Kent Academic Repository

Beadle, Lawrence, Charles, John (2009) ***Semantic and structural analysis of genetic programming.*** **Doctor of Philosophy (PhD) thesis, Computing.**

Downloaded from

https://kar.kent.ac.uk/30599/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.22024/UniKent/01.02.30599

This document version

UNSPECIFIED

DOI for this version

Licence for this version

CC BY-NC-ND (Attribution-NonCommercial-NoDerivatives)

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Semantic and Structural Analysis of Genetic Programming

Lawrence Charles John Beadle

July 2009

University of Kent at Canterbury

A thesis submitted by Lawrence Charles John Beadle to the University of Kent for the degree of Doctor of Philosophy in Computer Science, in July 2009.

FZ20017

Contents

List of Figures iii

List of Tables v

1 Introduction 1

1.1 Introduction 1

1.2 Contributions 2

1.3 Thesis Overview 4

2 Search and Genetic Programming 6

2.1 Search and Evolutionary Algorithms 6

2.2 Genetic Programming 9

2.3 Representation of Candidate Solutions 11

2.4 Initialising Programs 15

2.5 Selection 19

2.6 The Crossover Operator 22

2.7 The Mutation Operator 25

2.8 Other Genetic Programming Practises 27

3 Current Issues in Genetic Programming 28

3.1 Diversity in Genetic Programming 29

3.2 Bloat and Methods to Control Bloat 32

3.3 Building Blocks and Schema Theories 38

3.4 Summary 43

4 **Methodology** 44
4.1 A Test Problem Suite 44
4.2 General Genetic Programming Parameters 47
4.3 Modelling Behaviour 48

5 **Semantic Analysis of Initialisation in Genetic Programming** 57
5.1 Methods and Algorithms 58
5.2 Results 63
5.3 Discussion 81
5.4 Conclusions 86
5.5 Future Work 87

6 **Semantically Driven Operators** 88
6.1 Semantically Driven Crossover 89
6.2 Semantically Driven Mutation 106
6.3 Semantic Pruning 117
6.4 Intron Free Genetic Programming 125

7 **An Analysis of Program Structure in Genetic Programming** 133
7.1 Methodology 134
7.2 Profiling Program Structure 134
7.3 Program Structure and Problem Fitness 147
7.4 Discussion 156
7.5 Conclusions 158
7.6 Future Work 159

8 **Conclusions** 160
8.1 Contributions 160
8.2 Future Directions of Research 162

Bibliography 164

List of Figures

2.1 Example Fitness Landscape I 7
2.2 Example Fitness Landscape II 8
2.3 Example of a Tree Representation 12
2.4 Locality Example 14
2.5 Uniform Crossover Example 23

4.1 The Santa Fe Trail 46
4.2 Example of a Reduced Ordered Binary Decision Diagram 49

5.1 Enumeration of Unique Behaviours Present in Starting Populations. 65
5.2 Semantically Unique Programs Produced by Initialisation Algorithms. 66

6.1 Semantically Driven Crossover Rejections for Multiplexer Experiments 98
6.2 Semantically Driven Crossover Rejections for Even Parity Experiments 99
6.3 Semantically Driven Crossover Rejections for the Majority Experiments 100
6.4 Semantically Driven Crossover Rejections for Artificial Ant Experiment 101
6.5 Semantically Driven Crossover Rejections for Regression Experiments 102
6.6 Semantically Driven Mutation Rejections for the Multiplexer Experiments. . . . 111
6.7 Semantically Driven Mutation Rejections for the Even Parity Experiments. . . . 112
6.8 Semantically Driven Mutation Rejections for the Majority Experiments. 113
6.9 Semantically Driven Mutation Rejections for the Artificial Ant Experiment. . . . 114
6.10 Semantically Driven Mutation Rejections for the Regression Experiments. . . . 115
6.11 Pruned Program Length by Generation 121

7.1 Even Four Parity Shape Over Generations 135

7.2 Five Majority Shape Over Generations 136
7.3 Six Bit Multiplexer Shape Over Generations 137
7.4 Even Seven Parity Shape Over Generations 138
7.5 Nine Majority Shape Over Generations 139
7.6 Eleven Bit Multiplexer Shape Over Generations 140
7.7 Artificial Ant Shape Over Generations 141
7.8 Cubic Symbolic Regression Shape Over Generations 142
7.9 Quartic Symbolic Regression Shape Over Generations 143
7.10 Tree Shapes at Generation 50 145

List of Tables

5.1 Speed Comparison of Initialisation Algorithms. 63

5.2 Comparison of Unique Behaviours Produced by Initialisation Algorithms 67

5.3 Bias Results for Even Parity and Multiplexer Problems. 68

5.4 Bias Results for Artificial Ant, Symbolic Regression and Majority Problems. . . 71

5.5 Size and Shape Comparisons for Multiplexer and Even Parity Problems. 73

5.6 Size and Shape Results for Ant, Majority and Regression Problems. 75

5.7 Performance Results for Semantic Initialisation Algorithms. 78

5.8 Performance Results for Shape Comparison at Initialisation. 80

5.9 Average Program Length and Depth at Initialisation 82

6.1 Semantically Driven Crossover Performance Results (Uniform Version). 93

6.2 Semantically Driven Crossover Performance Results (Koza Version). 94

6.3 Program Length Results Using Semantically Driven Crossover. 95

6.4 Program Length Results Using Semantically Driven Crossover. 96

6.5 Semantically Driven Mutation Performance Results Table. 109

6.6 Program Length Results Using Semantically Driven Mutation. 110

6.7 Semantic Pruning Performance Results Table. 119

6.8 Program Length Results Using Semantic Pruning. 120

6.9 No Intron Performance Results Table. 127

6.10 No Intron Program Length Results Table. 128

6.11 Intron Free Rejection Rates. 129

7.1 Structural Change Measurements Results. 146

7.2 Structural Change Compared to Fitness Change Results. 148

7.3 Even Four Parity Shape and Structural Locality Comparison 150
7.4 Five Majority Shape and Structural Locality Comparison 151
7.5 Six Bit Multiplexer Shape and Structural Locality Comparison 151
7.6 Even Seven Parity Shape and Structural Locality Comparison 152
7.7 Nine Majority Shape and Structural Locality Comparison 152
7.8 Eleven Bit Multiplexer Shape and Structural Locality Comparison 153
7.9 Artificial Ant Shape and Structural Locality Comparison 153
7.10 Cubic Shape and Structural Locality Comparison 154
7.11 Quartic Shape and Structural Locality Comparison 154

List of Algorithms

5.1 Semantically Driven Initialisation 59
5.2 Hybridised Semantically Driven Initialisation 61
5.3 The MODFULL Algorithm with Additional WASHED Section. 62
6.1 Semantically Driven Crossover Algorithm 91
6.2 Semantically Driven Mutation Algorithm 108
6.3 Semantic Pruning Algorithm 118

Publications

Lawrence Beadle and Colin G Johnson (2008)

Semantically Driven Crossover in Genetic Programming

Proceedings of the IEEE World Congress on Computational Intelligence

Pages 111—116

IEEE Press

Lawrence Beadle and Colin G Johnson (2009)

Semantic Analysis of Program Initialisation in Genetic Programming

Genetic Programming and Evolvable Machines

Vol 10, Number 3

Pages 307—337

Springer

Lawrence Beadle and Colin G Johnson (2009)

Semantically Driven Mutation in Genetic Programming

Proceedings of the IEEE World Congress on Computational Intelligence

1336—1342

IEEE Press

Acknowledgements

There are several people I would like to thank for their support and guidance during my PhD. I would like to thank my tutor, Dr Colin Johnson, for guiding me through the PhD process. Colin's enthusiasm and seemingly always being open to new ideas has been an inspiration to me and have made three years of hard work, three years of enjoyable hard work. I would like to thank my panel, Dr Alex Freitas and Dr Andy King for their guidance during the PhD process. More specifically, I would like to thank Alex for taking the time to review papers and discuss ideas and Andy for always asking difficult questions. I would like to thank my office colleagues, Rob, Laurence, Ahmed and Tom, for helping me deal with the highs and lows of a PhD and providing me with entertainment and support as required. I would also like to thank Tom for his collaboration constructing the EpochX GP software. I would like to thank Sally Fincher for her invaluable discussions and organisation of so many transferable skills courses from which I have benefited. I would like to thank Vicky Phippen for taking the time to help proof read this thesis. I would like to thank my mum and dad for their never ending support, especially towards the completion of this thesis. Finally, I would like to thank Sarah for patiently always being there for me.

Abstract

Genetic programming (GP) is a subset of evolutionary computation where candidate solutions are evaluated through execution or interpreted execution. The candidate solutions generated by GP are in the form of computer programs, which are evolved to achieve a stated objective. Darwinian evolutionary theory inspires the processes that make up GP which include crossover, mutation and selection. During a GP run, crossover, mutation and selection are performed iteratively until a program that satisfies the stated objectives is produced or a certain number of time steps have elapsed.

The objectives of this thesis are to empirically analyse three different aspects of these evolved programs. These three aspects are diversity, efficient representation and the changing structure of programs during evolution. In addition to these analyses, novel algorithms are presented in order to test theories, improve the overall performance of GP and reduce program size.

This thesis makes three contributions to the field of GP. Firstly, a detailed analysis is performed of the process of initialisation (generating random programs to start evolution) using four novel algorithms to empirically evaluate specific traits of starting populations of programs. It is shown how two factors simultaneously effect how strong the performance of starting population will be after a GP run. Secondly, semantically based operators are applied during evolution to encourage behavioural diversity and reduce the size of programs by removing inefficient segments of code during evolution. It is demonstrated how these specialist operators can be effective individually and when combined in a series of experiments. Finally, the role of the structure of programs is considered during evolution under different evolutionary parameters considering different problem domains. This analysis reveals some interesting effects of evolution on program structure as well as offering evidence to support the success of the specialist operators.

Chapter 1

Introduction

Darwin's theory of evolution acts as an inspiration to the field of evolutionary computation within computer science. A relatively recent subset of evolutionary computation is that of genetic programming (Cramer [1985], Koza [1992], Poli et al. [2008]); a system designed to evolve process in the form of a population of computer programs. Human programmers can easily demonstrate that there is more than one program that can be used to encode a process; however, a human programmer will usually (or hopefully, depending on the level of ability) select an optimal way in which to encode the process in syntax. Since programs generated using genetic programming often only check the accuracy of a process, the latent intelligence and selection of appropriate programming methods of a human programmer are lost in the construction of an optimal process. This leads to several undesirable side effects when using genetic programming to evolve programs. The objective of this thesis is to examine, theoretically and empirically, the link between programming code and process evolved using genetic programming, in order to understand and combat some of the side effects of genetic programming.

1.1 Introduction

Genetic programming (GP) is a process whereby a population of programs are initialised and evaluated against an objective. The programs that are most fit are selected and act as parent programs during crossover and mutation operations, which are then used to generate a new population. The new population is in turn evaluated against the objective, and the fitter

individuals are selected for the genetic operations once more. In the form of GP discussed in this thesis, this process continues until a termination criterion is met; which could be either a program achieving total accuracy, or a certain number of iterative steps of the evolution process.

Though GP is a powerful tool in knowledge discovery producing patentable results (Koza et al. [2003]), it is not without weaknesses which impact the scalability and efficiency of GP. Three prominent issues are examined in detail in this thesis. The first is that of program growth or bloat, which is an increase in program size for no related increase in fitness (Tackett [1994], Banzhaf et al. [1998], Soule and Foster [1998], Luke [2003], Langdon and Poli [1997], Dignum and Poli [2007]). Secondly, there is the evaluation of the role or non role of introns in evolution, which are areas of code that do not contribute to fitness (Nordin et al. [1995], Soule and Foster [1997]). Finally, the one to many relationship between process (or behaviour) and the many programs that can be used to represent a particular behaviour. This analysis reaches into both issues in diversity (Gustafson [2004]) and representation theory (Rothlauf [2006]).

This thesis presents methods to address bloat, remove introns, formally evaluate the behaviours of programs and increase diversity in GP based on theoretically motivated techniques to model the behaviours of programs evolved using GP. Initialisation, crossover, mutation and the reduction of evolved programs are considered in empirical evaluation to test how methods based on increased links between syntax and behaviour can impact at each stage of the GP process.

1.2 Contributions

This thesis presents contributions in three areas to the field of genetic programming.

Firstly, four algorithms are presented to provide a theoretical analysis of the complexities of program initialisation. The motivation for this research is a study of a traditional initialisation algorithm. This study highlights limitations of the traditional initialisation method, more specifically focusing on the issues of repeated sampling of particular program behaviours and how program structure at initialisation influences GP performance. The four algorithms are developed to test different aspects of program behaviours and structures in an initial po-

pulation. An empirical study is performed comparing these algorithms to the performance of a traditional and widely accepted technique in order to assess how the issues of bias and program structure affect initialisation. This research demonstrates the difficulty of developing an initialisation algorithm that is consistently effective in terms of performance over different domains and highlights areas for future research.

Secondly, three algorithms are presented which leverage the ability to assess the behaviour of a program and apply this knowledge during GP search. The first of these algorithms is a modification to the crossover operation to enforce the semantic novelty of the child programs when compared to the parent programs. The second algorithm is a modification to the mutation search operation in order to enforce the semantic novelty of the mutated program in comparison to its original state. Finally, a semantic pruning algorithm is developed in order to reduce programs to a minimalistic syntactic form during evolution. The three algorithms are compared to traditional techniques in an empirical study and their relative merits and disadvantages discussed. As a final experiment, the three algorithms are combined to evaluate the cumulative effect which can both influence performance and program size simultaneously.

Thirdly, based on the results using the algorithms for initialisation, crossover, mutation and pruning, an analysis of structure of program during evolution is performed. The aim of this analysis is to highlight previously unseen traits in program structure during evolution and using the different algorithms which have been discussed in the previous chapters. The results show statistical differences in the shapes of programs produced by different algorithms over several different problems and a discussion of how this differences may help or hinder evolution is presented. The level of structural change required in order to find a solution is compared for different problems and show that some problems require a larger amount of structural change in order to find a solution. The level of structural change is compared to the level of fitness change during evolution and the resulting measure of structural locality is used to compare the level at which different problems are dependant on structural change in order to achieve optimal solutions.

Finally, during the course of this thesis, GP analysis software known as Epoch X (Beadle and Castle [2007]) has been developed and made available on the World Wide Web. This software is open source and includes the specialist methods presented in this thesis for other

users to experiment with.

1.3 Thesis Overview

Chapter 2 introduces search and evolutionary algorithms and forms the first part of the literature review. The traditional GP process is fully explained, including detailed descriptions of initialisation, selection, crossover and mutation with reference to more recent developments in these areas.

Chapter 3 discusses current issues in GP and forms the second part of the literature review, evaluating three relevant theoretical aspects of GP. The diversity of programs, program bloat and building blocks and schema theories are discussed in detail with reference to recent literature.

Chapter 4 presents the general methodologies used in this thesis. A test problem suite is defined including nine benchmark GP experiments with a set of general parameters for the empirical work presented in this thesis. Crucially, this chapter sets out methods to canonically represent program behaviour, which provides the functionality to test the proposed algorithms in this thesis. The test problems suite is composed of problems from three common domains (Boolean, symbolic regression and artificial ant) and canonical representation methods for each of these domains are outlined in this chapter.

Chapter 5 presents an empirically based evaluation of theory relating to the process of initialisation in genetic programming. Firstly, an existing and popular initialisation technique is examined in detail in terms of the semantic diversity of the programs produced and the structure of the programs produced. Secondly, based on the analysis of the traditional technique, four new algorithms are presented to test different levels of semantic diversity and changes to program structure during initialisation. The results presented demonstrate that both semantic diversity and program structure can influence evolution using GP.

Chapter 6 presents an empirical evaluation of semantically driven operators. Firstly, semantic comparison is combined with the crossover operation in order to check child programs are semantically distinct in comparison to their parent programs. Empirical analysis demonstrates that semantically driven crossover is beneficial to GP in almost all cases. Secondly, semantic comparison is combined with the mutation operator to enforce the creation

of semantically distinct mutated programs in comparison to the original program. Empirical analysis shows that semantically driven mutation is beneficial in the majority of experiments. Thirdly, programs are pruned through being modelled behaviourally and then using the behaviour to produce a minimalistic syntax. Empirical analysis shows this technique to be effective when performing code reduction, yet not always desirable when considering the performance of a GP run. Finally, all of the new algorithms are combined and the cumulative effects evaluated. Empirical analysis shows the technique to be beneficial in the majority of experiments as well as providing insight into existing theories of the GP search mechanism.

Chapter 7 evaluates the structure of programs during evolution. Firstly, the way in which program structure changes over time during evolution is studied to search for clues as to whether particular structure types can be linked with good GP performance. Secondly, the amount of structural change required for the different experiments in the problem suite is calculated and compared, indicating that some experiments require more structural change than others to reach their solutions. Thirdly, structural change is compared to fitness change in order to evaluate the level of correlation using a range of different GP parameters. Analysis reveals that program structure during evolution can be effected by a variety of GP settings indicating that some setting may perform better for different types of problem.

Chapter 8 presents general conclusions and discusses the contributions made by this thesis. The conclusions and contributions are based around the three contribution areas of program initialisation, search operators and the analysis of program structure. Several suggestions for future work are presented based upon the three contribution areas.

Chapter 2

Search and Genetic Programming

The aims of this chapter are to introduce the field of genetic programming and discuss current issues within the field. Section 2.1 establishes the place of genetic programming in the context of the development of search algorithms. Section 2.2 introduces genetic programming and describes the genetic programming process. Sections 2.3 — 2.7 provide detailed descriptions of common techniques available to perform the different operations within genetic programming.

2.1 Search and Evolutionary Algorithms

There are numerous problems which require search techniques in order to find an optimal solution. Different search techniques have evolved out of an increasing level of problem difficulty. Whilst an exact measurement of problem difficulty remains elusive, it is relatively easy to illustrate different example cases of some typical factors that will effect problem difficulty.

The first issue is that of visualisation. Consider the function $z = f(x, y)$. Given that z is dependent on two variables, it would be possible to visualise the results of using different values of x and y using three dimensional graphics. This is not normally the case for such problems, but for the purpose of this discussion is beneficial.

If x and y are both considered as discrete variables operating over a small range, it may be appropriate to run an exhaustive search that would calculate every value of z for every combination of x and y. Whilst this strategy has its place, it may be the case that x and y

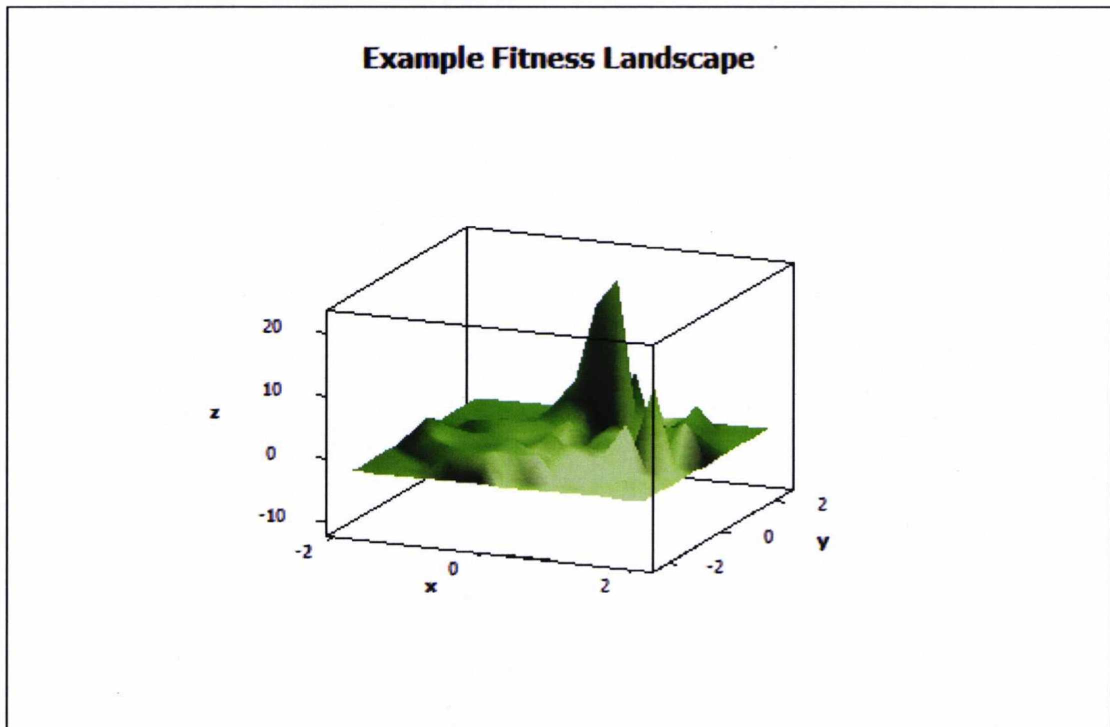


Figure 2.1: Example Fitness Landscape I

are continuous and operate over a large range; resulting in the calculation of every possible value of z being excessively time consuming or impossible.

The next step would be to consider the nature of the fitness landscape. The three dimensional representation of the fitness landscape could look something like figure 2.1. On the assumption that the highest peak represents the best fitness of a candidate solution, in this situation, it is possible to identify peaks and troughs.

The next logical step would then be to develop a search algorithm to move uphill. A simple hill climbing algorithm would assess the fitness values (in this case z) of the area around its current x, y location and then relocate to the highest neighbouring region (or most fit) and perform the calculation again. Once the hill climbing algorithm reaches the highest point and all its neighbours are lower values, the algorithm terminates. Whilst this would be ideal in a situation where only one hill is present, there are numerous peaks and troughs, as demonstrated in figure 2.1. In this situation, the hill climbing algorithm may become trapped on one of the smaller peaks (known as *local optima*) and unable to reach the tallest peak (*global optimum*).

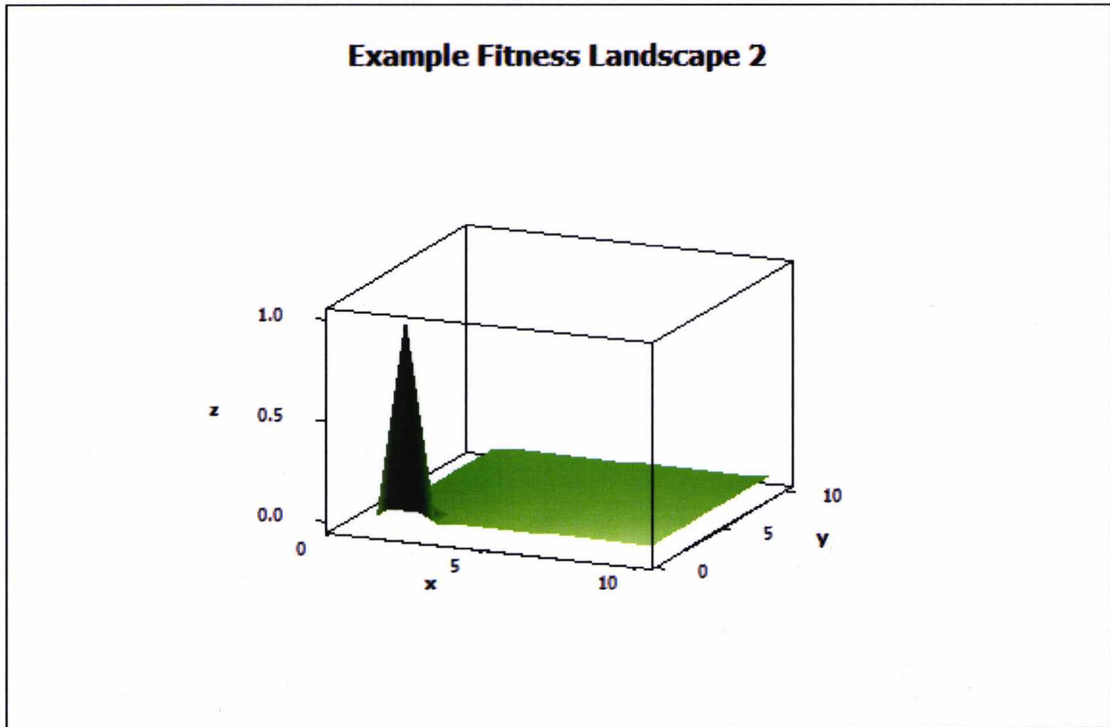


Figure 2.2: Example Fitness Landscape II

In addition to premature convergence at a local optimum, there are two other major problems. Firstly, consider figure 2.2 where most of the landscape is flat. In this situation, the hill climbing agent (or search agent) would not know which way to move, unless the agent started at a location by the peak. This kind of landscape is known as *needle in a haystack*, because the search agent is searching for something very small in a field of fitness neutrality. The second problem would be if the single peak was surrounded by a trough, since this would render the peak inaccessible to the search agent.

