" of code to be reused and subsequently modified to produce a variant program. In either case a strict or rigid definition of a building block may be overly restrictive. If a building block preserving GP extension were to be used it would have to allow the modification of the contents of a building block.

Fig. 4.2: Best individual programs for GP on Bubble Sort

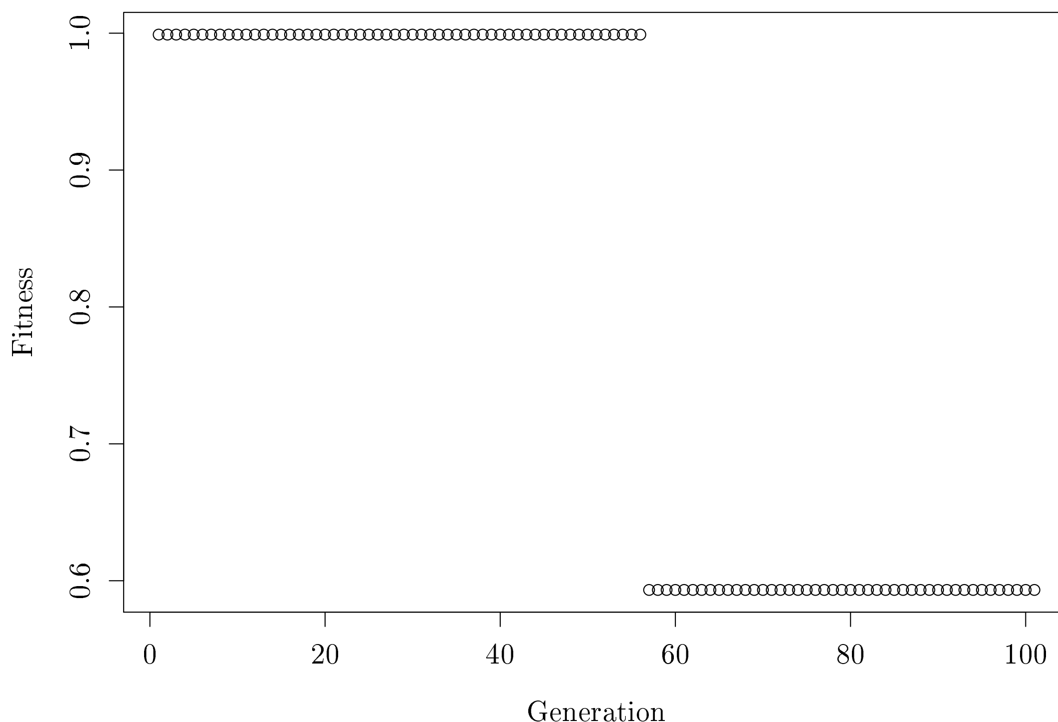


Figure 4.2 shows the fittest (i.e lowest value) individual from each generation in Figure 4.1. The particular GP run we picked shows a single distinct jump to 0.6. This represents a saving of 40% of the execution cost of this algorithm in terms of bytecodes

executed (note this does not equate to 40% wall-clock time saving). The graph presented here demonstrates a typical GP run. Many runs on these problems produce a small number of stepped improvements which is in contrast with traditional GP graphs which show many finer improvement steps. Of the GP runs that do find an improvement, many show a single step to an improved version of the program. The graph shown here is largely representative of improvements on this problem. We could however have included more interesting graphs showing more steps of improvement. It is rare to find a problem and GP setup that will give a very smooth gradient from seed to an improved version of a program such is the makeup of the fitness landscape. Once the improvement is found superfluous code can sometimes be removed in subsequent generations resulting in a number of visually distinct improvement steps.

The best fitness individuals from each generation in a single run are shown in Figure 4.2. When using GP on existing software the algorithm is starting from a local optima and is attempting to find one of very few improvements. The best fitness does not show incremental improvements and the graph remains flat for the majority of the search process. Where GP is able to grow a program outright better programs can be found in almost every generation of the GP run. When GP frequently finds improvements the resulting graph is smooth as opposed to a sudden step improvement as seen in Figure 4.2. This comparison can distinguish between the scale of the problem and the jaggedness of the search space where a smooth gradient of program improvement is not possible. Visually, what we see in Figure 4.2 looks like what we would see if we zoomed in on the tail end portion of a GP run which had began with random programs. GP may make a better program in every generation when starting from randomly generated programs as opposed to a seed program which is already considered highly fit. We think of GP on existing software as attempting to find one of a very few final improvements. The graph of best programs during GP on existing software can be thought of as a "zoomed in" view on the final portion of a GP which is creating a solution from scratch.

4.2.1 Problem Set

We apply GP to sort algorithms and a Huffman codebook generating algorithm. The sort implementations were taken mostly unmodified from online sources [135]. The Huffman codebook algorithm was written to include one of our sort algorithms as a subfunction as listed in Appendix B and characterised in Table 4.1. An improved version of each program was found for all test programs in our problem set.

These programs consist of common algorithms that are well studied. There are many versions in existence and there is reasonable interest in their improvement. They are relative small but their size is large enough that we can assume GP is unlikely to generate these programs from scratch without being given domain specific primitives [2]. They are big enough to pose as difficult problems for GP to improve but small enough that any improvements may be understood. The problems are neither too easy or prohibitively difficult for GP, and as such, are sufficient to measure the difference bias makes on the search process for program improvement.

We create a further sort problem by modifying Bubble Sort to contain an extra outer loop over the entire algorithm. This adds extra locations in the code which can be modified and greatly increases the execution cost of the algorithm. The extra outer loop is executed twice but almost doubles the execution cost of the entire program. This problem, termed “BubbleLoops”, is used as a deceptive problem in subsection 4.4.2.

The “Huffman codebook” problem introduces a new fitness function and associated candidate solution for improvement. The Bubble Sort algorithm is embedded in the candidate solution as we are familiar with the improvements which exist. We inspect only one candidate solution for prefix codebook generation as that is all that is needed to show our ruleset does not scale well to a larger program. Until we can understand why our rules are not advantageous on this problem, or until we can find a rule that does work on this problem, we do not gain much further information about dynamic bias allocation by applying rules to larger problems.

We use related problems so that we may control the experiments and make step-wise inferences. If we use programs that are too different, it is more difficult to make inferences

about the differing results. We can see this when we try our approach on Huffman. Our approach does not show the same improvements as it does on our sort programs. The Huffman codebook problem, although incorporating Bubble Sort (and in turn the same improvement opportunities), has a different fitness function, larger seed program size, and is broken into a number of methods.

Problem Name	AST Nodes	LOC	Best Fitness	% Discarded
Insertion Sort	60	13	0.91	73.3
Bubble Sort	62	13	0.55	71.4
BubbleLoops	72	14	0.29	71.8
Selection Sort 2	72	16	0.99	70.9
Selection Sort	73	18	0.98	71.2
Shell Sort	85	23	0.95	71.4
Radix Sort	100	23	0.99	80.5
Quick Sort	116	31	0.46	72.7
Cocktail Sort	126	30	0.85	73.7
Merge Sort	216	51	0.95	73.2
Heap Sort	246	62	0.59	71.1
Huffman Codebook	411	115	0.57	83.8

Table 4.1: Problem improvement overview

Table 4.1 shows the improvements found in our test problems when GP was applied. This shows that for a number of common algorithm implementations GP can find a modification which reduces execution count. Sort implementations were taken “off-the-shelf” from various sources [135] as reproduced in Appendix B. Of the sort implementations which were taken without modification, and which we can assume were not written specifically to be inefficient, a reduction in bytecode executed was found. A count of the AST nodes available for modification in each program shows the number of modifications points which can be chosen during the application of GP. Lines of code are

counted including those containing braces [143]. The number of discarded programs is a ratio of the number of programs which do not make it into each generation over the total number of programs produced. Discarded programs include those which do not compile and intermediate programs generated where mutation and crossover are applied together. This is different to software mutational robustness [115] which measures how the number of modifications which leave a program's behaviour unchanged. In our experiments, a minuscule number of programs, certainly less than 1%, show no behavioural change when modified. This may be a result of our GP configuration which has built-in parsimony. Superfluous code is less likely to be present under this setup. Bloat is not an issue and we are operating on relatively small programs throughout the GP process.

4.3 Statistical Comparison

In the following sections the effect of bias on the GP process is graphed for GP with and without bias as two separate data series. The GP process is repeated 100 times. Data points for each GP configuration are derived by resampling the 100 GP runs. For each generation 100 mean values were calculated from 1000 samples (with replacement) and used to surround data points with grey bands showing quantiles at 97.5 and 2.5 to give an estimated 95% confidence interval. Separate data series lines for GP with and without bias can be seen in Figure 4.3.

In subsequent graphs, starting with Figure 4.6, we graph the *difference* between two data series to observe at what generations during the GP process there is a statistically significant difference. Quantiles at 97.5 and 2.5 are drawn surrounding these differenced values to give an estimated 95% confidence interval around the difference using sampling with replacement 1000 times. When the confidence interval bands intersect the x-axis we can say that there is no statistically significant difference between GP with and without bias. The power of this test is dependent on the difference between the means for GP with and without bias. The difference between the two data series lines in Figure 4.6 is shown as a single difference line in Figure 4.6.

4.4 Static Bias

This section introduces the application of bias to the initial program before the application of GP. By "static bias" we are referring to node bias which is not updated or modified in magnitude during a GP run. Bias is applied to the seed program before running GP. The bias affects the probability that certain nodes are chosen for modification. The optimal effect that bias can have in a static context is shown through the application of hand-made bias values which match the locations of known improvements. Here we show that static bias can improve GP on our Bubble Sort example.

4.4.1 Hand-Made Bias

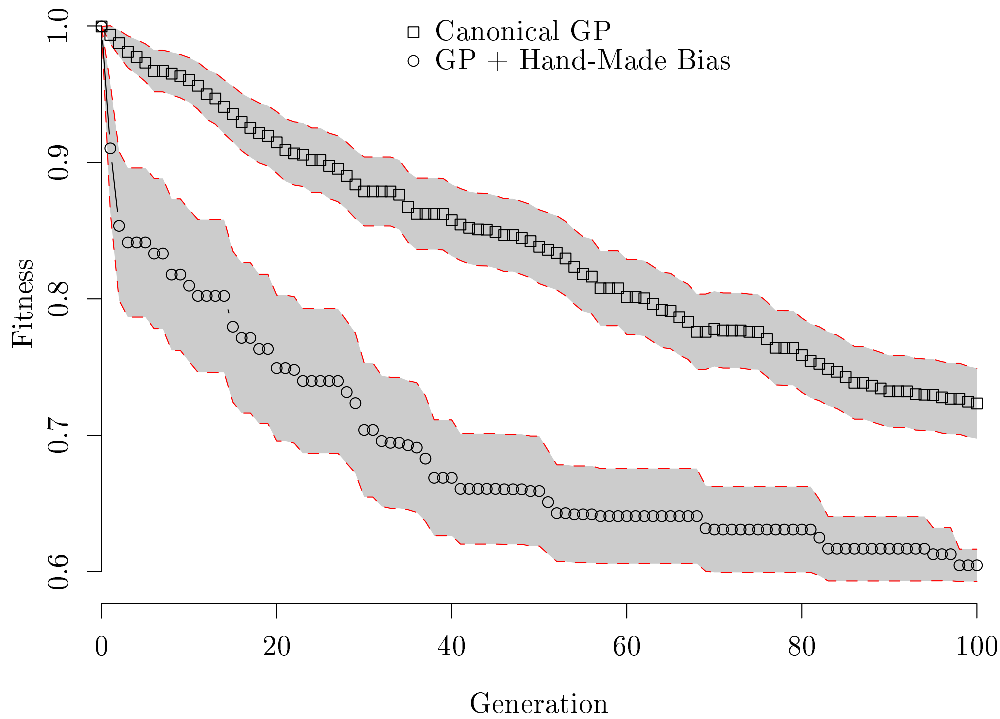
To observe the effect of biasing locations in a program we manually set bias high for locations in Bubble Sort which we know can be changed to produce an improved version of the program. This hand-made bias represents an ideal focus of GP change for optimising Bubble Sort and represents what we aspire to produce automatically. We set values to 1 for each node in the program which needs to change separately for both the known improvements in Bubble Sort.

We take known improvements in Bubble Sort as shown in Listing 2 and Listing 3 and hand-craft bias values which would be beneficial for guiding random mutation toward the improvement. This provides us with an indicator for what the optimal impact bias can have on the GP process in ideal conditions.

The rate at which both improvements are found with GP is demonstrated to show differing improvements are more difficult, depending on the edits required for each improvement as well as the fitness of individuals along this edit path. An improvement requiring more changes is going to take more individuals to find using GP.

Figure 4.3 shows how GP is improved when bias is set to 1 for nodes involved in the second improvement as shown in Listing 2 (" i++" replaced with "length−−" on line 3). The rate of improvement appears the same as in standard GP but the improvement found by both approaches is different. Even though the overall rate of improvement

Fig. 4.3: GP with Hand-Made Outer-Loop Bias on Bubble Sort



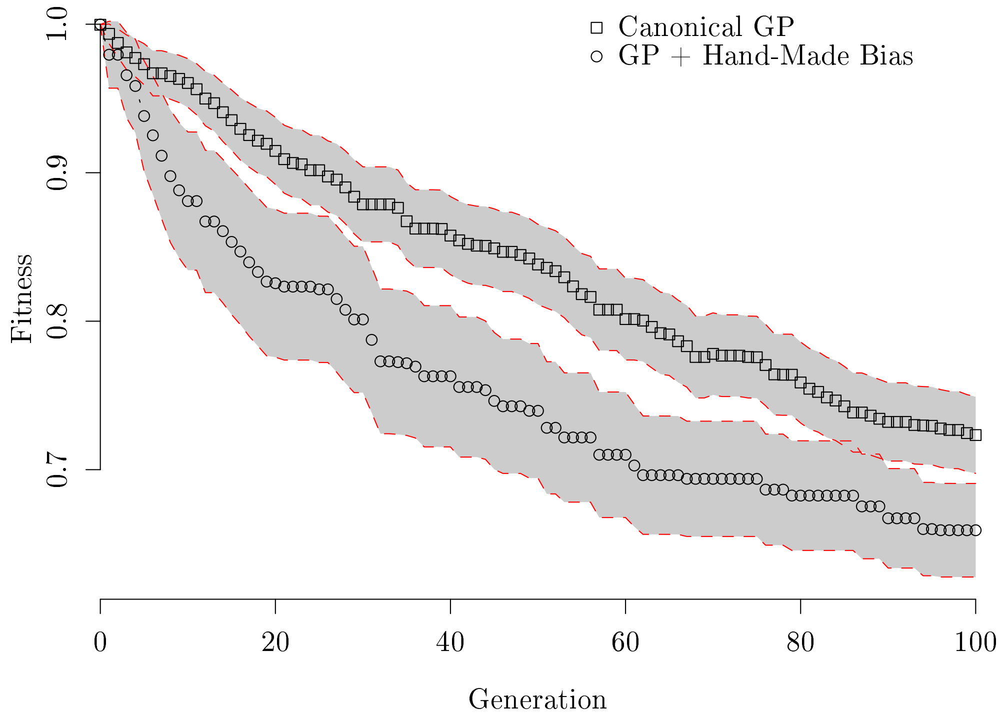
is roughly the same, GP with bias in this figure finds an improvement which is more "obscure" in that it is more difficult to find.

Data points in Figure 4.3 are derived by resampling 100 GP runs as described in section 4.3. For each generation 100 mean values were calculated from 1000 samples (with replacement) and used to surround data points with grey bands showing quantiles at 97.5 and 2.5 to give an estimated 95% confidence interval.

Figure 4.4 shows how a GP run has a higher chance of optimising Bubble Sort when bias is set high for the simpler of the two improvements (''length − 1'' replaced with ''length − i'' on line 4) Listing 3. As this is the simplest improvement it can be expected that our unbiased GP run will find this improvement more often than the more complex.

Considering both these graphs gives insight into what makes program improvement difficult. There are a number of ways of achieving the same improvement even if the measurable improvement is the same overall. These different cases show what is meant

Fig. 4.4: GP with Hand-made Inner-Loop Bias on Bubble Sort



by problem difficulty. The length of lineage required to produce a particular improved program variant influences the chances of finding that improvement. The fitness of the program variants along this lineage also affects how likely the programs are to be selected for subsequent modification and how likely the lineage is to be traversed.

Of the improved programs found when using standard GP the simpler improvement (Inner Loop Listing 3) is found 96% of the time and the more complex version (Outer Loop Listing 2) is found 4% of the time. When the inner loop is highlighted and an improvement found as shown in Figure 4.4, the inner loop improvement is found 94% of the time and the outer loop 6% of the time. When the outer loop is highlighted and an improvement is found as shown in Figure 4.3, the outer loop improvement is found 88% of the time and the easier improvement being found 12% of the time. We distinguish between these different improvements only in passing and scope our work to the rate of improvement of a program which GP is capable of. We could use the distinction of these improvements to assess the complexity of improvement that a bias technique is

likely to uncover. This raises the point that although we may not be able to make GP find improvements more quickly on all problems, we may be able to affect the diversity of the solutions which GP may find. It may be useful to discover different variants of an improvement which may have different applications in different contexts but this is limited to future work. Our fitness function does not capture the different improvements found.

Our problem is the automated generation of an "optimal" bias which can guide software change with respect to a program measure.

4.4.2 Profiler-Derived Bias

We show how a bias derived using profiler techniques compares to optimal bias. Techniques for profiling have been used previously to apply bias to large programs [73]. We investigate the effects of profiling in a static context before GP is applied.

When profiling is used a program is instrumented to count the number of times a line of code is executed when run. Profiling provides an execution profile for all lines of code in a program. The execution frequency can be used to bias locations in a program. Profiling a program is shown to improve GP almost as good as a hand-made optimal bias but can however be deceived when the location of an improvement is different to the location of a heavily executed line of code. The deceptive problem used is a variant Bubble Sort problem where an extra redundant outer loop is added to the program. Improved versions of the program can be attained by modifying the outer loop. The problem is deceptive for a profiler as this outer loop is only executed twice which means the profiler technique ranks this code as having among the lowest execution count.

Table 4.2 shows the line count for each line of code when the program is executed over 20 calls. For every AST node in each line we set the bias the same in relation to the number of times the line is executed. Bias is set to 1 where the execution count is highest with bias being allocated as a fraction of this maximum value for other nodes.

Figure 4.5 shows how a profile-derived bias affects GP. GP is able to find a more improved version of the program more quickly. The chance of finding the inner or outer

Line	Count	Bias Value	Rank
for (**int** i=0; i < length; i++)	284	0.5	3
for (**int** j=0; j < length − 1; j++)	5894	1.0	1
if (a[j].compareTo(a[j + 1])<0)	5630	0.95	2
int k=a[j]	793	0.13	4
a[j]=a[j + 1]	793	0.13	4
a[j + 1]=k	793	0.13	4

Table 4.2: Profiler-Derived Bias Values for Bubble Sort

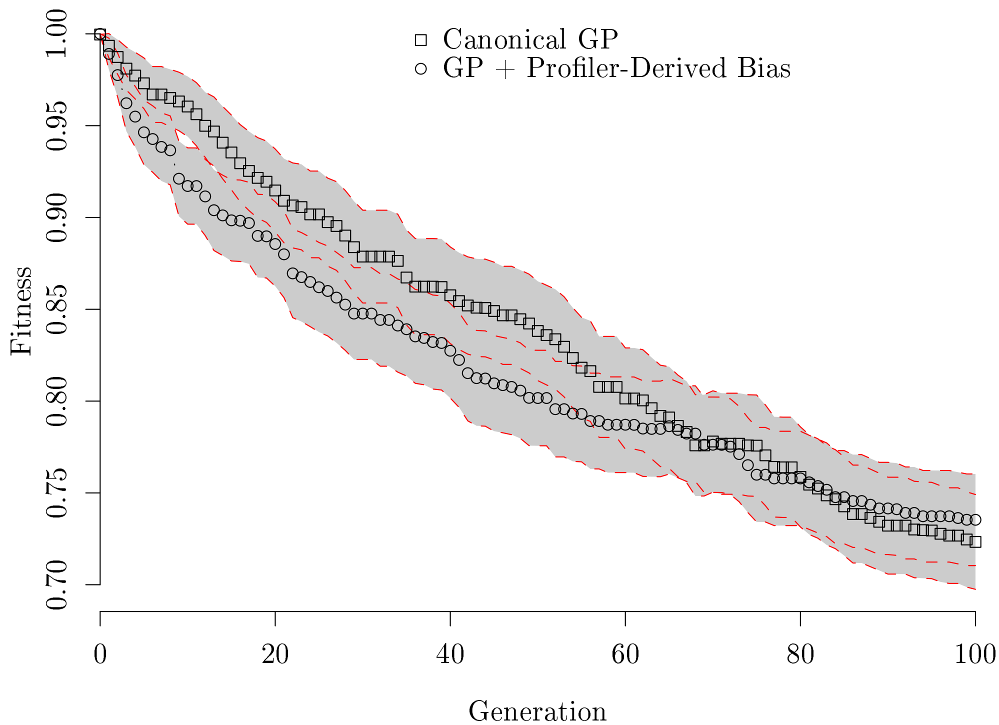


Fig. 4.5: GP with Profiler-Derived Bias on Bubble Sort

loop improvement was equal.

We redraw the results of Figure 4.5 in Figure 4.6 to show the *difference* between 100 values for GP with and without bias as previously described in section 4.3. Quantiles

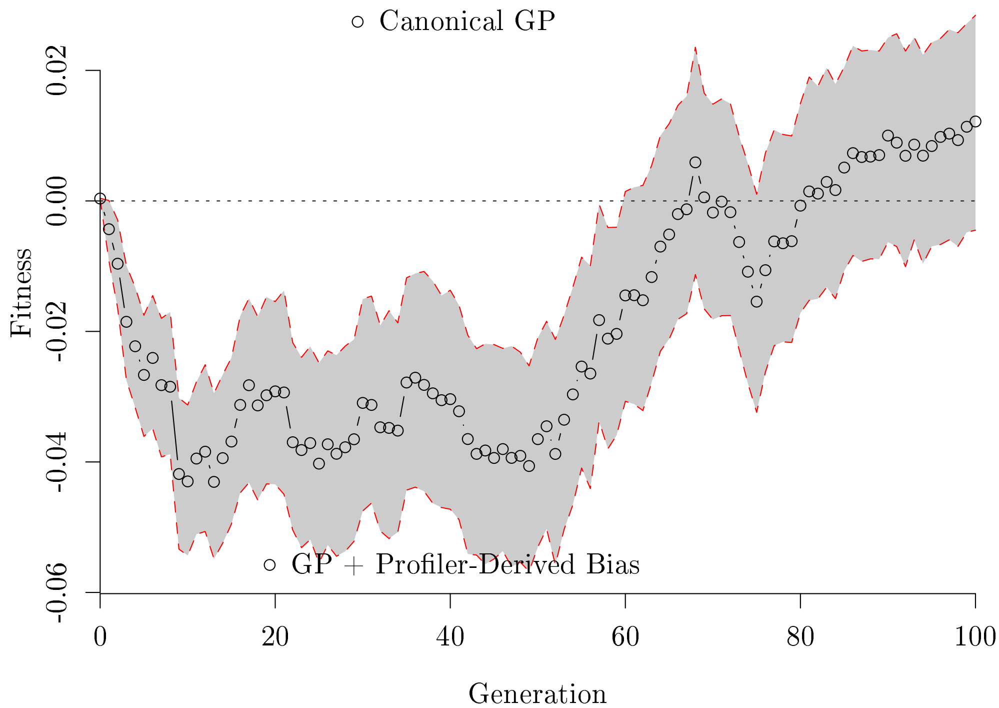


Fig. 4.6: Profiler-Derived Bias Difference on Bubble Sort

at 97.5 and 2.5 are drawn around these differenced values to give an estimated 95% confidence interval around the difference using sampling with replacement 1000 times. When the confidence interval bands intersect the x-axis we can say that there is no statistically significant difference between GP with and without bias. The power of this test is dependent on the difference between the means for GP with and without bias.

An improvement is more likely to be found up until the 90th generation when standard GP is then more likely to find an improvement. This may be due to static bias becoming less effective as the search progresses and the population becomes more diverse in comparison to the seed program. Profile-derived bias provides an advantage to the GP process on this problem.

4.4.2.1 Deceptive Problem

Profiling techniques can show where a problem manifests in a program and can be used to guide GP. In this section we show that a profiler can be deceived as to the location

of improvements. The location of a bottleneck is not always the same as where change should be applied to improve a program. Where a performance bottleneck manifests is not always the same as where the code should be changed to reduce the execution cost of a program.

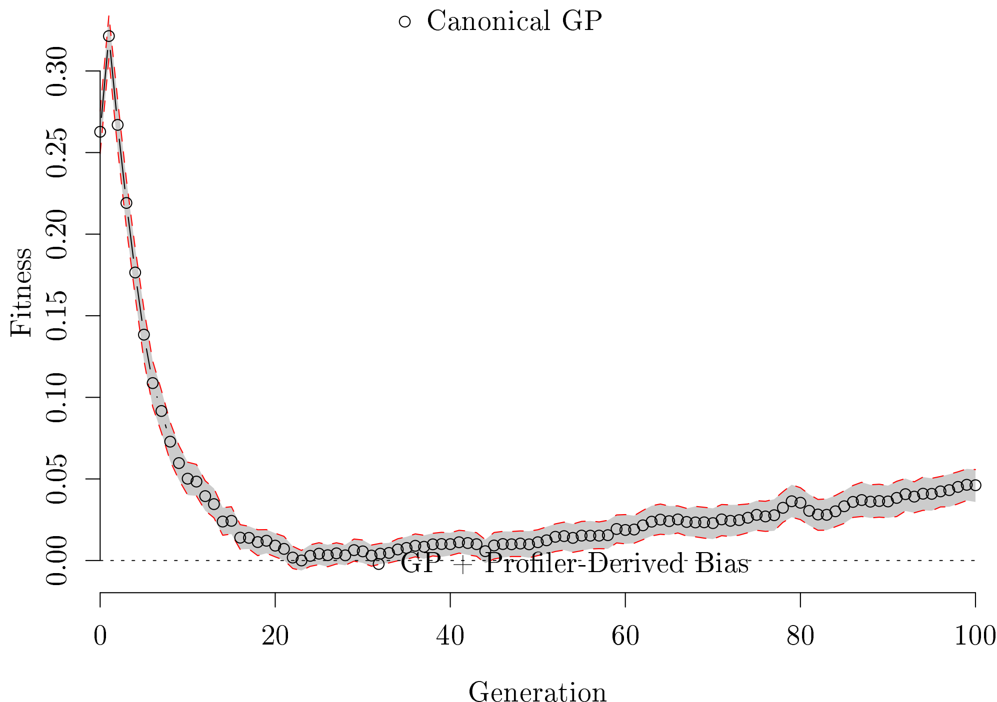


Fig. 4.7: GP with Profiler-Derived Bias Difference on Deceptive Bubble Sort

By modifying Bubble Sort to include more redundant iterations a deceptive problem is created. When a profiler is used on this problem lines of code which are executed most frequently are highlighted. The extra outer loop which causes the redundant iterations is given the lowest bias ranking when attributed with a profiler. Figure 4.7 shows the difference between Profiler derived bias and canonical GP. Profile-derived bias is only equivalent to GP between the 20th and 30th generations but presents a net disadvantage to the process on this problem. When analysed we find that the use of a profiler finds the most improved version of the program half as often as standard GP.

A profiler is particularly useful where an improvement exists at the same location as costly code. How often an improvement is co-located with costly code would require a

broad review of software and is considered out of scope of this thesis. When the location of a bottleneck is not the same as the location of an improvement opportunity a profiler will highlight the bottleneck. Focusing change on the bottleneck will draw search effort away from the improvement and toward the bottleneck, slowing the search process.

4.4.3 Mutation-Derived Bias

We attribute bias to locations in a program in response to the measurable difference between an original program and its modified variant. By repeatedly modifying a program and attributing bias to the modified locations, bias values for all locations in the program can be built up. We term this process "mutation-derived" bias. The purpose of mutation-derived bias is to find locations in a program which show some hint relevant to to program improvement when modified. GP can be guided to locations in the program which have shown hints of being particularly relevant to improvement opportunities. Mutation-derived bias can guide GP on Bubble Sort and is not misdirected on deceptive Bubble Sort. Although the bias generated is accurate, it is costly to analyse software in this way. We create mutation-derived bias in a static context before the application of GP, but as mutation-derived bias through the same cycle of modification and evaluation as in GP, we can derived bias during the GP process, and is so termed "Dynamic Bias".

This section shows the bias overlay that mutation produces on our bubble sort example program. The bias values and ranking of nodes in a program due to these values is shown in Table 4.3. Nodes which must change to produce an improved version of Bubble Sort are highlighted in this table.

Rank	Absolute Node Value	Node Number	Node Subtree
15	0.920865	1	for (int i=0; i < length; i++) { for (int j=0; j < length - 1; j++) { if (a[j] > a[j + 1]) { int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k; } }}
12	0 .921351	2	int i=0
22	0.919202	3	i=0
26	0.918849	4	i
41	0.917408	5	0
6	0.926602	6	i < length
8	0.923221	7	i
9	0.922238	8	length
1	0.942054	9	i++
7	0.923594	10	i
18	0.920286	11	{ for (int j=0; j < length - 1; j++) { if (a[j] > a[j + 1]) { int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k; } }}
49	0.916529	12	for (int j=0; j < length - 1; j++) { if (a[j] > a[j + 1]) { int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k; }}
4	0.939287	13	int j=0
59	0.914065	14	j=0
60	0.913459	15	j
42	0.917401	16	0
3	0.939547	17	j < length - 1
52	0.916287	18	j
58	0.914073	19	length - 1
54	0.916139	20	length
56	0.915849	21	1
57	0.915054	22	j++
36	0.91798	23	j
51	0.916346	24	{ if (a[j] > a[j + 1]) { int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k; }}
44	0.917204	25	if (a[j] > a[j + 1]) { int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k;}
2	0.940164	26	a[j] > a[j + 1]
27	0.918658	27	a[j]
14	0.920887	28	a
37	0.917949	29	j
24	0.918934	30	a[j + 1]
13	0.921169	31	a
46	0.917115	32	j + 1
34	0.918134	33	j

25	0.918873	34	1
31	0.918354	35	{ int k=a[j]; a[j]=a[j + 1]; a[j + 1]=k;}
5	0.939187	36	int k=a[j];
50	0.916511	37	k=a[j]
61	0.913347	38	k
39	0.917689	39	a[j]
17	0.920663	40	a
32	0.918266	41	j
19	0.919975	42	a[j]=a[j + 1];
40	0.917664	43	a[j]=a[j + 1]
16	0.92072	44	a[j]
11	0.921573	45	a
23	0.919057	46	j
21	0.919381	47	a[j + 1]
10	0.922172	48	a
43	0.91738	49	j + 1
48	0.916938	50	j
47	0.91711	51	1
33	0.918226	52	a[j + 1]=k;
55	0.915924	53	a[j + 1]=k
28	0.91863	54	a[j + 1]
20	0.91945	55	a
53	0.916264	56	j + 1
45	0.917116	57	j
38	0.917846	58	1
35	0.918028	59	k
30	0.918483	60	return a;
29	0.91853	61	a

Table 4.3: Mutation-derived Node Rankings for Bubble Sort

As bias mechanisms from the related work have been applied statically, we compare the bias produced by our mutation-based technique for software improvement in this context. Though expensive, we generate bias by creating program variants of a seed program and updating bias until bias values "stabilise" or converge on particular values. Bias is then applied to the seed before running GP.

When the bias from Table 4.3 is applied to Bubble Sort the resulting rate of improve-

ment can be seen in Figure 4.8. 250 generations of 5 individuals were created to produce this node ranking. When an improvement was found the inner loop improvement was found 8% of the time while the outer loop improvement was found 92% of the time. We would interpret this result as mutation-derived bias being able to improve the program quickly while finding the more difficult of the two improvements. Whether this approach is particularly beneficial for finding more difficult-to-find improvements would require further investigation.

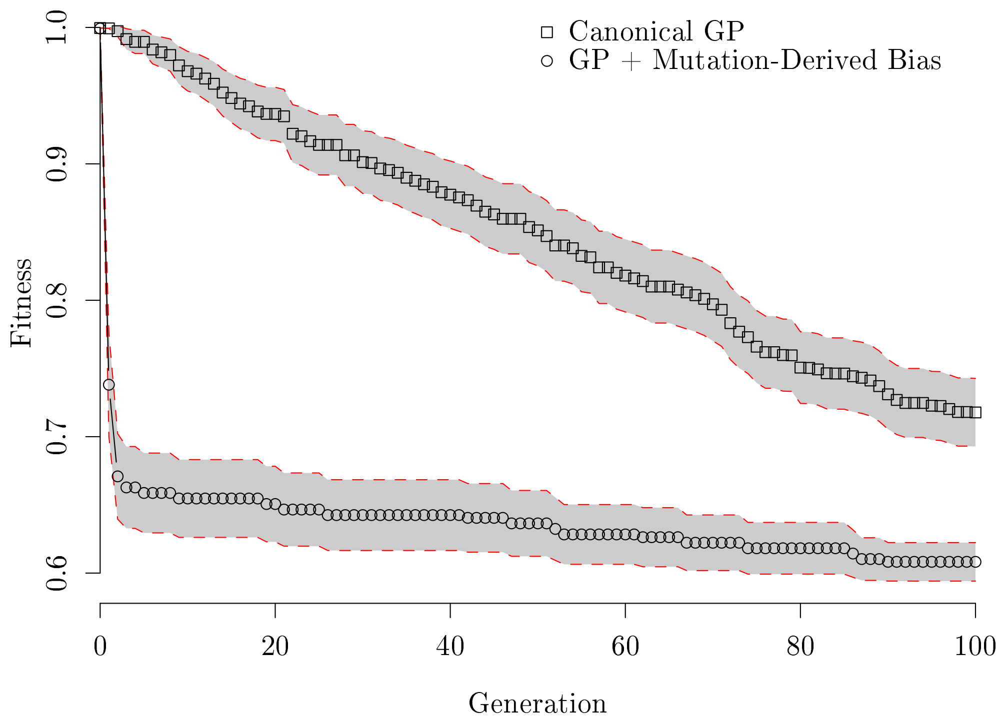


Fig. 4.8: GP with Mutation-Derived Bias on Deceptive Bubble Sort

4.4.3.1 Deceptive Problem

We apply the same approach of repeatedly mutating the deceptive Bubble Sort problem as seed to produce a bias overlay to see how it compares to the use of a profiler. Although mutation-derived bias does not increase the chances of finding an improvement over canonical GP, the use of mutation to derive bias is not as deceived as profiler-derived

bias on this problem as shown in Figure 4.9.

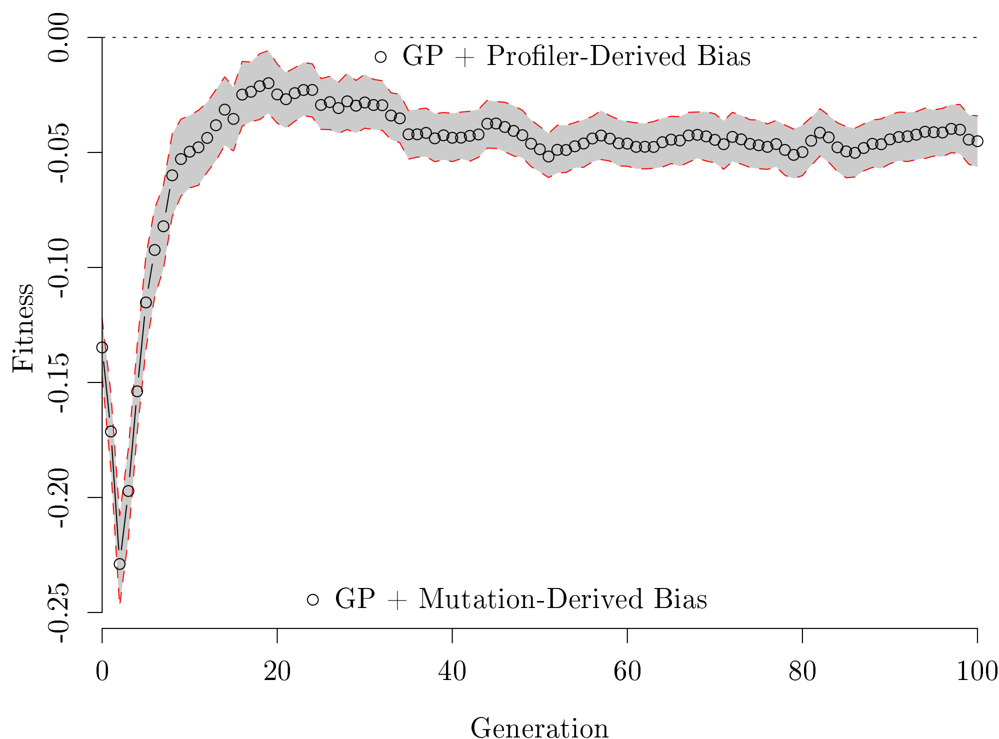


Fig. 4.9: Profiler and Mutation-Derived Bias Difference on Deceptive Bubble Sort

This provides an example where the performance analysis approach of profiling shows where a problem manifests as opposed to where a solution is likely to exist.

4.5 Dynamic Bias

The last section showed us that a profiler technique is good for finding out where a problem manifests but can be deceived where the cause of a bottleneck is not co-located with code that takes the most amount of execution time. The location where a performance bottleneck problem manifests may not always be the same location as an improvement which can alleviate the bottleneck. In short, the location of the problem may not be where the solution should be applied. We also showed that mutation-derived bias in a static context is less influenced by deceptive problems.

We are interested in using mutation-derived bias dynamically during GP as deriving bias in this way is an expensive analysis technique. As GP performs mutation as part of the algorithm the cost can be offset when used as part of GP to localise mutation and search for solutions through the same process.

Deriving bias dynamically during GP means that bias is attributed to a wider range of program variants. During GP the program being modified can change and the number of modifications to any one program may be less than the number of modifications made by repeated modification of a single seed program. As modification is spread out among a population of programs and applied less frequently to each individual program there is only partial information for each location within a program. In this section we show how mutation-derived bias can improve GP when used as an integral part of the algorithm. It is possible to influence GP through bias allocation to find improvements more quickly, even though bias is allocated over a range of program variants.

We show dynamic mutation-derived bias across a number of sort and Huffman codebook algorithms and find that although we can derive beneficial bias on some problems, the approach does not generalise. In Figure 4.10 dynamic bias in GP is compared to canonical GP, showing dynamic bias marginally outperforms standard GP until the 70th generation.

On the Huffman codebook problem our dynamic bias does not focus GP toward a known improvement. This is curious as the “Huffbook” problem contains the Bubble Sort implementation. This indicates that either the fitness function or the implementation of Huffbook is obscuring the patterns required to find the improvement. The same improvement was found in Bubble Sort previously so the fitness function and the program have changed in such a way to prevent the improvement being measured. We tested all of the rule-sets but none highlighted the improvement. Thus it would appear that our approach is limited due to some programs not exposing the location of improvements. If the nodes in a program cannot be measured as different then the bias mechanism may focus on other nodes reducing the chances of finding an improvement. This is dependent on the program being modified and the structure of the fitness function being used.

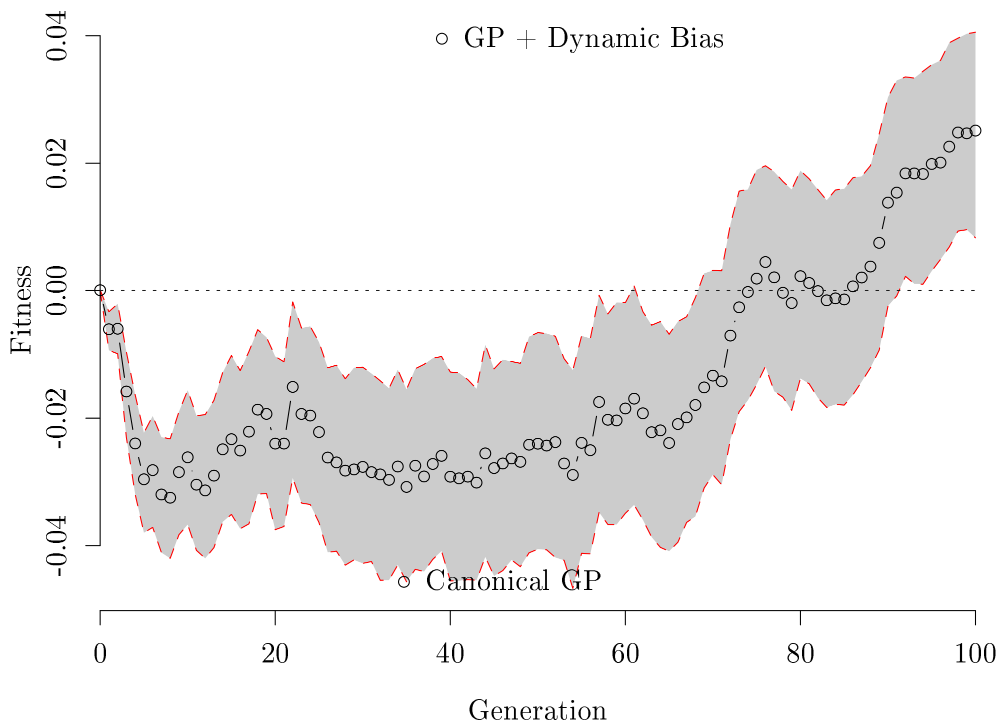


Fig. 4.10: GP with Dynamic Bias Allocation on Bubble Sort

Further work would be required to fully understand why this problem is resistant to any bias allocation rules. In short, if there are no distinguishable patterns produced when nodes are modified bias allocation will not improve GP.

4.5.1 Static VS Dynamic

We compare mutation-derived bias in both static and dynamic contexts. This compares the improvement found by making bias allocation part of the GP algorithm. Mutation-derived bias applied statically is beneficial and we show here that the same benefit can be made to GP through the use of mutation-based bias allocation as part of the GP algorithm.

When we allocate bias statically, it is with respect to one seed program. The vast bulk of program variants generated from this seed, will have lower fitness values. Throughout this process the seed program does not change and the bias allocated to it can be more

specific to that program. When bias is allocated "dynamically" as part of GP the fitness differences between parent and child program will include increases and decreases of fitness.

Deriving bias during GP is outperformed by static bias as compared in Figure 4.11. Dynamic bias on Bubble Sort finds the inner loop improvement 97% of the time. This is a curious result as the opposite is true when mutation-derived bias is used in a static context, the outer loop improvement is found most often (96%). The discrepancy in the operation of mutation-derived bias when applied statically and dynamically may be due to the range of program variants which bias is applied to during GP. When bias is allocated dynamically during GP the number of opportunities to update bias for each individual is less.

We pick generation 100 to compare as this gives enough generations for the GP approaches to separated with regard to their effect on the GP process.

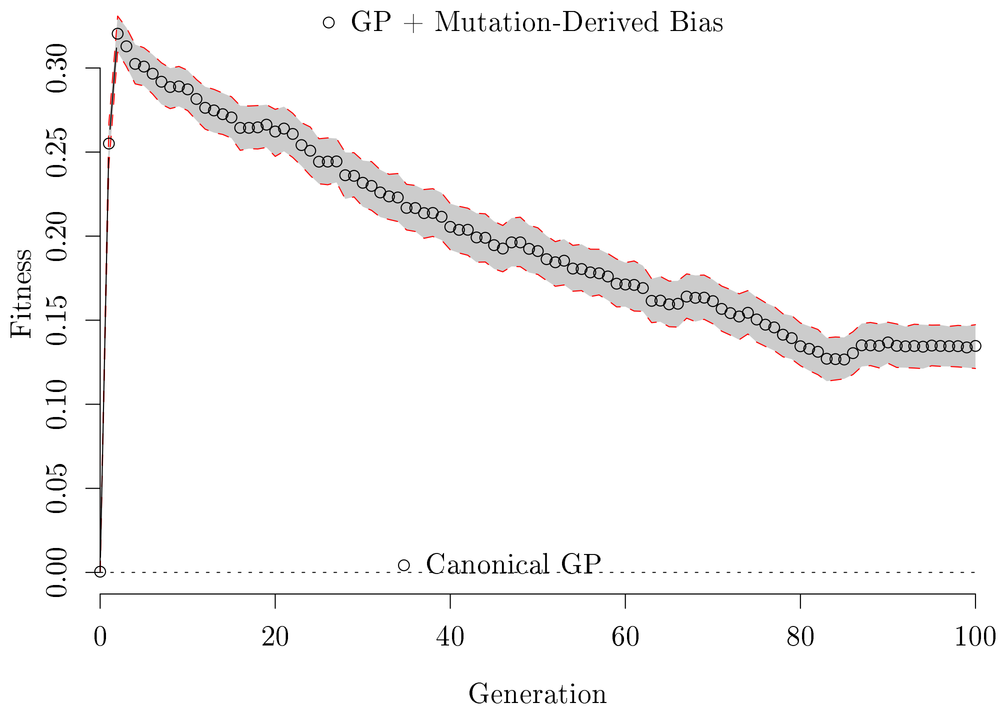


Fig. 4.11: Dynamic and Static Derived Bias Difference on Bubble Sort

4.5.2 Deceptive Problem

In Figure 4.12 the improvement pace made by dynamic rule-based bias allocation is shown over canonical GP on the deceptive loops problem. Although bias is beneficial in the early generations canonical GP catches up. The use of dynamic mutation-derived bias is better than mutation-derived bias applied statically on this problem. Dynamic bias (as well as mutation-derived bias applied statically) is not as deceived as the use of a profiler statically on this particular problem.