Situations such as *needle in a haystack* and inaccessible peaks inspired more complex search algorithms in order to be able to cope with the increased complexity of the fitness landscape. Several strategies have been proposed; one notable example being *simulated annealing* (Kirkpatrick S. [1983]), in which the search agent can move down hill (to try to escape a local optima) with a certain probability based on a temperature function.

In order to build more effective search algorithms, one of the next steps was developing evolutionary algorithms inspired by nature's selection processes. Evolutionary algorithms had the benefit of using multiple interacting search agents, which could evaluate a popula-

tion of individuals, keep the most fit individuals, use these individuals to make children subject to a crossover or mutation operation and re-evaluate the population iteratively. These evolutionary algorithms gave researchers the ability to search more complex search spaces with more variables, whilst being able to reduce the chance of getting trapped in a local optimum compared to simple hill climbing.

As evolutionary algorithms progressed, a class of algorithms known as *genetic algorithms* was developed by Holland [1992], which included crossover and mutation operations inspired by genetics and allowed researchers to evolve optimal solutions to problems. The next step was to evolve optimal processes, in the form of programs (rather than optimal solutions), which could be used to generate optimal solutions.

2.2 Genetic Programming

GP can be described as a subset of evolutionary algorithms which has the unique requirement that their candidate solutions are executed or subjected to interpreted execution in order to assess their fitness value. The difference between genetic algorithms and genetic programming could be described as follows: an evolved solution using a genetic algorithm describes an optimal solution to a problem, whereas a candidate solution evolved by genetic programming describes a process (in the form of a program) which, through application, results in a fit, possibly optimal, solution. In a more practical sense, the key differences between genetic algorithms and GP is that programs in GP are represented in such a way that they may be executed (commonly a tree form), whereas candidate solutions in genetic algorithms are represented as a list of input values which would require interpretation by a program to assess fitness.

Genetic programming first appeared as a progression from genetic algorithms (Holland [1992]). Whilst Cramer [1985] was the first to develop GP in the commonly applied tree format, GP was popularised by Koza [1992], who applied the process to several practical problems. Since then GP has developed substantially, in several cases providing patentable solutions (Koza et al. [2003]) to real world problems.

The genetic programming process features several different processes executed in sequence. The basic process is described as follows:

1. Decide on a representation format, function and terminal sets and a fitness function for the proposed problem.
2. Initialise a starting population of candidate solutions.
3. Assess the fitness of the candidate solutions.
4. Select a number of highly fit candidate solutions.
5. If a termination condition has been reached, exit. If not, continue to step 6.
6. Use crossover and/or mutation operations on selected candidate solutions to create a new population of candidate solutions.
7. Return to step 3.

During a GP run, the GP practitioner will decide on the representation structure of the candidate solutions and choose problem specific function and terminal sets. To start the genetic program, a seed population of randomly created candidate solutions is created to serve as a starting point for the algorithm. The fitness function is then used to provide a measure of fitness for each of the candidate solutions in the population. The termination condition in step five is true when either a candidate solution has reached full score or a set number of new candidate solutions has been generated using genetic operations and assessed by the fitness function. The termination could be one, or both of the mentioned criteria.

Using the measure of fitness, it is more likely that the best of the candidate solutions is selected (depending on selection method used) and used by the genetic operations (crossover and/or mutation). GP practitioners may choose to run crossover or mutation either individually or simultaneously. The crossover operation produces more candidate solutions which are added to the population, whilst the mutation operation causes changes to programs in the population. The candidate solutions are assessed for fitness and against the termination criterion and the cycle repeats until the termination criterion is met.

Steps one to seven contain a number of key operations in the genetic programming process including; the representation of candidate solutions (section 2.3), initialisation of candidate solutions (section 2.4), selection mechanisms (section 2.5), crossover (section 2.6) and mutation operations (section 2.7). These operations are described in greater detail in the respective sections.

2.3 Representation of Candidate Solutions

The are two major aspects to consider when deciding on the representation of candidate solutions. Firstly, the GP practitioner must decide on the choice of functions and terminals to represent the functionality, inputs and constants required to solve the problem. Secondly, the GP practitioner must decide upon the way in which the executable structure is represented.

Functions and Terminals

One of the first choices a GP practitioner is forced to make is to choose a function and terminal set. A function set is a number of operations that the user specifies as required for the genetic programming run. An example of a function would be logic operations such as AND a b , OR a b , IF a b c where a, b and c represent substructures which are either functions or terminals. *Arity* is a term used to describe how many inputs are considered by a function. In the case of the IF function, the arity would be three. A terminal set will include both inputs and/or constants. For example, when considering the symbolic regression domain, the GP practitioner may choose the terminal set {X, Y, 0, 1, 2}. All terminals will have an arity of zero.

According to Koza [1992], function and terminal sets must have *closure*, defined as the ability of each member of the function set to handle every possible input it might receive gracefully. The random combinations of programs created using GP means that the closure property will be tested. A common example would be using the division function in the symbolic regression domain. Division by zero results in infinity, and situations such as this need to be handled gracefully to facilitate the random nature of GP runs and prevent errors.

A second aspect of the function and terminal sets which is essential to solving problems using GP is *sufficiency*. Sufficiency requires that the function and terminal sets must have enough expressive power to be able to model the perfect solution. This is a difficult balance since using too many functions and terminals results in an enlarged search space, which in turn makes the solution harder for GP to attain. Minimising the search by using just enough functions is known as *parsimony*. This is desirable because the use of less functions will result in a smaller search, which should make the solution easier to attain.

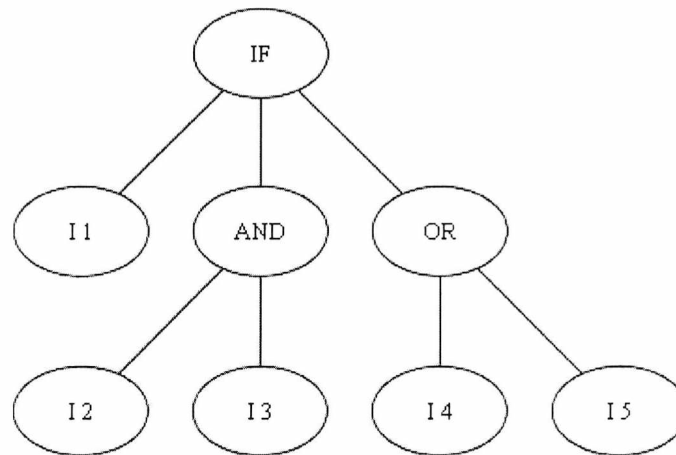


Figure 2.3: Example of the tree representation for the candidate solution IF I1 (AND I2 I3) (OR I4 I5). I's refer to different Boolean inputs.

Representation Structure

There are several different structural representation types which have been used in GP. The most common representation structure is the S-expression tree structure, which used in the context of genetic programming, was first described by Cramer [1985] and popularised by Koza [1992].

An example of the tree representation is set out in figure 2.3. The tree representation uses a system of nodes and edges to connect the different inputs (either terminals or sub tree structures) to the functions, allowing for the construction of more complex programs. Execution of a tree structure involves a recursive evaluation of each child node in relation to its parent node. This has the effect of resolving the tree in a bottom up fashion. Whilst other representation methods are described for completeness, the tree representation is the focus of the work in this thesis because it is the most commonly used representation.

One of the alternatives to the tree structure is linear representation (Banzhaf et al. [1998]). In this case, a sequential list of instructions (containing functions) is used to apply functions to members of the terminal set. During execution the instruction list is executed and a result returned after the last instruction.

A second alternative is grammar based genetic programming developed by Whigham [1995a]. This differs from both linear and tree based genetic programming in that all functions and terminals are considered to be terminal nodes. In Whigham's examples, two non

terminals are used; one to describe expressions and the other, actual terminals. The grammars are used to construct derivation trees upon which crossover takes place. Whigham has used grammatically based GP in order to demonstrate the effects of bias of the representation (Whigham [1995b]).

An entirely different approach to representation in GP is the graph structure. Whilst tree, linear and grammatical structures can be described by graphs and edges, they have very particular ordering and execution constraints. In contrast to this, the PADO (Parallel Algorithm Discovery and Orchestration) system, developed by Teller and Veloso [1996], allows evolution to decide the structure of the edges connecting the nodes. This kind of system permits functionality that is difficult to produce and utilise in other structures such as looping and recursion[1].

A fourth approach to representation, which is known as grammatical evolution (developed by Ryan et al. [1998]) works in a slightly different way to traditional GP at the representation level. Here, programs are evolved as a list of numbers, which is decoded using a grammar into a program tree. Crossover takes place on the list of numbers which causes a substantially different effect on the program tree in comparison to traditional GP. Using grammatical evolution, trees are effectively cut across one or more branches as crossover occurs rather than the exchange of one sub tree as with traditional GP.

According to Rothlauf [2006], there are three key theories to consider in relation to representation for evolutionary algorithms; these are redundancy, building block scaling and locality. These three theories are based on the distinction between the *genotype* and the *phenotype*. In the case of GP, the genotype refers to the actual representation of syntax in the tree and the phenotype refers to the behaviour of the program during execution. This is an important distinction in the context of this work as chapter 4 presents novel ways to canonically represent the behaviour of programs (the phenotype) which had not previously been available to GP practitioners.

Considering redundancy for the GP paradigm, in this case more than one genotype can map to a phenotype. This is a common occurrence in GP and numerous results to support this are presented in section 5.2.2. If two programs result in the same phenotype, then GP is duplicating behavioural search steps which reduces the search capability of GP. A GP

[1] Looping and recursion have been described as difficult (Li and Ciesielski [2005]) as it is difficult to be sure of a termination situation for the program containing either function when it is executed for fitness assessment.

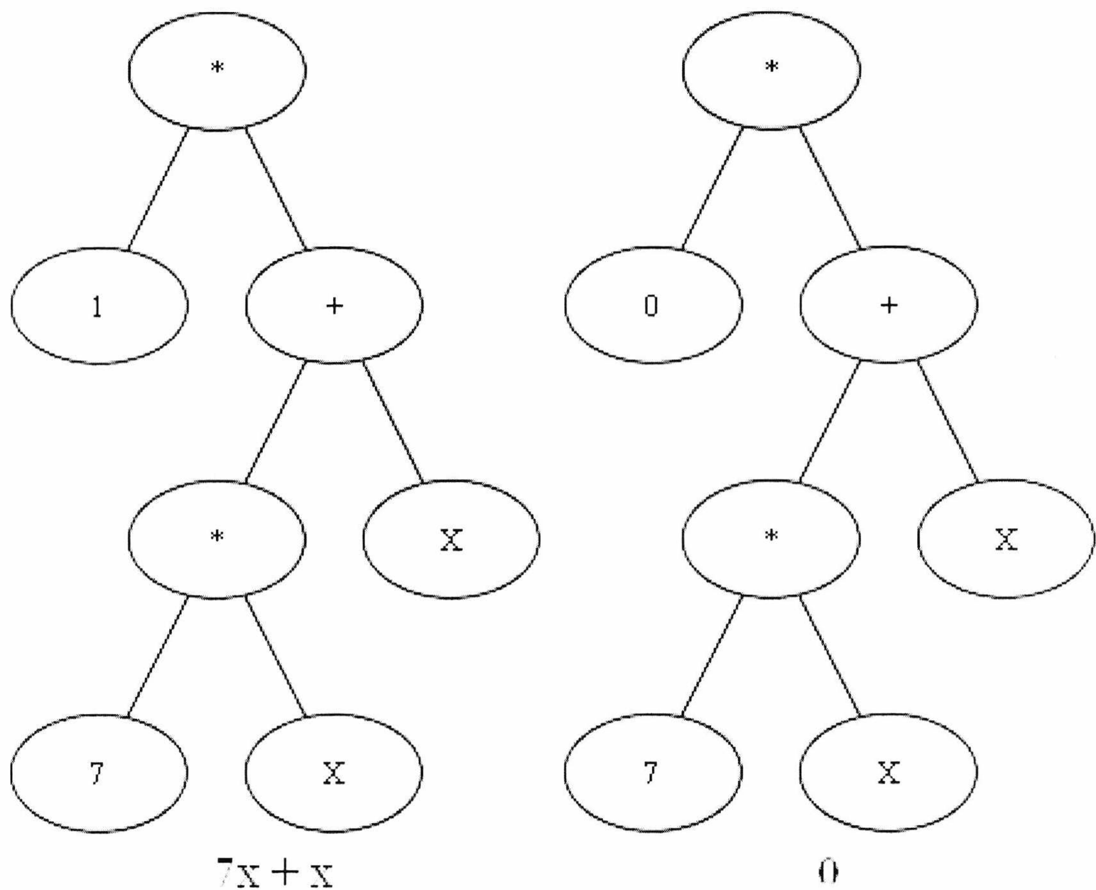


Figure 2.4: Example of Locality. The program on the left receives a one node modification changing it to be the program on the right. The resultant formulae are quoted underneath the program tree.

run has a fixed population size and usually a limited run time (or number of generations), therefore, behavioural duplication reduces the search potential during a GP run which may result in a loss of performance. Sections 6.1 and 6.2 use modifications to the crossover and mutation operators respectively to try to reduce redundant search.

Building block scaling refers to the level at which the components of the genotype contribute to the overall fitness. In the event that all parts of the genotype contribute equally, the scaling is considered to be uniform. In the GP context, this scaling is not uniform since very small changes in a genotype can result in large changes in fitness and vice versa (consider for example a negation function). This further complicates the matter of understanding the relationship between the genotypic space and the phenotypic space.

Finally, locality refers to the situation in which small changes in the genotype map to small changes in the phenotype. In GP, locality is considered to be low because it is easy to demonstrate that small changes in syntax can result in relatively large changes to the phenotype. This is demonstrated in figure 2.4, a one node change of 1 to 0 (left most node at depth 1) results in a completely different behaviour or phenotype.

The representation of programs in GP adds to the complexity of problems due to the fact that GP features genotypic redundancy (in some cases in large quantities, see section 5.2.2), exponentially scaled building blocks and low locality. According to Rothlauf [2006] these three factors will increase problem complexity. This indicates that GP not only has to navigate the undulating search space, but also address representational inefficiencies to find a global solution.

2.4 Initialising Programs

Thinking back to figures 2.1 and 2.2 showing the two fitness landscapes, an ideal initialisation algorithm should be able to distribute the different search agents (candidate solutions) across the fitness landscape in order to give GP the best possible chance of finding the global optimum. In addition to this, the initialisation algorithm needs to be relatively quick to execute as it will be run multiple times in different runs for statistical averaging and comparison.

A general solution to this is to use an algorithm capable of randomly constructing program trees in order to get a good distribution of programs across the search space. The size of the initial programs needs to be limited so the initialisation can happen quickly. Due to the random nature of the creation of the trees, there would also have to be checking to prevent duplicate trees being inserted into the initial population. Several algorithms following this general theme have been constructed and are summarised in the following paragraphs.

The most popular method for population initialisation is the *Ramped Half and Half* (RHH) method. It was introduced by Koza [1992], who set out three methods for creating a diverse starting population: GROW, FULL and RHH. Koza elected to use the RHH technique for the majority of his experiments after conducting several experiments comparing FULL and GROW to the RHH because "*...the ramped half-and-half method creates trees having a wide variety of sizes and shapes.*" Koza [1992, p93]. Koza also recognised that whilst the RHH

method creates a variety of programs, there was the possibility that programs could be duplicated. Therefore, he added syntactic duplicate checking of the programs to ensure the syntactic diversity of the starting population.

The RHH algorithm is comprised of the FULL and GROW algorithms. To assemble a program using the FULL algorithm, all branches of the program tree are filled to a predefined maximum depth parameter. Whilst the branches are shallower than the maximum depth, function nodes are selected uniformly and added to the tree. When the maximum depth of the tree is reached, terminal nodes are selected uniformly.

The GROW algorithm is different from FULL in that the branches are filled randomly with a maximum limit of the predefined maximum depth. Starting at the top of the program tree, each node is selected randomly (following a uniform distribution) from all of the function and terminals available to the problem. If a node is a function, all of the child nodes are subsequently selected in the same way. As noted by Luke [2000a], GROW is particularly susceptible to a poor choice of function and terminal sets. In the situation where more functions are present than terminals, GROW will tend towards at full tree sized at the predefined depth limit.

The RHH algorithm is a combination of GROW and FULL over a predefined maximum depth range which is typically 2-6. The first 20% of the population is generated at depth 2. 10% is generated using FULL and 10% is generated using GROW. Once this process is complete, the depth limit is increased for the next 20% of the programs and again 10% is allocated to GROW and 10% is allocated to FULL for program creation. This process is repeated for each 20% of the population. Whilst the RHH algorithm will prevent the duplication of initialised programs, no consideration is made for bias at either the genotypic or phenotypic level.

A *language bias*, or genotypic bias (defined by Whigham [1995b] as "*...bias is the set of all factors that influence the form of each program*") is present when there is a bias in the choice of items from the function or terminal sets. Whigham [1995a,b, 1996] analysed such a bias and its effects on grammatically based genetic programming by conducting experiments that explicitly added segment(s) of code to a program in the population. For example, in one experiment he biased *if* statements so that the condition could only be a specific terminal. This narrowed the exploration of the search space and increased the probability of

finding an ideal solution by artificially including segments of a known perfect solution. This demonstrates that specific language bias, whilst beneficial, could also severely compromise the ability of GP to find a solution to a problem with the highest fitness.

In order to address the concept of language bias Iba [1995] devised an approximately uniform tree generation method (RAND_TREE). In parallel to Iba's efforts, Bohm and Geyer-Schulz [1996] independently devised a method called *Exact Uniform Initialisation* based on statistical theory with the same objective in mind. Though both of these papers report slight improvements in the success of GP runs compared to RHH, only a small number of examples were studied, which left their results open to issues of problem-dependence (as noted by Bohm & Geyer-Schultz). In turn, this issue was addressed by Luke and Panait [2001], who conducted a more comprehensive survey and comparison of population initialisation methods. This survey concluded that there was no significant statistical difference in performance between the RHH and the uniform initialisation methods.

Langdon [2000] developed ramped uniform initialisation as part of experiments to control code bloat through changing the bias in the distribution of syntax. The algorithm is similar to Iba's and focuses upon a method of distributing syntax, rather than controlling the semantics of programs. Another potential approach for resolving this type of question would be to ask whether the RHH and uniform creation methods create a similar behavioural (or phenotypic) bias. Low locality and high redundancy may prevent a uniform distribution of syntax (or genotypes) providing a uniform distribution of behaviours (or phenotypes). This may explain why a theoretically grounded uniform initialisation method cannot improve results when compared to an ad hoc method such as RHH.

Later results by Looks [2007] (covering four Boolean domain problems), Beadle and Johnson [2009a] (covering more Boolean problems and the artificial ant domain) and the results presented in chapter 5 (which include the symbolic regression domain) demonstrate that redundancy is a key issue at initialisation. Whilst results are presented using semantically driven algorithms and semantic heuristics at initialisation show mixed results in terms of GP performance, possibly indicating a change in semantic diversity at initialisation, an exact analysis of locality at initialisation would be an interesting avenue for future research.

Despite the risk of imposing bias in a starting population, it is still a requirement that the GP practitioner has some control over the size of the programs produced (measured in depth

or length) in order to prevent excessively large programs being produced. Producing a starting population should take a relatively short amount of time, allowing for the fact that GP runs need to be executed many times to provide some degree of consistency in the results. To address this issue, Chellapilla [1997] devised an algorithm called RANDOMBRANCH which utilised a specified length rather than depth and produced approximately uniform programs. One problem with this algorithm (highlighted by Luke and Panait [2001]) is that because the RANDOMBRANCH, algorithm divides up the branch depths evenly, there are many trees that this initialisation method cannot produce. This would result in a language (or genotypic) bias in the starting population; however, the exact effect on (or phenotypic) behavioural bias remains unstudied.

A later effort by Luke [2000a] addressed the related issue of control over program initialisation. This resulted in two *Probabilistic Tree Creation* algorithms (PTC) of which there are two types, known as PTC1 and PTC2. These algorithms differed from those previously mentioned in that they allowed more user control. PTC1 allowed the user to provide the probability of appearance of individual functions as well as defining an average size of the initial programs. However, this method does not give the user any control of the variance of these programs. PTC2 addresses this issue by allowing the user to set a probability distribution of tree sizes which gives control over the variance in tree sizes. In comparison to the uniform based algorithms, PTC1 and PTC2 are simpler to implement and provide the user with much more control over the size and variation of programs in the initial population. In a similar way to Whigham's work, PTC1 and PTC2 give the user the ability to bias initial populations in such a way that may or may not focus the starting population more towards the global optimum. The risk with this is that if the wrong kind of bias is used, then the exploration could be steered away from the global optimum.

The latest works on initialisation are based on the use of semantics (Looks [2007], Beadle and Johnson [2009a], Jackson [2009]). Whilst there appears to be general consensus that semantic initialisation can improve performance on a selection of problems, it also appears that semantic initialisation performs worse on other problems. Though Looks [2007], Jackson [2009] report positive results using semantics to enhance initialisation, Beadle and Johnson [2009a] use a larger experiment suite and demonstrate that behavioural diversity and program structure affect the performance of GP. This topic is discussed in greater detail in

chapter 5.

2.5 Selection

The purpose of selection is to provide a consistent process which makes it more likely that the best candidate solutions will be selected as parents for the next round of genetic operations. This process involves two parts. Firstly, each of the programs considered by selection has its fitness score evaluated. Secondly, a selection method is applied to consistently choose the candidate solutions with the best fitness scores.

Assessing Solution Fitness

When considering scoring, the scoring function is uniquely constructed for the problem under consideration. In most cases, this involves assessing the performance of the candidate solution over a range of, or all of, the possible input scenarios. The performance of the candidate solution is compared to the ideal answer or a raw data set and a numeric value of a level of error is produced as a measure of fitness. Fitness scores can be described as either *raw fitness* or *standardised fitness* (Koza [1992]). Raw fitness denotes the number of cases in which the candidate solution produced the correct answer. In this situation, the higher the score, the better the answer. Standardised fitness denotes a measure of the number of input to output tests the candidate solution provided an incorrect answer for. As such, the lower the score, the more fit the candidate solution and when the score is zero, the candidate solution is perfect for the input range considered.