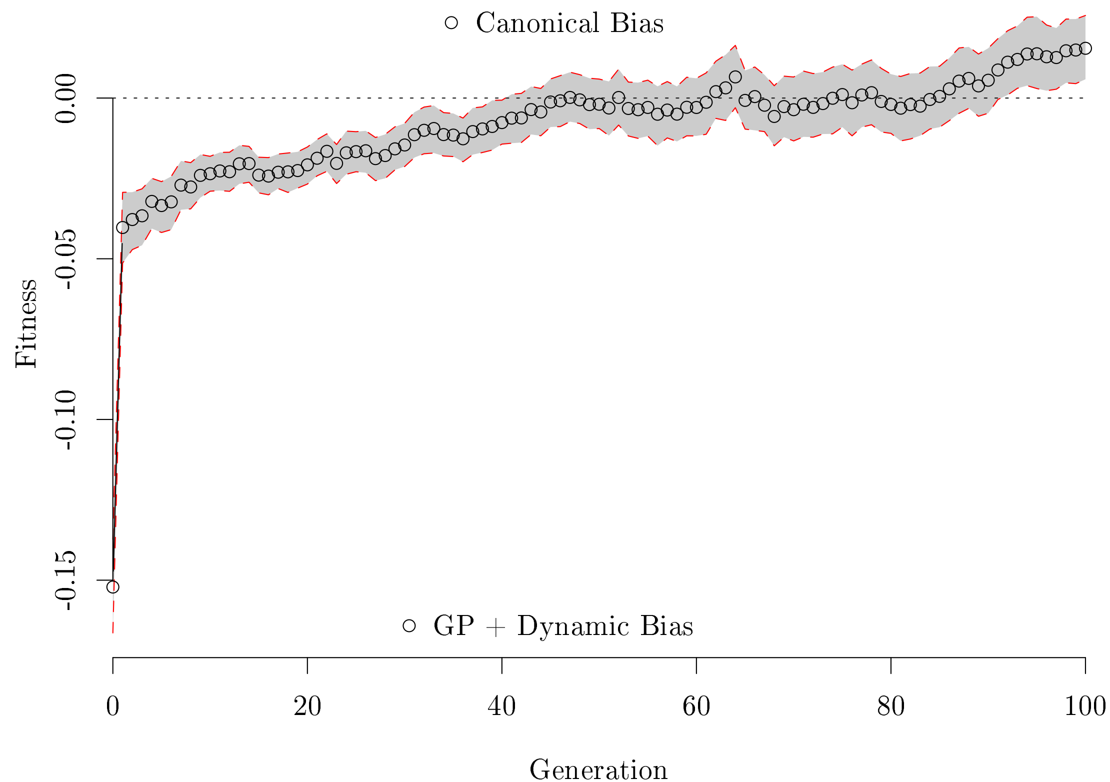


Fig. 4.12: GP with Dynamic Bias Allocation on Deceptive Bubble Sort

Dynamic bias allocation appears to be the best of the three bias mechanisms for this type of problem. The deceptive Bubble Sort requires a change in two separate places to find the known improvement. The changes can be made independent of each other so a static bias which accurately highlights both of these locations should be more effective on this problem.

4.5.3 Generality of Dynamic Bias

In this section we run GP with dynamic bias allocation across all our test problems to inspect generality. We compare dynamic bias with canonical GP in these tests to see if dynamic bias can find performance improvements even over the range of programs created during GP. Where a single seed program is repeatedly mutated to derive a static bias many variant programs are mutated to derive bias when mutation-derived bias is used during GP. In this section, we also look for trends across the problem set which may influence the operation of dynamic bias.

Figure 4.13 shows the results of dynamic bias against canonical GP on all problems. We show all problems on a single graph to observe on which of our problems bias improves the GP process. GP with dynamic bias increases the chances of finding an improvement over canonical GP in 7 out of the 12 problems up until about generation 90. After generation 90 Canonical GP is better on 7 out of the 12 generations. As dynamic bias is better during some generations, but not others, it is necessary to show results across all generations on all problems. Because of this variance it is difficult to pick any one generation as being representative for comparison. Of course a radical change in the improvements found could happen after generation 100, but this is unlikely given that GP tends to converge after a number of generations. This view is also supported by scatter graphs of all fitness values in a generation, which in our experiments, reduces over time as exemplified in Figure 4.1. We could also analyse the time taken for GP to find a known improvement, but the issue is that some runs may not converge on the known improved version of a program regardless of how much time is allowed.

These results are encouraging as they show that on some problems dynamic bias is capable of highlighting the location of improvements. We continue our analysis to see if the positive effects of dynamic bias are correlated with problem characteristics such as program size or improvement size.

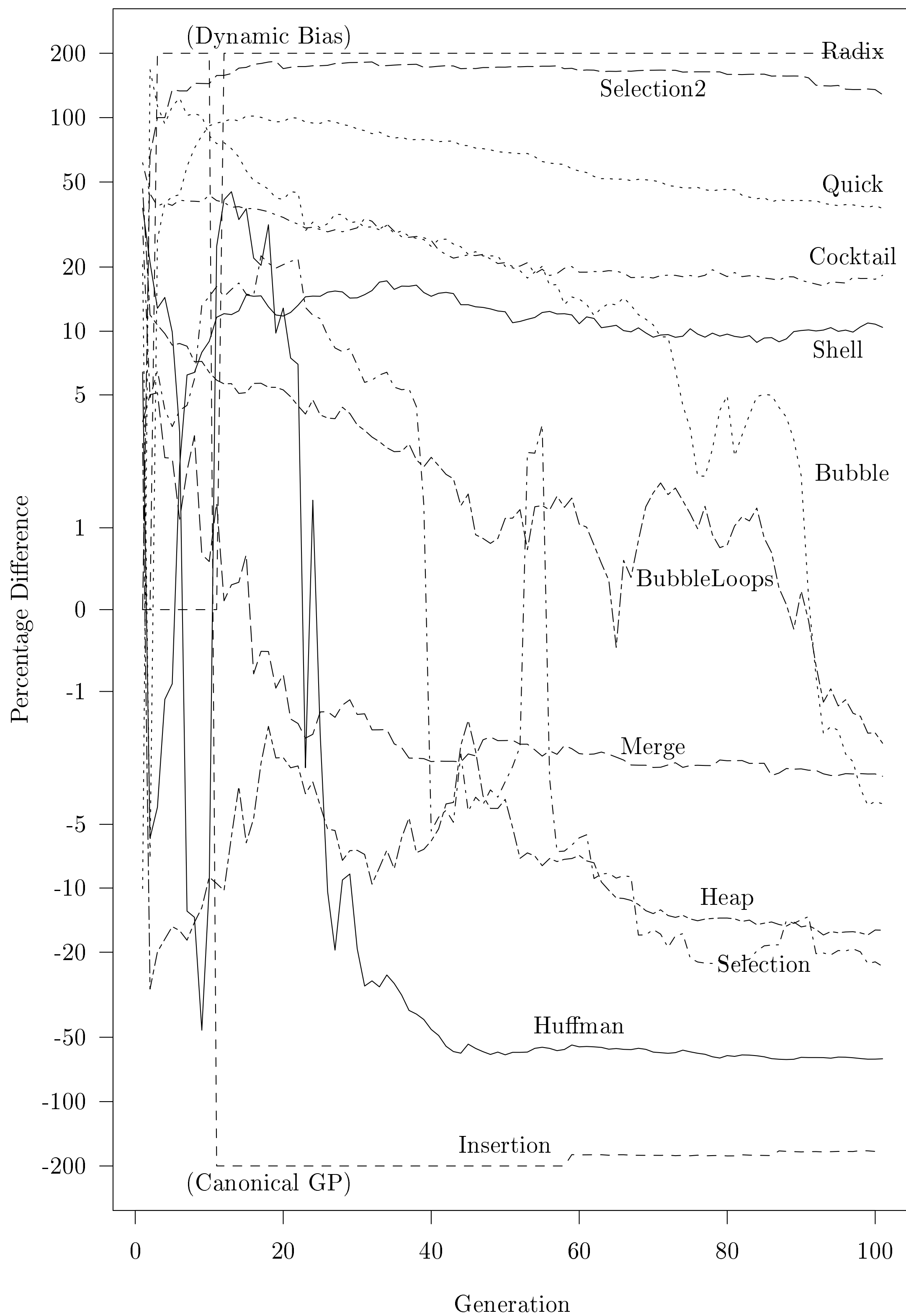


Fig. 4.13: Dynamic Mutation-Derived Bias Across All Generations

4.5.4 Trend Analysis

In Table 4.4 a number of program characteristics are listed. We measure these characteristics to see if, for example, the benefit of dynamic bias is correlated with program size. We want to see if general program characteristics can predict how well dynamic bias can be expected to perform on a problem. Program size is listed in terms of number of AST nodes in a problem. "Tree-edits" is a count of the minimum number of tree edits required to find an improved version of the problem. Although a number of edits are possible the minimum required is sometimes less where multiple equivalent improvements are possible.

Dividing the number of edits required by the program size give "Edits / Nodes". This is a measure of the portion of the problem which has to change for an improved version to be found. We use this as a rough measure of the breadth of search which must be undertaken. The lower the number of nodes which must be "found" and the larger the program is, the harder these improvements can be expected to be found.

The percentage of programs discarded during the GP process is also shown. This gives an idea of how difficult the problems are to modify using the GP operators. Certain problems are more difficult to modify, and we can say that these problems are more complex or have more interdependencies, as more of the programs have to be discarded.

We take the comparison of dynamic bias and canonical bias at generation 100 across all problems and order the results per the measures shown in Table 4.4.

Figure 4.14 shows the difference between Canonical GP and Dynamic Bias in the last generation of all problems. This diagram provides an overview for how many problems dynamic bias was able to improve the chances of finding improved programs. The ordering of problems is in increasing size in terms of program size measured in *AST nodes*. We show this graph here to inspect any trends that may be visible as the program size increases. From the diagram there is no obvious trend when considering program size. Dynamic bias slows down the GP process on a number of problems. Our choice of generation 100 is somewhat arbitrary and does not provide a definitive answer to any

Problem Name	AST Nodes	Tree-Edits	Edits / Nodes	% Discarded
Insertion Sort	60	3	.05	73.3
Bubble Sort	62	2	.032	71.4
Deceptive Bubble Sort	72	4	.055	71.8
Selection Sort 2	72	1	.0138	70.9
Selection Sort	73	1	.0137	71.2
Shell Sort	85	3	.035	71.4
Radix Sort	100	3	.03	80.5
Quick Sort	116	2	.0172	72.7
Cocktail Sort	126	1	.0079	73.7
Merge Sort	216	1	.0079	73.2
Heap Sort	246	2	.0081	71.1
Huffman Codebook	411	2	.0048	83.8

Table 4.4: Size of Program and Minimum Change Required for Improvement

questions about trend. As the operation of dynamic bias on these problems varies over time it is difficult to pick the correct generation to compare values over. From Quicksort onwards the effect of dynamic bias appears to decrease. Up to Quicksort, the range of nodes is from 60 to 100. From Quicksort onwards the range is from 116 to 411. Applying dynamic bias to larger programs is of particular interest for future work. The ordering of problems in Figure 4.14 is the only one that shows any hint of a correlation.

In Figure 4.15 the ordering is based on the minimum number of *tree edits* required to produce an improved program. We refer to this as the "Improvement Size". If the size of the improvement determines how difficult an improvement is to find then potentially the use of bias is affected by the number of nodes which must change to produce an improved variant. The more nodes which must change the longer the "tree-edit distance" or lineage of programs between the seed and the improved program. As this lineage grows longer a longer sequence of edits must be performed. As more edits are required the higher the

chance that some of these edits have poor fitness. The longer the lineage of edits and the poorer the fitness of program variants along the lineage, the less likely it is that GP will be able to traverse this lineage. There does not appear to be any noticeable trend under this ordering.

In Figure 4.16 results are ordered by the *improvement size over the program size*. This measure of problem difficulty seeks to capture the size of the changes we are searching for over all possible changes. Figuratively, we are measuring the size of the needle in comparison to the haystack.

In Figure 4.17 results are ordered by the number of programs discarded. The concept behind this ordering is that more complex programs which are more tightly knit or have a larger number of interdependencies between code elements will produce more degraded programs when modified randomly. This ordering does not show any correlation either.

A larger problem set may show correlations more clearly but is left to future work. There may also be further explanatory variables which have yet to be identified. Such analysis is left to future work and further discussed in subsection 5.4.3.

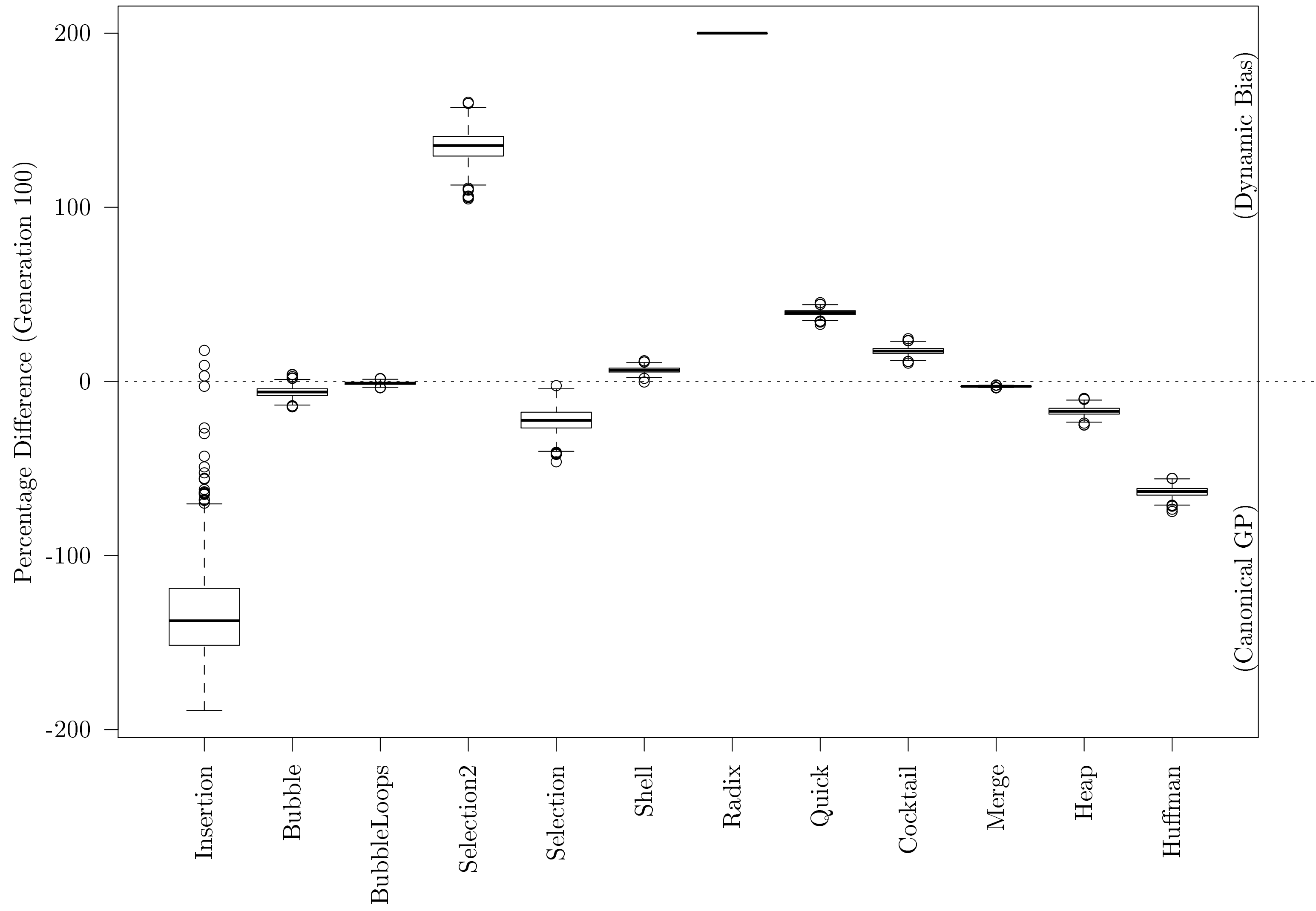


Fig. 4.14: Dynamic Mutation-Derived Bias Ordered by Problem Size

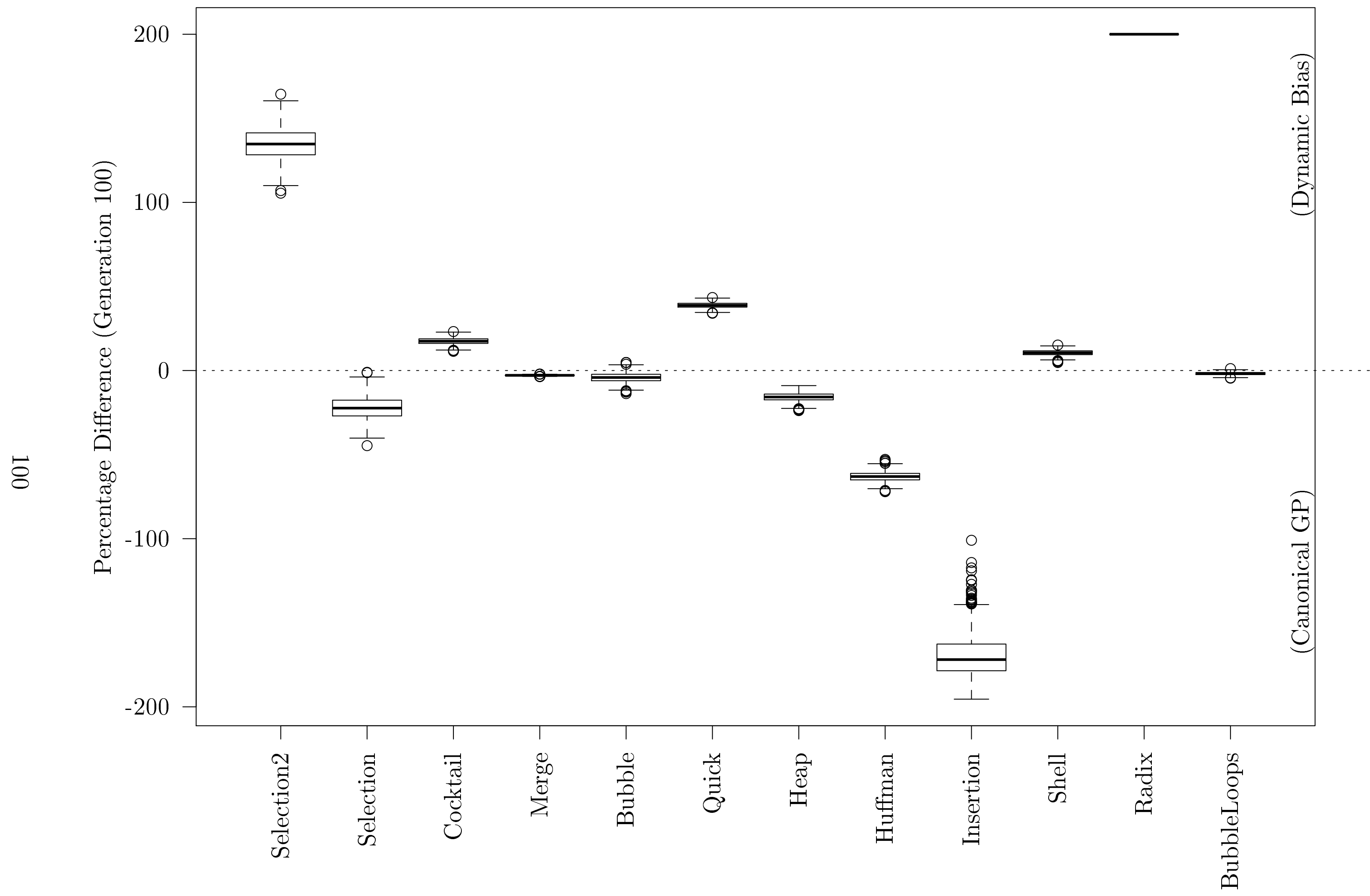


Fig. 4.15: Dynamic Mutation-Derived Bias Ordered by Improvement Size

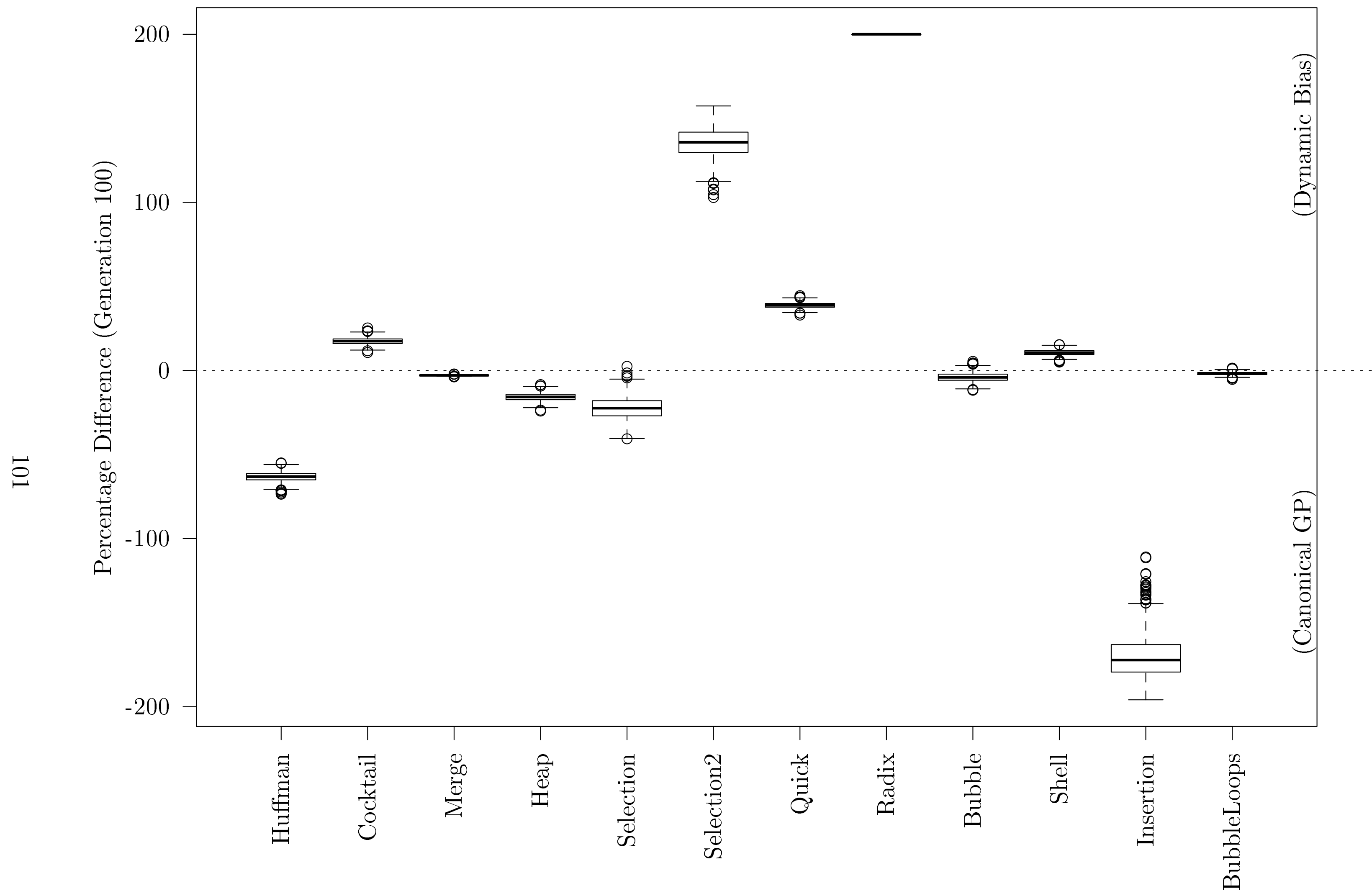


Fig. 4.16: Dynamic Mutation-Derived Bias Ordered by Improvement Size over Problem Size