Assessing a range of or all input-output scenarios is often the most computationally expensive part of genetic programming experiment and can expand exponentially with an increased number of inputs. Consider, for example, a problem in the Boolean domain involving 3 inputs. To test every input scenario involves $2^3 = 8$ tests. If this were expanded to consider 11 input variables, then each fitness assessment would involve $2^{11} = 2048$ tests of input combinations in order to ensure the candidate solution will work in every situation. When considering that populations of candidate solutions in the thousands are used for larger problems, this will substantially increase the computational effort required to solve the problem. If one considers a Boolean problem with 20 inputs, then $2^{20} = 1048576$ input combinations

will need to be tested to ensure the candidate solutions is accurate. It is at this point that it becomes necessary to assess a sample of inputs in the interests of being able to complete an experiment with the computational resources available, albeit at the risk of losing accuracy. Whilst the Boolean domain is finite, others, such as the symbolic regression domain is not. Considering the symbolic regression domain, where GP is trying to evolve $y = f(x)$, x could represent an infinite range of numbers. In this case, it becomes necessary to consider a limited range of inputs (values of x) in order to facilitate running the genetic program.

An alternative has been suggested (notably Yanagiya [1995]) where canonical semantic representations of the candidate programs and solution are compared and the differences used to calculate standardised fitness values. This technique has the advantage of not requiring the processing of all the input values, which provides potential to massively scale up problems. However, this technique has several disadvantages; first and foremost, is that it requires the GP practitioner to be in possession of a canonical representation of the perfect solution. In many cases, possessing the perfect canonical representation of the solution is the desired outcome of GP, and as such the possession of the answer makes GP irrelevant. In addition, if it were possible to construct the canonical representation of the perfect behaviour from the input output scenarios, GP would not be required either. In the defence of this method, this technique is ideal for testing theoretical concepts on larger problems and minimising computational load.

Selection Methods

Several selection mechanisms have been suggested over the years, each with their own advantages and disadvantages. In this section, the three most common selection systems in GP are described. These are tournament, fitness proportionate and ranked selection.

Tournament selection (as described by Banzhaf et al. [1998]) involves selecting (at random) *n* programs (tournament size) from the population of candidate solutions, evaluating their fitness and selecting the most fit candidate solution. This method has two advantages which have made it popular in the GP community. Firstly, as the tournaments can be run independently, they can be run in parallel saving time. In addition to this, it may be the case that tournament selection does not require the whole population to be assessed, however, it is usual that GP practitioners use statistics generated from the assessment of the whole

population. With larger problems, this can be computationally expensive, so assessing tournaments separately and if possible only the programs in the tournament may result in saving substantial processing time. Secondly, by increasing or decreasing n, the GP practitioner can alter the selection pressure of the experiment adding another level of tunable control to GP. A large tournament size will result in a high selection pressure. Whilst this can be useful, it can also result in premature convergence of solutions.

A second common selection mechanism, fitness proportionate selection (Holland [1992], Koza [1992]) can be described as a biased roulette wheel where the size of the individual slots is proportional to the fitness of each candidate solution. This system relies on the variance of the fitness of the candidate solutions being neither too large nor too small. If the variance is too small, the fitness proportionate selection effectively become a random selection method as it will move towards a uniform probability of selection as the variance decreases to zero. If the variance is very large and there are one or two very fit candidate solutions, the slots for these candidate solutions become very large, resulting in these solutions being repeatedly selected. This may result in the premature convergence of the candidate solutions in a local optimum rather than a globally optimal solution. To smooth out high variance of fitness, methods such as sigma scaling (Mitchell [1996]) have been developed. Sigma scaling generates an expected value as a function of fitness based on the individuals fitness, the mean fitness of the population and the population standard deviation. The expected value represents the expected number of off spring to be generated using the individual. Methods such as sigma scaling do not resolve the problem of situations where the variance of a population fitness is low.

A third common selection mechanism, ranked selection, applies a rank to each candidate solution based on the solution's fitness and generates a selection probability based on an individual's rank. Obscuring the absolute fitness can be beneficial in that highly fit candidate solutions are unable to dominate the population as quickly. Notwithstanding, in some cases it may be necessary to know whether some candidate solutions are far more fit than others. There are different schemes in use to generate the selection probability. Two common methods are linear and exponential ranking (Banzhaf et al. [1998]).

In addition to selection, many practitioners use elitism (DeJong [1975]). Elitism copies a small number of highly fit individuals through to the next generation unaffected by crosso-

ver or mutation. This has the effect of preserving highly fit programs from the destructive nature of crossover and mutation, as a result preserving highly fit building blocks for later generations.

2.6 The Crossover Operator

Crossover is one of the core operations of the GP process. In this section, crossover is described for tree based GP. The first version of crossover to be considered is standard crossover with uniform swap points.

During standard crossover, two parent programs are selected using one of the selection techniques as described in section 2.5. Once the two parents have been selected, they are copied into two child program trees. Using a uniform distribution the node swap points are selected for the sub tree swap. Once the sub trees are swapped, the new child programs are added to the population. A diagrammatic example of the process using two small examples is presented in figure 2.5.

A common modification to the crossover process, first described by Koza [1992], is to place a bias on the swap points. In his version, a bias of 90% is applied to the function nodes and 10% on the terminal nodes with the intention of resulting in more swaps nearer the root of the program tree, potentially resulting more dramatic change to the genotype and resulting phenotype. The counter argument is that once the structure has converged and some results suggest this occurs relatively quickly (McPhee and Hopper [1999]), it becomes more difficult to fine tune the leaves of the tree to add precision to a solution. As discussed earlier, however, large structural changes at the genotypic level may not result in similarly large changes at the phenotypic level due to low locality and high redundancy present in the genotypic and phenotypic search space. This biased crossover is regularly used by GP practitioners and in this thesis is referred to as Koza crossover.

2.6.1 Improving Crossover in Genetic Programming

Whilst crossover is the central operation to many GP algorithms (mutation is typically applied with a low probability in comparison to crossover), it is a random process resulting in two issues. Firstly, it has been documented that crossover is a mostly destructive process (Banz-

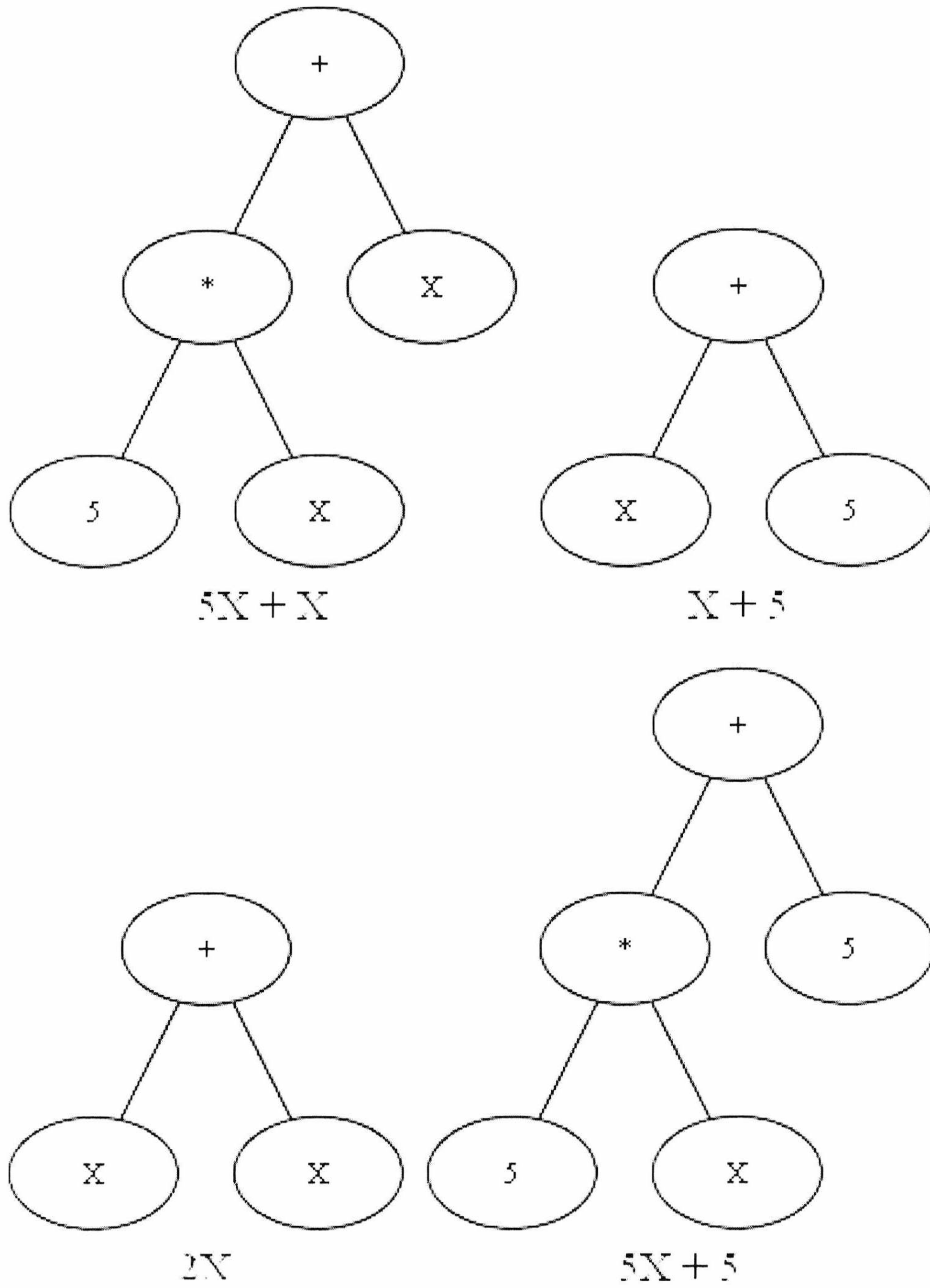


Figure 2.5: An example tree diagram showing the results of tree based crossover. The trees in the top row are the parent trees and the trees in the bottom row are the two child trees. The crossover occurred at the "*" node in the top left parent and the "X" node in the top right parent.

haf et al. [1998]) and theories have been build on this fact. Secondly, due to the random nature of crossover it is quite possible to produce child programs that are phenotypically (or behaviourally) equivalent to their parent programs. This represents an inefficiency in the GP search as the search agent is not moving around the search space. Both of these factors have resulted in several attempts to improve the crossover algorithm.

There are three approaches when considering improving the crossover mechanism. These approaches are: to change the way in which parents are chosen for the crossover, choosing the swap points within the parent programs and using a system of pre and post crossover evaluation.

When choosing parents based on a particular characteristic, common sense suggests that two parents with that characteristic should be selected more frequently in order to improve the performance of the GP run. This characteristic could be fitness based, or it could be due to another attribute (e.g. containing all the distinct terminals when it is prior knowledge that all terminals will be required to solve the problem). The danger of this process is that it could cause the GP to converge on a local optimum and may prevent the search process producing large enough movement (because of the parsimony pressure of the parent selection method) in the search space to escape the local optima.

Modifications to the method of choosing swap-points have been shown to have some effect on performance. Whilst Koza [1992] has already been discussed, there are other approaches to applying bias to swap points during crossover. The concept is that by putting a bias on particular areas of the program tree, it is more likely to cause a bigger movement in the search space. Rosca and Ballard [1999] proposed a similar idea based on a negative binomial distribution over tree depth. Their experiments demonstrate a positive effect on GP performance; while, an incorrect choice of swap-points could, however, have a negative effect on performance. Another example of using the choice of swap points to influence crossover is homologous crossover (Banzhaf et al. [1998], Langdon [2000], Poli and Langdon [1998], Page et al. [1999]), where structurally similar sub trees (defined using edit distances) are more likely to be crossed over.

Pre and post crossover evaluation compares the result of the crossover to the parents. An example of this is presented by O'Reilly and Oppacher [1995], in which GP crossover is hybridised with two hill climbing techniques. Whilst this technique demonstrated that fewer

fitness evaluations were required to reach the ideal solution, there is always the risk that the candidate solutions can be caught in local optima, as well as the additional computational requirement needed to implement this technique. Other approaches using semantic analyses (Beadle and Johnson [2008], Nguyen et al. [2009]) to compare the phenotypic or behavioural aspects of crossover have yielded positive results. These approaches will be discussed further in section 6.1.

2.7 The Mutation Operator

Sub tree mutation selects a random point in a randomly chosen program tree and swaps the sub tree with another randomly generated sub tree. Koza [1992] introduced sub tree mutation, but questioned the value of the operator (later demonstrated by Luke and Spector [1997, 1998] to be comparable to crossover) and chose to perform most of his experiments without the mutation operator in use. Whilst the concept of sub tree mutation is relatively simple, there are more detailed practicalities that influence the relative performance of the GP with the use of the mutation operator.

The mutation process invokes selection to choose one of the candidate solution programs from the population and also randomly selects a point within that program. For standard sub tree mutation, the selected node within the candidate program is swapped for a random node. This may be a terminal or a function. If it is a function, then all of the child nodes are completed to make the tree structure valid. The function which was swapped out of the original program is no longer required and deleted.

The main variation between mutation methods is how different authors have constructed the new sub tree to replace the sub tree that was removed. Two examples of solutions to this are by Kinnear, Jr. [1993] and Langdon [1998]. Kinnear created sub trees that could not increase the program depth by more than 15% after mutation. Langdon's size-fair sub tree mutation utilised a system which ensured that the new sub trees were on average the same sizes (50%-150%) as the previously removed sub tree.

A further variant of sub tree mutation is known as shrink mutation. In this system, a random sub tree is replaced by a terminal. Whilst Angeline [1996] uses this type of mutation to aid his investigation into the sensitivity of the frequency of leaf selection in GP, he also

shows that it helps to reduce program size.

Point mutation (or node replacement mutation) McKay et al. [1995] picks a node and replaces it with a node of equivalent arity. This essentially simulates a single bit flip mutation from genetic algorithms. As noted by Poli et al. [2008], a similar idea is that of permutation, which selects a node and mutates the arguments of this node. Whilst Koza [1992] used this technique in one experiment with little success, Maxwell [1996] had more success with a variant of permutation called swap.

Hoist mutation selects a sub tree from the program to be mutated and uses this sub tree to replace the full tree from which the sub tree was copied. Kinnear, Jr. [1993, 1994] presented and made use of this technique with some success; though it is potentially highly destructive technique because of the loss of root functionality. Later research by McPhee and Hopper [1999] indicated that specific patterns of code within successful programs can be traced back to very early programs in most GP runs. Mutation such as hoist may be highly destructive in that, if it were to alter the root of one of these common ancestors, it would cause a serious decrease in performance.

In a similar fashion to crossover, instead of altering the mutation technique, a few authors have attempted pre and post evaluation of the mutation in order to improve it. In the related field of grammatical evolution, Majeed and Ryan [2007] demonstrate a technique known as context aware mutation. The technique evaluates sub trees and works to prevent the mutation operator causing a destructive change in fitness. One of the limitations noted by the authors is the necessity of building up a repository of "good" subtrees to work with. Potentially, however, this technique could turn the standard mutation operator into another hill climbing operator. The danger of it becoming a hill climbing operator is that the fitness distribution across the search space may be rugged, increasing the possibility of premature convergence on a locally optimal solution compared to a non hill climbing algorithm.

Continuing the theme of pre and post evaluation, Beadle and Johnson [2009b] present results demonstrating that using semantic analysis to enforce a phenotypic change as a result of the mutation operation improves the performance of mutation. The technique, known as *semantically driven mutation* proved successful in comparison to a control of standard sub tree mutation in seven experiments across two problem domains. (Section 6.2 discusses this process in further detail and includes more results from the symbolic regression domain.)

2.8 Other Genetic Programming Practises

For completeness, basic descriptions of *automatically defined functions* (ADFs) and *steady state GP* have been included, though they are not utilised in the work presented in this thesis.

ADFs were developed by Koza [1994] and hinges on a very small number of frequently appearing sub trees being captured as a function. The benefit of this is that the sub tree, expected to be a highly fit building block, does not get destroyed during the destructive crossover process. GP candidate programs can use these functions as much or as little as possible. Koza also reports that ADFs help to reduce code bloat.

ADFs require extra care in their implementation in three ways. Firstly, it has to be established that the ADF is not a constant value (defined in this thesis in section 4.3.4). It would be a waste of resources to capture a value which resolves to a constant as the ADF is not providing any functionality. Secondly, recording multiple ADFs may be problematic due to the recording mechanism selecting code equivalent to parts of an already composed ADF. This would be a waste of the resources available to GP. Finally, there is a danger of recursion. Without careful screening in the recording phase, it may be the case that the ADF calls itself, which would result in an infinitely deep program tree which would not be possible to calculate any result to.

The GP algorithm presented in section 2.2 is in the generational form. An alternative to the generational approach is that of the steady state algorithm. In a steady state algorithm, there are no clear time step iterations. The process performs selection on some of the population, removes the weakest candidate solutions and uses the selected programs as parents to generate new programs using GP operations to fill up the population again. The process repeats until a solution is found or a predefined number of individual evaluations have been performed.

Chapter 3

Current Issues in Genetic Programming

The objective of this chapter is to provide an overview of the latest research evaluating issues in GP that relate to the goals of this thesis. The goals of this thesis are to increase search efficiency through encouraging diversity, remove inefficient segments of code and, as a result, control program growth and finally, to investigate the role of program structure in GP.

In section 3.1, the issue of diversity of candidate solution populations in GP is discussed. In the context of this thesis, diversity is considered at the behavioural level, and tools developed to model behaviour provide new techniques to describe the behaviour of programs. The tools can be applied on two levels; firstly, they can be used to measure the level of diversity present in a GP run and secondly, they can be used to enhance the search ability of existing GP operators.

In section 3.2, program growth or bloat and methods to control bloat are reviewed. The ability to canonically model behaviour and back translate the behaviour into reduced code that all contributes to the behaviour of the program, allows the possibility to reduce program to a minimal form removing inefficient segments of code. This technique is useful as it allows the ability to completely remove bloat from a GP run and evaluate the effects of GP on programs in their minimal form.

In section 3.3, the schema theories are evaluated. The motivation to review building block and schema theories is not directly to consider the numbers of building blocks persisting from generation to generation, but to understand the effects of repeated patterns of code in the

population in the form of program structure.

3.1 Diversity in Genetic Programming

GP is a search process and it is important for the candidate solutions to represent as many different programs (and behaviours) as possible. Thinking back to figure 2.1, more search agents (or in the GP case candidate programs) allow greater coverage of the search space. More candidate programs are only better when they are traversing different areas of the search space simultaneously, rather than a large number of these search agents assessing the same areas of the search space.

Multiple programs assessing the same fitness result represent an inefficiency in the GP search as it becomes less probable that GP will be able to find an acceptable solution. Diversity in GP is further complicated due to the nature of genotypic redundancy (Rothlauf [2006]) present in GP which results in two different kinds of diversity.

When considering diversity in GP populations, it is important to distinguish between the two distinct types of diversity. The first type is syntactic or genotypic diversity, that is, programs in the population being syntactically different. Koza [1992] argues that this is important both as a method of generating programs with different behaviours, and as a means of providing a pool of material from which programs can be evolved.

The second type is behavioural or phenotypic diversity, that is, diversity of the input-output behaviour. It is easy to find examples of sets of programs that are all syntactically distinct, yet which have identical behaviours (genotypically redundant) and theories such as fitness causing bloat (Langdon and Poli [1997]), and formal semantic analysis of population diversity support this fact (Beadle and Johnson [2009a] and chapter 5).

Managing or increasing diversity both at the genotypic and phenotypic level has been poorly implemented within GP with a tendency to only ensure genotypic diversity at initialisation through the traditional Ramped Half and Half technique (Koza [1992]) by preventing the insertion of duplicate program trees into the population of programs. Later efforts by Looks [2007] and Beadle and Johnson [2009a] (and chapter 5) applied phenotypic diversity at initialisation and demonstrated that whilst this can be very successful at increasing GP performance in some experiments, it is not the only aspect to be considered during initialisation

as successful results appeared problem dependant.

After initialisation, standard GP runs are completed with no checks as to the diversity level of the GP population. This combines with the random nature of GP operations such as crossover, mutation and selection which will have a tendency to reproduce existing programs at both the genotypic and phenotypic level, reducing diversity in the search space.

There have been studies aimed at increasing diversity during crossover (O'Reilly and Oppacher [1994a], Poli and Langdon [1998], Beadle and Johnson [2008], Nguyen et al. [2009] and section 6.1) and mutation (Beadle and Johnson [2009b] and section 6.2). The earlier approaches by O'Reilly and Oppacher, and Poli and Langdon, focused upon testing different crossover operators and studying their effects on GP performance. Later approaches by Beadle and Johnson and Nguyen et al. focused on controlling the semantic change of the crossover to ensure a change took place. Whilst these techniques will not result in complete phenotypic diversity (a completely behaviourally diverse population), they do force phenotypic change at the crossover and mutation operations thus resulting in more movement around the phenotypic search space and result in a nearly universal increase in performance.

Studies of the level of semantic diversity present in GP populations are not new to GP. Gustafson et al. [2004], Burke et al. [2004], Gustafson [2004] conducted multiple analyses of behavioural diversity in GP. Burke et al. [2004] conducted an analysis comparing behavioural diversity measures with fitness. These behavioural measures were based on two edit distances and this analysis concluded that the edit distance showed a strong correlation with the level of change in fitness. Gustafson et al. [2004] present three different methods for measuring the behaviours of the programs they study; however, the authors mention that even these mechanisms do not provide an exhaustive representation of the behaviour of the programs. One of the limitations of Gustafson's work (Gustafson et al. [2004], Burke et al. [2004]) was that a behaviourally canonical representation was not used to check for isomorphism. The use of behavioural representations in this work (presented in chapter 4.3) provides that particular ability. As a result of this, it is possible to accurately evaluate the level of diversity present in a population and this has been incorporated into an analysis of initialisation (Beadle and Johnson [2009a] and chapter 5).

Further use of semantic analysis has been applied when assessing the context of crossover operations and attempting to identify semantic building blocks. McPhee et al. [2008]

used truth tables to analyse behavioural changes in crossover. McPhee et al. highlight both the concept of semantic building blocks and the idea of context, which is how the program around the missing the sub tree will influence the behaviour of the sub tree. Context is important because no matter how fit the sub tree, if the context invalidates it, the sub tree will not have any effect as part of the new program it is inserted into. McPhee reports rates as high as over 75% of crossovers having no movement in the semantic search space due to the context invalidating the swapped sub tree.

McPhee et al. [2008] highlight the inherent problems facing the crossover operation in several Boolean problems. As McPhee et al. discuss, their work could be extended to other domains (other than Boolean). This could be achieved by making use of the abstraction techniques presented in chapter 4 for the artificial ant and symbolic regression domains. The work in the Boolean domain could also be scaled up by using *Reduced Ordered Binary Decision Diagrams* (Bryant [1986]) described in chapter 4. Further to this, results generated by both Beadle and Johnson [2008], Nguyen et al. [2009] and in chapter 6.1 take into account the context of crossovers as part of their mechanism of assessing behavioural change. As a result, the extended results for semantically driven crossover may be able to add more data to the argument for importance of context in domains other than the Boolean domain.

Previous research (McPhee et al. [2008], Looks [2007], Beadle and Johnson [2008]) has demonstrated the inability of GP to efficiently move around the behavioural search space. This clearly has an effect on the efficiency of GP search and as a result, phenotypic diversity will become of the key areas of research in GP in the years to come. One possible objective would be the concept of a completely behaviourally diverse run, where every initialised program is behaviourally unique and every GP operation results in a new behavioural step in the phenotypic search space.

Combating the inefficiency in the phenotypic or behavioural search space is one of the primary motivations of this thesis and methods are developed to promote semantic diversity at every stage of GP.