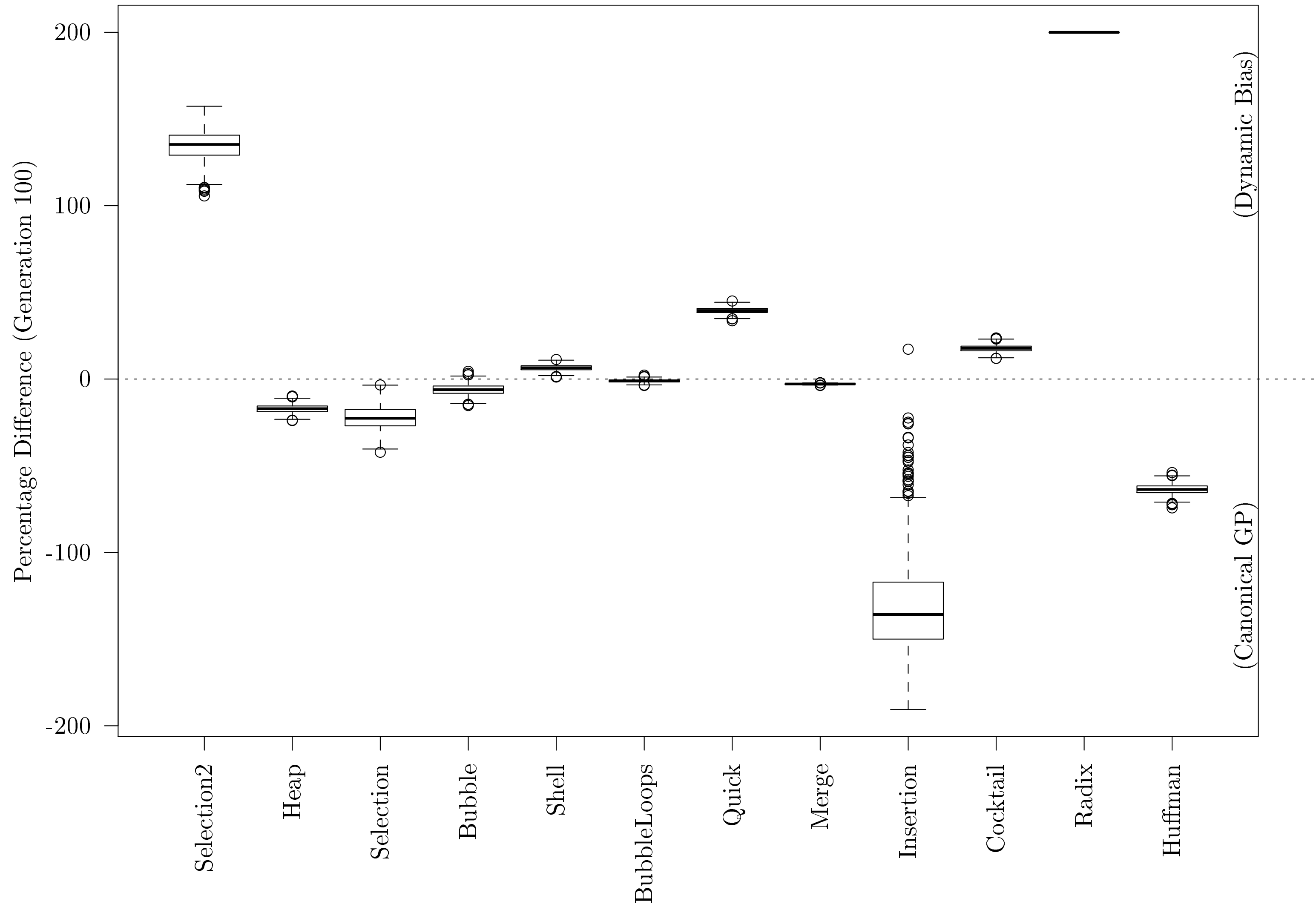


Fig. 4.17: Dynamic Mutation-Derived Bias Ordered by Programs Discarded

4.5.5 Gaussian Bias

Bias can also be allowed to emerge by randomly modifying bias values during the GP algorithm [6]. Bias is shaped through the selection of programs as per their fitness. Programs with higher fitness are likely to have whatever bias they contain propagated. Bias is subject to shaping under the standard evolutionary pressure is implicit as opposed to being directly modified per any deterministic ruleset. In this analysis we inspect whether explicit bias allocation, due to fixed update rules, is necessary to find program improvements [6]. Our findings are that explicit bias allocation outperforms the use of random bias allocation on half the problems tested. From these results we can say that both allocation methods target different improvement types.

Figure 4.18 shows the comparison of canonical GP and GP with random bias allocation drawn from a Gaussian distribution as described in [6]. Random bias increases the chances of finding an improvement on 7 out of 12 of the test problems.

In Figure 4.19 we compare random bias with dynamic mutation-derived bias. We find that dynamic bias is an improvement over random bias on half of the problems we inspect.

A possible explanation for these results is that improvement locations are the same as locations which produce high fitness individuals when modified. This would indicate that variant programs along the lineage between seed and improved variant programs have relatively high fitness. If this were the case then random bias should be better on problems where the fitness along seed to improved variant lineage is high. Dynamic bias promotes changes which allow the program to compile and reduces the functionality. As such, it should produce programs which are of average lower fitness than random bias or canonical GP. The average fitness should be lower when dynamic bias is used. It focuses on the modifications which degrade fitness. As dynamic bias is subject to the same evolutionary pressure as random bias we can speculate that dynamic bias provides large bias changes than evolutionary pressure.

It is claimed in the literature that there is no need for explicit credit assignment [6]. Our results show that in certain cases explicit bias can be advantageous in the context

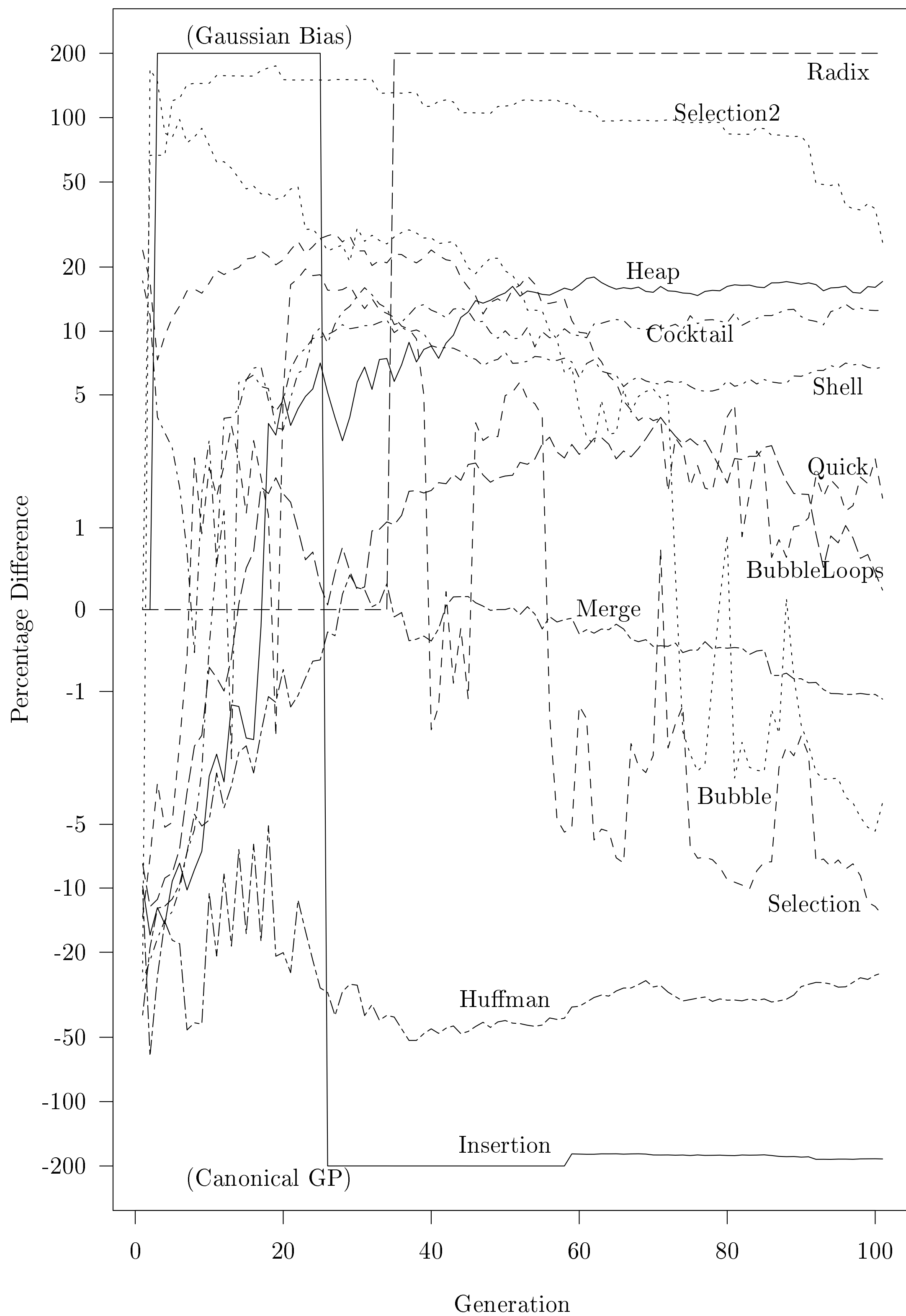


Fig. 4.18: Canonical GP against Gaussian Bias

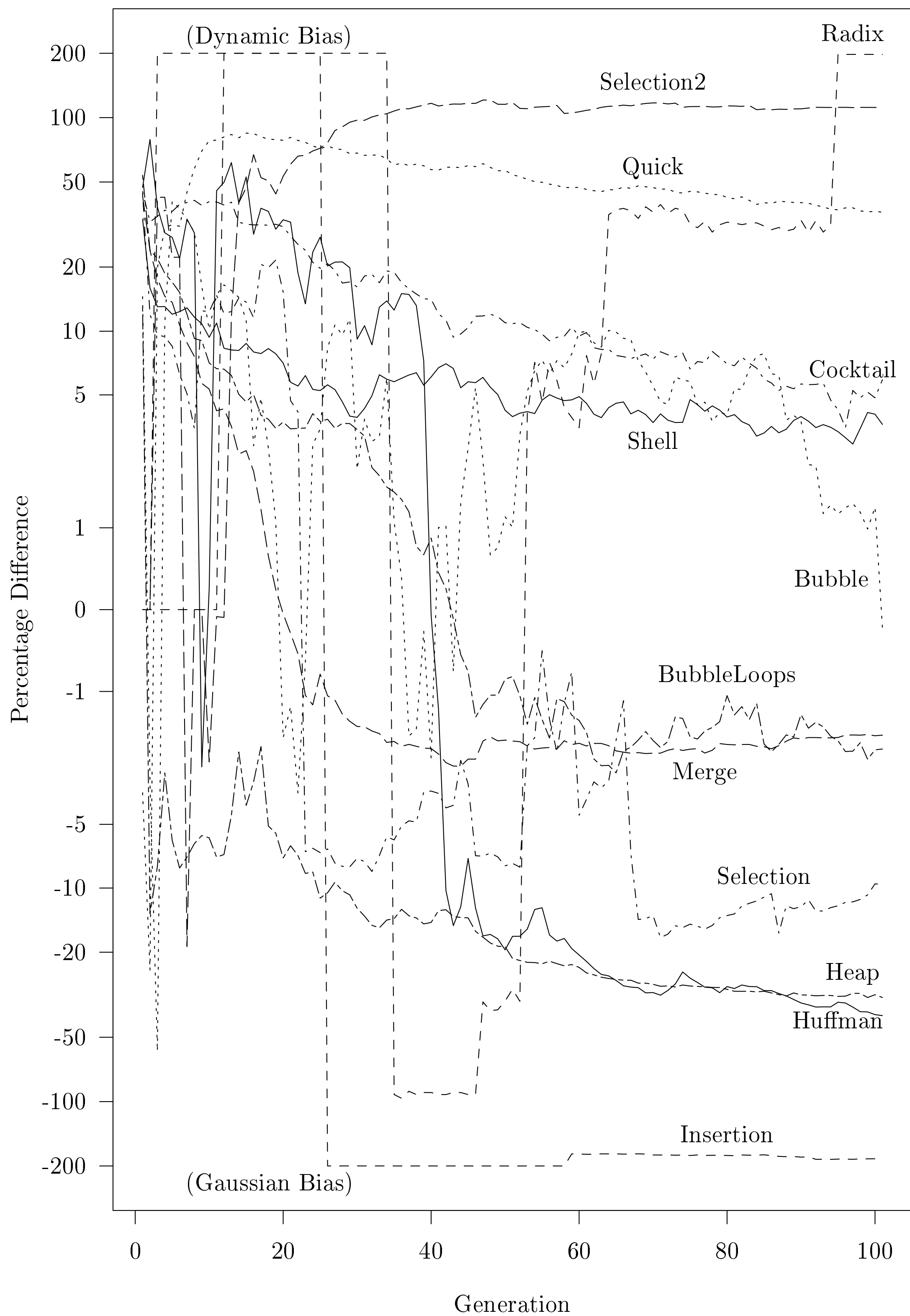


Fig. 4.19: Gaussian Bias against Dynamic Mutation-Based Bias

of improving existing software when compared with randomly perturbed bias.

4.6 Mutation Spectrum Analysis

Experiments so far have shown the effect of bias on the chance that GP will find improvements in our problem set. As these results are somewhat mixed, we more closely inspect the range of functionality and performance measures possible when mutating at different locations in a program.

We address the assumption that nodes can be told apart by analysing random modification. We find that exhaustive modification at each node produces program variants which evaluate with unique sequence of functionality and performance measures. To a high degree this is true for locations of program improvements. We show that for most improvements in most problems, whether a program compiles and how it affects the performance metric are both factors which highlight potential improvements in the programs.

Our previous ruleset for mutation-derive bias allocates bias where the functionality produced through modification is decreased. To evaluate functionality the program must compile. As 70% of program modifications are destructive, when a program is modified and still compiles there is a high likelihood that the modification reduced the functionality. This is the basis of the bias allocation rule tested in previous sections of this chapter. Further to our bias rule the findings of this section suggest that measuring performance changes is also an important factor in highlighting program improvements.

4.6.1 Exhaustive Modification

To observe the range of program variant fitness values that can be generated from a program we take every modifiable node in a program and make all possible modifications to that node. The range of possible modifications is determined by the other nodes in the program which are taken as the set of replacement nodes. We attempt to replace every node in the program with every other node in the program. A variant program is

produced for each node replacement performed. The variant programs are then evaluated with the relevant fitness function.

While this appears a formidably large search problem the set of evaluated programs is reduced considerably due to node typing and the semantics of the programming language that we use. Many nodes are only replaceable by a small number of other nodes from the original program. Replacing some nodes may lead to programs which do not compile. This excludes a very large number of programs from being evaluated for fitness. Thus the search space produce by a single edit can be considered "sparse" having many inviable programs. While we utilise typing in our experiments we do not attempt to enforce semantic constraints. Even though a large number of generated programs do not compile the search process is not adversely affected as non-compiling programs do not require costly execution during evaluation. Compilation and evaluation time are the most costly events during our GP search.

Exhaustive modification is not a fully accurate deterministic view of what happens within mutation-derived bias. When mutation-derived bias is used modifications are made at random. The same modifications can be made multiple times meaning that the bias is updated cumulatively. In this experiment we control exactly how many times each node is modified. Over time mutation-derived bias should equate roughly to the exact values. When dynamic bias is used during GP bias fluctuation is more uncertain as multiple programs are considered and bias is inherited between programs. By making all possible modifications to all locations in a program exactly once we hope to gain a clearer and less "noisy" view of the range of values which can be observed when modifying a program.

4.6.2 Fitness Analysis

We consider here the most prominent improved program variants from our GP search results. By comparing these improved variants with seed programs we find the nodes which need to be modified to produce the improved variants. We rank all program nodes by averaging the set of fitness values produced through modification at each node.

The median of rank values is taken for the set of "improvement nodes". The purpose of using median ranking is to see if improvement nodes can be separated from a reasonable proportion of the total number of nodes in a program. If they can be separated by the fitness measures produced, either positively or negatively, then we can ignore the modification of irrelevant nodes during the GP process by updating bias as per these fitness measures. If we can effectively rule out a set of program nodes as irrelevant to improvement we can reduce the search space.

4.6.3 Results

Table 4.5 provides a summary of the evaluation measures for locations of improvements under modification in our test problems. The number of variants which are syntactically correct and compile along with measures of functionality error and performance values are shown.

In 8 out of 12 of the problems the median compiled % rank for the set of improvement locations were ranked in the top 50% of nodes which compile. Based on this it appears that on a majority of the problems we could half the number of nodes which should be modified during GP search. This should increase GP's chances of finding improvements on these problems.

One design concept that comes from this analysis is the use of not just a single bias value but maintaining a running summary of the number of times program variants compile, the functionality change average and the performance change average. When selection is performed the highest value from these 3 metrics could be taken for comparison during tournament selection. Future work on bias update rule design is in subsection 5.4.2.

Problem	# Opt Nodes	Median Compiled % Rank	Median Func % Rank	Median Perf % Rank	% Unique
Insertion	3	61.6	78.3	36.6	84.1
Bubble	5	74.2	61.3	82.3	73.5
BubbleLoops	8	51.4	76.0	8.9	75.5
Selection2	1	58.3	63.3	59.7	74.6
Selection	1	67.1	32.9	69.8	72.2
Shell	3	35.3	50.6	47.1	80.3
Radix	3	32	27	48	77.4
Quick	2	55.2	44.4	45.3	97.4
Cocktail	1	61.1	31	50.1	84.4
Merge	1	42.1	85.6	28.7	89.1
Heap	2	43.5	51.8	33.9	64.8
HuffBook	5	85.6	77.5	34.8	78.9

Table 4.5: Average Ranking for Improvement Nodes when Modified

Opt Nodes refers to the number of nodes in the program which need to be modified to produce an improved variant of the original program. For example, 5 node modifications are needed to produce an improve Bubble Sort.

Median Compiled % Rank is the median rank for the number of times improvement nodes produced compilable programs when modified. This is calculated by taking a count of successful compiles for each of the improvement nodes. Then the percentage rank for each of those nodes is created. Finally we then get the median of these percentage rank values. The trend on this measure is that nodes which compile more often are generally more likely involved in a program improvement. For example, on Bubble Sort at least half of the improvements nodes, have compile counts higher than 74.2% of the other nodes.

Median Fun % Rank is the median rank of averaged functionality values for improvement nodes. A lower Functionality % Rank means that more of the other nodes had an average functionality measure higher than the current one. When a program is modified it generally tends to have a higher functionality error measure than its parent (as program functionality generally degrades under modification). It is likely that modification will degrade the which will yield a a higher functionality error.

Median Perf % Rank is the median rank of improvement nodes in terms of performance measure. We rank the performance values for each location. A higher Performance % Rank means that more of the other nodes had an average performance measure higher than the current one. This is deliberately the opposite of functionality % Rank as the trend is toward lower performance under modification. Programs which compile after modification tend to have degraded functionality and although not functionally correct, will have reduce execution cost. A prominent example of this is an empty program, which has greatly reduced execution cost, but a large functionality error count. The finding from Median Performance % Rank is that improvement nodes generally produce variants with a smaller amount of execution cost reduction. For these nodes execution count is reduced but reduces less than other nodes in the program.

% Unique is the percent of nodes which can be uniquely identified with a combination

of compilation, infinite loops, performance and functionality measures.

4.7 Threats to validity

External Threats include issues which affect the ability to generalise our results. The problem set contains a relatively small number of programs which can be evaluated with only two different functionality measures. How the results found generalise beyond these problem types would require a wider consideration of programs and their respective functionality measures.

We use a deceptive problem to differentiate between profiler-derived and mutation-derived bias, yet we do not know how prevalent such deceptive problems are. It may transpire that our deceptive problem is somewhat artificial and unlikely to occur in practice. Our deceptive problem introduces inefficiencies for the sake of our experiments. With no other purpose for such a blatant inefficiency it seems unlikely that any developer would create such a program. In short, it is unlikely for such an inefficiency to be present without it having some desirable purpose.

In addition, the size of any one of problems is relatively small. The ability to uniquely identify nodes may dissipate as more nodes are available to modify though we do not see this trend in results so far. In general we can say that the problem set is not as broad as we would like though is nonetheless varied enough to show the limits of dynamic bias allocation.

Internal Threats include considerations for our fitness functions. Measures of functionality and performance have been cross-checked numerous times during evaluation and have repeatedly shown to be deterministic. Functionality measures have been tested on known working programs, especially in the case of sort algorithms, where multiple sort implementations have been used as oracles including the sort implementation from the Java core library.

When measuring functionality, the test data used is potentially an issue. Although a number of different test cases are used the data in each of these test cases is fixed

throughout experiments. There is a chances that programs can overfit to the fixed data, although enough test cases are used to make the chances of this occurring slight. Even though this may be possible we have not encountered any improved program which overfit to the test cases.

All of our samples have been taken with at least 100 GP runs, and so we are content that our experiments have been run enough to separate the effects of bias with statistical confidence. The variance can be seen in graphs such as Figure 4.15

4.8 Summary

In this chapter we have shown that mutation-derived bias can improve the chances of finding an improved program variant when used with GP. It outperforms canonical GP and profiler-derived GP on a deceptive problem particularly where an improvement is not at the same location as code which can be considered a “bottleneck”.

Further to this, we show that deriving bias during GP can increase the chances of finding improvements in certain problems. On these problems dynamic bias outperforms canonical GP and GP with random bias. The results show that each of these approaches increase the chances of finding improvements on certain problems.

A subsequent analysis of the spectrum produced under exhaustive program mutation produces values which could be used to isolate the locations of performance improvements from other locations in code.

The core take-away from this work is that while profiling programs can guide random change, mutation derived-bias is capable of highlighting different improvements in programs. We find that explicit credit assignment can be beneficial in improving the GP process on some problems. We also find that using random bias shaped by evolutionary pressure alone is also beneficial on the remaining problems.

Chapter 5

Conclusion

In this thesis we have further explored the use of GP for automated support for software engineering. We scope our work to the improvement of existing source code in terms of execution cost measured in Java bytecode executed.

In related work GP is shown to have scaling issues when applied to large problems. A common solution approach is to prune the search space by focusing change where it is most likely to improve the software and ignore locations that are irrelevant. Finding locations likely to yield an improved version of a program when modified is the challenge we are faced with. Most GP literature describes growing programs outright as opposed to improving existing programs. Many of the bias mechanisms in GP are geared toward outright improvement. We compare our bias allocation mechanism with random bias modification.

In particular we inspect whether fitness changes which result from program mutation can highlight performance improvements. Our hypothesis is that the measurable effects of change on a programs performance and functionality can be used to highlight potential locations of improvement. We provide evidence that mutation analysis can be used as an integral part of GP to improve its operation for certain problems.

We find where in a program is relevant by observing changes in the fitness as a program is modified. This is based on the assumption that modifications at different locations in a program produce different fitness values. This assumes that even when

a node is replaced by nodes which break the program, affecting the fitness drastically, the nodes where improvement is possible show a different pattern when modified than do other locations irrelevant to improving a program. If the locations can be told apart from these fitness values GP can focus change on these locations and ultimately increase the probability of finding an improvement. Our work attempts to validate the existence of these patterns and that these patterns can be exploited by GP for specific problems.

To validate our claims we applied GP to a range of sort algorithms as well as a prefix code-book algorithm. We compare canonical GP with self-focusing GP across problems. We also perform repeated mutations of the sort and prefix code-book programs to observe how bias influences the ranking of nodes. We can localise improvement opportunities by modifying the initial program repeatedly. We inspect a range of bias allocation rules to understand what parts of our fitness function are the best at finding improvement opportunities.