3.2 Bloat and Methods to Control Bloat

In the early GP literature, it was noted that programs grew during evolution with little or no related increase in fitness (Koza [1992]). This increase in program size has commonly been referred to as *code bloat* or just *bloat*. Several theories have been developed based on experimentation (further discussed in section 3.2.1), however, there appears to be no conclusive explanation or combination of explanations to formally describe the occurrence of bloat. What is clear is that, unless GP practitioners can control bloat, it will negatively impact the scalability of GP (Luke [2000c]) and the robustness (human readable programs that can be tested and validated formally) of the programs GP produces. If programs grow without a related improvement in fitness for small scale problems, then increased growth on larger scale problems may prohibit GP finding a good solution simply due to the computational load required to process these larger scale problems. Furthermore, larger programs are harder to formally test the input-output behaviour and maintain, should any improvements be required.

Three of the existing theories have been based on the presence of *introns* within syntax trees. Since the phrase intron was first used in the field of evolutionary algorithms (Angeline [1994]), it has had a confused definition (Blickle and Thiele [1994], Nordin et al. [1995], Luke [2000c]) in the literature. The confusion arises from the consideration of whether suboptimal or redundant code should be treated as an intron as well as inviable or unreachable code.

An intron can be defined as a node (either a single node or sub tree) that does not have an effect on the behaviour (or phenotype) of the program. Within this definition, introns fall into one of two subclasses separated by execution. Firstly, an *unreachable intron* describes a node which cannot be executed under any input state of the program. In other literature, this is sometimes referred to as *inviable* code. An example of this would be:

IF A1 (IF A1 A2 ***A3***) A4

In this situation A3 can never be evaluated because of the nested IF statement based upon the A1 condition. The second kind of intron is the *redundant intron*. This describes a node which is executed but does not affect the behaviour of the program. In other literature, this is referred to as *unoptimised* code. Two examples of this would be:

AND A1 A1 $\equiv$ A1

$$\text{MUL 0 (ADD 5 5)} \equiv 0$$

In the first case, the behaviour will just be A1 and in the second case the answer will result in a constant behaviour 0.

A number of authors have done experimentation examining unreachable introns (notably Blickle and Thiele [1994], Luke [2000b]), however, there is little experimentation examining redundant introns. One of the contributions described by this work are the methods presented in chapter 4.

3.2.1 Existing Bloat Theories

Program growth during GP evolution (or bloat) has affected GP since its inception. Several theories have been presented in an attempt to at least partially describe the causes of bloat, although the precise cause or causes of code bloat is still contentious and elusive. Hitchhiking (Tackett [1994]), protection from deletion (Altenberg [1994b], Blickle and Thiele [1994], Banzhaf et al. [1998]), removal bias (Soule and Foster [1998]), the fitness function causing bloat (Langdon and Poli [1997]), modification point depth (Luke [2003]) and the crossover bias theory (Dignum and Poli [2007]) have all been proposed as causes of bloat each with different merits. Hitchhiking, protection from deletion and removal bias require the presence of introns whereas the other theories do not.

Hitchhiking occurs when introns as well as subtrees that contribute to fitness are crossed over as part of the crossover operation. Selection occurs, selecting the most fit programs and as a result, the successful programs continue to contain the introns and grow as this same process repeats over the generations.

Tackett [1994] provided experimental evidence to demonstrate that bloat was related to the level of selection pressure. In Tackett's experiments, a random selection scenario resulted in no bloat, and this is further confirmed by results presented by Langdon and Poli [1997], Langdon and Banzhaf [2000]. Tackett disagrees with the protection from deletion theory as when applying a less destructive brood crossover system, there was no proportional decrease in bloat.

Protection from deletion states that introns are required to protect fit programs from the mostly destructive crossover operation. Successful individuals are able to survive generation

to generation due to the fact that potentially destructive crossovers take place in intron areas of the program, and as such does not degrade their fitness for selection. Luke [2000b] rejects defence against crossover based on experiments using marking (Blickle and Thiele [1994]). In these experiments, Luke marked inviable introns and demonstrated that bloat persisted despite crossovers taking place in active areas of code. Further to this, semantically driven crossover (Beadle and Johnson [2008]), showed that preventing behaviourally neutral crossovers (an as a result increasing the proportion of destructive crossovers) can significantly reduce bloat and significantly increase performance in some problems. This supports the approach that in some problems introns do not appear to be required to achieve good results and also that protection from deletion may be problem dependant.

Removal bias occurs when inviable subtrees near the leaves of programs more frequently are replaced by larger subtrees with no effect on fitness. As most crossover is destructive, these programs survive due to their fitness being unchanged, whereas, more programs that had crossovers in areas of active code are culled at selection. Removal bias is a variation on the protection from deletion theme and would suffer from the same criticisms as protection from deletion in the form of the marking experiments and semantically driven crossover. Whilst some authors ran experiments using non-destructive crossover (O'Reilly and Oppacher [1995], Soule and Foster [1997]) in order to demonstrate that the protection from deletion theory and removal bias theories are valid because they reduced the level of code growth, Luke [2003] suggests this may be down to replacing less fit children with the parents, thus causing the full bloat effect to be delayed.

The fitness function causing bloat is due to a greater number of programs (or genotypes) representing a particular behaviour (phenotype) in the search space. Given that crossover is a destructive process, when more fit programs become hard to find, then programs of equal fitness are favoured. As such, it is more likely that the size and shape of programs will move to the region of the search space where programs are fit enough to survive. This theory is grounded by a substantial amount of theoretical research (Langdon and Poli [2002]). In addition to this, Rothlauf [2006] has discussed in detail the concept of genotypic redundancy and the results in Beadle and Johnson [2009a] demonstrate the presence of genotypically redundant programs, even from the initialisation stage of GP.

Modification point depth links the size of the program to the depth of the swap point in

the program. Given that most crossover is destructive, programs that experience swaps near the leaves of the trees suffer less decrease in their fitness compared to trees that include a crossover near the root. As a result, more code gets added near the leaves of the tree and longer trees are favoured in terms of survival when the fitness function and selection are applied. As shorter programs do not make it through selection, it suggests that code size is related to survivability. Streeter [2003] also suggested a concept of resilience of programs which was directly related to program size.

The crossover bias theory states that shorter programs will be more frequently sampled as a result of crossover. These small programs will typically have poor fitness so will not be selected during the selection process. This results in the average size of the population increasing. Dignum and Poli [2007] support the crossover bias theory providing strong theoretical evidence showing how the distribution of program sizes follows a *Lagrange distribution of the second kind*.

One of the key issues behind bloat appears to be the selection mechanism. As shown by Tackett [1994] random selection results in no bloat. The moment the fitness function is applied, underlying bias becomes present and have numerous effects on GP performance and program size.

Given the semantic analysis tools described in chapter 4, results will be presented (in chapter 6) which demonstrate the effects of intron free GP on bloat and performance of a GP run for the experiments in the test suite.

3.2.2 Methods to Reduce Bloat

Given the lack of one concise theory to explain the presence of bloat, GP practitioners have developed and applied numerous different techniques to counter bloat. A selection of the most popular techniques are presented in this section.

The first of the efforts to reduce bloat are simple program size limits applied during the genetic operations. The first of these were depth and length (total number of nodes) limits. Koza [1992] used a depth limit of 17, such that if a program was greater than 17 nodes deep after a crossover, the parent program was used instead of the child program. Similar tactics have been used limiting the length (or total number of nodes) of a program. Later research by Gathercole and Ross [1996] indicates that programs are attracted to a predefined depth limit

if one is in place. In addition, recent work by Dignum and Poli [2008b] shows that program size will increase to limits in the early section of GP run defeating the point of them.

Dynamic limits (da Silva [2008], Silva and Almeida [2003]) are an extension of the simple limits system. Programs can be limited by either depth or length (with the maximum limit starting at the maximum initialisation size) and when a new, fit individual is found the limit can be increased to allow the new individual to be entered into the population. Da Silva applies this process in conjunction with other parsimony pressure based techniques and as such the process is not a complete solution to bloat.

Resource limits (da Silva [2008], Silva et al. [2005]), unlike simple and dynamic limits, do not apply to individual programs in the population. The resource limit is applied to the population, for example, a limit of how many nodes are available to the population. This is a difficult parameter as too many will result in bloat, and too little may result a situation where programs grow and as a result the population decreases as there are no more resources for additional programs. Silva and Costa [2005] report results comparing dynamic and resource limited GP. These results support the superiority of dynamic program limits, although it was noted that resource limited GP can achieve some good results earlier in the run.

Attempts have been made to combat bloat as part of the crossover operation itself. Langdon [1999, 2000] implemented a system which developed size fair crossover in order to tackle bloat with some success. Further work by Beadle and Johnson [2008], using semantically driven crossover unintentionally showed significant success at reducing code bloat on some problems (discussed further in chapter 6).

Luke and Panait [2006] presented a substantial comparison of a selection of bloat control methods and compared their results over a range of benchmark problems. This analysis included linear and Pareto based parsimony pressure techniques, double tournament selection, proportional tournament selection, the Tarpeian selection method (Poli [2003]), waiting room and death by size techniques.

The Tarpeian method (Poli [2003]) introduces a parameter which represents a probability of assigning a very poor fitness value to a large program without assessing the fitness of the program. The method is tunable and Luke and Panait report that a probability of 0.3 works well in their experiments. If the probability is too low, there is not enough pressure against the larger programs and if the probability is too high, then the Tarpeian method will reject large

and highly fit individuals without assessing the fitness when some of the large individuals may represent very fit programs. This method has the advantage of not having to assess the fitness of a percentage of the larger individuals, thus saving computational resources.

Linear parametric parsimony pressure considers size as well as fitness as contributing to the overall fitness of a candidate program. The potential problem with this method is the fact that without scaling, one element of the equation $fitness = f(fitness) + f(size)$ will overwhelm the other element creating bias in the fitness function. As a result, the fitness and size components are scaled in an effort to prevent this bias occurring. The choice of coefficients in order to balance fitness and program size is difficult, however, recent work by Poli and McPhee [2008] shows how to apply Price's covariance theorem to dynamically set and optimise the parsimony coefficients.

Pareto-based parsimony pressure considers size and fitness as two separate objectives and use Pareto-dominance optimisation (Ekart and Nemeth [2001]). In similar circumstances to the linear parsimony pressure, a potential problem using this methods is that either size or fitness dominate the result at the Pareto front. Ekart and Nemeth [2001] dealt with this issue using an algorithm known as SPEA2 to enforce diversity at the Pareto front.

Double tournament selection applies two tournaments during selection in sequence in order to select each parent for crossover or mutation. One tournament is based on fitness and the other on program size. Luke and Panait tried both orders of tournament, i.e. fitness before size or fitness after size. Proportional tournament selection, based on a probability, either uses a fitness function based on fitness or a fitness function based on program size.

The waiting room method forms an ordered queue of programs that are waiting to enter the population with the biggest programs at the back. This has the effect of forcing the evaluation of the smaller programs first in order to find high fitness within smaller programs. Death by size is operational within the steady state version of GP. Instead of using generations, a number of programs are killed off every so often and crossover is used to refill the population. In the case of death by size, larger programs are more likely to be killed off.

Luke and Panait [2006] concluded that individually, the linear parametric approach was a good choice to control bloat across all the experiments they present. In addition to this, they noted that any of the bloat reduction techniques combined with individual simple limits were nearly always superior to the individual bloat control method alone.

A final tactic for controlling bloat is to reduce programs, also referred to as *code editing*. Recent work by Garcia-Almanza and Tsang [2006] presents a pruning mechanism known as the *Scenario Method* which was shown to effectively reduce programs; however, this resulted in a variance of performance. Another approach by Eggermont et al. [2004] presents a method to detect and prune trees using static analysis techniques. This method reduces production rules and can use the reduced trees to check for semantic equivalence. The work of Eggermont et al. was used on classification problems (supervised machine learning) and reduces programs at the genotypic representation rather than the phenotypic (or behavioural) level. In opposition to code editing, Haynes [1998] presented results which suggested that code editing could be responsible for the premature convergence of solutions at a local optima.

These techniques give GP practitioners formal mechanisms to evaluate all introns and in chapter 6, explicit experiments are conducted aimed at removing all introns from programs at the different stages (initialisation, crossover, mutation and pruning) in the GP run. The resulting effects of these techniques can be used to analyse the effects on GP of the removal of introns in relation to existing theories of bloat.

3.3 Building Blocks and Schema Theories

In this section, the complex issue of repeated patterns within candidate solutions is discussed. Firstly, building blocks and schema theories are discussed, outlining their difficult birth in the context of GP to their most recent general and exact schema theories for GP. Secondly, a related side issue of program structure (related in that repeated structures of programs are built from repeated blocks of code) and its emerging role in GP are considered.

3.3.1 Schema Theories

In a most general sense, schema theory identifies a set of elements from the search space (an individual *schema*) and provides mathematical methods in order to be able to predict the probability of a schema being present in a future generation. In line with the *building block hypothesis* (Goldberg [1989]), which is that small but highly fit individuals are combined hierarchically as programs move towards a global optimum, schema theory can be used to

predict the presence of these building blocks. The reason this is important is that, whilst crossover and mutation are essentially random operators, it is difficult to understand the mechanics that result in GP finding optimal solutions to problems. Combined with this, if GP researchers can understand and model the transitional and random nature of crossover and mutation, they may be able to suggest theoretically motivated techniques to enhance the performance of GP runs.

Schema theories were originally developed for genetic algorithms (Holland [1992]) (original text in 1975, but republished in 1992) and expressed in the form of a lower bound of the number of a particular schema that would be present in the next generation. Holland's original schema theory was informally extended into GP by Koza [1992].

The first of the schema theories dealt with *non rooted* schemata. Non rooted schema are sub trees that can occur at any point in a program tree and potentially repeatedly. Altenberg [1994a] was the first to suggest a schema theory specifically for GP. Altenberg's theory was developed under a number of assumptions including: the population being very large, no mutation and that fitness proportionate selection was used. Altenberg's schema theory differs from Koza's in that Altenberg considered a schema as a sub expression, whereas Koza considered that a schema could be made up of multiple sub expressions. A final fundamental difference is that Altenberg took account of schema creation which results in his theory being an equality rather than a pessimistic lower bound.

O'Reilly and Oppacher [1994b] refined Koza's work on schema theory by considering the addition of a *don't care* node. Schemata are defined as a set of unordered code fragments and sub trees. Like Koza, O'Reilly and Oppacher do not take account of the position, and therefore, do not take into account the context (the way in which the schema is executed in the program) of the schema. One of the problems with potential for multiple contexts is that the theory describes the way in which components of a representation change over time rather than how the number of programs representing a specific schema changes over time.

In a related area of GP using context free grammars, Whigham [1995c] used a slightly different definition of a schema to O'Reilly and Oppacher, however, this was still a non rooted definition of a schema. As a result, the schema theory again describes the way in which the components of a representation change over time rather than the number of programs containing specific schema changing over time. The main difference is that the derivation

tree fragments would always represent a single sub expression rather than multiple sub expressions as with O'Reilly and Oppacher's definition.

These early difficulties in obtaining a precise description of how schema propagates from one generation to the next left schema theory open to some criticism. Langdon and Poli [2002] describe three major weaknesses of the theory up to this point. Firstly, the schema theories so far (with exception to Altenberg [1994a]) were inequalities and not equalities, which means that they only provide a lower bound of the expected number of instances of a particular schema in the next generation. Secondly, unless the population is infinite, the expectation operator means that it is difficult to predict the behaviour of the genetic program over multiple generations. Finally, as most of the theories only provide a lower bound, it is questionable how useful a lower bound is even one generation ahead for the prediction of the occurrences of a particular schema.

A further criticism of the non rooted schemata is that, given that candidate solutions are executed to assess fitness, a non rooted schema takes no account of the context (the way a schema is executed due to its position in the tree). As a result, the behaviour of a schema may differ from one program to the next. Independently and simultaneously Rosca and Ballard [1999] and Poli and Langdon [1997] developed rooted tree schema theories. The theories were constructed slightly differently, however, they both reintroduced the importance of position of components of a schema and could only be instantiated once per program because the context information fixed the root of the tree. Whilst Rosca and Ballard [1999] use a *don't care* node to represent complete subtrees, Poli and Langdon [1997] use a *don't care* node to represent exactly one function or exactly one terminal. This has the effect of fixing the size and shape of the schemata, which as a result made it easier to calculate all the possible combinations of program using a particular schema in comparison to non rooted schema.

One of the main issues resulting in schema theories producing a lower bound rather than an equality was that the task of predicting the frequency of occurrence of a particular schema in a later generation depended on two factors. The first is modelling the destructive effect of crossover and mutation in combination with a schema being selected. Existing schema theory produced a lower bound because it could only predict survival against destruction or elimination through crossover, mutation and selection. The element that was missing is the

prediction of how many instances of a particular schema could be created from crossover or mutation events. Identifying all of the ways that a particular schema can be created after having identified all the ways in which it could be destroyed is a complex task.

Whilst the first exact schema theories were described for genetic algorithms by Stevens and Waelbroeck [1997], Stephens and Waelbroeck [1998], these theories were extended into GP to provide an exact GP schema theory for one point crossover (Poli [2001]). This work represented a step forward; however, it does not account for the usually applied sub tree crossover with the uniform selection of crossover points (or the 90% bias on functions and 10% on terminals) favoured by the GP practitioners, only fixed size and shape representations in GP.

The difficulty with representing standard crossover is that the program trees may change size and shape after crossover and mutation operations. As such, the fixed size and shape schema needed to be extended to take advantage of both *don't care* nodes as single nodes (Poli and Langdon [1997]) and a second kind of *don't care* node which could represent a sub tree (Rosca and Ballard [1999]). This was described as *variable arity hyperschema*. A variable arity hyperschema could contain functions, terminals, a *don't care* node that represented either a function or a terminal or a *don't care* node that could represent a sub tree.

A second problem that needed addressing was that a node referencing system needed to be designed in order to uniquely identify every node in the program tree. The node referencing system would enable functions and probability distributions to be defined over the node references. The full description of the node referencing system and how probability distribution are made use of in the context of different types of crossover is set out by Poli and McPhee [2003a]. This step allowed the creation of general and exact schema theories by Poli and McPhee [2003b] for standard crossover in GP (and the 90% and 10% bias on functions and terminals respectively) as well as a number of other types of crossover.

So the real question is; how can schema theory be put to practical use in GP? Langdon and Poli [2002] discuss the potential for practical uses in three areas. Firstly, given that schema theories require the modelling of the effects of different operators, an advanced understanding of these effects may lead to better choices of operators to solve particular problems or the development on new theoretically motivated operators (for example, Poli

and Langdon [1998]) to enable different effects in the GP search. Secondly, understanding operator bias may enable the creation of theoretically driven initialisation algorithms (for example, Langdon [2000]) which are able to benefit from or neutralise the effect of operator bias. Finally, Langdon and Poli [2002] suggest that further research may make it possible (in conjunction with Stephen and Waelbroek's schema theory for genetic algorithms) to derive a method of calculating population sizes in order to maximise success in GP. In turn, this would enable the estimated calculation of how much computational effort is required to solve a particular problem and may lead to being able to classify some problems as "easy" and some as "hard" as a function of computational effort.

Whilst so much research has gone in to predicting the frequency of repeated blocks of syntax within program trees, in comparison very little research has considered looking at semantic or phenotypic building blocks within GP. For instance, it would be interesting to build on the work of Haynes [1997] and evaluate a set of schema for a particular problem to compare whether the schema represent the same number of schema in the semantic search space. There is the potential for substantial redundancy in the form of genotypic schema to be present and further analysis using phenotypic representations or even phenotypic schema may allow a simpler understanding of the combination of behaviours during a GP run.

Langdon and Banzhaf [2005] developed further work analysing building blocks both concerning the shape and size of programs and the semantics of programs. Langdon found that the semantic repetition of building blocks was much higher than the syntactic repetition of building blocks. Additionally, Langdon found that in the two problems chosen, the programs produced were similar shapes which brings into focus a fractionally different area of research in the form of program structure. Program structure could be considered as a low ordered hyperschema, that is a tree structure full of *don't care* nodes, which would just define the shape of a program.

Previous work on the analysis of changes to program structure has demonstrated that tree shape does have an influence on the success or failure of GP. Punch et al. [1996] and Gustafson et al. [2005] present artificial problem domains in the form of royal trees and the tree-string problem. These artificial problem domains are tunable and designed to evaluate the relationship between program structure and the ability of GP search to navigate through the search space. Whilst Langdon et al. [1999], Langdon and Banzhaf [2005] provide

evidence that programs evolve towards particular shapes, Daida and Hilss [2003], Daida et al. [2003] suggest not only that program structure has a role to play in evolution, but that predictions can be made as to which shapes are the most evolvable. Further work by Langdon [2000] and Daida et al. [2003] both discuss how tree shape can impact upon the search power of GP, and as a result the performance. Daida et al. go further and discuss how tree structure may determine problem difficulty.

3.4 Summary

Returning to the three goals of improving diversity, controlling bloat and evaluating the role of program structure during evolution, this thesis presents novel techniques to evaluate these three areas in details.

When addressing the search inefficiency in the form of low diversity in the phenotypic or behavioural search, research presented in chapters 5 and 6 based on methods of canonically modelling behaviour described in chapter 4, seeks to encourage phenotypic diversity at all stages of the GP process. More specifically, algorithms are presented to encourage phenotypic diversity at the initialisation, crossover and mutation stages of GP and tested on a suite of nine benchmark problems.

Considering controlling bloat, based on the methodology presented in chapter 4, one algorithm is presented which completely rebuilds programs in a minimal efficient form in order to analyse the effects of intron free GP. Combined with this, the semantically based crossover algorithm shows some promise at controlling bloat. Further research in chapter 7 indicates different characteristics of bloat when using crossover and mutation and demonstrates that the choice of operator may help control program size.

Program structure in this thesis is analysed in two ways. Firstly, the effect of different shaped programs at the point of initialisation is analysed in section 5.2.4. This small analysis motivated a greater detailed analysis of program shape during evolution and particularly its role at initialisation. The analysis is contained within chapter 7.

Chapter 4

Methodology

This chapter presents the methodologies used for the experiments and analyses in chapters 5, 6 and 7. In section 4.1, the test problems are described and, in section 4.2, the standard parameters for each of the test problems are described.

In order to model behaviour, three different descriptions of behaviour are developed for three substantially different problem domains. The description of the behaviour of Boolean problems is described in section 4.3.1. The description of the behaviour of the artificial ant problem domain is described in section 4.3.2. Finally, the behaviour of programs in the symbolic regression domain is described in section 4.3.3.

4.1 A Test Problem Suite

The nine problems which are studied have been chosen because they represent a selection of benchmark problems from three substantially different problem domains. This difference in the problems domains is required in order to test that the algorithms and theories presented in this thesis are applicable to more than one problem domain. Within the Boolean domain, the majority and multiplexer problems have been chosen for their contrasting nature and the even parity problems have been chosen as they are deceptive (Langdon and Poli [2002]). The experiments chosen have been scaled to make the problems more difficult for GP to find accurate solutions to and test the theories and algorithms in a more complex setting. In the artificial ant domain, the benchmark Santa Fe trail has been selected, which is known to be a hard problem (Langdon and Poli [1998]) for GP to solve. In the symbolic regression domain,

a version of a cubic and a quartic polynomial have been chosen as test problems.

The objective of the 6 bit multiplexer problem (referred to in all tables and figures as 6MUX) is to interpret two control bits {A0, A1} as a binary number and choose the correct input bit from the terminal set {D0, D1, D2, D3} to output based on the binary number. The fitness is the number of correct choices over over all possible 64 combinations of inputs for the six Boolean bits. The function set used is {IF, AND, OR, NOT} and the terminal set is {A0, A1, D0, D1, D2, D3}.