We compare our approach to the use of a profiler. A profiler is very effective if the location where a performance bottleneck manifests can be alleviated by modifying the same code. We show that it is possible for a profiler to focus on the symptoms of poor performance but that the changes required to alleviate the symptoms may be located elsewhere in the program. Due to the interdepencies in imperative code it is possible for a profiler to be "deceived" as to the location of an improvement. When used with GP to focus modification, change being focused on locations which will not reduce execution of bottleneck code.

5.1 Review of the Research Question

Our central research question is:

- Can differences in functionality and performance measures caused by arbitrary program modifications indicate the likelihood that further modification will create a program with reduced execution cost?

From this question we have a follow-on question:

- What change in the fitness function is particularly telling about a possible improvement?

When we consider these in terms of GP we get a GP-specific question:

- Can updating bias during GP increase the chances of finding a code improvement?

5.2 Overview of the Evidence

Nodes can be told apart by their effect on fitness. The production of different fitness patterns has been demonstrated by repeatedly modifying the original sort programs and observing how the fitness changes. As more modifications are performed bias is updated in response to fitness changes. The bias associated with each node can be used to rank the nodes in order of likelihood of being modified again. Taking an average of the node ranking throughout this process shows us that some nodes get a higher rank than others and we can tell the nodes apart as seen in Table 4.5.

Highlighting certain nodes can improve GP. We can see that a hand-made optimal bias can improve GP's ability to find an improvement in Bubble Sort as seen in subsection 4.4.1. We then observe how the bias produced by our rule-sets also increases the chances of finding an improvement with GP. We then show that a rule-set can be used as part of GP, first on Bubble Sort and then on subsequent sort variants. The rules which we developed for Bubble Sort also worked for improving some other sort programs. This tells us that rule-sets are not generally applicable across all sort problems, but shows that there is at least some common characteristic shared amongst some of the programs which our rule-set is able to capture.

Modification-derived bias can improve certain programs that a profiler can not. We use a profiler to count the frequency of execution of lines of code. The line count is used as bias for the program when GP is applied. The results show that using a profiler on this problem will allow GP to find an improved version of Bubble Sort quicker. We then repeat the same process using a modified version of Bubble Sort which introduces a redundant outer loop which causes an additional iteration through the whole Bubble

Sort algorithm. The idea here is to introduce some code which makes the program perform worse but is not itself executed at a high frequency and thus is not highlighted by a profiler. The execution count of the introduced code is very low in comparison to the core Bubble Sort algorithm meaning that GP is slowed down by focusing on the location of the problem instead of where the solution actually lies. This provides evidence that a profiler is good at focusing on where the problem manifests with the caveat that where the problem manifests may not always be the same as where a solution lies. The ramifications of this when using a profiler to bias GP change is that a profiler may draw attention away from relevant locations if the location of the problem and the solution do not overlap in terms of execution count per line of code. In this context, our bias allocation rules are less influenced by this "deceptive" problem finding the improvement quicker than the bias produced from a profiling technique as seen in Figure 4.9. The approaches of dynamic bias allocation and using profiler techniques do not appear to be at odds with each other and could be used in conjunction.

Explicit credit assignment can improve chances of finding improvements. Explicitly assigning credit or increasing bias can in some cases outperform randomly modifying bias after every program variant is generated. On half the problems tested evolutionary pressure alone did not produce the highest change of performance improvement. This shows that explicit credit assignment can be useful as observed in subsection 4.5.5.

Functionality and Performance measures could be used to highlight performance improvements. Each problem is exhaustively modified to observe the variance in performance and functionality measures for each modification point in a program. We find that the vast majority of nodes can be uniquely distinguished by unique combinations of compilation, performance and functionality measures. The measures for improvement locations in programs are observed for their difference to other locations. We find that if a range of these values were to be used we should be able to differentiate more clearly the location of performance improvements as can be seen in Table 4.5.

GP as a useful approach for improving programs All programs we applied GP to were improved in terms of bytecode executed as listed in Table 4.1. The test programs

were taken from a programming chrestomathy collection [135] and as such are exemplary implementations of common algorithms. This is however a weak claim given that we don't know how representative these programs are of software in general.

5.3 Review of Contributions

In summary, the contributions of this thesis are:

- Provide evidence that modifying a program produces fitness patterns which differentiate nodes. Nodes which are required to change to improve a program show a different pattern to other nodes. Using these patterns we can attribute fitness changes to locations in a program and subsequently direct code modification to improve a program quicker using GP
- Provide evidence for the utility of explicit credit assignment in Genetic Programming
- Show a profiler approach can be deceived and that mutation-derived bias is less deceived on a specific problem
- A GP system for evolving Java code.
  The system by which we evolve code has been demonstrated. It's purpose was to inspect bias on java code.
- Further demonstrate the evolution of programs using GP for performance improvement.

5.4 Future Work

A number of items have been identified throughout the course of this thesis which are listed here before moving discussion on towards future work in a broader scope.

5.4.1 Analysis

During the experimental work in this thesis a large amount of data was been produced. Work can be done to analyse some of this information further.

Immediate work which could be performed is to test whether Gaussian bias finds improvements where the lineage of individuals between seed and improved variant has high fitness as mentioned in section 4.1. Similarly, the improvements for which dynamic bias is beneficial to GP can be analysed to see if the variant programs have lower fitness on average. This would go some way to explaining why Gaussian and dynamic bias are better on some problems than they are on others. Why some improvements have lower or higher fitness lineages is unknown.

Due to the exploratory nature of this thesis, our contributions and claims could benefit from an extended analysis of the data produced during experimentation.

Our fitness values for program improvement normalised against an original seed program does not produce a smooth graph. This is due to the use of functionality as part of the fitness function where each discrete functionality "score" represents the state of a program. Where programs are different, but give the same answer, the difference in the programs cannot be measured using our functionality metric. We have looked at the use of Kaplan-Meir or "survival" curves [58] to model fitness improvement of the seed program disregarding programs worse than the seed. Survival curves capture then probability of being in one of two states but does not capture a multi-state system. A hidden Markov multi-state model with discrete time intervals [55] appears appropriate for further analysis of our results. Our analysis so far considers improvement as an absolute and does not consider the different possible programs which are equivalent in performance. Understanding what improvement types are more likely at an in-depth level may be possible by the application of a multi-state model and be interesting in itself in the area of statistics.

5.4.2 Bias Update Rules

As mentioned in section 4.1 further factors, such as performance change between parent and offspring program, may also be used to expose the location of performance improvements. Such findings encourage the further design of more effective bias update rules.

In subsection 4.6.3 findings suggest that maintaining a count of the number of times a node modification resulted in a compilable program, the functionality change and performance change could all be maintained per modifiable node. This would be represented by three summary values per node in a program and could be used during the selection of nodes during tournaments.

Additional experimentation could also be performed to dissect what parts of our bias update rules are useful. We employ bias decay, inheritance and updates in the parent and offspring programs. As GP is a complex and dynamic environment it is unknown specifically how much each of these contributes to the accurate generation of bias.

As general software metrics do not appear suitable for guiding GP [17] program spectra or other measures specific to the program domain may be useful metrics [45]. Our approach uses black-box measurements to evaluate the fitness of a program. By adding more information about a programs internal state and operations, the "semantics" of a program can be traced and used to provide even more accurate information about the effects of a code change.

Another possible avenue of further work could inspect whether the unique signatures generated through random modification of a node could be used to identify similar nodes in other programs. If these nodes can be uniquely identified across problems we may be able to find similar improvement opportunities across problems. As such this would represent a signature-based method of searching for similar code.

5.4.3 Problem Set

New implementations of sorting algorithms can be easily added to our experiments. Addition fitness functions would allow programs for other purposes to be tested under this system. Having a wider range of software to test our system on would provide better

information regarding in what circumstances bias allocation is particularly useful.

As mentioned in section 4.5.4 inspecting the effect of bias on larger programs is of importance. Finding where to change in particularly large programs is an important ability when improving real-world programs and would more comprehensively explore any scaling advantages for GP.

5.4.4 Broader Work

A range of broader follow on work can be conducted. We experiment in this thesis with the use of GP. A number of modifications could be made to the algorithm to restrict the type of program that can be generated to prevent uncompilable programs [18] or repeat evaluation of the same programs [32].

We evolve programs represented as an AST. Another approach which aids GP's applicability to larger programs is the evolution of code patches [73]. It is unclear how evolving patches effects the search process but the approach appears useful when modifying larger programs. A patch-centric approach also puts emphasis on what is to be changed in a program. When dealing with software improvement this is a more appropriate model for designing code enhancement techniques where the portion of code which must change in a program is relatively small, on the order of 1%. It is appealing conceptually at least, as it draws attention to the evolution of software change.

Our experiments allow a single mutation and/or a single crossover to be performed in the generation of programs. This could be extended to allow multiple modifications at the same time to produce more varied programs and aid reachability (subsubsection 2.3.2.1). Using patches as the unit of evolution may support this.

It is unclear how a program and its execution environment can expose improvements under modification. As certain programming languages are more suited to evolution with GP perhaps certain programming languages are more likely to expose performance improvements through modification.

5.4.5 Open Questions

Although we have made progress in describing bias allocation we do not have a general way to find potential improvements in a program. The problem of localising modification for performance improvement in the general case remains an open problem.

Why a program exposes performance improvements is not known. Perhaps there is some programming language whose semantics particularly expose performance improvements. The nature of the language used in this case and why different modifications produce largely unique program spectra is not known.

5.5 Closing Remarks

At the core of our work is the ability to inspect code by measuring the effect of change on a programs behaviour. Applying modification randomly throughout a program can produce a range of behaviour which can be analysed to understand that code. Further investigation into the plausibility of this concept is worth pursuing in our opinion especially if it can be generalised. The goal is that it may then be more practically used as an additional program analysis technique for software engineers.

More broadly, the field of Search Based Software Engineering appears appropriately poised to have a further impact on the discipline of Software Engineering. Given that a huge amount of code exists and is readily accessible there is great potential for reuse. As the cost of computing reduces, leading to more comprehensive software management environments such as test and build infrastructure, it appears a logical next step to automate the reuse of code. If quality can be maintained, and the cost of distributed program evaluation reduces, then randomised search may be widely practical as a "trail-and-error" approach to the nuanced reuse of existing software for improvement.

Bibliography

[1] Abbott, R., and Parviz, J. G. B. Guided genetic programming. In *International Conference on Machine Learning; Models, Technologies and Applications (MLMTA)* (2003), Citeseer, pp. 28–34. 2.3.4

[2] Agapitos, A., and Lucas, S. M. Evolving modular recursive sorting algorithms. In *European Conference on Genetic Programming (EuroGP)* (2007), vol. 4445, Springer Science & Business Media, p. 301. 4.2.1

[3] Ammons, G., Choi, J.-D., Gupta, M., and Swamy, N. Finding and removing performance bottlenecks in large systems. In *Object-Oriented Programming (ECOOP)*. Springer, 2004, pp. 172–196. 2.6.1

[4] Angeline, P. J. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence: A Dynamic Systems Perspective* (1995), IEEE Press, pp. 152–163. 2.4

[5] Angeline, P. J. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In *Conference on Genetic Programming* (1996), MIT Press, pp. 21–29. 3.2.2

[6] Angeline, P. J. Two self-adaptive crossover operators for genetic programming. In *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. MIT Press, Cambridge, MA, USA, 1996, ch. 5, pp. 89–110. 1.2, 1.8, 2.4, 2.5.2, 3.1, 3.3.2, 3.4, 4.5.5, 4.5.5

[7] Aporntewan, C., and Chongstitvatana, P. A quantitative approach for validating the building-block hypothesis. In *IEEE Congress on Evolutionary Computation (CEC)* (2005), vol. 2, IEEE, pp. 1403–1409. 2.4.1.2

[8] Arcuri, A. *Automatic software generation and improvement through search based techniques*. PhD thesis, University of Birmingham, 2009. 2.3, 2.3.1, 2.3.3.2

[9] Arcuri, A., White, D., Clark, J., and Yao, X. Multi-objective improvement of software using co-evolution and smart seeding. *Simulated Evolution and Learning* (2008), 61–70. 2.3.3.2, 2.3.6, 3.2

[10] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2012), pp. 307–320. 2.6.2

[11] Azeemi, N. Z. Architecture-aware hierarchical probabilistic source optimization. In *International Conference on Parallel and Distributed Computing Systems* (2006), pp. 90–95. 2.3.5

[12] Bartolomé, J., and Guitart, J. A survey on java profiling tools. Tech. rep., Research Report number: UPC-DAC-2001-13/UPC-CEPBA-2001-10, 2001. 2.6.1

[13] Binder, W., and Hulaas, J. Using bytecode instruction counting as portable cpu consumption metric. *Electronic Notes in Theoretical Computer Science 153*, 2 (2006), 57–77. 2.6.1

[14] Binder, W., Hulaas, J., and Moret, P. Advanced java bytecode instrumentation. In *Symposium on Principles and practice of programming in Java* (2007), ACM, pp. 135–144. 2.6.1

[15] Brandolese, C., Fornaciari, W., Salice, F., and Sciuto, D. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers 11*, 05 (2002), 477–502. 2.3.2.1

[16] Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice 17*, 5 (2005), 309–332. 2.2

[17] Castle, T. *Evolving High-Level Imperative Program Trees with Genetic Programming*. PhD thesis, University of Kent, 2012. 2.6, 5.4.2

[18] Castle, T., and Johnson, C. G. Evolving high-level imperative program trees with strongly formed genetic programming. In *European conference on Genetic Programming (EuroGP)*. Springer, 2012, pp. 1–12. 2.3.1, 3, 5.4.4

[19] Chang, P. P., Mahlke, S. A., and Hwu, W.-M. W. Using profile information to assist classic code optimizations. *Software: Practice and Experience 21*, 12 (1991), 1301–1321. 2.2.2

[20] Chen, Y., Yu, T.-L., Sastry, K., and Goldberg, D. E. A survey of linkage learning techniques in genetic and evolutionary algorithms. *IlliGAL report 2007014* (2007). 2.4.1.4

[21] Chen, Y.-p. *Extending the Scalability of Linkage Learning Genetic Algorithms: Theory & Practice*, vol. 190. Springer, 2006. 2.4.1.4

[22] Chen, Y.-P. Estimation of distribution algorithms: basic ideas and future directions. In *World Automation Congress (WAC)* (2010), IEEE, pp. 1–6. 2.4.1.4

[23] Chuang, P.-f., Chen, H., Hoflehner, G., Lavery, D., and Hsu, W.-C. Dynamic profile driven code version selection. In *Interaction between Compilers and Computer Architecture* (2007). 2.6.1

[24] Chung, E.-Y., Luca, B., DeMicheli, G., Luculli, G., and Carilli, M. Value-sensitive automatic code specialization for embedded software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21*, 9 (2002), 1051–1067. 2.3.3.1

[25] Cinnéide, M. O. *Automated application of design patterns: a refactoring approach*. PhD thesis, Trinity College Dublin., 2001. 2.3.1, 2.3.5

[26] Clarke, J., Dolado, J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., et al. Reformulating software engineering as a search problem. *IEE Proceedings Software 150*, 3 (2003), 161–175. 1

[27] Coello, C. C., and Becerra, R. L. Evolutionary multiobjective optimization using a cultural algorithm. In *Swarm Intelligence Symposium (SIS)* (2003), IEEE, pp. 6–13. 2.4.1.5

[28] Cooper, K. D., Schielke, P. J., and Subramanian, D. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices 34*, 7 (1999), 1–9. 1

[29] Coppa, E., Demetrescu, C., and Finocchi, I. Input-sensitive profiling. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 89–98. 1.2, 2.6.3

[30] Crawford-Marks, R., and Spector, L. Size control via size fair genetic operators in the pushgp genetic programming system. In *Genetic and Evolutionary Computation Conference (GECCO)* (2002), pp. 733–739. 2.4.2.2

[31] de Jong, E., Watson, R., and Thierens, D. On the complexity of hierarchical problem solving. In *Genetic and Evolutionary Computation Conference (GECCO)* (2005), ACM, pp. 1201–1208. 2.3.2.1

[32] Díaz, E., Tuya, J., Blanco, R., and Dolado, J. J. A tabu search algorithm for structural software testing. *Computers & Operations Research 35*, 10 (2008), 3052–3072. 5.4.4

[33] Downing, K. L. Reinforced genetic programming. *Genetic Programming and Evolvable Machines 2*, 3 (2001), 259–288. 2.4.1.4

[34] Edmonds, B. Meta-genetic programming: Co-evolving the operators of variation. *Turkish Journal of Electrical Engineering & Computer Sciences 9*, 1 (2001), 13–30. 2.4.1.1

[35] Fitzgerald, J., and Ryan, C. Individualized self-adaptive genetic operators with adaptive selection in genetic programming. In *World Congress on Nature and Biologically Inspired Computing (NaBIC)* (2013), IEEE, pp. 232–237. 2.5.1

[36] Forrest, S., Nguyen, T., Weimer, W., and Goues, C. L. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference (GECCO)* (2009), F. Rothlauf, Ed., ACM, pp. 947–954. 2.4, 3.2.4.2, 3.2.6

[37] Francone, F. D., Conrads, M., Banzhaf, W., and Nordin, P. Homologous crossover in genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)* (1999), pp. 1021–1026. 2.4.2.2, 2.5.3

[38] Franke, B., O'Boyle, M., Thomson, J., and Fursin, G. Probabilistic source-level optimisation of embedded programs. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 78–86. 2.2

[39] Friedlander, A., Neshatian, K., and Zhang, M. Meta-learning and feature ranking using genetic programming for classification: Variable terminal weighting. In *IEEE Congress on Evolutionary Computation (CEC)* (2011), IEEE, pp. 941–948. 2.4.1.2

[40] Gabel, M., and Su, Z. A study of the uniqueness of source code. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2010), ACM, pp. 147–156. 2.6.1

[41] Goldberg, D. E., and Bridges, C. L. An analysis of a reordering operator on a ga-hard problem. *Biological Cybernetics 62*, 5 (1990), 397–405. 2.3.2.1

[42] Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. Measuring empirical computational complexity. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (2007), ACM, pp. 395–404. 1, 2.6.1

[43] Harman, M., and Clark, J. Metrics are fitness functions too. In *International Symposium on Software Metrics* (2004), IEEE, pp. 58–69. 2.3.3

[44] Harman, M., Langdon, W. B., Jia, Y., White, D. R., Arcuri, A., and Clark, J. A. The gismoe challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *IEEE/ACM International Conference on Automated Software Engineering* (2012), ACM, pp. 1–14. 1, 2.3.4

[45] Harrold, M. J., Rothermel, G., Wu, R., and Yi, L. An empirical investigation of program spectra. *ACM SIGPLAN Notices 33*, 7 (1998), 83–90. 2.6, 2.6.4, 5.4.2

[46] Haynes, T. Duplication of coding segments in genetic programming. In *National Conference on Artificial Intelligence* (1996), vol. 1, AAAI Press, pp. 344–349. 2.4.1.2

[47] Helmuth, T., and Spector, L. Empirical investigation of size-based tournaments for node selection in genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO) Companion* (2012), ACM, pp. 1485–1486. 2.4.2.2

[48] Helmuth, T., Spector, L., and Martin, B. Size-based tournaments for node selection. In *Genetic and Evolutionary Computation Conference (GECCO)* (2011), ACM, pp. 799–802. 2.5.3, 3.3.5.1

[49] Hengpraprohm, S., and Chongstitvatana, P. Selective crossover in genetic programming. In *International Symposium on Communications and Information Technologies (ISCIT)* (2001). 1.2, 2.4.2.1

[50] Hewett, R. Program spectra analysis with theory of evidence. *Advances in Software Engineering 2012* (2012), 1. 2.6.4

[51] Hoste, K., and Eeckhout, L. Cole: compiler optimization level exploration. In *IEEE/ACM International Symposium on Code Generation and Optimization* (2008), ACM, pp. 165–174. 1

[52] Iba, H., and de Garis, H. Extending genetic programming with recombinative guidance. *Advances in Genetic Programming 2* (1996), 69–88. 2.5.3

[53] Ito, T., Iba, H., and Sato, S. Depth-dependent crossover for genetic programming. In *World Congress on Computational Intelligence* (1998), IEEE, pp. 775–780. 1.2, 2.3.6

[54] Ito, T., Iba, H., and Sato, S. Non-destructive depth-dependent crossover for genetic programming. In *Genetic Programming* (1998), Springer, pp. 71–82. 2.4.2.2

[55] Jackson, C. H. Multi-state models for panel data: the msm package for r. *Journal of Statistical Software 38*, 8 (2011), 1–29. 5.4.1

[56] Jackson, D. Self-adaptive focusing of evolutionary effort in hierarchical genetic programming. In *IEEE Congress on Evolutionary Computation (CEC)* (2009), IEEE, pp. 1821–1828. 2.4.1.2

[57] Jones, J. A., and Harrold, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering* (2005), ACM, pp. 273–282. 1.2, 2.6, 2.6.5

[58] Kaplan, E. L., and Meier, P. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association 53*, 282 (1958), 457–481. 5.4.1

[59] Karuri, K., Al Faruque, M. A., Kraemer, S., Leupers, R., Ascheid, G., and Meyr, H. Fine-grained application source code profiling for asip design. In *Design Automation Conference* (2005), IEEE, pp. 329–334. 2.6.1

[60] Kim, M. H., McKay, R. I. B., Hoai, N. X., and Kim, K. Operator self-adaptation in genetic programming. In *Genetic Programming*. Springer, 2011, pp. 215–226. 2.4.1.1

[61] Kinnear Jr, K. Evolving a sort: Lessons in genetic programming. In *IEEE International Conference on Neural Networks* (1993), IEEE, pp. 881–888. 2.1, 2.3.3, 2.3.4, 3.2.4.2

[62] Kinnear Jr, K. *Alternatives in automatic function definition: A comparison of performance*. MIT Press, Cambridge, MA, 1994. 2.3.4

[63] Knuth, D. E. Structured programming with go to statements. *ACM Computing Surveys (CSUR) 6*, 4 (1974), 261–301. 2.1

[64] Koza, J. *Scalable learning in genetic programming using automatic function definition*. MIT Press, 1994. 2.4.1.2

[65] Koza, J., Andre, D., Bennett III, F., and Keane, M. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In *Conference on Genetic Programming* (1996), MIT Press, pp. 132–140. 2.1, 2.3.4

[66] Koza, J. R. *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992. 2.3, A

[67] Kuperberg, M., and Becker, S. Predicting software component performance: On the relevance of parameters for benchmarking bytecode and apis. In *International Workshop on Component Oriented Programming (WCOP)* (2007), July. 2.6.1

[68] Kuperberg, M., Krogmann, M., and Reussner, R. ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation (European Joint Conferences on Theory and Practice of Software)* (2008). 2.6.1, 3.2.4.1, A

[69] Küstner, T., Weidendorfer, J., and Weinzierl, T. Argument controlled profiling. In *International Conference on Parallel Processing, Parallel Processing Workshops* (2010), Springer, pp. 177–184. 2.6.1

[70] Lambert, J. M., and Power, J. F. Platform independent timing of java virtual machine bytecode instructions. *Electronic Notes in Theoretical Computer Science 220*, 3 (2008), 97–113. 2.3.3.1, 2.6.1