The 11 bit multiplexer problem (referred to in all tables and figures as 11MUX) is a larger version of the 6 bit multiplexer problem. There are three control bits which, represented as a binary number, can select one of 8 inputs to be output. The fitness is the number of correct choices over over all possible 2048 combinations of inputs for the eleven Boolean bits. The 11 bit multiplexer is substantially more complex compared to the 6 bit multiplexer as the size of the search space increases from 2^{2^6} (6 bit multiplexer) to $2^{2^{11}}$(11 bit multiplexer). The function set used for the 11 bit multiplexer is {IF, AND, OR, NOT} and the terminals set is {A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7}.

The objective of the even 4 parity problem (referred to in all tables and figures as 4PAR) is to return true if and only if an even number of the inputs are true. The function set is {IF, AND, OR, NOT} and the terminal set is {D0, D1, D2, D3}. The score in this case is the number of correct outputs for all of the 16 different input combinations.

The 7 parity problem (referred to in all tables and figures as 7PAR) is a larger version of the 4 parity problem with the objective or returning true, if and only if, an even number of inputs are true. The function set used is {IF, AND, OR, NOT} and the terminal set is {D0, D1, D2, D3, D4, D5, D6}. The fitness score is the correct classification of the even number of inputs for all 128 input combinations.

The objective of the 5 majority problem (referred to in all tables and figures as 5MAJ) is to return true, if and only if, the majority of the inputs are true. The function set is {IF, AND, OR, NOT} and the terminal set is {D0, D1, D2, D3, D4}. the score in this case is the correct classification of all 32 input combinations.

The 9 majority problem (referred to in all tables and figures as 9MAJ) is an extension of the 5 majority experiment with the same function set and the terminal set {D0, D1, D2, D3, D4, D5, D6 , D7, D8}. The larger number of terminals increases the difficulty as the score is

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0		x	x	x																												
1				x																												
2				x																						x	x	x				
3				x																					x					x		
4				x																					x					x		
5				x	x	x	x		x	x	x	x	x									x	x									
6													x																	x		
7													x								x											
8													x								x											
9													x								x									x		
10																					x											
11													x																			
12													x																	x		
13													x								x											
14													x								x						x	x	x			
15																		x						x								
16																																
17													x																			
18													x				x								x							
19													x				x											x				
20													x				x															
21													x				x															
22													x														x					
23													x											x								
24				x	x			x	x	x	x	x					x															
25		x															x															
26		x															x															
27		x							x	x	x	x	x	x	x																	
28		x						x																								
29								x																								
30			x	x	x	x																										
31																																

Figure 4.1: The Santa Fe trail for the artificial ant problem. The ant starts at 0:0 facing east. X symbols represent the location of food pellets.

based on the number of correct classifications out of 512 input combinations.

The artificial ant *santa fe* problem (referred to in all tables and figures as AASF) models an ant operating over a trail of food pellets on a grid. The ant must collect all the food pellets in order to achieve full score. The benchmark *Santa Fe* trail (Langdon and Poli [2002]) as shown in figure 4.1 is used, which is a trail of 89 food pellets (and is also a broken trail in places) on a 32X32 toroidal grid.

The function set for the ant problem is {IF-FOOD-AHEAD, PROGN2, PROGN3} and the terminal set is {MOVE, TURN-LEFT, TURN-RIGHT}. The function IF-FOOD-AHEAD is an if-then-else structure with the condition representing whether the ant has a food pellet in the grid square directly in front of it. PROGN2 and PROGN3 execute the instructions they hold in sequence. The only difference between them is that PROGN2 has an arity of two and PROGN3 has an arity of three. The score is represented by how many food pellets the ant can pick up in a set number of time steps, where a time step is either a move or a turn. In

this case, 600 time steps are used.

In the cubic polynomial problem (referred to in all tables and figures as CUBIC), GP is attempting to generate an equation equivalent to $y = 3x^3 + 2x^2 + x$. The function set used is {ADD, SUBTRACT, MULTIPLY, PROTECTED_DIVISION} and the terminal set is {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, X}. Protected division prevents division by zero calculating the result as zero rather than infinity. Fitness is assessed by calculating the set of values between $-5 \leqslant x \leqslant 5$ in intervals of 0.5, and summing the absolute error obtained for each reading.

In the quartic polynomial problem (referred to in all tables and figures as QUART), the objective of GP is to generate an equation equivalent to $y = x^4 + x^3 + x^2 + x$. The function set and terminal set is the same as that used for the cubic polynomial problem. Fitness is assessed using the same method as for the cubic polynomial problem.

4.2 General Genetic Programming Parameters

These parameters have been chosen based on the experimental settings presented in Koza [1992] as they are relatively common parameters used by GP practitioners. Using zero probability for mutation removes an additional variable of consideration. Whilst most experiments presented in this work will use these parameters, individual parameters may be modified for experimental purposes. For example, an experiment testing initialisation may make use of different initialisations to that of the Ramped Half and Half initialisation method.

The general parameters used for all experiments, unless otherwise stated were:

- 0.9 crossover probability
- Koza standard crossover with 90% bias on functions and 10% on terminals
- 0.1 reproduction probability
- Breeding pool at 10% of population size
- 0 mutation probability
- Ramped half and half initialisation (depths 2 - 6)
- 10% elites moved through to the new population

- Maximum depth limit of 17 enforced during crossover and mutation operations. Programs deeper than depth 17 are replaced by their parent programs.
- 50 generations
- 100 runs
- 7 competitor tournament selection
- Population size of 500 for 4PAR, 5MAJ, 6MUX, AASF, CUBIC, QUART and population size 4000 for 7PAR, 9MAJ, 11MUX

4.3 Modelling Behaviour

The problems described in section 4.1 fall into one of three categories of problem domain. These are; the Boolean domain, the artificial ant domain and the symbolic regression domain. This section presents the methods used to reduce syntax to canonical representations of behaviour in the respective domains.

4.3.1 Boolean Domains

In order to canonically model the behaviour of Boolean programs, *Reduced Ordered Binary Decision Diagrams* (ROBDDs) have been chosen due to their well established reduction rules, as set out by Bryant [1986]. The important functionality that this provides is the ability to reduce program representation to its canonical form by removing redundant and unreachable arguments. The key functionality is that the resulting canonical representations can be used to compare programs for semantic equivalence. Any two programs that reduce to the same ROBDD are semantically equivalent, and *vice versa*. In the GP context, not only can this technique be used to reduce programs to a canonical form for semantic equivalence checking, it can also be used to reduce programs and translate them back to a smaller, fully effective syntax.

In a practical sense, to enable the semantic evaluation of Boolean programs during a GP run, a Java implementation of GP (Beadle and Castle [2007]) which is linked to the *Colorado University Decision Diagram Package* (CUDD—Somenzi [1998]) using the *JavaBDD*

(Whaley [2007]) interface has been constructed. This provides the ability to both translate GP syntax into reduced representation and translate the reduced representations back into syntax.

This section is further divided into an explanation of key features of ROBDDs, the translation and reduction mechanism and back translation mechanism.

Key Features of Reduced Ordered Binary Decision Diagrams

An ROBDD is a node tree where each node represents a Boolean decision variable. These nodes are linked by true and false branches to either other nodes or the final output of the diagram (true or false). An example of an ROBDD can be found in figure 4.2.

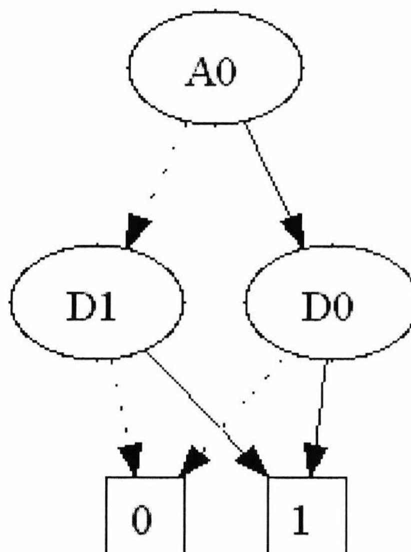


Figure 4.2: This example ROBDD is a canonical representation of behaviour. In the diagram, circles represent variables (terminals in the GP context), solid arrows represent true paths and dotted arrows represent false paths. The squares marked 1 and 0 represent output of true and false respectively. This behaviour could be represented by many different parse trees. Two examples of parse trees that would result in this behaviour are IF A0 D0 D1 and IF (NOT A0) D1 D0.

Two important measurements used in the analysis of ROBDDs are *SatCount* and *NodeCount*. *SatCount* is a value between 0 and 1 that represents the number of input combinations resolving to true in the ROBDD, divided by the total number of input combinations possible. *NodeCount* will return the number of variables present in the ROBDD (in the GP context the number of terminals used). This function can be used in conjunction with *Sat-*

Count to classify behaviours. For example, a SatCount of 0.25 with a NodeCount of 2 would represent a function such as AND A0 A1, given that there are four input combinations in total of which only one results in true. In a second example, a SatCount of 0.75 with a NodeCount of 2 would indicate a function such as OR A0 A1. If the *NodeCount* is 1 and the *SatCount* is 0.5, the ROBDD of the program parse tree will reduce to just one variable (terminal) and possibly a function such as NOT.

A *tautology* is a program which produces the output *true* regardless of input. In the case of a tautology, the value of *SatCount* is 1. A *contradiction* is a program which produces the output *false* regardless of input. For a contradiction, the value of *SatCount* is 0. Unlike in program parse trees, ROBDD functions are not represented explicitly as nodes in the parse tree, but by using true and false links between the variables. Due to the reduction mechanism (Bryant [1986]), it is possible to reduce some ROBDDs to just true or false (i.e. tautology or contradiction). If one considers the example AND A1 (NOT A1), this program will always result in false and the ROBDD of this program will reduce to false. In the GP context this is very undesirable because it indicates that the result is not dependent on any of the variables and always returns the same answer (true or false).

Translation to ROBDD

With a limited function set such as F = {IF, AND, OR, NOT} and a arbitrary fixed ordering of variables, translation to an *Ordered Binary Decision Diagram* is a matter of following a rule set as follows:

- Terminal, assemble variable with true and false links to correct outputs.
- Function NOT, swap the true and false links of the child node to the outputs.
- Function AND, first node with false link to false output and true link to second node, second node with false link to false output and true link to true output.
- Function OR, first node with true link to true output and false link to second node, second node with false link to false output and true link to true output.
- Function IF, true link from condition node to true branch node and false link from condition node to false branch node.

After these rules have been applied the *Ordered Binary Decision Diagram* can be reduced. Bryant [1986] defines an ROBDD as "A function graph *G* is *reduced* if it contains no vertex *v* with a *low(v)=high(v)*, nor does it contain distinct vertices *v* and *v'* such that the sub graphs rooted by *v* and *v'* are isomorphic." (where low and high are the false and true branches respectively connecting the child nodes). This provides the basis for the reduction mechanism and Bryant describes an example algorithm to perform the reduction method.

Back Translation from ROBDD

Considering the same limited function set, F = {IF, AND, OR, NOT}, translation back from ROBDD format is a matter of following the rule set:

- Node with true link to true output and false link to false output becomes single terminal.
- Node with true link to false output and false link to true output becomes NOT and single terminal.
- Node with false link to false output and true link to another node becomes AND function.
- Node with true link to true output and false link to another node becomes OR function.
- Node with both links to other nodes becomes an IF function.
- Node with true link to false output and false link to another node becomes OR function with nested NOT function.
- Node with false link to true output and true link to another node becomes AND function with nested NOT function.

Limitations

Bryant [1986] notes there are limitations to making use of ROBDDs. From the point of view of GP, one limitation is the fact that node trees including ten or more variables can in some situations result in node trees containing over 100,000 vertices. Given the random nature of GP, this could be possible when reducing a program to ROBDD form. Having expressed these limitations, throughout the running of all of the experiments (including the 11MUX), the creation of the ROBDDs has never been excessively slow or unusable (see results presented

in table 5.1). This maybe due to a bias towards simplistic behaviour (see discussion in chapter 5) or the increased power of modern computers considering this algorithm was devised in 1986.

In contrast, the back translation mechanism does have limitations. Problems with more variables/terminals than the 11MUX do hinder the ability to back translate the programs, although the loss of performance appears to occur when constructing the syntax tree and the fact that the resultant syntax trees can be much larger than the standard GP depth limitations. For further discussion on this topic, see chapter 5.

4.3.2 Artificial Ant Domain

In order to represent the behaviour of ants, a behavioural model can be considered as a sequence of moves and orientations that represent the path which the ant has travelled during only one execution of the ant control program (or GP candidate solution). When the artificial ant is simulated in GP, the candidate solution is repeatedly executed until the ant has travelled a set number of time steps (600 in this case) Langdon and Poli [2002]. In this behavioural model, the ant control code is only executed once. In addition to this, the ant code is executed on a toroidal grid (32X32) that contains no food pellets and the path of both the true and false branches of the IF-FOOD-AHEAD (if-then-else) function are calculated.

An example program in this domain is as follows:

PROGN2 (PROGN3 (MOVE, (IF-FOOD-AHEAD (PROGN2 (MOVE, TURN-RIGHT)) MOVE) MOVE) TURN-LEFT)

An example of the syntax, equivalent to the above program, is as follows:

$$\text{Ant Representation} = \langle \text{M}, \langle \text{ M, S } \rangle, \langle \text{ M } \rangle, \text{M, N } \rangle$$

The character M represents one move and the characters N, S, E, W represent the orientations north, south, east and west respectively. The sub sequences within the set indicate when a branch of an IF-FOOD-AHEAD statement is being accessed and coordinates within those brackets indicate the path travelled during each branch of the condition. When considering the ends of the IF-FOOD-AHEAD statements, rather than duplicate the path following

the IF branches in each branch, the orientation is reset to the orientation before the IF statement and a single path follows the branches of the IF statement. Because modelling the shape of the trail is the only concern (consider the picture of the trail as one looks down on the grid), it is unimportant whether or not the ends of each of the if-blocks have different orientations for the trail to continue upon, only that the relative meaning for modelling change of position (or picture of the trail) is captured (once and not duplicated) after the IF statement.

More formally, the representation can be described in Backus-Naur Format:

$$\begin{aligned} rep &::= \langle < expr > \rangle \\ expr &::= M|N|S|E|W| < bracketExpr > | < expr >, < expr > \\ bracketExpr &::= \langle < expr >, < expr > \rangle \end{aligned}$$

As with the Boolean domain, the random combination of syntax produced by GP can produce very simplistic behaviours. The most common of these is an ant that does not move. As a result, this produces a zero trail and effectively is similar to the tautology and contradiction in the Boolean domain.

Translation to Representation

In addition to this model structure, three checks are added which condense the abstract representation to a canonical form. These three checks are:

- Remove duplicate sub branches of the same *if* statement and incorporate the paths as part of the fixed path the ant was on before the *if* statement.
- Search for sequences of orientations and reduce them to the last orientation in the sequence. This has the effect of removing redundant turns from the ant abstract model.
- Moving through the representation, remember the current orientation and remove any duplicate calls to turn to the current orientation. This serves to remove redundant turn instructions.

Back Translation from Representation

Back translation of the ant representation is performed in two stages. Firstly, a linear list of instructions is created from the representation using the terminals available for the ant representation. Secondly, the list is transformed into a tree using the function set {IF-FOOD-AHEAD, PROGN2, PROGN3}. The IF-FOOD-AHEAD branches can be interpreted from the linear list and after this is resolved it is a matter of using PROGN2 and PROGN3 to build up the tree.

Due to a weakness in the ant representation, extra syntax is required to back translate from the representation. The reason for this is that it is possible to produce a non moving ant that ends in the same orientation as one of the branches of the IF-FOOD-AHEAD function. Without any method to cater for this the result would be a blank branch, which syntactically would be invalid. The fix that has been applied to this is to create an extra terminal called SKIP. The only functionality of SKIP is to cost the ant one move, because the ant could get trapped on the execution of SKIP due to the nature of the IF-FOOD-AHEAD function.

4.3.3 Symbolic Regression

In order to model symbolic regression, the representation of behaviour is considered as an ordered list of powers, variables and coefficients very similar to the concept of a simplified polynomial equation with some cosmetic differences. In the implementation that provides the results for this thesis, the translation and simplification methods have been created considering a single variable, any number of numeric constants and the function set F = {ADD, SUB, MUL, PDIV} (where PDIV is a protected division resolving numbers divided by zero to zero rather than infinity). This is expandable as the representation could be made to manage multiple variables and the simplification methods could be improved to include for more functionality. This minimalistic version is presented as a proof of concept.

Consider the example:

ADD (MUL X 4) (ADD (MUL 4 5) (ADD X 1))

The example syntax would resolve to:

{CVP(21 X 0), CVP(5 X 1)}

Where CVP represents a coefficient, variable, power object.

Written in mathematics notation $5x + 21$ is the result of the formula. The CVP object format is used to provide a standard format to model both constants and variables as well as allow mathematical manipulation of algebraic terms. In the CVP list the element with the lowest power is listed first so that the list is ordered to simplify checks for isomorphism.

Readers may note that functions using PDIV will result in zero when divided by zero causing a plot of the regression formula to spike towards zero when X is zero. As this reduction mechanism does not calculate the result of the regression formula, this problem does not effect the reduction mechanism. Consider the example PDIV 1 (ADD 5 X). This would result in a representation value of {CVP(1 X -1), CVP(0.2 X 0)}.

As with the Boolean and ant domains, there is the potential for constant behaviour representations to be produced. A simple example would be a multiply by zero operation at the root of the tree, thus resolving with the whole formula to 0. This is not ideal as the result is not dependant on any of the GP terminals. Moreover, any CVP object with a power of zero is not reliant on any input variables as, for example, $x^0 = 1$. As a result of this, regression representations that resolve to a constant value are considered as undesirable. There may be the argument that sometimes the solution is a constant, however, it would be doubtful that GP would be used to construct a constant solution not dependant on any variables.

Translation to Representation

Several steps are used to translate the syntax tree to the CVP list format. These are:

- Resolve all multiply by zeros.
- Resolve all PDIV by zeros and PDIV with isomorphic subtrees.
- Resolve all constant calculations.
- Reduce to the CVP format by first changing all terminals into CVP format and then conducting algebraic simplification to remove all multiplication and protected division operations.
- Simplify any CVP addition and subtraction operations where possible.

- Extract the remaining CVP objects from the tree and place them into the ordered CVP list.

This process effectively mimics that of algebraic simplification for polynomial equations.

Back Translation to Syntax

Translating back to syntax from the ordered CVP list is straightforward in comparison to the reduction mechanism. Firstly, build a basic syntax tree using the list of CVPs in the representation containing ADD and SUB functions. Secondly, recurse through the node tree and expand out all the CVP objects. For those with powers of zero, then insert a constant, for positive powers, expand using MUL and those with negative powers using PDIV.

4.3.4 Constant Behaviours

Within the Boolean, ant and regression domains, it is easy to demonstrate programs that do not depend on the input variables. Some examples of these programs include:

AND A1 (NOT A1)

PROGN2 TURN-LEFT TURN-RIGHT

SUB X X

Throughout this work, programs that do not depend on inputs are referred to as constant behaviours. The ant is somewhat different in design, however, an ant that contains no move instructions is stuck (as a constant) in one place and cannot pick up any food. In symbolic regression, a program may resolve in such a way that no variables are present and as such will be a constant value. In the Boolean domain, truth tables dictate that some programs will reduce to tautology or contradiction form, and while these programs do not depend on the values of their input variables, they count as a constant result for every input.

Chapter 5

Semantic Analysis of Initialisation in Genetic Programming

The goal of this chapter is to attain an advanced understanding of the issues involved with program initialisation in genetic programming (GP) through the analysis of different aspects of program initialisation. Although four different program initialisation algorithms are presented with varying levels of success, the focus of this chapter is primarily on testing theories rather than presenting algorithms for practical use. This chapter extends the results presented in Beadle and Johnson [2009a] to include results from experiments in the symbolic regression domain.

The ideal initialisation of random programs in a many to one genotype to phenotype situation (such as GP) presents a challenge when performing uniform population initialisation. Unlike other forms of evolutionary computation, GP relies on the execution (or interpreted execution) of programs in order to attain fitness values (although some work has included program structure as a factor, for example, Punch et al. [1996], Daida et al. [2003], Gustafson et al. [2005]). In terms of creating random programs to seed a GP run, the fact that fitness is based on the execution of the program means that a semantically diverse starting population should be investigated, rather than a starting population that is only syntactically diverse. It seems reasonable that this would increase the search power of GP because the initialised programs would have a greater range in the semantic search space.

In order to test the theory that increased semantic diversity affects GP results, an entirely Semantically Driven Initialisation (SDI) algorithm (built on ideas developed from analysis

of the traditional ramped half and half (RHH) technique created by Koza [1992]) has been developed, which produces one hundred percent effective code (Nordin et al. [1995]) at initialisation. In addition to the performance results generated when using this algorithm, the size and shape metrics of the programs produced by SDI are compared to the RHH technique over a range of benchmark problems. In a further set of experiments, the SDI algorithm is hybridised with the existing FULL (Koza [1992]) initialisation technique. When compared to the results produced by the RHH and SDI techniques, the experiments presented demonstrate that full semantic diversity is not the only major influence on program initialisation for GP.

Finally, a set of experiments is devised in order to test the importance of the role of program structure during initialisation. The experiment compares programs constructed using an existing algorithm and then semantically prunes them to result in behaviourally equivalent programs of a different structure containing no introns.

At the time of writing, initialisation in GP is an area that has received relatively little research attention in comparison to other aspects such as crossover. The contribution made by this work is formally examine semantic diversity and program structure at initialisation and attempt to understand the reasons these two factors affect GP performance and program metrics.

Section 5.1 presents the algorithms used for program initialisation. Section 5.2 presents the results, which include unique behaviour analysis, program bias analysis, program metrics analysis, GP performance results and evolvable shape experiments over a range of benchmark problems. Section 5.3 presents a discussion of the results and in section 5.4 the conclusions are presented. Section 5.5 discusses several avenues for future related work.

5.1 Methods and Algorithms

In order to test the theory that increasing the semantic diversity of the starting population will increase GP performance, two algorithms have been developed. The first of these is the *Semantically Driven Initialisation* (SDI) algorithm. This algorithm builds starting populations purely from a combination of semantically unique programs. All programs produced by SDI are semantically distinct and one hundred percent effective code. The second algorithm is

Algorithm 5.1 Semantically Driven Initialisation

Phase 1:
for each *Terminal* in *Terminal_Set*
    create *Representation* of *Terminal*
    add to *Representation_Store*
end for each
Phase 2:
while *Representation_Store_size* < *population_size*
    choose a *function* from the *Function_Set* (uniform probability)
    choose *function_arity Representations* from the *Representation_Store*
        at random (uniform probability)
    apply the *function* to the *function_arity Representations* at tree root
    if *resulting_Representation* is a new behaviour and not a constant behaviour
        add *resulting_Representation* to *Representation_Store*
    end if
end while
Phase 3:
for each *Representation* in *Representation_Store*
    translate *Representation* to *program*
    add to *First_Generation*
end for each

the *Hybridised Semantically Driven Initialisation* (HSDI) algorithm. Again, this algorithm will produce a semantically distinct starting population, however, it is seeded using the existing FULL (Koza [1992]) algorithm.

In order to test the importance of program structure during initialisation, a further two algorithms have been developed. The first is a modified version of the FULL algorithm (known as MODFULL) which will not produce *constant behaviours* (tautologies, contradictions, no move ants, constants in regression). MODFULL will produce "fat" programs with unreachable and redundant code present. To produce the contrasting smaller "thin" programs with fully effective code, semantic pruning is applied to the resulting programs of the MODFULL algorithm. This combination of the MODFULL algorithm and semantic pruning is referred to as WASHED initialisation.

The ability to reduce programs to canonical representation and back translate them to syntax is as described in section 4.3 and the nine experiments used are as set out in section 4.1. The parameters used are as set out in section 4.2 unless otherwise stated.