[71] Langdon, W., et al. Directed crossover within genetic programming. *Advances in Genetic Programming 2* (1996). 2.1, 2.4.1.2, 3.1, 3.3.3

[72] Langdon, W. B. Performance of genetic programming optimised bowtie2 on genome comparison and analytic testing (gcat) benchmarks. *BioData Mining 8*, 1 (2015), 1. 2.3, 2.3.1

[73] Langdon, W. B., and Harman, M. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012. 1, 1.2, 2.1, 2.3.1, 2.6.3, 3.1, 4.4.2, 5.4.4

[74] Langdon, W. B., and Harman, M. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* (2013). 2.4

[75] Langdon, W. B., and Nordin, J. Seeding genetic programming populations. In *European conference on Genetic Programming (EuroGP)*. Springer, 2000, pp. 304–315. 2.1

[76] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 3–13. 2.3.4

[77] Le Goues, C., Weimer, W., and Forrest, S. Representations and operators for improving evolutionary software repair. In *Genetic and Evolutionary Computation Conference (GECCO)* (2012), ACM, pp. 959–966. 2.3.1, 2.3.3.2

[78] Lee, H. B., and Zorn, B. G. Bit: A tool for instrumenting java bytecodes. In *USENIX Symposium on Internet Technologies and Systems* (1997), pp. 73–82. 2.6.1

[79] Li, X., Garzaran, M. J., and Padua, D. Optimizing sorting with genetic algorithms. In *International Symposium on Code Generation and Optimization* (2005), IEEE Computer Society, pp. 99–110. 2.3

[80] Liu, Y. Operator adaptation in evolutionary programming. In *Advances in Computation and Intelligence*. Springer, 2007, pp. 90–99. 2.4

[81] Llora, X., Verma, A., Campbell, R. H., and Goldberg, D. E. When huge is routine: scaling genetic algorithms and estimation of distribution algorithms via data-intensive computing. In *Parallel and Distributed Computational Intelligence*. Springer, 2010, pp. 11–41. 2.3.6

[82] Looks, M. Scalable estimation-of-distribution program evolution. In *Conference on Genetic and evolutionary computation* (2007), ACM, pp. 539–546. 2.4.1.4

[83] López, E., Poli, R., and Coello, C. Reusing code in genetic programming. *Genetic Programming* (2004), 359–368. 2.3.4, 2.4.1.2

[84] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices 40*, 6 (2005), 190–200. 2.6.1

[85] Luke, S. *Issues in scaling genetic programming: Breeding strategies, tree generation, and code bloat.* PhD thesis, Department of Computer Science, University of Maryland, College Park., 2000. 1.1, 2.3.6

[86] Luke, S. Modification point depth and genome growth in genetic programming. *Evolutionary Computation 11*, 1 (2003), 67–106. 2.3.6, 2.4.2.2

[87] Majeed, H., and Ryan, C. A less destructive, context-aware crossover operator for gp. *Genetic Programming* (2006), 36–48. 2.4.2, 2.4.2.1, 2.4.2.2

[88] McMinn, P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability 14*, 2 (2004), 105–156. 1

[89] McPhee, N. F., Ohs, B., and Hutchison, T. Semantic building blocks in genetic programming. In *Genetic Programming*. Springer, 2008, pp. 134–145. 2.4.2.3

[90] Mitchell, M., Holland, J. H., Forrest, S., et al. When will a genetic algorithm outperform hill climbing? In *International Conference on Genetic Algorithms* (1993), pp. 51–58. 2.2.3

[91] Moon, S., Kim, Y., Kim, M., and Yoo, S. Ask the mutants: Mutating faulty programs for fault localization. In *International Conference on Software Testing, Verification and Validation (ICST)* (2014), IEEE, pp. 153–162. 2.6.5

[92] Muller, G., Marlet, R., and Volanschi, E. N. Accurate program analyses for successful specialization of legacy system software. *Theoretical Computer Science 248*, 1-2 (2000), 201–210. 2.6.1

[93] Myers, S., Bottolo, L., Freeman, C., McVean, G., and Donnelly, P. A fine-scale map of recombination rates and hotspots across the human genome. *Science 310*, 5746 (2005), 321–324. 3.1

[94] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Evaluating the accuracy of java profilers. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 187–197. 2.6.1

[95] O'Keeffe, M., and Cinnéide, M. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice 20*, 5 (2008), 345–364. 2.3.1

[96] Ondas, R., Pelikan, M., and Sastry, K. Scalability of genetic programming and probabilistic incremental program evolution. In *Genetic and Evolutionary Computation Conference (GECCO)* (2005), ACM, pp. 1785–1786. 2.4.1.4

[97] O'Reilly, U.-M., and Oppacher, F. The troubling aspects of a building block hypothesis for genetic programming. In *Foundations of Genetic Algorithms (FOGA)* (1994), pp. 73–88. 2.4.2

[98] Orlov, M., and Sipper, M. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation 15*, 2 (2011), 166–182. 2.3.1

[99] Pan, Z., and Eigenmann, R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization* (2006), IEEE, pp. 319–332. 2.2.1

[100] Papadakis, M., Delamaro, M. E., and Le Traon, Y. Proteum/fl: A tool for localizing faults using mutation analysis. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2013), IEEE, pp. 94–99. 2.6.5

[101] Pellacini, F. User-configurable automatic shader simplification. In *ACM Transactions on Graphics (TOG)* (2005), vol. 24, ACM, pp. 445–452. 2.2.3

[102] Peng, B., and Reynolds, R. G. Cultural algorithms: Knowledge learning in dynamic environments. In *Congress on Evolutionary Computation (CEC)* (2004), vol. 2, IEEE, pp. 1751–1758. 2.4.1.5

[103] Perez, A. Dynamic code coverage with progressive detail levels. *arXiv preprint arXiv:1306.4546* (2013). 2.6.1

[104] Petke, J., Harman, M., Langdon, W. B., and Weimer, W. Using genetic improvement & code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming (EuroGP)* (2014). 2.3.4

[105] Petke, J., Langdon, W. B., and Harman, M. Applying genetic improvement to MiniSAT. In *Symposium on Search-Based Software Engineering (SSBSE)* (Leningrad, Aug. 24-26 2013), G. Fraser, Ed. Short Papers. 2.3, 2.3.3.2

[106] Poli, R., Langdon, W., and McPhee, N. *A field guide to genetic programming.* Lulu Enterprises UK Ltd, 2008. 1, 2.3, 3.2.5

[107] Poli, R., and McPhee, N. F. A linear estimation-of-distribution gp system. In *Genetic Programming.* Springer, 2008, pp. 206–217. 2.4.1.4

[108] Räihä, O. A survey on search-based software design. *Computer Science Review 4*, 4 (2010), 203–249. 1

[109] Reynolds, R. G., and Zhu, S. Knowledge-based function optimization using fuzzy cultural algorithms with evolutionary programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part B 31*, 1 (2001), 1–18. 2.4.1.5

[110] Roberts, S. C., Howard, D., and Koza, J. R. Evolving modules in genetic programming by subtree encapsulation. In *Genetic Programming.* Springer, 2001, pp. 160–175. 2.4.1.2

[111] Rosca, J. P. Generality versus size in genetic programming. In *Conference on Genetic Programming* (1996), MIT Press, pp. 381–387. 2.4.2

[112] Rosca, J. P., and Ballard, D. H. Causality in genetic programming. In *International Conference Genetic Algorithms (ICGA)* (1995), vol. 95, pp. 256–263. 2.4.1.2

[113] Schmidhuber, J., Zhao, J., and Wiering, M. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning 28*, 1 (1997), 105–130. 2.4.1.2

[114] Schulte, E., Dorn, J., Harding, S., Forrest, S., and Weimer, W. Post-compiler software optimization for reducing energy. In *International conference*

on Architectural support for programming languages and operating systems (2014), ACM, pp. 639–652. 2.3.1

[115] Schulte, E., Fry, Z. P., Fast, E., Weimer, W., and Forrest, S. Software mutational robustness. *Genetic Programming and Evolvable Machines 15*, 3 (2012), 281–312. 2.3.3.2, 4.2.1

[116] Shan, Y., McKay, R. I., Essam, D., and Abbass, H. A. A survey of probabilistic model building genetic programming. In *Scalable Optimization via Probabilistic Modeling*. Springer, 2006, pp. 121–160. 2.4.1.4

[117] Shen, L., and He, J. Evolutionary programming using a mixed strategy with incomplete information. In *Workshop on Computational Intelligence (UKCI)* (2010), IEEE, pp. 1–6. 2.4.1.1

[118] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM SIGSOFT symposium and the European conference on Foundations of software engineering* (2011), ACM, pp. 124–134. 2.2.3

[119] Simons, C. *Interactive evolutionary computing in early lifecycle software engineering design*. PhD thesis, University of the West of England, 2011. 2.3.1, 2.3.3, 2.3.4, 2.3.5

[120] Sitthi-Amorn, P., Modly, N., Weimer, W., and Lawrence, J. Genetic programming for shader simplification. In *ACM Transactions on Graphics (TOG)* (2011), vol. 30, ACM, p. 152. 2.2.3, 2.3.4, 2.4.2.1

[121] Soule, T., and Foster, J. A. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation 6*, 4 (1998), 293–309. 2.3.3.1

[122] Spector, L. Autoconstructive evolution: Push, pushgp, and pushpop. In *Genetic and Evolutionary Computation Conference (GECCO)* (2001), pp. 137–146. 2.3.1, 3.1

[123] Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 77–90. 2.2.2

[124] Stephenson, M., O'Reilly, U., Martin, M. C., and Amarasinghe, S. Genetic programming applied to compiler heuristic optimization. In *European conference on Genetic Programming (EuroGP)*. Springer, 2003, pp. 238–253. 2.3

[125] Su, Y.-Y., Attariyan, M., and Flinn, J. Autobash: improving configuration management with operating system causality analysis. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 237–250. 2.2.1

[126] Tackett, W. A., and Carmi, A. The unique implications of brood selection for genetic programming. In *IEEE World Congress on Computational Intelligence (CEC)* (1994), IEEE, pp. 160–165. 2.2.3, 2.4.2.1

[127] Tallent, N. R., Mellor-Crummey, J. M., Adhianto, L., Fagan, M. W., and Krentel, M. Diagnosing performance bottlenecks in emerging petascale applications. In *Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 51. 2.6.1

[128] Teller, A. Evolving programmers: The co-evolution of intelligent recombination operators. *Advances in Genetic Programming 2* (1996), 45–68. 2.4.1.1

[129] Teller, A. *Algorithm evolution with internal reinforcement for signal understanding*. PhD thesis, University of Tokyo, 1998. 2.4.1.2

[130] Terrio, M. D., and Heywood, M. Directing crossover for reduction of bloat in gp. In *Electrical and Computer Engineering (CCECE)* (2002), vol. 2, IEEE, pp. 1111–1115. 2.1

[131] Terrio, M. D., and Heywood, M. I. On naive crossover biases with reproduction for simple solutions to classification problems. In *Genetic and Evolutionary Computation (GECCO)* (2004), Springer, pp. 678–689. 2.5.4

[132] The Eclipse Foundation. Java development tools. http://www.eclipse.org/jdt/, Nov. 2012. 3.2.1, A

[133] Uy, N. Q., Hoai, N. X., OâĂŹNeill, M., McKay, R. I., and Galván-López, E. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines 12*, 2 (2011), 91–119. 2.4.2.3, 2.6.4

[134] Vanneschi, L., Castelli, M., and Silva, S. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines 15*, 2 (2014), 195–214. 2.4.2.3

[135] Various. Rosettacode.org. http://rosettacode.org. Accessed: 2014-09-30. 4.2.1, 4.2.1, 5.2, B, 5, 6, 7, 8, 9, 10, 12, 13, 14

[136] Villarreal, J., Suresh, D., Stitt, G., Vahid, F., and Najjar, W. Improving software performance with configurable logic. *Design Automation for Embedded Systems 7*, 4 (2002), 325–339. 2.2.1

[137] Vuduc, R., Demmel, J. W., and Bilmes, J. A. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications 18*, 1 (2004), 65–94. 1, 2.3.6

[138] Ward, M. *Proving program refinements and transformations.* PhD thesis, University of Oxford, 1989. 2.2

[139] Weimer, W., Forrest, S., Goues, C. L., and Nguyen, T. Automatic program repair with evolutionary computation. *Communications of the ACM 53*, 5 (2010), 109–116. 2.3.3.2, 2.6.5

[140] Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. Automatic program repair with evolutionary computation. *Communications of the ACM 53*, 5 (2010), 109–116. 1, 1.2, 2.3.1, 3.1

[141] Weimer, W., Nguyen, T., Goues, C. L., and Forrest, S. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)* (2009), IEEE, pp. 364–374. 2.1, 2.6.4

[142] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)* (2009), IEEE Computer Society, pp. 364–374. 2.1

[143] Wheeler, D. A. Sloccount, 2001. 4.2.1

[144] Whigham, P. Inductive bias and genetic programming. In *International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications* (1995), IET, IET Digital Library, pp. 461–466. 2.4.1.3

[145] Whigham, P. A. Search bias, language bias and genetic programming. In *Conference on Genetic Programming* (1996), MIT Press, pp. 230–237. 2.4, 2.4.1.3

[146] White, D., Arcuri, A., and Clark, J. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 99 (2011), 1–24. 1, 2.3, 2.3.1, 1, 4.2, 4

[147] White, D. R. *Genetic programming for low-resource systems*. PhD thesis, University of York, 2009. 2.2.2

[148] White, D. R., Clark, J., Jacob, J., and Poulding, S. M. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Genetic and Evolutionary Computation Conference (GECCO)* (2008), ACM, pp. 1775–1782. 2.3.3.1

[149] White, D. R., and Poulding, S. A rigorous evaluation of crossover and mutation in genetic programming. In *European Conference on Genetic Programming*

(EuroGP) (Tuebingen, Germany, 15-17 April 2009), vol. 5481, Springer, pp. 220–231. 2.3.2

[150] WOODWARD, J. R., AND BAI, R. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In *ACM/SIGEVO Summit on Genetic and Evolutionary Computation* (2009), ACM, pp. 593–600. 2.3.1

[151] WOODWARD, J. R., AND SWAN, J. The automatic generation of mutation operators for genetic algorithms. In *International conference on Genetic and evolutionary computation conference companion* (2012), ACM, pp. 67–74. 2.4.1.1

[152] YANAI, K., AND IBA, H. Estimation of distribution programming: Eda-based approach to program generation. In *Towards a New Evolutionary Computation*. Springer, 2006, pp. 103–122. 2.4.1.4

[153] YOO, S., XIE, X., KUO, F.-C., CHEN, T. Y., AND HARMAN, M. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Research Note RN/14/14, Department of Computer Science, University College London, 2014. 3.3.5

[154] YUEN, C. C. *Selective crossover using gene dominance as an adaptive strategy for genetic programming*. PhD thesis, University College London, 2004. 2.4.1.2

[155] ZANNONI, E., AND REYNOLDS, R. G. Learning to control the program evolution process with cultural algorithms. *Evolutionary Computation 5*, 2 (1997), 181–211. 2.4, 2.4.1.5

Appendix A

Implementation: locoGP

To inspect our research questions we use a GP system developed specifically for this thesis named "locoGP". locoGP is a Genetic Programming (GP) [66] system written in Java for improving program performance using a Java source code representation. locoGP utilises many of the mechanisms from the related work such as seeding and modification at the language level. The main distinction between the related work and locoGP is the use of Java source code. locoGP is seeded with compilable and fully functional source code which is modified at the Java language level. The system performs the cycle of parsing, compilation, instrumentation and execution of programs in memory. Though relatively expensive, this process of software evolution is manageable on modern hardware. Logging is written to disk periodically.

Java source code is hard-coded as a string in locoGP or can be read in from files. The system operates by parsing Java source text to an Abstract Syntax Tree (AST) representation using the Eclipse Java Development Tools [132] which specifies the typing of nodes and structure of a program in the Java language. An AST gives us a representation devoid of source code artifacts such as parentheses and line terminators. The Java language syntax is enforced by the AST and is used to restrict node compatibility. These tools provide methods for cloning, traversing and modifying program trees. We use these tools to gather statements, expressions and variables from a program. The primitive set in GP is defined by Java language elements which exist in the program to be improved

and may include statements, expressions, variable names or operators.

Code elements in an AST are selected randomly or with a weighting and can be modified as per the GP operators. Where a node such as an expression is chosen, it is possible that the node can be replaced with another expression or a variable name. Where a statement is selected, it may be replaced by another statement, or statement sub-type such as an `IF` or `WHILE` statement. A replacement node can be a whole sub-tree, e.g. in the case of a statement replacement. In this case, the node which is the root of the sub-tree is of concern. If we pick a block, we clone some other random line of code as an addition to the block. Node selection is not uniform in an AST as each node is not represented in the same way within an AST. For example, Java language operators do not exist as nodes in an AST but are attributes of expression nodes.

After modification, the modified AST representation is converted back to source code text and compiled to bytecode. While typing prevents some invalid code replacement, such as replacing an operator with a variable, the AST can nonetheless be modified to produce syntactically incorrect programs which do not compile. Due to this, the number of discarded programs created due to compilation errors is relatively high.

Bytecode is instrumented using a bytecode counting library [68]. Instrumentation adds extra code to the program to count the bytecodes that are executed when the code is run. Counting bytecodes executed gives a measure which does not vary between runs, and can be expected to be portable. The instrumented bytecode is executed in a separate thread. Although the locoGP implementation is multi-threaded, the bytecode counting library relies on static methods to update execution counters, limiting the program execution phase of evaluation to serial execution. The counting results are collected after execution and are used as a performance measure.

If the code (and thread) does not finish executing within a certain time (30 seconds is adequate), the thread is stopped. Assuming a program halts within the timeout period, counting and test case results can be collected. The bytecode is executed numerous times with various test input data and the returned values are compared with known correct values. Program results (return values) are measured for correctness with a problem-

specific function by counting functionality errors. Programs which compile and execute, but exhibit runtime errors or do not finish within the time bounds are given the worst fitness values possible and are very unlikely to be selected again for modification during GP.

The evaluation mechanism, and how performance is measured affects the type of program improvement that is likely to be found. Measuring operations performed ignores platform specific differences in performance, and means the GP algorithm differentiates programs only on their ability to reduce execution count. This is expected to promote search for general code improvements.

Modifying programs in Java is highly likely to produce disfunctional programs which are ultimately given low fitness or non-compilable which are simply discarded. We end up with many programs which are close to the seed due to selection, and many programs which have very poor fitness due to destructive modifications. A timeout is used to halt longer running programs. To balance out the distribution of programs along the fitness scale, we use elitism.

locoGP can be readily extended to further Java improvement problems or used as a comparison point with other code improvement systems. New sort and prefix code classes could be added easily as functionality measures are implemented. Adding additional problems (other than sort and prefix codebook) requires writing a functionality function and adding relevant test data. There is ample room for improvement of locoGP itself in terms of implementation[1]

[1]The notion of using locoGP to improve locoGP poses an enticing challenge.

Appendix B

Code Listings

We list here the programs that make up our problem set. The programs have been taken from online sources [135] with minimal modifications to interfaces. For each problem listed, possible improvements are noted as comments within the code.

B.1 Bubble Sort

```
1  class Sort1Problem {
2    public static void sort( Integer[] a,  Integer length){
3      // ``i=0'' -> ``i=1'' or ``i++'' -> ``length--''
4      for (int i=0; i < length; i++) {
5        // ``length - 1'' -> ``length - i - 1''
6        for (int j=0; j < length - 1; j++) {
7          if (a[j] > a[j + 1]) {
8            int k=a[j];
9            a[j]=a[j + 1];
10           a[j + 1]=k;
11         }
12       }
```

```
13     }
14     return a;
15   }
16 }
```

Listing 4: Bubblesort implementation [146]

B.2 Shell Sort

```
1  public class Sort1Shell {
2    public static Integer[] sort(  Integer[] a,  Integer length){
3      int increment=length / 2;
4      while (increment > 0) {
5        for (int i=increment; i < length; i++) {
6          int j=i;
7          int temp=a[i];
8          while (j >= increment && a[j - increment] > temp) {
9            a[j]=a[j - increment];
10           j=j - increment;
11         }
12         a[j]=temp;
13       }
14       if (increment == 2) {
15         // ``='' -> ``_=''
16         increment=1;
17       }
18       else {
19         increment*=(5.0 / 11);
20       }
```

```
21      }
22      return a;
23    }
24  }
```

Listing 5: Shell Sort Implementation [135]

B.3 Selection Sort

```
1   public class Sort1SelectionProblem {
2     public static Integer[] sort(  Integer[] a,  Integer length){
3       for (int currentPlace=0; currentPlace < length - 1; currentPlace++) {
4         int smallest=Integer.MAX_VALUE;
5         // ``currentPlace + 1'' -> ``currentPlace''
6         int smallestAt=currentPlace + 1;
7         for (int check=currentPlace; check < length; check++) {
8           if (a[check] < smallest) {
9             smallestAt=check;
10            smallest=a[check];
11          }
12        }
13        int temp=a[currentPlace];
14        a[currentPlace]=a[smallestAt];
15        a[smallestAt]=temp;
16      }
17      return a;
18    }
19  }
```

Listing 6: Selection Sort Implementation [135]

B.4 Selection Sort 2

```
1  public class Sort1Selection2Problem {
2    public static Integer[] sort( Integer[] a,  Integer length){
3      double p=0;
4      int k=0;
5      // ''0'' -> ''k''
6      for (int i=0; i < length - 1; i++) {
7        k=i;
8        for (int j=i + 1; j < length; j++) {
9          if (a[j] < a[k])          k=j;
10       }
11       p=a[i];
12       a[i]=a[k];
13       a[k]=(int)p;
14     }
15     return a;
16   }
17 }
```

Listing 7: Selection Sort 2 Implementation [135]

B.5 Radix Sort

```
1  public class Sort1Radix {
2    public static Integer[] sort( Integer[] a,  Integer length){
```

```
3     // "Integer.SIZE - 1" -> "0" and "shift > -1" -> "shift == 0"
4     for (int shift=Integer.SIZE - 1; shift > -1; shift--) {
5       Integer[] tmp=new Integer[a.length];
6       int j=0;
7       for (int i=0; i < length; i++) {
8         boolean move=a[i] << shift >= 0;
9         if (shift == 0 ? !move : move) {
10          tmp[j]=a[i];
11          j++;
12        }
13        else {
14          a[i - j]=a[i];
15        }
16      }
17      for (int i=j; i < tmp.length; i++) {
18        tmp[i]=a[i - j];
19      }
20      a=tmp;
21    }
22    return a;
23  }
24 }
```

Listing 8: Radix Sort Implementation [135]

B.6 Quick Sort

```
1 public class Sort1Quick {
2   public static Integer[] sort( Integer[] a, Integer length){
```

```
3      return sort(a,0,length - 1);
4    }
5    public static Integer[] sort(  Integer[] a,  Integer p,  Integer r){
6      if (p < r) {
7        int q=0;
8        int x=a[p];
9        int i=p - 1;
10       int j=r + 1;
11       while (true) {
12         i++;
13         // "r" -> "j"
14         while (i < r && a[i] < x)         i++;
15         j--;
16         // "j > p && a[j]" -> "a[j]"
17         while (j > p && a[j] > x)         j--;
18         if (i < j) {
19           int temp=a[i];
20           a[i]=a[j];
21           a[j]=temp;
22         }
23         else {
24           q=j;
25           break;
26         }
27       }
28       sort(a,p,q);
29       sort(a,q + 1,r);
30     }
31     return a;
```

```
32    }
33  }
```

Listing 9: Quick Sort Implementation [135]

B.7 Merge Sort

```
1  public class Sort1Merge {
2    public static Integer[] sort( Integer[] a, Integer length){
3      mergesort_r(0,length,a);
4      return a;
5    }
6    public static Integer[] merge( Integer[] a, int left_start, int left_end,
7                                                int right_start, int right_end){
8      int left_length=left_end - left_start;
9      int right_length=right_end - right_start;
10     int[] left_half=new int[left_length];
11     int[] right_half=new int[right_length];
12     int r=0;
13     int l=0;
14     int i=0;
15     for (i=left_start; i < left_end; i++, l++) {
16       left_half[l]=a[i];
17     }
18     for (i=right_start; i < right_end; i++, r++) {
19       right_half[r]=a[i];
20     }
21     for (i=left_start, r=0, l=0; l < left_length && r < right_length; i++) {
22       if (left_half[l] < right_half[r]) {
```

```
23          a[i]=left_half[l++];
24        }
25        else {
26          a[i]=right_half[r++];
27        }
28      }
29      for (; l < left_length; i++, l++) {
30        a[i]=left_half[l];
31      }
32      // replace whole for statement with "i=left_start"
33      for (; r < right_length; i++, r++) {             //
34        a[i]=right_half[r];                            //
35      }                                                //
36      return a;
37    }
38    public static Integer[] mergesort_r(  int left,  int right,  Integer[] a){
39      if (right - left <= 1) {
40        return a;
41      }
42      else {
43      }
44      int left_start=left;
45      int left_end=(left + right) / 2;
46      int right_start=left_end;
47      int right_end=right;
48      mergesort_r(left_start,left_end,a);
49      mergesort_r(right_start,right_end,a);
50      merge(a,left_start,left_end,right_start,right_end);
51      return a;
```

```
52    }
53  }
```

Listing 10: Merge Sort Implementation [135]

B.8 Deceptive Bubble Sort

```
1  public class Sort1Loops1Problem {
2    public static Integer[] sort(  Integer[] a,  Integer length){
3      // "2" -> "1" or "0" to "1"
4      for (int h=2; h > 0; h--) {
5        // clone the inner loop, then delete the whole outer loop
6        // "0" -> "1" or "i++" to "length--"
7        for (int i=0; i < length; i++) {
8          // "length - 1" -> "length - i - 1"
9          for (int j=0; j < length - 1; j++) {
10           if (a[j] > a[j + 1]) {
11             int k=a[j];
12             a[j]=a[j + 1];
13             a[j + 1]=k;
14           }
15         }
16       }
17     }
18     return a;
19   }
20 }
```

Listing 11: Deceptive Bubble Sort Implementation (BubbleLoops)

B.9 Insertion Sort

```
1  public class Sort1Insertion {
2    public static Integer[] sort( Integer[] a,  Integer array_size){
3      int i, j, index;
4      // "=" -> "+="
5      for (i=1; i < array_size; ++i) {
6        // clone inside this loop to above
7        index=a[i];
8        for (j=i; j > 0 && a[j - 1] > index; j--) {
9          // in cloned version, "j - 1" -> "1 - 1"
10         a[j]=a[j - 1];
11       }
12       a[j]=index;
13     }
14     return a;
15   }
16 }
```

Listing 12: Insertion Sort Implementation [135]

B.10 Heap Sort

```
1  public class Sort1HeapProblem {
2    public static Integer[] sort( Integer[] a,  Integer array_size){
3      int i;
4      // "/" -> "+"
5      for (i=(array_size / 2 - 1); i >= 0; --i) {
6        int maxchild, temp, child, root=i, bottom=array_size - 1;
```

```
7       while (root * 2 < bottom) {
8         child=root * 2 + 1;
9         if (child == bottom) {
10          maxchild=child;
11        }
12        else {
13          if (a[child] > a[child + 1]) {
14            maxchild=child;
15          }
16          else {
17            maxchild=child + 1;
18          }
19        }
20        if (a[root] < a[maxchild]) {
21          temp=a[root];
22          a[root]=a[maxchild];
23          a[maxchild]=temp;
24        }
25        else {
26          break;
27        }
28        root=maxchild;
29      }
30    }
31    for (i=array_size - 1; i >= 0; --i) {
32      int temp;
33      temp=a[i];
34      a[i]=a[0];
35      a[0]=temp;
```

```
36        int maxchild, child, root=0, bottom=i - 1;
37        while (root * 2 < bottom) {
38          child=root * 2 + 1;
39          if (child == bottom) {
40            maxchild=child;
41          }
42          else {
43            if (a[child] > a[child + 1]) {
44              maxchild=child;
45            }
46            else {
47              maxchild=child + 1;
48            }
49          }
50          if (a[root] < a[maxchild]) {
51            // delete
52            temp=a[root];
53            a[root]=a[maxchild];
54            a[maxchild]=temp;
55          }
56          else {
57            break;
58          }
59          root=maxchild;
60        }
61      }
62      return a;
63    }
64  }
```

Listing 13: Heap Sort Implementation [135]

B.11 Cocktail Sort

```
public class Sort1Cocktail {
  public static Integer[] sort(  Integer[] a,  Integer length){
    boolean swapped;
    do {
      swapped=false;
      for (int i=0; i <= length - 2; i++) {
        if (a[i] > a[i + 1]) {
          int temp=a[i];
          a[i]=a[i + 1];
          a[i + 1]=temp;
          swapped=true;
        }
      }
      if (!swapped) {
        break;
      }
      swapped=false;
      for (int i=length - 2; i >= 0; i--) {
        if (a[i] > a[i + 1]) {
          int temp=a[i];
          a[i]=a[i + 1];
          a[i + 1]=temp;
          swapped=true;
        }
```

```
25        }
26      }
27      while (swapped);
28      return a;
29    }
30  }
```

Listing 14: Cocktail Sort Implementation [135]

B.12 Huffman Codebook Generator

Improvements are the same as for Bubblesort.

```
1  package huffmanCodeTable;
2  public class BasicHuffman {
3    public static String[] getCodeBook(  Byte[] bytes){
4      BubbleSort.sort(bytes,bytes.length);
5      Byte[] uniqueChars=getUniqueChars(bytes);
6      huffmanNode[] freqTable=getCharFreq(bytes,uniqueChars);
7      huffmanNode huffTree=buildTree(freqTable);
8      String[] codeBook=new String[0];
9      codeBook=getCodes(huffTree,"",codeBook);
10     return codeBook;
11   }
12   private static String[] getCodes(  huffmanNode huffTree,  String prefix,
13                                                        String[] codeBook){
14     if (huffTree.uniqueChar != null) {
15       codeBook=addString(prefix,codeBook);
16     }
17     else {
```

```
18        codeBook=getCodes(huffTree.left,prefix + "1",codeBook);
19        codeBook=getCodes(huffTree.right,prefix + "0",codeBook);
20      }
21      return codeBook;
22    }
23    private static String[] addString(  String aStr,  String[] otherStrings){
24      String[] newStrings=new String[otherStrings.length + 1];
25      for (int i=0; i < otherStrings.length; i++) {
26        newStrings[i]=otherStrings[i];
27      }
28      newStrings[newStrings.length - 1]=aStr;
29      return newStrings;
30    }
31    private static huffmanNode buildTree(  huffmanNode[] freqTable){
32      BubbleSort.sort(freqTable,freqTable.length);
33      huffmanNode aRight=freqTable[freqTable.length - 1];
34      huffmanNode aLeft=freqTable[freqTable.length - 2];
35      huffmanNode newNode=new huffmanNode(aRight.getFreq() +
36      aLeft.getFreq(),aRight,aLeft);
37      huffmanNode[] newList=new huffmanNode[freqTable.length - 1];
38      for (int i=0; i < newList.length; i++) {
39        newList[i]=freqTable[i];
40      }
41      newList[newList.length - 1]=newNode;
42      if (newList.length == 1) {
43        return newList[0];
44      }
45      else {
46        return buildTree(newList);
```

```
47      }
48    }
49    private static huffmanNode[] getCharFreq(  Byte[] bytes,  Byte[] uniqueChars){
50      int[] freqInts=new int[uniqueChars.length];
51      int charIndex=0;
52      for (int i=0; i < bytes.length; i++) {
53        if (bytes[i].compareTo(uniqueChars[charIndex]) == 0) {
54          freqInts[charIndex]++;
55        }
56        else {
57          charIndex++;
58          freqInts[charIndex]++;
59        }
60      }
61      huffmanNode[] freqTable=new huffmanNode[uniqueChars.length];
62      for (int i=0; i < uniqueChars.length; i++) {
63        freqTable[i]=new huffmanNode(uniqueChars[i],freqInts[i]);
64      }
65      return freqTable;
66    }
67    private static Byte[] getUniqueChars(  Byte[] bytes){
68      Byte[] returnChars=new Byte[1];
69      returnChars[0]=bytes[0];
70      for (int i=0; i < bytes.length; i++) {
71        if (returnChars[returnChars.length - 1].compareTo(bytes[i]) != 0) {
72          Byte[] tempChars=returnChars;
73          returnChars=new Byte[tempChars.length + 1];
74          for (int j=0; j < tempChars.length; j++) {
75            returnChars[j]=tempChars[j];
```

```
76          }
77          returnChars[returnChars.length - 1]=bytes[i];
78        }
79      }
80      return returnChars;
81    }
82  }
83
84  package huffmanCodeTable;
85  public class BubbleSort {
86    public static <T extends Comparable<? super T>>void sort(  T[] a,
87                                                           Integer length){
88      for (int i=0; i < length; i++) {
89        for (int j=0; j < length - 1; j++) {
90          if (a[j].compareTo(a[j + 1]) < 0) {
91            T k=a[j];
92            a[j]=a[j + 1];
93            a[j + 1]=k;
94          }
95        }
96      }
97    }
98  }
99
100 package huffmanCodeTable;
101 public class huffmanNode implements Comparable {
102   Byte uniqueChar=null;
103   int freq=0;
104   huffmanNode left, right;
```

```
105   public int getFreq(){
106     return freq;
107   }
108   huffmanNode( byte aChar, int freq){
109     uniqueChar=aChar;
110     this.freq=freq;
111   }
112   huffmanNode( int freq, huffmanNode left, huffmanNode right){
113     this.freq=freq;
114     this.right=right;
115     this.left=left;
116   }
117   @Override public int compareTo( Object hN){
118     return this.freq - ((huffmanNode)hN).freq;
119   }
120 }
```

Listing 15: Huffman Codebook Implementation

Appendix C

Bias Rules

During the course of this thesis a number of bias update mechanisms were inspected. Performance and functionality were tested for ability to highlight performance improvements. The magnitude of bias attributed during an update was also tested. The most appealing predictor was chosen to be functionality at a fixed magnitude update, which was used in the body of this thesis.

C.0.1 Bias Allocation Rules

We measure bias derived from rules on a seed program as an indicator for how the rules may perform when applied during a GP run. To illustrate this difference. We use bias generated on a seed program as an indicator of bias's success. The best bias allocation rules may not be best across static and dynamic instances.

The emergence of bias during a GP run is particularly resistant to simple observation as the bias generated is different for each program in the population, and the effect of bias on a GP run is an aggregate of the different programs, fitnesses and even program lineage. Two programs derived from different lineages of programs may have different bias values. Thus it is not easy to inspect the effectiveness and operation of bias during GP.

We inspect a range of rules for allocating bias to locations in a program. We use

bubblesort to inspect how rules allocate bias. The seed program is repeatedly modified and the bias updated per our rules. The bias in the seed program is noted for every 5 individuals successfully generated to show how bias emerges. Tracing the bias updates can show how often interesting nodes in bubblesort or ranked highest per their node values, and in turn their likelihood of being selected for modification.

Table C.1 shows the results of a number of rules which were tested. The test involved randomly modifying bubblesort and updating bias values over the creation of 2500 program variants. An average rank of each node was determined, and ranking nodes which are required to change to improve the program are noted.

"% top nodes" - The percentage of time improvement nodes were ranked in the top 5.

"# inter nodes top" - The number of times the improvement nodes were ranked in the top 5.

"last ind #" - Showing the number of variant programs which were attempted but did not compile. 2500 programs were evaluated, but this count shows the number evaluated and number which didn't compile.

"# most inster top" - This distinguishes between nodes that could be involved in improvement, and those which are definitely involved in improvement. It is a smaller set of nodes. "# top changes" - The number of times the nodes in the top 5 changed.

"# uniq int top" - The number of interesting nodes which were top at any point in the process. This gauges the churn of interesting nodes at the top.

"# uniq top nodes" - The number of nodes which made it to the top 5. This is a rough gauge of churn.

"R15" is the ruleset which was finally used in the evaluation of this thesis. This was because it ranked improvement nodes relatively high (17% of the time) and performed well on other problems in early trials. Other rules with higher rankings of improvement nodes had other issues, such as creating a large number of program variants (R26 series) or performing poorly in subsequent tests during GP. The use of exhaustive modification of a program as shown in section 4.6 should be used to instruct rule design before the

use of extensive bias accumulation using GP as shown in this appendix. These results are included here briefly to show how the experimental ruleset was chosen for the thesis. It can also be noted that this approach is not very accurate at predicting what rule will work well during GP.

	% top nodes	# inter nodes top	last ind #	# most inster top	# top changes	# uniq int top	# uniq top nodes	% uniq nodes inter
R4	0	0	2700.42	0	1	0	1	0
R4rr	0	0	2252.66	0	1	0	1	0
R5	0	0	2278	0	1	0	1	0
R7	0	0	2338	0	1	0	1	0
R21	0	0	4440	0	83.4	0	6.2	0
R3	0	0	3656.8	0	23.2	0	3	0
R3rr	0	0	4098.6	0	21	0	3.8	0
R4	0	0	3348.2	0	96.4	0	8.8	0
R7	0	0	2895	0	4.07142	0	1.6428	0
R8	0	0	3398.25	0	1	0	1	0
R9	0	0	3327.5	0	1	0	1	0
R22	0.0024	0.6	4383.8	0.6	77.6	0.2	6.6	0.0303
R17	0.0024	0.6	4353.2	0.6	55.4	0.4	11	0.0363
R18	0.0032	0.8	2689.4	0.8	74.8	0.2	7.8	0.0256
R23	0.0056	1.4	3679.4	1.2	43.6	0.6	7.8	0.0769
R20	0.0104	2.6	4323	1.6	51.2	1.4	11.4	0.1228
R25	0.022	5.5	4199.5	4.75	72	2	31.5	0.0634
R3	0.035	8.75	2385.29	8.75	2.25	0.0833	1.7916	0.0465
R24	0.0368	9.2	4135.2	7	62.6	2.8	26.8	0.1044
R13	0.045	11.25	3553.25	11.25	58	0.75	9	0.0833
R3rr	0.0472	11.8	3306.4	11.8	6.8	0.2	3.2	0.0625
R14	0.077	19.25	2605	0	41.75	0.75	6.25	0.12
R12	0.077	19.25	2484.75	0	55.75	1	6.75	0.148
R11	0.09	22.5	2598	0	53	1	7.25	0.1379
R25	0.10145	25.363	3987.272	18.9090	104.636	3.4545	44.3636	0.0778
R10	0.114	28.5	2610	0	45.5	1	8.25	0.1212
R5	0.12	30	2383.6	0	60.8	1	7.6	0.1315
R6	0.13	32.5	6372	25	63.75	2	10.75	0.1860
R19	0.1304	32.6	3991.2	19.6	156.6	4.6	53.4	0.0861
R4rr	0.1648	41.2	2757.2	0	87	1	9.6	0.1041
R26tinyDecay	0.166	41.5	12910.5	12.5	170.5	2	9.5	0.2105
R15	0.172	43	3403.8	38.4	133.2	3	44.6	0.0672
R26	0.1792	44.8	3970	30.8	169.4	4.8	53	0.0905

R9	0.186	46.5	2635.75	0	2.5	0.25	1.25	0.2
R26	0.208	52	3740.75	38.75	145.5	4	50.75	0.0788
R16	0.2344	58.6	3919.2	58.6	48.6	0.6	9.4	0.0638
R8	0.325	81.25	2667.5	0	2.75	0.5	1.25	0.4
R26	0.4355	108.888	5230.833	96.222	183.5	4.2222	46.83	0.09015
R6	0.444	111	10261.25	93.5	98	2	8.75	0.2285
R26smallDecay	0.5843	146.0795	12516.2045	130.011	136.3295	2.0909	10.3636	0.2017
R26noDecay	0.6636	165.9183	12908.469	165.918	1.5714	1	1.571	0.6363

Table C.1: Rulesets and the Movement of Top Ranked Nodes

Appendix D

Absolute Values for Dynamic Bias

The majority of graphs in this thesis show a measure of the *difference* between GP with and without bias to observe at what points during the GP process there is a statistically significant difference. This section contains graphs for each problem showing separate data series for the absolute values of canonical GP and dynamic bias. The GP process is repeated 100 times. Data points for each GP configuration are derived by resampling the 100 GP runs. For each generation 100 mean values were calculated from 1000 samples (with replacement) and used to surround data points with grey bands showing quantiles at 97.5 and 2.5 to give an estimated 95% confidence interval.