5.1.1 Semantically Driven Initialisation

Pseudo code for the SDI algorithm is presented in algorithm 5.1. In phase 1 the terminals are translated to behavioural representations and added to the representation store. This is required as at least one example of each input variable is needed to be present as building blocks for phase 2. Without any representations in the representation store from phase 1, phase 2 would fail as it would have nothing to combine using the functions. Phase 2 starts to combine the terminals (or single variable representations) using functions: when a semantically unique representation is produced, it is saved in the representation store. As phase 2 continues, the algorithm is able to take advantage of all of the representations held in the representation store and this encourages more complex behaviour to be generated. Phase 3 translates the representations back to syntax trees. One other important factor is that this algorithm will not produce constant behaviours.

Close observers of the SDI algorithm will notice that there is no syntax that will result in the guarantee of termination. Termination in this case occurs because the behavioural search space is larger than the number of programs being initialised. In GP, it would be very unlikely that a practitioner would be trying to evolve a solution when they could generate every possible behaviour in a reasonable amount of time, otherwise it would defeat the point of using GP to solve a particular problem. Theoretically, for small finite problems (such as a 3 bit multiplexer), if the SDI was asked to generate a population of more than 254 behaviours (256 in total minus the tautology and contradiction), the algorithm would not terminate.

There are underlying differences between the three problem domains. The ant domain is toroidal, and therefore potentially infinite. When a domain is infinite, it is necessary to constrain the size. This has been achieved by constraining the behaviour, whereas in previous work (Koza [1992]) this has been done by constraining the syntax. Furthermore, whilst programs in the Boolean domain would be run only once, in the ant domain the program is executed repeatedly up to a limit of 600 time steps. As a result of both the toroidal nature and the repeated executions, a behavioural size limit of ten moves (chosen as an arbitrary reasonable value) has been applied to enforce a syntactic size limit. This limit was set so that if the function PROGN3 is used, it allows enough moves to traverse the grid in one execution.

Algorithm 5.2 *Hybridised Semantically Driven Initialisation*

Phase 1:
Generate *FULL_First_Generation (depth 4)*
for each *program* in *FULL_First_Generation*
  translate *program* into *representation*
  if *representation* is not a constant behaviour and not in *Representation_Store*
    add *representation* to *Representation_Store*
  end if
end for each
Phase 2:
while *Representation_Store_size* < *population_size*
  choose a *function* from the *Function_Set* (uniform probability)
  choose *function_arity Representations* from the *Representation_Store*
    at random (uniform probability)
  apply the *function* to the *function_arity Representations* at tree root
  if *resulting_Representation* is a new behaviour and not a constant behaviour
    add *resulting_Representation* to *Representation_Store*
  end if
end while
Phase 3:
for each *Representation* in *Representation_Store*
  translate *Representation* to *program*
  add to *First_Generation*
end for each

5.1.2 Hybridised SDI

In addition to the SDI algorithm, a hybridised version of the algorithm (HSDI) was developed. The objective of the hybridised version of the algorithm was to combine behaviours both in the simplistic and complex areas of the search space, aiding a wider search. The pseudo code for the HSDI is displayed in algorithm 5.2.

In the hybridised version of the SDI, phase 1 of the initialisation algorithm is altered in comparison to the SDI algorithm. Instead of using representations of terminals as the initial seed to build on, the existing FULL (Koza [1992]) algorithm is used to create the first round of representations. The FULL algorithm is used because of the increased semantic diversity it provides (shown in figure 5.2 and table 5.2) and the fact that because the programs are converted into representations the shape of the trees is unimportant to the outcome of this algorithm. Upon creation of each FULL program, the representation is stored, if and only if, it is unique and is not a constant behaviour. Section 5.2.2 shows results to demonstrate the current level of unique behaviours within different starting populations using the RHH and FULL techniques. Once this seed is complete, the HSDI algorithm continues in the same

Algorithm 5.3 The MODFULL Algorithm with Additional WASHED Section.

```
while first_Gen_Size < pop_Size
   generate FULL_Program
   generate FULL_Program_Representation
   if FULL_Program_Representation is NOT a tautology or contradiction
      add FULL_Program to first_Gen
   end if
end while
—Additional code to reduce the programs produced above for "Washed" code.—
for each program in first_Gen
   translate program to representation
   back-translate representation to reduced_program
   replace program with reduced_program in population
end for each
```

way as the SDI algorithm and combines behaviours at the root, therefore encouraging more complex behaviour. In phase 3, the behaviours are translated back to one hundred percent effective syntax.

In the hybridised version of the artificial ant problem, phase 3 can become problematic due to the fact that the abstract representation can reduce branches of the IF-FOOD-AHEAD statement to nothing (for example if it contained a turn left and then a turn right instruction). As mentioned in chapter 4, an addition has been made to the ant syntax which can only occur in back translation. This addition is the SKIP operation. This has no effect on the ant apart from costing it one move. The move cost is required because some IF structures result in the ant always falling into the SKIP function when it is executed.

5.1.3 Evolvable Structure Analysis Algorithms

In order to evaluate how program structure affects GP performance, the MODFULL and WASHED algorithms are used to provide contrasting tree shapes. The MODFULL and WASHED algorithms are presented in algorithm 5.3.

The MODFULL algorithm is based on the FULL algorithm (presented by Koza [1992]) with a maximum depth of four. The notable difference is that the MODFULL algorithm will not produce a constant behaviour. The reason for this is that, if a program is reduced semantically to a constant behaviour, it cannot always be translated back to syntactic form. The reason for developing the MODFULL algorithm is to compare equivalent trees that are composed of different structures. As the WASHED algorithm is an extension of the MODFULL

Experiment	SDI	HSDI	RHH
4PAR	161	203	87
5MAJ	243	279	89
6MUX	414	415	87
7PAR	650	614	90
9MAJ	1885	1516	87
11MUX	5581	3882	86
AASF	548	625	129
SYM_REG	123	108	58

Table 5.1: One hundred populations of size five hundred are initialised using SDI, HSDI and the RHH techniques. The times quoted are in milliseconds and represent the average number of milliseconds taken for one initialisation of a population to take place.

algorithm and translates representation to syntax, constant behaviours cannot be present, otherwise it would not be possible to translate from representation to code.

The key differences between the the first generation produced by the algorithms are two-fold. Firstly, by applying the WASHED algorithm all introns are removed from the starting population and in conjunction with this there are no constant behaviours either. Secondly, the programs produced by MODFULL and WASHED are structured distinctly, yet are semantically equivalent.

5.2 Results

Results are presented for the experiment suite presented in section 4.1, which analyse different characteristics of the GP run by comparing different initialisation methods. These experiments include a speed comparison, analysis of unique behaviours, behavioural bias analysis, size and shape analysis, a comparison of overall GP performance, and a comparison of performance using program structures at initialisation.

5.2.1 Speed Comparison of Initialisation Methods

To address initial fears that these new algorithms might take impractical amounts of computation time, a comparison of speed of initialisation is presented showing the time it takes to initialise a starting population using SDI, HSDI and the traditional RHH technique.

Whilst table 5.1 shows that the RHH technique takes less time to generate programs, this experiment does not give any indication of the comparable quality of the resulting programs.

Considering that the results quoted are in milliseconds, it is reasonable to use semantically driven initialisation in order to attain a semantically diverse starting population, as even the slowest initialisation is only six seconds.

A second aspect of this experiment which is not comparable is that RHH has a built in depth limit and, as such, cannot build programs greater than that depth. As a result of these depth limitations we know that the average depth of programs will fall in the range of two to six, therefore limiting the size of the programs generated. The SDI and HSDI algorithms do not use a depth limit[1] to constrain the size of the programs. Therefore, it seems reasonable to expect that the process might take longer as given that the programs generated by the SDI and HSDI algorithms only represent the behaviour in the form of effective code, the SDI and HSDI are having to do more work in terms of code generation, and the size and shape results in section five support this. The difference in speed of generation is not considered to be a significant factor in choosing between these different methods as the difference is comparable in seconds and these algorithms are being used in this work to test and evaluate theories only.

5.2.2 Behaviour in Starting Populations

Analysis of Unique Behaviours

The behavioural representations set out in section 4.3 are used to analyse GP starting populations. Given a starting population, each program in the population is transformed into behavioural model form. The number of unique behaviours in the population is enumerated by testing for model equivalence. In addition to testing for representational equivalence, the number of programs associated with a specific behaviour is calculated to analyse any bias towards specific behaviours. In these experiments, one hundred populations at each population size are initialised and all results reported are averages of these one hundred initialisations.

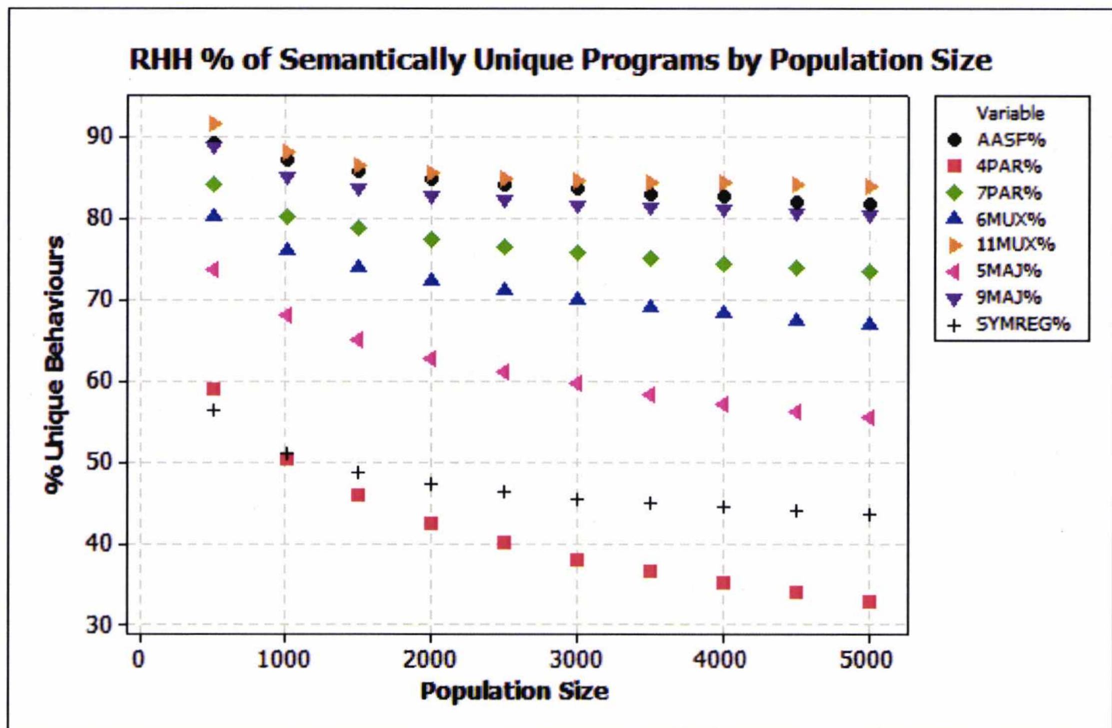


Figure 5.1: Enumeration of unique behaviours present in starting populations expressed as a percentage of the total population size. SYMREG% represents experiments in the symbolic regression domain because the same functions and terminals are used for initialisation in both symbolic regression experiments. This analysis is repeated for population sizes ranging from 500-5000 for each experiment. All results quoted are an average of one hundred initialisations.

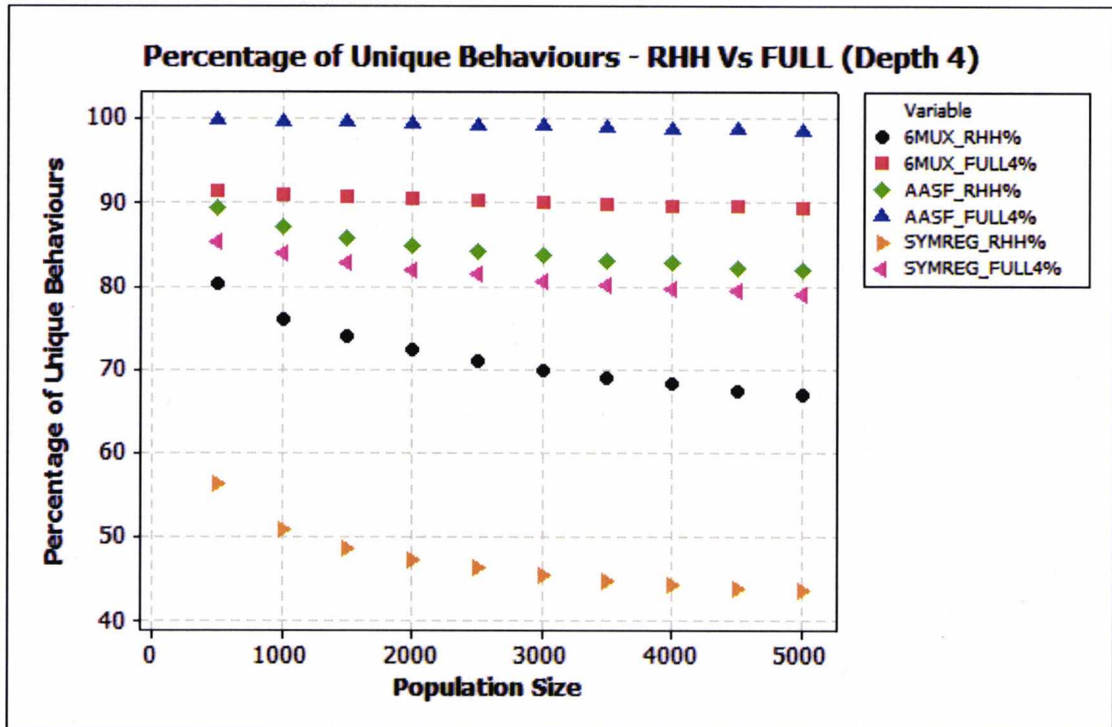


Figure 5.2: The figure shows the percentage of semantically unique programs generated by the FULL (depth 4) and RHH techniques against population size for the 6MUX, AASF and CUBIC problems.

Unique Behaviours

Figure 5.1 shows that in every model, there is a notable percentage of duplicated behaviours. The 4PAR experiment represents the worst performing result in terms of the number of unique behaviours: when the population increases past 2500, less than 40% of the programs produced by RHH are semantically unique. Furthermore, the 11MUX experiment demonstrated the least duplication of behaviours with a maximum of 15%. One feature applicable to all models is that, at different levels, all percentages of unique programs decrease as the population increases. This could indicate a type of bias such that some behaviours are favoured and repeatedly produced by the RHH. The final feature of note is that as the number of terminals increases for the Boolean domain, there is less duplication of behaviour.

Figure 5.2 shows a sample of three of the test problems (one from each domain considered) comparing the semantic diversity when using the FULL (depth 4) and RHH techniques. In two out of three cases, the FULL technique generated more semantic diversity overall

[1] Except as already explained for the artificial ant SDI with the behavioural building block size of ten moves.

Problem	Experiment	Mean Unique Behaviours	PT
6MUX	RHH	1911.86 ±987.11	-
	FULL4	2471.93 ±1347.99	0.001
AASF	RHH	2292.90 ±1225.27	-
	FULL4	2719.33 ±1488.46	0.001
CUBIC	RHH	1248.73 ±635.58	-
	FULL4	2210.75 ±1181.79	0.000

Table 5.2: The table shows statistical tests comparing the number of unique behaviours generated by the RHH and FULL (depth 4) algorithms. Problem indicates the problem considered, experiment indicates the initialisation method used, Mean Unique Behaviours indicates the average number of unique behaviours $\pm$ the standard deviation. PT indicates the P-Value of a Paired T-test comparing the numbers of unique behaviours produced by the RHH and FULL4 algorithms. In the case of significance at the 95% confidence level, the result is aligned with the most diverse starting population.

(for the range of populations sizes presented) and paired T-tests (table 5.2) revealed that this difference is significant at the 99% confidence levels for the 6MUX, AASF and CUBIC problems.

This is an interesting result as previous authors Koza [1992] have reported that FULL does not perform as well as RHH in terms of GP performance. This result lends weight to the theory that there is an ideal evolvable shape of program tree. In the context of the HSDI algorithm, if a semantically diverse seeding mechanism is required and the shape of the programs is not important (because they will be modelled as behaviour), then the FULL algorithm would be ideal to seed the HSDI.

5.2.3 Bias Analysis

In a second experiment, aimed at analysing any bias present in an initialised population, 100 populations of size 1000 are initialised. Every behaviour produced by the 100,000 initialised programs is recorded and if behaviours are produced multiple times, a record of the frequency is saved.

Table 5.3 shows that for the initialisation of the populations considering 4PAR experiment, the RHH favours simplistic behaviours with the readings at rank one and two being the contradiction and tautology. As explained in section 4.3.1, tautologies and contradictions are a constant behaviour and their result does not depend on the input value of any of the variables (or terminals in the GP context). In every initialisation of population size 1000, an

4PAR			
Rank	Frequency	Node Count	Sat Count
1	42.25	0	0
2	40.94	0	1
3	18.59	1	0.5
4	18.32	1	0.5
5	18.14	1	0.5
6	18	1	0.5
7	10.7	2	0.25
8	10.55	2	0.75
9	10.53	2	0.75
10	10.5	2	0.25

7PAR			
Rank	Frequency	Node Count	Sat Count
1	17.52	0	1
2	17.05	0	0
3	7.21	1	0.5
4	7.05	1	0.5
5	7.02	1	0.5
6	6.69	1	0.5
7	6.67	1	0.5
8	6.66	1	0.5
9	6.47	1	0.5
10	3.29	2	0.75

6MUX			
Rank	Frequency	Node Count	Sat Count
1	21.78	0	1
2	21.68	0	0
3	9.15	1	0.5
4	8.93	1	0.5
5	8.85	1	0.5
6	8.7	1	0.5
7	8.65	1	0.5
8	8.64	1	0.5
9	4.13	2	0.25
10	4.1	2	0.25

11MUX			
Rank	Frequency	Node Count	Sat Count
1	8.62	0	1
2	7.67	0	0
3	3.6	1	0.5
4	3.59	1	0.5
5	3.54	1	0.5
6	3.54	1	0.5
7	3.53	1	0.5
8	3.52	1	0.5
9	3.47	1	0.5
10	3.42	1	0.5

Table 5.3: Bias results for the parity and multiplexer models. Rank indicates the relative frequency with one being the most frequent behaviour. The frequency is the total number of occurrences of this behaviour divided by 100 which indicates the number of times this behaviour is expected to occur per initialisation. Node count and sat count are as explained in section 4.3.1.

average of 40.94 tautologies and 42.25 contradictions occur. This results in 8.32% (combination of tautology and contradiction) of the programs generated in each initialisation not depending any any of the input variables.

Moreover, none of the behaviours in the ten most frequent behaviours contain all the terminals and therefore these behaviours represent partially blind candidate programs. Behaviours in ranks three to six represent single terminals with or without the possibility of a NOT function and the behaviours in ranks seven to ten represent simple AND or OR functionality. The 46th ranked most frequent record (with a frequency of 2.92) is the first record which has a node count of four. This indicates that behaviours that use four inputs (and possibly all the terminals) are being infrequently created when compared to the simplistic structures we see in table 5.3.

In table 5.3, the 7PAR results are similar to that of the 4PAR results. Again, the constant behaviour states feature at ranks one and two, and single terminals at positions three to nine and simplistic OR functionality in position ten. Whilst the behavioural structures are similar to the 4PAR experiment, the frequencies are slightly reduced such that tautologies only represent 5.78% of the behaviours generated in an average initialisation. The introduction of more terminals adds but a little more diversity to creation of behaviours using the RHH technique. It is not until the 1515th ranked most frequent behaviour with a frequency of 0.5 occurrences per initialisation, that we see a node count of seven for the first time. Again, this indicates a bias towards simplistic behaviours being generated by the RHH technique.

In table 5.3, the 6MUX and 11MUX experiments show similar results to that of the even parity experiments. Again, the tautology and contradiction states feature as the two most frequently constructed behaviours. These are followed by single terminals, and then simple two terminal structures. As the number of terminals in the problem increases, the chances of constructing a behaviour with all terminals present becomes even worse.

The first occurrence of a behaviour with a node count of six (for the 6MUX problem) is ranked 559th, with a frequency of 0.13 occurrence per initialisation. This information is worth considering as the 6MUX problem frequently has its population size parameter set at 500. Therefore, considering an average initialisation, the population is unlikely to contain one candidate program which has all terminals present in the behaviour.

Table 5.3 shows that the bias results for the 11MUX experiment have similar characte-

ristics to the other Boolean models: the increase in terminals results in a decrease in the frequency of behaviours which use all inputs. In this analysis, it was not until the 345th ranked most frequent program (with a frequency of 0.16) was reached until only three nodes were used to create a behaviour.

In keeping with the other Boolean domain experiments, table 5.4 shows that the 5MAJ and 9MAJ experiments suffer a similar bias. In the case of the 5MAJ experiment, it was the 150th most common behaviour with a frequency of 0.67 when a node count of 5 was first achieved. In the case of the 9MAJ experiment, it was the 2614th most frequent behaviour before a node count of 9 was achieved.

The AASF results in table 5.4 exhibit similar simplistic behaviours, albeit not in the same way as the problems in the Boolean domain. The most frequent structures assembled by the RHH technique are simple one move or turn structures. It is not until the 6th most frequent reading that a behaviour contains two operations. It is not crucial to have behaviours with large numbers of moves (because of re-execution of the ant control code), but in order to achieve full score, the ant will have to have a behaviour containing several moves and turns. To put this into perspective, it is not until the 197th most frequent reading (0.32 frequency) when five moves are first accomplished.

Both the CUBIC and QUART experiments use the same function and terminal set so only a single bias analysis has been carried out in table 5.4. In keeping with the other results obtained, the most frequent behaviours are all simplistic. All of the top ten most frequent behaviours are constant behaviours not dependant on any variables. It is not until the 18th most frequent behaviour with a frequency of 5.24 that the first non constant behaviour is present in the initialised population.

If one considers the distribution of behaviours in the search space, behaviours that include all of the variables will compose the majority of the search space in comparison to behaviours that do not rely on all of the input variables. Therefore, the level of bias to achieve the favouring of constant behaviours presented in tables 5.3 and 5.4 is strong enough to bias to the distribution of behaviours to a very simplistic area of the search space.

5MAJ			
Rank	Frequency	Node Count	Sat Count
1	29.88	0	1
2	29.11	0	0
3	12.73	1	0.5
4	12.18	1	0.5
5	11.99	1	0.5
6	11.73	1	0.5
7	11.31	1	0.5
8	6.2	2	0.25
9	6.08	2	0.75
10	6.04	2	0.75

9MAJ			
Rank	Frequency	Node Count	Sat Count
1	11.26	0	0
2	10.61	0	1
3	5.01	1	0.5
4	4.88	1	0.5
5	4.84	1	0.5
6	4.76	1	0.5
7	4.6	1	0.5
8	4.57	1	0.5
9	4.46	1	0.5
10	4.42	1	0.5

Regression			
Rank	Frequency	Terms	Constant
1	125.1	1	T
2	25.67	1	T
3	24.6	1	T
4	21.21	1	T
5	20.66	1	T
6	19	1	T
7	18.5	1	T
8	18.45	1	T
9	18.32	1	T
10	15.93	1	T

AASF			
Rank	Frequency	Move Count	Final Orientation
1	11.11	1	E
2	9.82	0	S
3	9.70	0	N
4	7.97	0	W
5	7.93	0	E
6	5.48	2	E
7	5.19	1	S
8	5.14	1	N
9	5.13	1	N
10	4.96	1	S

Table 5.4: Bias results for the artificial ant and majority models. Rank indicates the relative frequency with one being the most frequent behaviour. The frequency is the total number of occurrences of this behaviour divided by 100 which indicates the number of times this behaviour is expected to occur per initialisation. Node count and sat count are as explained in section 4.3.1. For the AASF table, the move count represents the number of moves in the behaviour and the final orientation is the direction the ant is facing on the grid after the moves have taken place. For the regression table, Terms indicates the number of terms present in the reduced formula and Constant indicates whether any variables are present. T = true (F = false), which implies the reduced formula is a constant and therefore, no variables are present.

Discussion

Both the analysis of unique behaviours present in starting populations, and the focused bias analysis for each model have revealed several biased features of the output of the RHH initialisation technique.

Despite preventing syntactically identical code being produced, there are still notable quantities of duplicated behaviours present in the starting population. Further bias analysis of all problem domains considered revealed that the RHH technique favours simplistic behaviours or *constant behaviours*. Constant behaviours represent the RHH's inability to create building blocks dependent on terminal values and its ability to produce ineffective code, as it is not possible to directly construct true or false with the syntax available. The fact that they are the most frequent behaviours in the Boolean problems indicates that this is the main weakness of using the RHH technique as an initialisation method.

When considering the ant domain, a similar bias in the form of a constant behaviour is present. Unfortunately, this characteristic is the second to the fifth most produced behaviour by the RHH technique generating syntax in the artificial ant domain. In the symbolic regression domain, out of the top ten most frequently produced behaviours, all of them were constant behaviours. Across all three domains, this simplistic bias during RHH initialisation is present.

A final aspect of concern is the apparent inability of the RHH to construct more complex behaviour. In the case of the Boolean domain, the low rank of the first occurrences of candidate behaviours with node counts capable of representing all inputs present were noted. With increasing numbers of terminals, it was harder for the RHH to generate behaviours that used all the inputs. This effect is not limited to the Boolean domain. When considering the ant domain, it was not until the 197th most frequent behaviour (with frequency of 0.1) that the RHH achieved an ant that was capable of moving five positions. This situation is repeated in the symbolic regression domain and it is not until the 33rd most frequent behaviour that two terms are present in the reduced regression formula.

An advantage of the SDI algorithm is that it knows when behaviours reduce to the constant behaviour, and therefore these behaviours can be removed. In addition the SDI will prevent bias in behaviours because it can enforce semantically the uniqueness of programs in the population.

4PAR	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	3.8276	13.7575	6.7903	6.9672	3.8385	3.5943
RHH	3.6121	29.9810	14.4263	15.5546	3.2304	8.3002
HSDI	3.5902	12.003	5.9322	6.0710	3.6950	3.3433

7PAR	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	7.3996	46.351	23.302	23.049	6.1663	6.2640
RHH	3.2604	28.197	13.537	14.661	4.3065	8.6483
HSDI	6.1480	32.348	16.230	16.118	5.5344	5.2615

6MUX	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	6.2213	31.8924	15.9970	15.8954	5.5051	5.1263
RHH	3.3571	28.5263	13.7093	14.8169	4.0049	8.4970
HSDI	5.2024	22.811	11.429	11.382	4.9242	4.3847

11MUX	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	14.2564	203.931	102.762	101.168	8.7484	14.3045
RHH	2.9887	27.199	13.041	14.158	5.2075	9.1006
HSDI	10.253	107.91	53.994	53.916	7.5249	10.5247

Table 5.5: Size and shape comparisons. SDI represents readings produced by semantically driven initialisation. RHH represents readings produced by the ramped half and half technique. HSDI represents reading produced by the hybrid SDI. All readings quoted are averages of 100 runs of 1000 population size.

5.2.4 Size and Shape Analysis

Analysis of Size and Shape of Programs

As previously mentioned there is no size or depth limit on programs produced using SDI. This section aims to present an analysis and comparison of the size and shapes of programs produced by the RHH, SDI and HSDI algorithms. 100 populations of size 1000 are initialised for each problem. The results are averaged over the 100 initialisations and the metrics examined are program depth, program length (total number of nodes in the tree), number of functions, number of terminals and the number of distinct terminals.

Results

Table 5.5 shows size and shape analysis from the parity and multiplexer experiments. Two sample T-tests revealed that the length and depth metrics of the SDI and HSDI are significantly different compared to the length and depth metrics of the RHH initialisation at the 99% confidence level. The two most notable points that apply to all but the 11MUX are that the SDI produces deeper, thinner trees, compared to the RHH technique; and that the SDI increases the numbers of distinct terminals in all cases. The HSDI falls somewhere between the SDI and RHH extremes. In problems such as the 11MUX, this is useful as it creates smaller programs that are less likely to grow beyond the crossover depth cap (Koza [1992]) (in this case 17) during evolution as a result of the bloat phenomenon (Banzhaf et al. [1998], Luke [2003], Poli et al. [2007], Dignum and Poli [2007]).

Table 5.6 shows the results for the majority, ant and symbolic regression experiments. Two sample T-tests revealed that the lengths and depths of the SDI and HSDI initialisations are significantly different when compared to those of the RHH initialisation at the 99% confidence level. The majority experiments reflect the multiplexer and parity results in that the SDI produces deeper, thinner trees, except for the larger 9MAJ problem and the 11MUX. The 11MUX and 9MAJ results show a large increase in the size (all metrics) of the programs produced. This relates to the earlier discussion in section 5.1 regarding the limitations of generating syntax directly from behaviour. As there are no size checks in the SDI, it will produce syntax to represent the behavioural complexity of the problem faced. In the majority experiments (table 5.6) the SDI and HSDI consistently produce more distinct terminals during the initialisation which indicates a better ability for programs to deal with all possible inputs presented. Similar characteristics of size and shape are shown in the ant domain. Deeper thinner programs are produced by the SDI and HSDI compared to the RHH technique and both the SDI and HSDI produce a statistically higher level of distinct terminals. The symbolic regression domain is somewhat closer to the 11MUX. The SDI produces large programs near the depth cap (17), however, the HSDI produces programs that are significantly thinner in tree shape than the RHH initialisation. Whilst the SDI uses a larger number of distinct terminals in the symbolic regression domain, the HSDI uses the least. One speculative explanation for this is that the HSDI simplifies constants present in the program tree, therefore causing less distinct terminals to be present when programs are reduced from their FULL

5MAJ	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	4.9326	20.6616	10.3361	10.3255	4.6724	4.1888
RHH	3.4762	29.1390	14.0198	15.1193	3.6557	8.3824
HSDI	4.3329	16.141	8.0663	8.0745	4.3020	3.7252

9MAJ	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	10.5130	99.9944	50.3765	49.6179	7.5707	9.5115
RHH	3.1005	27.4803	13.1808	14.2995	4.7949	8.8632
HSDI	8.1057	61.135	30.632	30.503	6.6202	7.5422

AASF	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	6.3281	49.542	19.486	30.056	2.9631	7.8289
RHH	3.7340	56.369	23.718	32.652	2.7955	15.0961
HSDI	5.4073	37.788	15.392	21.867	2.9906	6.9883

SYMREG	Depth	Length	Functions	Terminals	Distinct Terminals	Length/Depth
SDI	16.091	146.79	72.89	73.89	12.729	9.1225
RHH	2.8784	26.692	12.846	13.846	5.4836	9.2732
HSDI	2.6590	8.4406	3.7203	4.7203	3.2643	3.1744

Table 5.6: Size and shape comparisons. SDI represents readings produced by semantically driven initialisation. RHH represents readings produced by the ramped half and half technique. HSDI represents reading produced by the hybrid SDI. All readings quoted are averages of 100 runs of 1000 population size. SYMREG represents one test for both the symbolic regression problems as they use the same function and terminal set.

program form.

One final point to mention is the intron wash effect the HSDI will have on the populations it produces. It is seeded from the FULL method, but generated from behavioural representation of that code, so that it produces effective code, which, from studying the size and shape analysis, is supported by the consistently low length/depth values in tables 5.5 and 5.6 (with the exception of the 11MUX).

Discussion

One global feature of the size and shape experiments is that no matter which problem domain is being considered, the RHH always produces roughly similar sized programs, whereas the SDI produces different sized programs depending on the problem. It is reasonable to assume that this happens because different problem domains have different behavioural requirements, therefore this would result in different size and shape characteristics of programs.

In addition to this, as the number of terminals increases, the potential combinations of behaviour will increase, causing deeper programs that require more functions and terminals to attain specific behaviours. This is borne out in the results for the Boolean domain, where the increase in program sizes and depths is in line with the increase in the numbers of terminals present in our experiments.

The second point to be made when studying these results is the level of importance that should be given to the measurement of the depth of the tree. The majority of the SDI and HSDI results produced trees of greater depth than the RHH (but generally shorter in terms of length) and this would be consistent with using composite functions (trees of nested simple functions) to model specific behaviours. The length metric may be a better measure to use, in terms of flexibility, as it gives programs the opportunity to develop complex behaviour as well as controlling the overall size of the program.

The third observation is that in all experiments the SDI produced significantly more distinct terminals than RHH. Statistically, there are fewer possible behaviours that can be generated in the Boolean domain when not all the terminals are used. Once these are generated, the SDI has to generate more complex behaviour to retain the semantic variety it promises. As a result of this, the SDI will automatically have a tendency towards producing programs

with all the terminals present, especially in larger populations.

5.2.5 GP Performance Results

In the experiments presented in this section, the performance of GP runs that are initialised using the SDI, HSDI and RHH algorithms is compared. The parameters are as set out in section 4.2 with exception to the initialisation algorithms.

Table 5.7 shows that the performance of the RHH, SDI and HSDI techniques vary depending on the problem being analysed. In terms of maximum scores over each generation, the HSDI performs best in four experiments, the SDI in three and the RHH in two experiments (at the 95% confidence level). Three experiments present statistically similar results overall. This raises two complex questions about how the dynamics of creating the starting population can impact on the performance of GP runs.

The first issue concerns the way in which initialised programs are distributed in relation to the search space for a particular problem. The SDI performs well on the parity and 6MUX experiments, but poorly on the 11MUX experiment, when compared to the RHH. Further complicating the matter is the result for the CUBIC and QUART experiments, where the SDI is the statistically similar to RHH result overall for the CUBIC and QUART, but achieves better maximum scores at generation 50.

A simplistic theory is that the SDI produces too many complex behaviours in the wrong region of the search space when the ideal program may be a simplistic program in another region of the search space. This might explain why, for example, the SDI performs well at the 6MUX experiment, but poorly at the 5MAJ experiment. It might also explain how the HSDI is able perform well on both multiplexers, as it is seeded with simplistic behaviour, but can build more complex behaviour on top of this.

The second issue is the effect of initialising one hundred percent effective code compared to code with redundant and unreachable statements. The introduction of the SKIP operation was necessary in the HSDI applied to the artificial ant problem, to alleviate the problem whereby RHH and FULL produce dead branches for the IF-FOOD-AHEAD statement. Given the performance results shown in table 5.7, this SKIP statement is clearly required.

The obvious comparison to this work is that of Looks [2007] which presented results for a selection of multiplexer and parity experiments. Whilst Looks examines similar problems,

Problem	Experiment	Max Score Overall	RO	Max Score G50	RG50	Success
4PAR	SDI	**0.0500 (±0.0542)**	**1**	**0.0200 (±0.0396)**	**1**	77% G3
	HSDI	**0.0601 (±0.0590)**	**1**	**0.0238 (±0.0477)**	**1**	77% G6
	RHH	0.0949 (±0.0735)	3	0.0425 (±0.0494)	3	50% G9
5MAJ	SDI	0.0549 (±0.0363)	1	0.0300 (±0.0266)	3	32% G9
	HSDI	0.0467 (±0.0356)	1	**0.0213 (±0.0250)**	**1**	50% G9
	RHH	0.0427 (±0.0394)	1	**0.0188 (±0.0231)**	**1**	54% G10
6MUX	SDI	**0.0287 (±0.0534)**	**1**	**0.0052 (±0.0232)**	**1**	95% G4
	HSDI	**0.0212 (±0.0506)**	**1**	**0.0003 (±0.0031)**	**1**	99% G3
	RHH	0.0763 (±0.0498)	3	0.0431 (±0.0529)	3	51% G6
7PAR	SDI	**0.2966 (±0.0822)**	**1**	**0.1745 (±0.0388)**	**1**	0% –
	HSDI	0.3176 (±0.0775)	2*	0.2037 (±0.0361)	2	0% –
	RHH	0.3452 (±0.0621)	3	0.2582 (±0.0297)	3	0% –
9MAJ	SDI	0.1783 (±0.0379)	2	0.1338 (±0.0120)	3	0% –
	HSDI	0.1657 (±0.0381)	2	0.1202 (±0.0124)	2	0% –
	RHH	**0.1228 (±0.0467)**	**1**	**0.0722 (±0.0118)**	**1**	0% –
11MUX	SDI	0.1411 (±0.0790)	3	0.0755 (±0.0660)	3	22% G25
	HSDI	**0.0992 (±0.0848)**	**1**	**0.0377 (±0.0441)**	**1**	45% G16
	RHH	**0.1019 (±0.0906)**	**1**	**0.0339 (±0.0401)**	**1**	45% G15
AASF	SDI	0.3220 (±0.0547)	3	0.2781 (±0.0855)	1	0% –
	HSDI	**0.2889 (±0.0639)**	**1**	0.2407 (±0.1187)	1	10% G7
	RHH	0.3177 (±0.0812)	2*	0.2579 (±0.1299)	1	11% G5
CUBIC	SDI	788.9 (±287.7)	1	**539.3 (±261.4)**	**1**	0% –
	HSDI	850.8 (±181.9)	1	661.8 (±256.0)	3	0% –
	RHH	816.3 (±287.3)	1	**498.6 (±204.7)**	**1**	0% –
QUART	SDI	1114.2 (±587.9)	1	**517.8 (±358.7)**	**1**	0% –
	HSDI	1040.3 (±493.2)	1	625.7 (±298.8)	2	0% –
	RHH	1258.5 (±596.1)	1	711.9 (±311.5)	2	0% –

Table 5.7: Problem is the problem being analysed. Experiment shows the initialisation method. Max Score Overall shows the average of one hundred runs of standardised maximum scores for all generations. The scores have been normalised to be in the range 0-1 for easy comparison, except for the continuous regression domain which shows raw fitness. Standardised fitness is used so 0 is the best value. The ± values are the standard deviation of the maximum scores normalised to the 0-1 scale (except regression domain). RO shows the rank of the overall scores based on a one way ANOVA test conducted to the 95% confidence level. The individual ranks are based on results from a Tukey test comparing all three results. A rank of 2* indicates that the middle result is statistically equivalent to the best and worst results, however, the best and worst results are statistically different. Max Score G50 shows the average of maximum scores at generation 50 (the end of evolution) and RG50 shows the performance rank using a one way ANOVA and post hoc Tukey test on the generation 50 results. Success shows the percentage of runs that reach full score and the earliest generation that full scores is reached out of all 100 runs.

he uses a different function set and different parameters, and a different semantic sampling initialisation. Despite these differences, for the 6MUX and the 4PAR, there is still a similar relative change in performance based on semantic style sampling. Looks uses an even 5 parity experiment whereas a 7PAR experiment is used in this evaluation; given that these results support the value of semantic initialisation for parity problems overall, these GP performance results are in line with those of Looks. The 11MUX result is different to the results of Looks; a speculative reason is that this is because these experiments use no depth limits on the SDI, and as a result (supported by the size and shape results (table 5.5)) vastly increased program sizes are created changing the distribution of programs in the search space. By contrast, Looks does control the size of the programs he produces using the semantic sampling algorithm. As stated in the introduction to this chapter, the algorithms presented were designed to test theories in order to better understand the important issues in producing good quality starting populations.

Finally, there is an element of difference in program sizes when comparing the SDI, HSDI and RHH initialisation algorithms. These size and shape results (section 5.2.4) show that the SDI and HSDI algorithms produce programs at varying sizes depending on the problem. Some are smaller than the RHH output and some are larger. There is an argument to try to make the programs the same size for comparison, however, as the only control mechanism on the RHH algorithm is depth and section 5.2.4 supports that the HSDI and SDI tend to produce deep thin trees, the job of creating similar sized starting programs is very complex. As such, the traditional 2-6 depth range of the RHH algorithm has been used for comparison. Furthermore, crossover bias will quickly modify the size of the majority of the programs resulting in a change in the size distribution of programs. As a counter argument, given that semantic based initialisation results in starting programs of different sizes, crossover bias will occur using a different length distribution of programs compared to initialisation with RHH. This will affect the results of a GP run and makes it difficult to determine whether the effects of factors such as diversity and shape, if the mean size factor is not discounted. While distributions of functionality (and as a result fitness) reach a limit beyond a certain program size, there are potential program sizes which provide promising and poor functionality (and fitness) before that limit. The difficulty is establishing an initialisation method which can develop programs of the right size and or shape to assist evolution to a good solution.

Problem	Experiment	Max Score Overall	PT	Max Score G50	2T-50	Success
4PAR	WASHED	**0.0782 (±0.0631)**	**0.00**	0.0381 (±0.0554)	0.409	60% G6
	MODFULL	0.0923 (±0.0710)	-	0.0444 (±0.0513)	0.409	50% G9
5MAJ	WASHED	0.0503 (±0.0346)	-	0.0103 (±0.0167)	-	38% G9
	MODFULL	**0.0321 (±0.0373)**	**0.00**	**0.0253 (±0.0246)**	**0.00**	**70% G7**
6MUX	WASHED	**0.0235 (±0.0506)**	**0.00**	**0.0025 (±0.0152)**	**0.00**	**97% G4**
	MODFULL	0.0663 (±0.0577)	-	0.0270 (±0.0395)	-	60% G8
7PAR	WASHED	**0.3177 (±0.0784)**	**0.00**	**0.2025 (±0.0358)**	**0.00**	**0% –**
	MODFULL	0.3449 (±0.0687)	-	0.2491 (±0.0527)	-	0% –
9MAJ	WASHED	0.1656 (±0.0383)	-	0.1201 (±0.0117)	-	0% –
	MODFULL	**0.1177 (±0.0449)**	**0.00**	**0.0683 (±0.0083)**	**0.00**	**0% –**
11MUX	WASHED	**0.0971 (±0.0850)**	**0.00**	**0.0366 (±0.0505)**	**0.012**	**51% G18**
	MODFULL	0.1498 (±0.0879)	-	0.0529 (±0.0389)	-	13% G33
AASF	WASHED	0.3032 (±0.0625)	-	0.2563 (±0.1137)	0.359	8% G2
	MODFULL	**0.2968 (±0.0789)**	**0.011**	0.2413 (±0.1161)	0.359	10% G7
CUBIC	WASHED	935.74 (±141.02)	-	798.72 (±229.70)	-	0% –
	MODFULL	**799.14 (±250.15)**	**0.00**	**485.94 (±225.44)**	**0.00**	0% –
QUART	WASHED	1330.58 (±481.49)	-	932.18 (±307.98)	-	0% –
	MODFULL	**1200.72 (±533.65)**	**0.00**	**700.96 (±296.71)**	**0.00**	0% –

Table 5.8: Problem indicates the problem being analysed. Experiment represents the initialisation type (WASHED of MODFULL see 5.1.3). Max Score Overall is the average of the maximum scores ± the standard deviation of max scores. PT is a Paired T-test of the overall scores. 0.00 aligned with an experiment indicates a best result. Max Score G50 shows the average of the maximum scores at generation 50 and 2T-50 shows a 2 sample T test of these results. Success indicates the percentage of runs that reached full score at generation 50 and the first generation in which a single run attained full score.

5.2.6 Evolvable Shape

In order to examine evolvable shape, the MODFULL and WASHED algorithms (as set out in section 5.1.3) are used. The parameters are as stated in section 4.2. The MODFULL and WASHED algorithms are used at initialisation to examine the effect that changing the shape of programs without changing the semantics will have on GP performance. The MODFULL and WASHED algorithms are used because the data in tables 5.5 and 5.6 support that the semantically reduced code (WASHED) will provide a dramatic contrast in tree shape, whilst retaining the same behaviour as the MODFULL code.

Table 5.8 demonstrates that changing the shape of the initialised program trees can have a dramatic effect on the performance of GP runs. When considering the results, all experiments show a statistically significant difference between the MODFULL and WASHED algo-

rithms overall, and seven out of nine of the problems at the 95% confidence level at generation 50. However, whether MODFULL or WASHED gives a superior performance appears to be problem dependent. Overall, five of the experiments favour the MODFULL algorithm and four favour the WASHED algorithm. Even when comparing the initialised programs average lengths and depths (table 5.9) there appears to be no pattern linking a particular shape to the different results. Table 5.9 shows that the WASHED algorithm results in shorter and deeper trees trees in the majority of cases. The fact that average depth and length reading are statistically different at the 95% confidence level indicates that the WASHED algorithm is having a substantial effect on the composition of initialised programs. In the absence of any pattern, it is hard to establish the cause for the difference in performance other than a speculative reason that different problems may require different program structures in order to compose fit candidate solutions.

The other factor of note when considering the success rates on the experiments that do find ideal solutions is the level of the difference in success. The biggest difference is 37% in the case of the 6MUX. This shows the importance of program shape to evolvability in a GP run.

The AASF results statistically favour MODFULL overall, and is statistically similar at generation 50. This is surprising, as despite the size and shape results in tables 5.6 and 5.9. The back translation mechanism still constructs full trees (just smaller FULL trees) when it reassembles the ant movement instructions. The only difference is that redundant and unreachable code will have been removed in this reduced form. The fact the performance is statistically significant in favour of MODFULL for the artificial ant indicates that simply removing both unreachable and redundant introns does not improve performance overall.

5.3 Discussion

One of the main points to draw from this analysis, as well as the work of other authors (for example, Looks [2007], Luke and Panait [2001]) in the field, is that the choice of initialisation method may result in statistically significant variation in the performance of GP runs. In the context of this investigation, and the results presented in tables 5.7 and 5.8, it is clear that changing different aspects of program initialisation can have an impact on performance far

Problem	Experiment	Average Length	PL	Average Depth	PD
4PAR	WASHED	9.49 ($\pm$0.19)	0.00	3.11 ($\pm$0.05)	0.00
	MODFULL	31.48 ($\pm$0.61)	-	4.00 ($\pm$0.00)	-
5MAJ	WASHED	14.71 ($\pm$0.39)	0.00	4.08 ($\pm$0.08)	-
	MODFULL	31.35 ($\pm$0.55)	-	4.00 ($\pm$0.00)	0.00
6MUX	WASHED	21.78 ($\pm$0.62)	0.00	5.05 ($\pm$0.09)	-
	MODFULL	31.32 ($\pm$0.65)	-	4.00 ($\pm$0.00)	0.00
7PAR	WASHED	31.18 ($\pm$0.36)	0.00	6.02 ($\pm$0.04)	-
	MODFULL	31.38 ($\pm$0.24)	-	4.00 ($\pm$0.00)	0.00
9MAJ	WASHED	59.33 ($\pm$0.98)	-	7.99 ($\pm$0.07)	-
	MODFULL	31.29 ($\pm$0.22)	0.00	4.00 ($\pm$0.00)	0.00
11MUX	WASHED	104.25 ($\pm$1.90)	-	10.09 ($\pm$0.10)	-
	MODFULL	31.24 ($\pm$0.23)	0.00	4.00 ($\pm$0.00)	0.00
AASF	WASHED	37.84 ($\pm$0.48)	0.00	5.41 ($\pm$0.05)	-
	MODFULL	51.06 ($\pm$0.54)	-	4.00 ($\pm$0.00)	0.00
CUBIC	WASHED	6.24 ($\pm$0.13)	0.00	2.23 ($\pm$0.04)	0.00
	MODFULL	31.00 ($\pm$0.00)	-	4.00 ($\pm$0.00)	-
QUART	WASHED	6.23 ($\pm$0.12)	0.00	2.23 ($\pm$0.03)	0.00
	MODFULL	31.00 ($\pm$0.00)	-	4.00 ($\pm$0.00)	-

Table 5.9: Problem indicates the problem being analysed. Experiment indicates the type of initialisation algorithm used. Average Length is the average length of programs at initialisation averaged over 100 runs. PL indicates the result of a 2 Sample T-test comparing the average programs lengths. The P-Value is aligned with the shorter programs. Average Depth indicates the depth of the programs at initialisation averaged over 100 runs. PD is the result of a 2 Sample T-test comparing the initialised program depths. The P-Value is aligned with the shorter programs.

beyond that required for statistical significance.