D.1 Insertion Sort

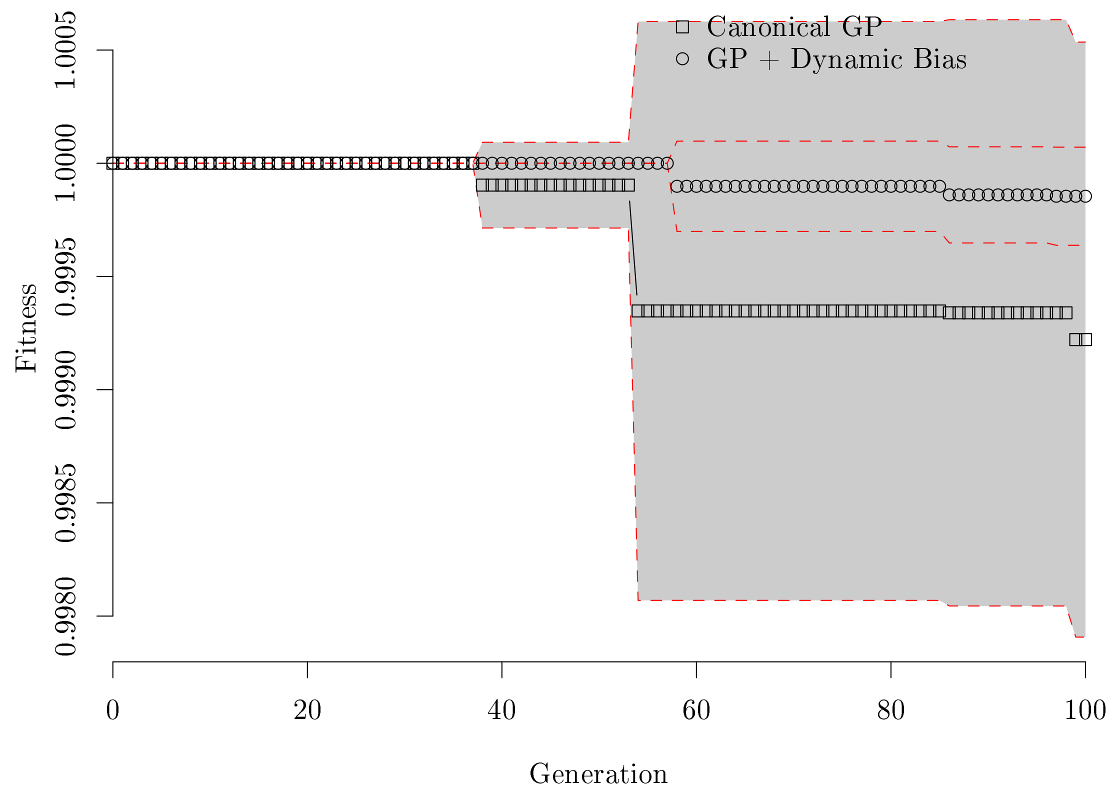


Fig. D.1: GP with Dynamic mutation-derived bias on Insertion Sort

D.2 Heap Sort

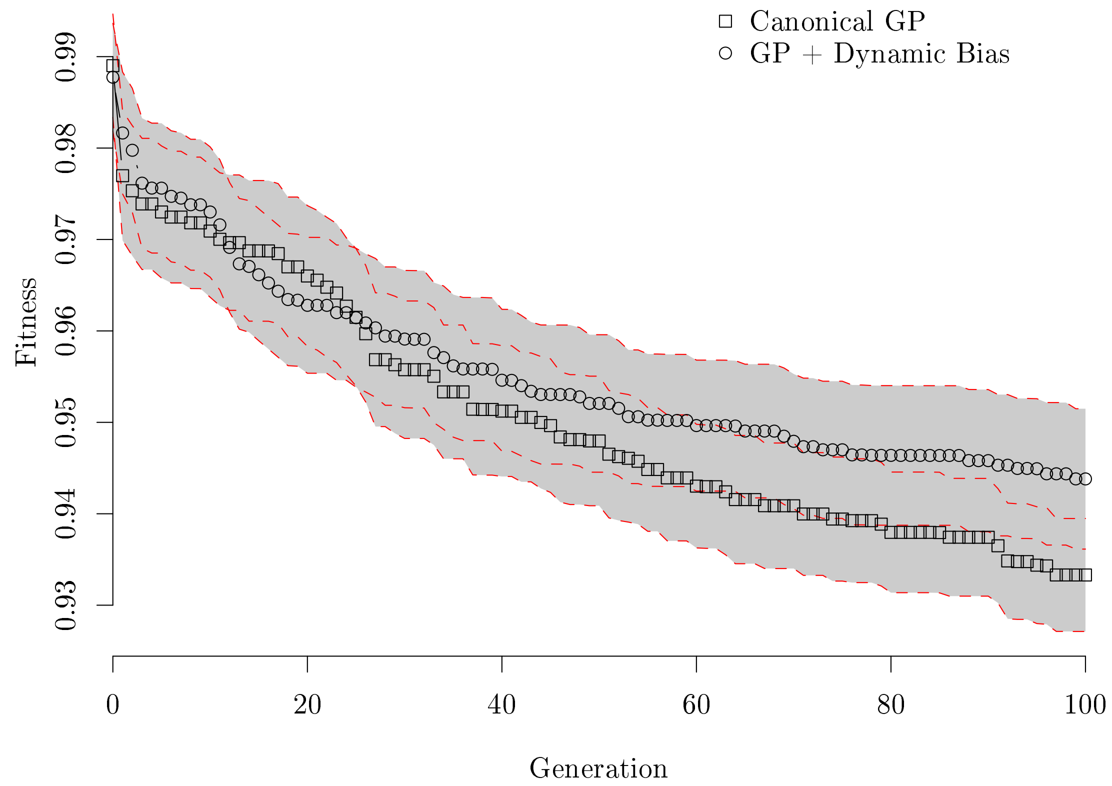


Fig. D.2: GP with Dynamic mutation-derived bias on Heap Sort

D.3 Merge Sort

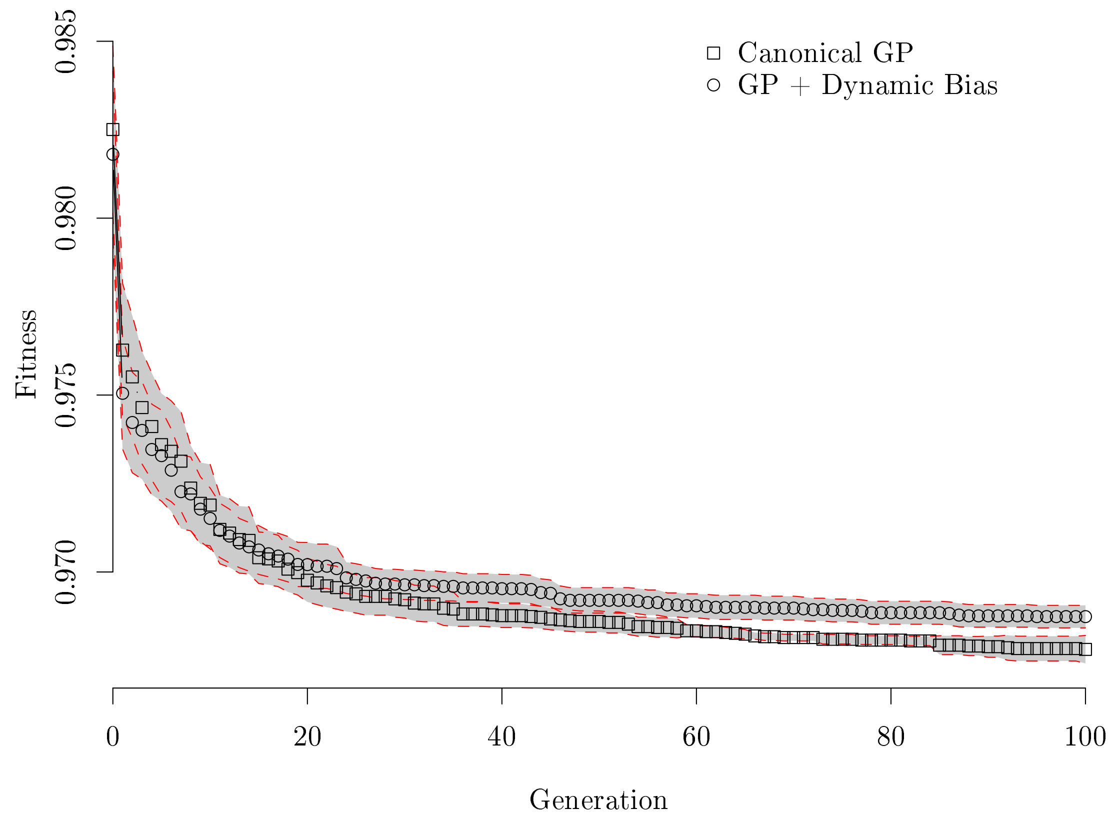


Fig. D.3: GP with Dynamic mutation-derived bias on Merge Sort

D.4 Radix Sort

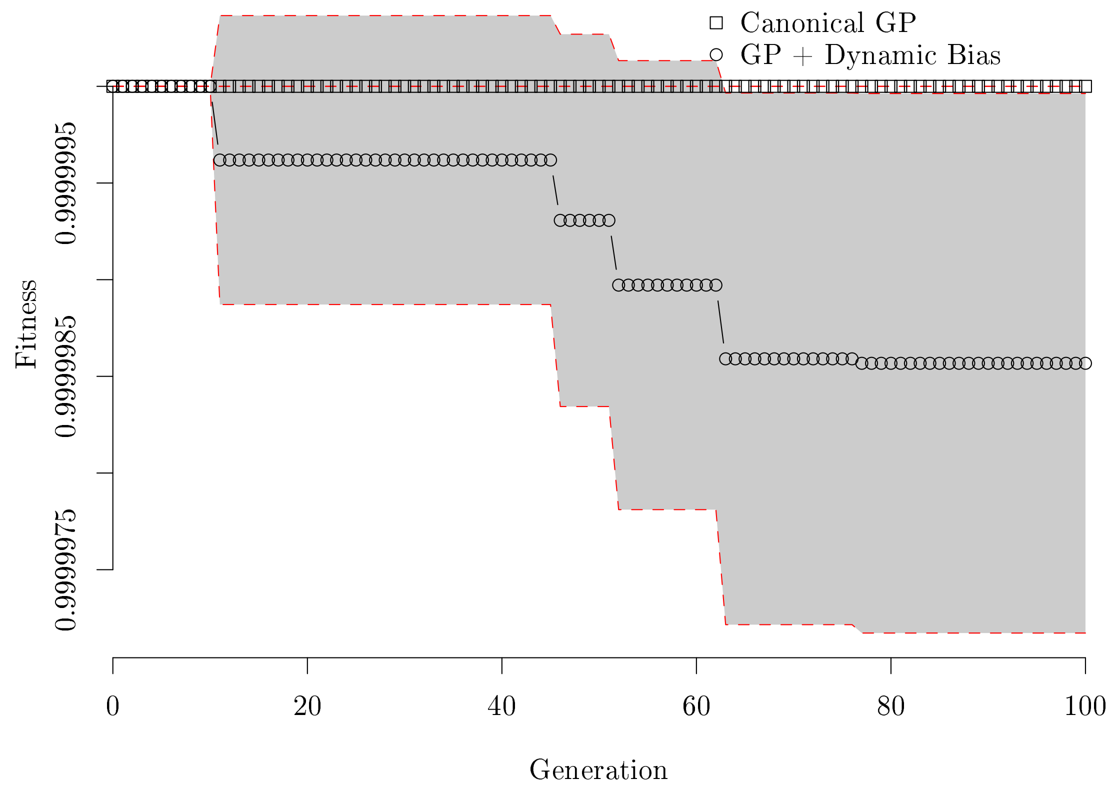


Fig. D.4: GP with Dynamic mutation-derived bias on Radix Sort

D.5 Selection Sort

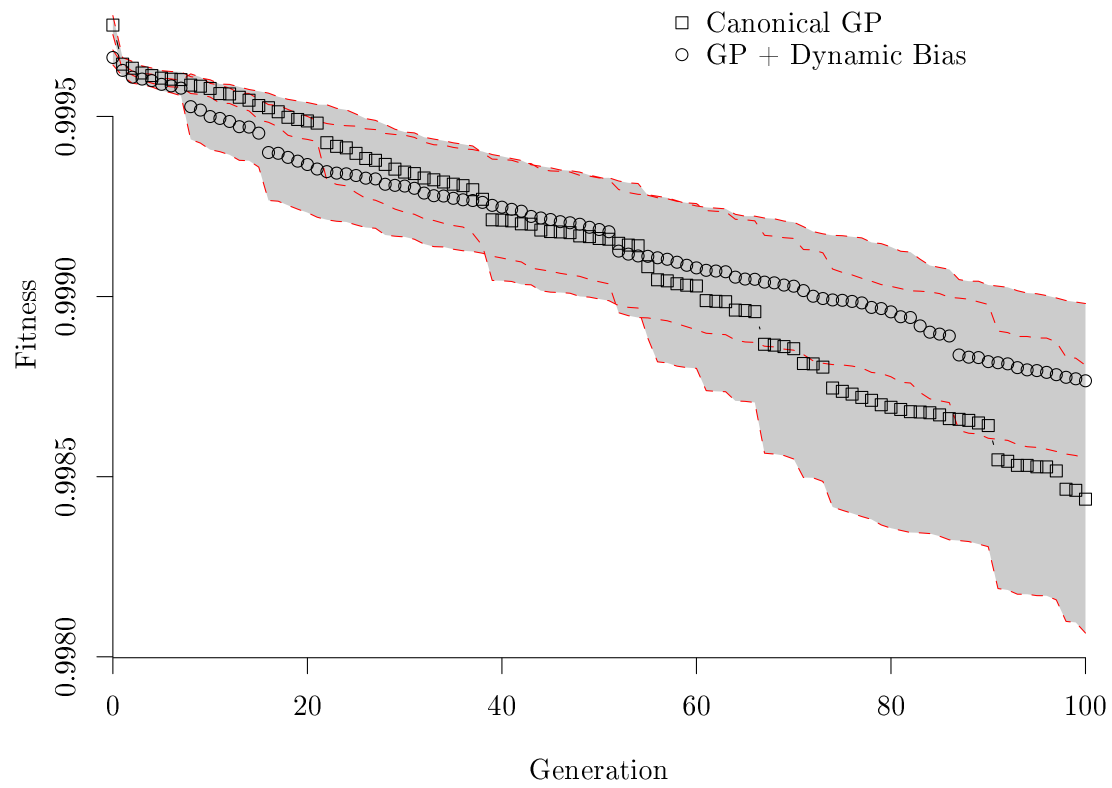


Fig. D.5: GP with Dynamic mutation-derived bias on Selection Sort

D.6 Selection Sort 2

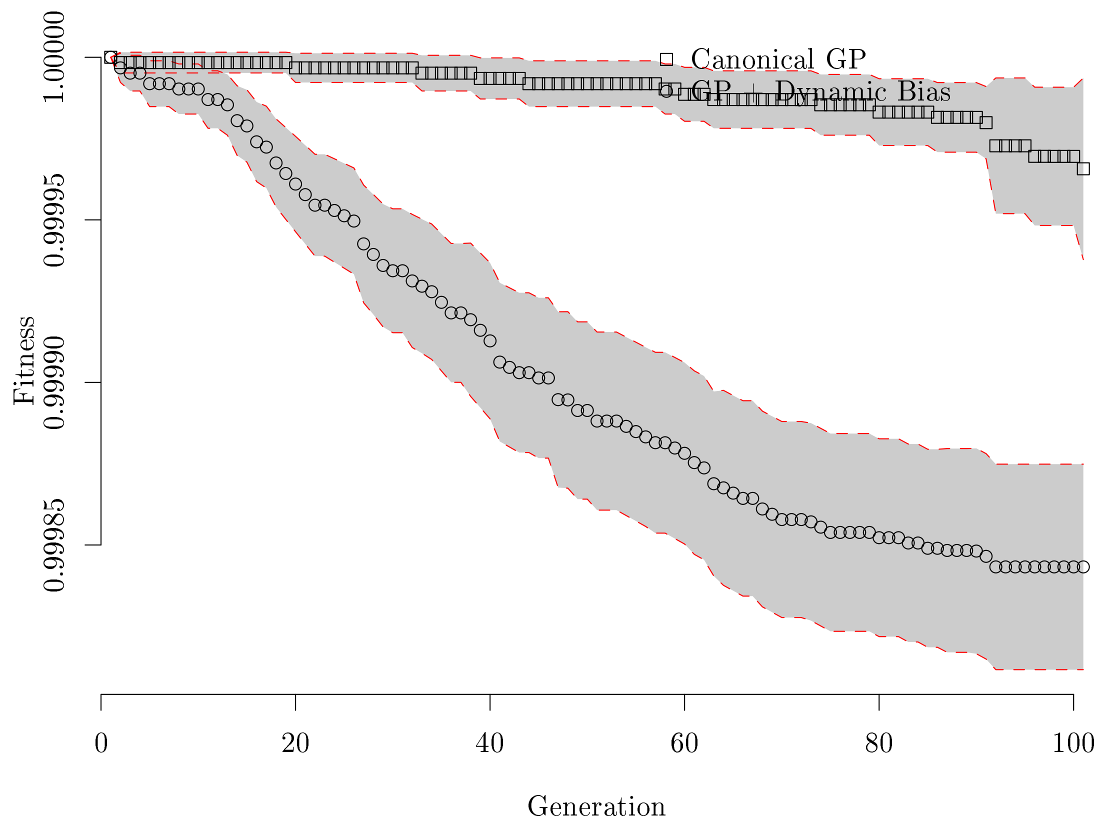


Fig. D.6: GP with Dynamic mutation-derived bias on Selection Sort 2

D.7 Shell Sort

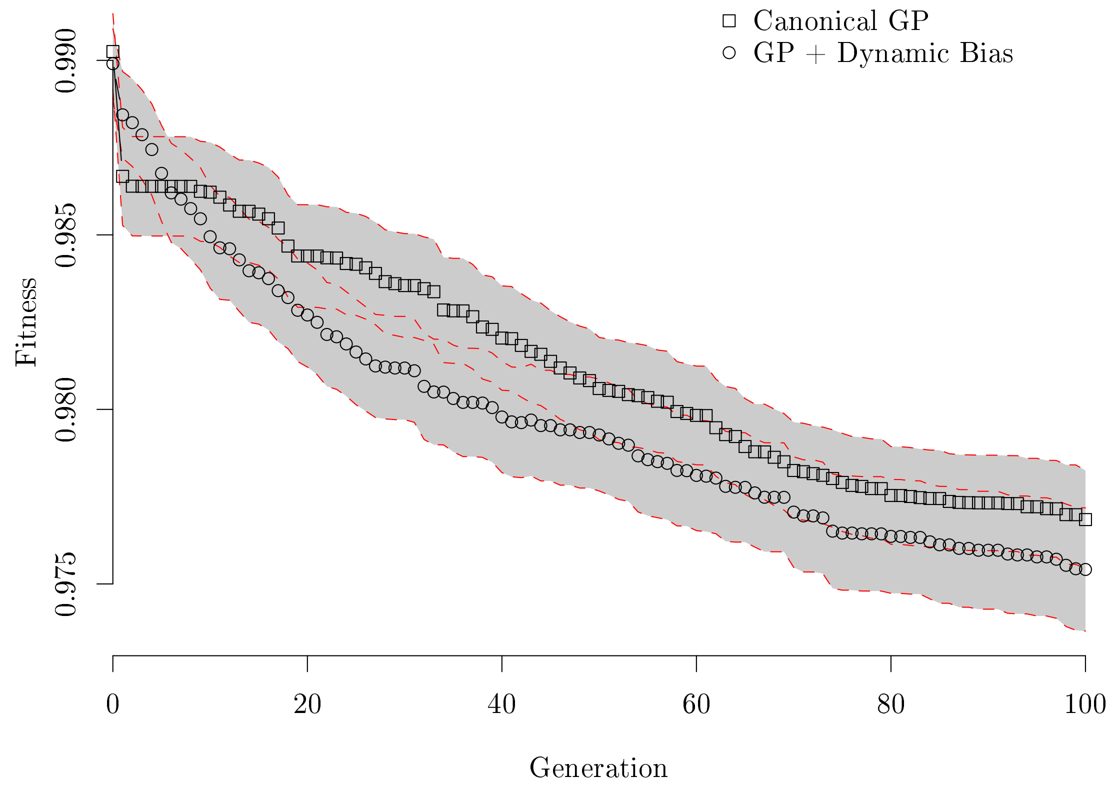


Fig. D.7: GP with Dynamic mutation-derived bias on Shell Sort

D.8 Quick Sort

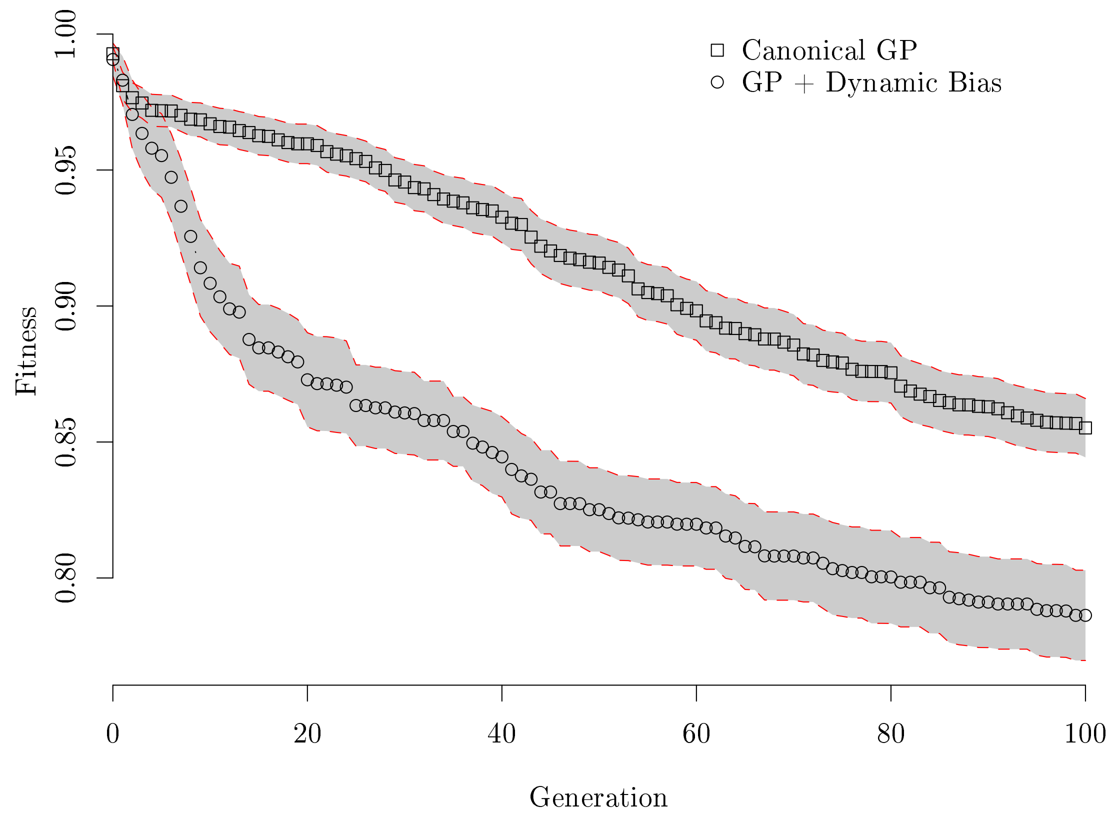


Fig. D.8: GP with Dynamic mutation-derived bias on Quick Sort

D.9 Cocktail Sort

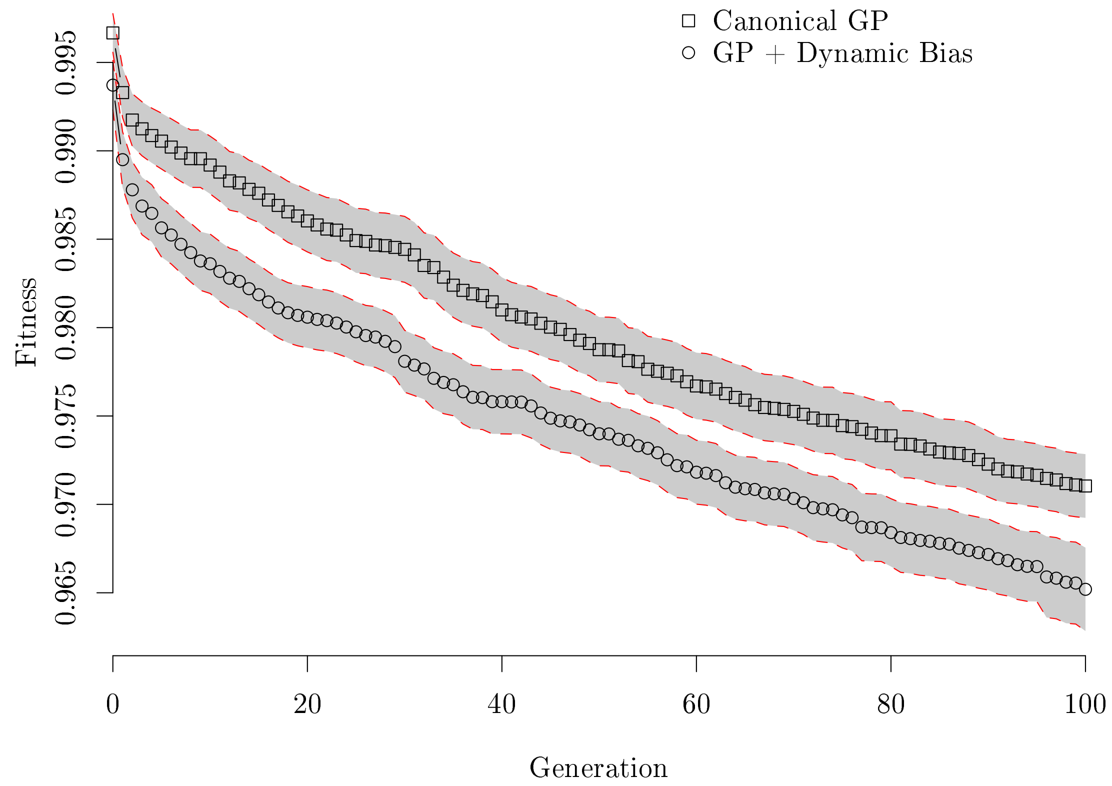


Fig. D.9: GP with Dynamic mutation-derived bias on Cocktail Sort

D.10 Huffman Codebook

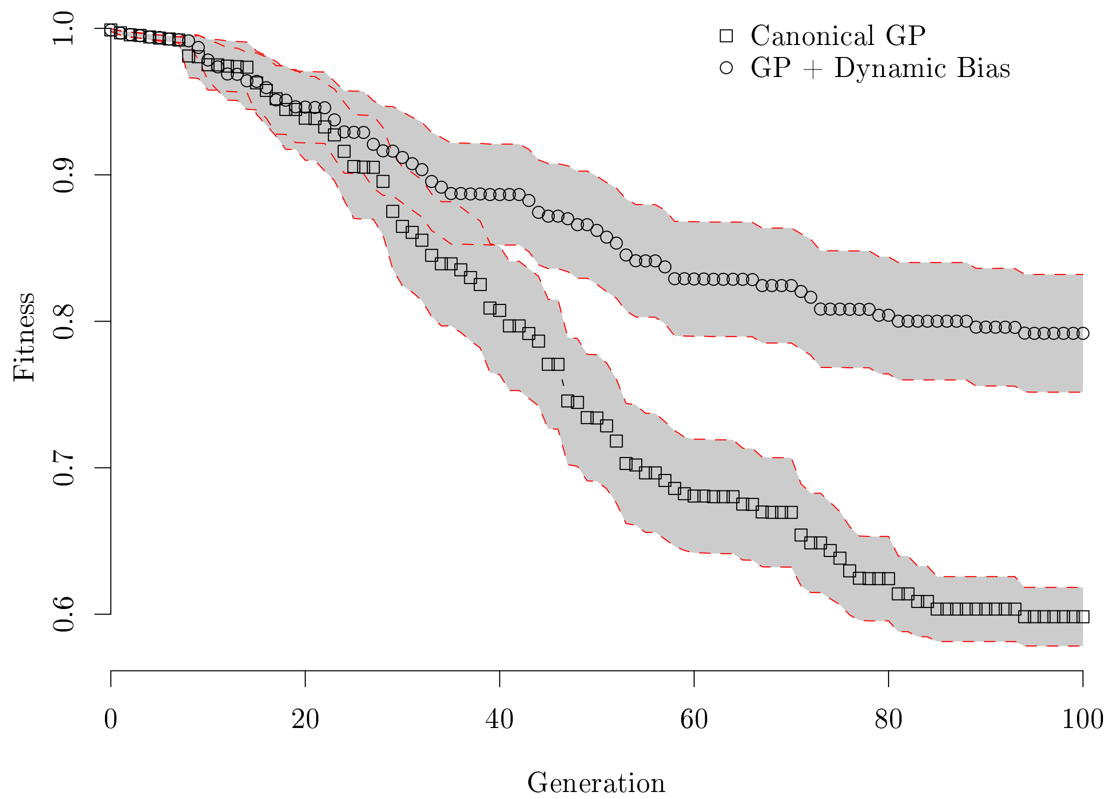


Fig. D.10: GP with Dynamic mutation-derived bias on Huffman Prefix Codebook