5.3.1 Distribution of Behaviours in the Search Space

The results in sections 5.2.2 and 5.2.3 clearly demonstrate how the existing RHH technique has bias, frequently duplicating simplistic and constant behaviours. The SDI algorithm was developed to counter these effects by producing complete behavioural diversity and building more complexity into the starting populations.

Preliminary analysis of the size and shape output (tables 5.5 and 5.6) of the SDI algorithm appeared positive for two reasons. The first is the ability of the SDI to create starting populations that vary their programs size depending on the problem. This gives the impression that the SDI is actually modelling behaviours specific to the search space of each problem rather than the "one size fits all" solution in the RHH. The second is that depth appears not to be the best way to constrain programs. This size and shape analysis (tables 5.5 and 5.6) showed that the SDI produced deeper but thinner trees in order to specifically model more complex behaviour.

Whilst the SDI might be theoretically superior in terms of distributing behaviours in the search space, table 5.7 shows that it only outperforms the RHH technique in the 6MUX, 4PAR, 7PAR experiments and is equal overall in 5MAJ, CUBIC and QUART. The parity problems are known to be a deceptive problem (Langdon and Poli [2002]), requiring a more intricate and complex program to solve them. This may be the reason that the SDI performed well on this problem. The multiplexer results are less clear, as it is intriguing that the SDI does not outperform RHH on the 11MUX problem. A possible explanation for this is that in terms of the overall search space the program required to secure 100% fitness is relatively simplistic in comparison to all the behaviours in the 11MUX search space. If one considers the exponential increase in search space size between the 6MUX search space of 2^{2^6} to the 11MUX search space of $2^{2^{11}}$, the SDI is having to model far more complexity to distribute programs through the breadth of the search space. It is possible that, as a result of this, the RHH is more biased to the correct area of the search space, therefore achieving a higher success rate.

The SDI failed on the majority experiments, which is arguably the most simple of the Boolean experiments. This would suggest that the RHH was better able to produce programs

in the most successful areas of the majority search space whereas the SDI's complexity worked against it biasing search away from high fitness areas. The SDI combines functions resulting in deep thin trees resulting in more complex functionality. Given that the MODFULL algorithm was best performing in the majority of experiments, it may have been the case that a shallow fat tree was more likely to facilitate a need for less functionality, but more choice of nodes at a shallow level in the tree. This may be the behaviour that is required to solve the majority experiments and as a result the SDI biased the search for behaviours in a different area of the search space.

Based on these results, the hybrid algorithm (HSDI) was created to take advantage of the increased semantic diversity generated by the FULL algorithm and the increased complexity aspect of the SDI algorithm. The HSDI produced an algorithm capable of being the best performer overall in one experiment and equal best overall in eight experiments (at the 95% confidence level). In generation 50, it was able to statistically equal the best result in five out of nine experiments. The performance of the HSDI was good overall, equalling eight best results, but disappointing at generation 50 considering the HSDI combined the different features of the SDI and RHH algorithms. This is a testament to the difficulty of creating a problem independent population generation algorithm.

This difficulty in creating one initialisation algorithm which can be applied to every problem could be a consequence of the no free lunch theorem (Woodward and Neil [2003]). If initialisation algorithms were all equally good at generating starting populations, results would be varied like those of RHH and SDI. The HSDI algorithm is a contrast in that, it performs to a good standard in comparison to RHH and SDI most of the time. This may be an indication that HSDI is able to obtain good results (a free lunch) over a wider problem space (Poli et al. [2009]).

5.3.2 Evolvable Shape

The results presented in table 5.8 clearly show that the shape of the trees can have a significant effect on GP performance which is in agreement with Daida et al. [2003], Daida and Hilss [2003] and Langdon et al. [1999]. In the Boolean domain, all of the experiments presented demonstrate not only statistically significant differences at the 95% confidence level, but also some dramatic changes in the performance of the GP runs. Given the variations in

these results, it would appear that the ideal evolvable shape for a program is problem dependent (based on the varying program sizes and depths in table 5.9) and therefore it would be difficult to predict ideal program structures for specific problems. This suggests that further work comparing these results with those of Daida and Hilss [2003] would be valuable, as well as, further evaluation of the structure of programs.

The difference between the artificial ant shape results overall is surprising, given that the translation mechanism between abstract meaning and syntax of the ant problem builds full trees (without redundant code). As a result, the experiment compares larger full trees with smaller, more effective, full trees. In this case, it is clear that removing introns and changing the size of programs through the WASHED process reduces evolutionary potential across all experiments as the MODFULL algorithm is statistically the best performer. A noteworthy point is that the SDI contains no introns for the ant domain, whereas the HSDI has simulated introns (in the form of SKIP), and this increases the performance of the algorithm in the ant domain. Counter intuitively, when the WASHED algorithm reduces code applying SKIP terminals the performance reduces. From these conflicting results, it is difficult to decide whether the code produced using WASHED reduces the evolutionary potential of programs, or whether the increased diversity generated by HSDI or whether a fundamental change in program size caused by both of the algorithms improves performance.

Another algorithm that gives the user the ability to influence tree shape is *Probabilistic Tree Creation* (Luke [2000a], Luke and Panait [2001]). This is because the user-controlled bias in the appearance of functions and terminals would have an overall effect on tree shape. Luke's success using this algorithm could, at least partially, be a result of evolvable shape. With reference to the results presented in table 5.8, it would be interesting to see if performance would increase if the probabilities of selection of the terminals and functions were changed from experiment to experiment, for example, to cater for the difference between the parity and majority problems.

Having shown that there is a preferred evolvable shape for specific problems (Table 5.8 and 5.9), it can be argued that this lends support to GP schema theory (Poli and McPhee [2003a,b], Langdon and Poli [2002]). In this interpretation, the schemata are in a more abstract form, consisting of a tree of a (problem specific) particular shape containing "don't care" nodes. Alternatively, one could follow a strategy such as Salustowicz and Schmidhuber

[1997] enforcing a structure and using node weightings to perform the learning. This would be an interesting direction for future work.

The issue of problem dependence in the context of generating starting populations is a complex one. Table 5.7 shows that the three algorithms presented seem to favour particular problems and the results in table 5.8 merely complicate the matter further. Based on this data, further research is required in the field of program initialisation in order to be in a position to recommend specific code generation algorithms for specific problems. Once this research is completed, it would be interesting to run a further study which did not treat initialisation as a separate issue from crossover and mutation and consider the array of different choices of crossover and mutation operators which may have further effects on populations initialised in different ways.

5.4 Conclusions

The results presented in this analysis of program initialisation in GP have shown that the initialisation method chosen can have a dramatic impact on the performance of GP runs. However, it appears that this impact is problem specific, and that it cannot concluded that one of the algorithms presented is best (or better than the other algorithms presented) for every problem considered. The results presented in sections 5.2.2 and 5.2.3 have clearly illustrated the limitations of the RHH algorithm, and tested theories using the algorithms presented. It is clear that none of the methods described, in the form of either the SDI, HSDI, MODFULL or WASHED algorithms present one clear solution to program initialisation that incorporates measures to deal with the behavioural distribution of programs and to control the shape of the syntax produced.

Clear evidence has been presented in sections 5.2.5 and 5.2.6 showing that both the distribution of programs in the search space and the structure of the program tree can have dramatic effects on the performance of GP. Both of these variables are strongly dependent on the problem being tackled, which therefore makes the challenge of constructing an initialisation algorithm that consistently produces a more diverse and better structured first generation of programs a highly complex one. As a result of this work, there is evidence to support the need to create initialisation algorithms that can explicitly exercise control over

both the behavioural distribution of programs and the shape of the programs they produce.

5.5 Future Work

Based on evidence presented, future work in the area of program initialisation needs to be able to address both behavioural diversity and program structures and the interactions between the two. Possible ideas, could be merging one of the algorithms presented (either SDI or HSDI) with one of Luke's PTC algorithms in order to gain a more granular control over program structure, whilst retaining behavioural diversity.

Alternatively, it would be a worthwhile experiment to build a new initialisation algorithm which is capable of explicitly and tunably controlling both behavioural diversity and program structure. From the results presented in the chapter, the correct use of these new controls would allow statistically significant improvement in GP performance.

Chapter 6

Semantically Driven Operators

The objective of this chapter is to discuss methods for increasing the efficiency of three different areas of the process of search in GP. The motivation for this work is two fold. In the case of semantically driven crossover and semantically driven mutation, the key concept is to change a syntax altering operation to become a semantics altering operation. This increases the power of GP search resulting in better performance in the experiments presented. In the case of semantic pruning and intron free GP, the motivation is to reduce programs to consist of only effective code, and attempt to combat the bloat phenomenon, as well as understanding the effects on the performance of GP when using intron free GP.

In section 6.1, semantically driven crossover is described. This section is based upon work presented by Beadle and Johnson [2008] and is presented with a greatly expanded problem suite (as described in section 4.1) compared to the original paper. The results presented demonstrate the increased search power of semantically driven crossover and in some cases, how this operator can work to reduce bloat.

Section 6.2 described the semantically driven mutation process. This section is based upon work presented by Beadle and Johnson [2009b] with further experimentation from a greater problem suite (as described in section 4.1). The results obtained using semantically driven mutation demonstrate how increased search power can improve performance in GP.

In section 6.3, semantic pruning is described. This operator has mixed effects on GP performance but also powerful effects on reducing program size.

Finally, in section 6.4, semantically driven crossover, semantically driven mutation, semantic pruning and semantically driven initialisation methods (chapter 5) are combined to

produce intron free GP. Results are presented using the problem suite outlined in section 4.1.

6.1 Semantically Driven Crossover

Semantically Driven Crossover (SDC) is a modification to the crossover algorithm which can be used to increase the performance of GP. The key concept of the SDC algorithm is to not allow the production of child programs which are behaviourally equivalent to their parent programs. This section is based on techniques presented by Beadle and Johnson [2008] using the greater test suite described in section 4.1.

The key feature of this work is the use of a canonical representation of behaviour of members of the population. These representations are as described in section 4.3. In the genetic programming context, being able to compare the behaviour of two syntactically distinct programs creates the ability to ensure that candidate programs are mobile in the semantic search space. This increased movement in the phenotypic search space will improve the search power of genetic programming.

Section 6.1.1 discusses the SDC algorithm in relation to other methods to improve the crossover operation. Section 6.1.2 presents the SDC algorithm. Section 6.1.3 presents results using the SDC algorithm and the associated discussion of the results. Section 6.1.4 links the effects of the SDC algorithm to existing theories of bloat and sections 6.1.5 and 6.1.6 present conclusions using the SDC algorithm and ideas for future work arising from the use of the SDC respectively.

6.1.1 Improving Crossover and Effective Fitness

The first priority of the SDC algorithm is to increase the performance of GP through the control of program behaviour. This increase in performance highlights an inefficiency in the crossover operation which is the possibility of the production of semantically equivalent child programs. The second effect of the SDC algorithm is significant changes in the level of bloat through not allowing the crossover of redundant or unreachable code in some of the test problems. In the majority of cases, the use of the SDC algorithm results in a decrease in bloat.

In relation to improving the performance of crossover, there are three categories of settings to amend to change the performance of crossover. The first of these is to change the way parent programs are selected (using different selection methods, some outlined in section 2.5), the second is to change the method which is used to select swap points on program representations (for example, Koza [1992] applying a 90% bias to functions and 10% to terminals as swap points) and the third is to perform some type of pre and post crossover processing to test for some desirable property of the crossover operation (for example, O'Reilly and Oppacher [1995]). Whilst the SDC algorithm falls into the category of pre and post processing, it is novel in that it does not look for some improvement, it merely tests that there is some new behaviour in the child program. This is important as, when using the SDC programs will take a semantic step in the search space in relation to the parent programs instead of stepping to an area of higher fitness such as during crossover hybridised with hill climbing.

Whilst several authors have started to consider the implications of semantics in GP (notably, Looks [2007], McPhee et al. [2008], Gustafson et al. [2004]), only two research teams have used semantics to directly influence the crossover operation (Beadle and Johnson [2008], Nguyen et al. [2009]), although McPhee et al. [2008] did present semantic analyses of the effects of the crossover operation on small scale problems.

When considering the changing effects on bloat, the SDC algorithm is an interesting case as it was never designed with bloat control in mind. Whilst other authors have proposed methods to control bloat during crossover, for example Koza [1992] imposing an absolute depth limit on child programs (for a more complete review of bloat control methods, see Luke and Panait [2006]) and Langdon [2000] devising a method for size fair crossovers, the SDC algorithm does nothing to explicitly control program size during crossover.

6.1.2 Semantically Driven Crossover Algorithm

The implementation of the SDC algorithm could be applied to any form of crossover algorithm as the SDC algorithm acts as a wrapper evaluating the behaviour before and after the operation. As such it could be used with single point, uniform swap point or Koza style crossover (Koza [1992]). It could also be applied around other pre and post crossover evaluation techniques. The SDC algorithm is presented in algorithm 6.1.

Algorithm 6.1 Semantically Driven Crossover Algorithm

```
while number_of_programs < population_size
   if random_number < crossover_probability
      while not (crossover_accepted1 and crossover_accepted2)
         select p1 randomly from breeding pool (parent 1)
         select p2 randomly from breeding pool (parent 2)
         copy p1 into c1 (child 1)
         copy p2 into c2 (child 2)
         choose swap_point1 on c1
         choose swap_point2 on c2
         perform crossover at swap points
         generate Representation of p1, p2, c1, c2
         if p1_Representation not equivalent to c1_Representation AND p2_Representation
               not equivalent to c1_Representation
            crossover_accepted1 set to true
         end if
         if p1_Representation not equivalent to c2_Representation AND p2_Representation
               not equivalent to c2_Representation
            crossover_accepted2 set to true
         end if
      end while
      add c1 to population
      add c2 to population
   else
      select program 1 randomly from breeding pool and add to new population
      select program 2 randomly from breeding pool and add to new population
   end if
end while
```

The SDC algorithm (6.1) repeats standard crossover until the child programs produced are not semantically equivalent to either parent. Whilst this means that there is an extra computational requirement in terms of both the numbers of crossovers to be performed, and the construction of the behaviours, this increased load is offset in some cases by decreased program size. In all of the experiments presented, the SDC does not produce excessive or unusable run times.

Experimental Parameters

The parameters used for these experiments are set out in section 4.2 with the notable exception of the crossover techniques. Four experiments have been conducted per test problem. These include using standard crossover with a uniform choice of swap points (UNIXO), a semantically driven version of standard crossover with uniform choice of swap points (UNIXO-SDC), standard crossover with Koza style choice of swap points (90% bias on functions and 10% bias on terminal swap points) (KOZAXO) and a semantically driven version of standard crossover with Koza style choice of swap points (KOZAXO-SDC). Statistical comparison has been conducted between UNIXO and UNIXO-SDC, and KOZAXO and KOZAXO-SDC. This provides the ability to test that semantically driven crossover will work against an equivalent control experiment rather than comparing experiments with different crossover mechanisms.

6.1.3 Results

When considering crossover with a uniform selection of crossover points, table 6.1 shows that semantically driven crossover is able to statistically improve the performance of GP, in terms of max scores overall, in every experiment. At generation 50, semantically driven crossover is able to significantly improve performance in all but two experiments which are statistically similar to the control experiment.

When considering crossover with Koza style bias on the choice of swap points, table 6.2 shows that semantically driven crossover improves performance overall in all experiments except the artificial ant problem, which is statistically similar to the control experiment. At generation 50, semantically driven crossover outperforms the control experiment significantly in all but two cases.

Problem	Experiment	Max Score Overall	PT	Max at G50	2T	Success (Gen)
4PAR	UNIXO-SDC	**0.0653 (±0.0487)**	**0.00**	**0.0222 (±0.0360)**	**0.00**	69% (G10)
	UNIXO	0.0854 (±0.0376)	-	0.0859 (±0.0594)	-	44% (G9)
5MAJ	UNIXO-SDC	**0.0282 (±0.0455)**	**0.00**	**0.0006 (±0.0044)**	**0.00**	98% (G7)
	UNIXO	0.0429 (±0.0414)	-	0.0144 (±0.0215)	-	63% (G8)
6MUX	UNIXO-SDC	**0.0484 (±0.0622)**	**0.00**	**0.0094 (±0.0285)**	**0.00**	88% (G6)
	UNIXO	0.0684 (±0.0534)	-	0.0291 (±0.0431)	-	62% (G8)
7PAR	UNIXO-SDC	**0.3382 (±0.0682)**	**0.00**	**0.2394 (±0.0493)**	**0.00**	0% –
	UNIXO	0.3533 (±0.0588)	-	0.2696 (±0.0277)	-	0% –
9MAJ	UNIXO-SDC	**0.1258 (±0.0482)**	**0.00**	**0.0718 (±0.0088)**	**0.00**	0% –
	UNIXO	0.1307 (±0.0460)	-	0.0796 (±0.0098)	-	0% –
11MUX	UNIXO-SDC	**0.0910 (±0.1033)**	**0.00**	**0.0098 (±0.0204)**	**0.00**	76% (G18)
	UNIXO	0.1014 (±0.0954)	-	0.0301 (±0.0351)	-	49% (G15)
AASF	UNIXO-SDC	**0.3367 (±0.0774)**	**0.00**	0.2730 (±0.1371)	0.98	11% (G3)
	UNIXO	0.3409 (±0.0782)	-	0.2719 (±0.1292)	0.98	10% (G1)
CUBIC	UNIXO-SDC	**807.70 (±288.77)**	**0.00**	**493.78 (±212.55)**	**0.004**	0% –
	UNIXO	877.47 (±261.54)	-	579.33 (±199.53)	-	0% –
QUART	UNIXO-SDC	**1246.77 (±605.85)**	**0.00**	682.69 (±282.01)	0.338	0% –
	UNIXO	1288.83 (±599.18)	-	721.37 (±287.42)	0.338	0% –

Table 6.1: Table showing results comparing semantically driven crossover to semantically driven crossover. Problem indicates the test problem being tackled. Experiment indicates the type of crossover being used. Max Score Overall indicates the average of maximum scores (standardised fitness so 0 is best score) (with standard deviation in brackets) over all generations. All results are scaled in relation to full possible score except regression domain results. PT indicates the P-Value of a Paired T-test comparing two experiments. In the event that one result is statistically superior to another, the P-Value will be aligned with the better result. Max at G50 indicates the mean of the maximum scores at generation 50 (with standard deviation in brackets). 2T indicates the P-Value result of a 2 sample T-test. In the case of statistical significance, the P-Value is aligned with the best result. Success indicates the number of runs that reached full score and (Gen) indicates the earliest generation at which full score was achieved.

Problem	Experiment	Max Overall	PT	Max at G50	2T	Success (Gen)
4PAR	KOZAXO-SDC	**0.0562 ($\pm$0.0512)**	**0.00**	**0.0131 ($\pm$0.0285)**	**0.00**	81% (G6)
	KOZAXO	0.0854 ($\pm$0.0378)	-	0.0506 ($\pm$0.0522)	-	43% (G12)
5MAJ	KOZAXO-SDC	**0.0266 ($\pm$0.0435)**	**0.00**	**0.0019 ($\pm$0.0075)**	**0.00**	94% (G6)
	KOZAXO	0.0364 ($\pm$0.0406)	-	0.0125 ($\pm$0.0199)	-	68% (G7)
6MUX	KOZAXO-SDC	**0.0457 ($\pm$0.0634)**	**0.00**	**0.0061 ($\pm$0.0186)**	**0.00**	87% (G3)
	KOZAXO	0.0732 ($\pm$0.0521)	-	0.0406 ($\pm$0.0578)	-	56% (G6)
7PAR	KOZAXO-SDC	**0.3378 ($\pm$0.0660)**	**0.00**	**0.2438 ($\pm$0.0320)**	**0.00**	0% –
	KOZAXO	0.3461 ($\pm$0.0611)	-	0.2588 ($\pm$0.0252)	-	0% –
9MAJ	KOZAXO-SDC	**0.1168 ($\pm$0.0493)**	**0.00**	**0.0640 ($\pm$0.0095)**	**0.00**	0% –
	KOZAXO	0.1217 ($\pm$0.0477)	-	0.0707 ($\pm$0.0116)	-	0% –
11MUX	KOZAXO-SDC	**0.0873 ($\pm$0.0999)**	**0.00**	**0.0102 ($\pm$0.0214)**	**0.00**	77% (G15)
	KOZAXO	0.0973 ($\pm$0.0905)	-	0.0327 ($\pm$0.0353)	-	46% (G16)
AASF	KOZAXO-SDC	0.2768 ($\pm$0.1264)	0.19	0.2764 ($\pm$0.1258)	0.19	5% (G11)
	KOZAXO	0.2985 ($\pm$0.1113)	0.19	0.2985 ($\pm$0.1113)	0.19	3% (G1)
CUBIC	KOZAXO-SDC	**771.97 ($\pm$302.31)**	**0.00**	**440.36 ($\pm$195.14)**	**0.041**	0% –
	KOZAXO	816.26 ($\pm$287.25)	-	498.64 ($\pm$204.69)	-	0% –
QUART	KOZAXO-SDC	**1210.30 ($\pm$288.71)**	**0.00**	637.57 ($\pm$268.58)	0.087	0% –
	KOZAXO	1258.54 ($\pm$596.09)	-	711.89 ($\pm$311.47)	0.087	0% –

Table 6.2: Table showing results comparing standard crossover to semantically driven crossover. Problem indicates the test problem being tackled. Experiment indicates the type of crossover being used. Max Overall indicates the average of maximum scores (standardised fitness so 0 is best score) (with standard deviation in brackets). All results are scaled in relation to full possible score except regression domain results. PT indicates the P-Value of a Paired T-test comparing two experiments. In the event that one result is statistically superior to another, the P-Value will be aligned with the better result. Max at G50 indicates the mean of the maximum scores at generation 50 (with standard deviation in brackets). 2T indicates the P-Value result of a 2 sample T-test. In the case of statistical significance, the P-Value is aligned with the best result. Success indicates the number of runs that reached full score and (Gen) indicates the earliest generation at which full score was achieved.

Problem	Experiment	Mean Program Length	PT
4PAR	UNIXO-SDC	**117.05 ($\pm$29.20)**	**0.00**
	UNIXO	186.76 ($\pm$59.49)	-
5MAJ	UNIXO-SDC	**81.05 ($\pm$29.46)**	**0.00**
	UNIXO	132.15 ($\pm$56.36)	-
6MUX	UNIXO-SDC	**48.98 ($\pm$13.50)**	**0.00**
	UNIXO	88.77 ($\pm$38.05)	-
7PAR	UNIXO-SDC	**274.06 ($\pm$93.95)**	**0.00**
	UNIXO	342.50 ($\pm$121.58)	-
9MAJ	UNIXO-SDC	**245.67 ($\pm$146.31)**	**0.00**
	UNIXO	254.82 ($\pm$140.86)	-
11MUX	UNIXO-SDC	**75.08 ($\pm$34.56)**	**0.00**
	UNIXO	101.13 ($\pm$53.38)	-
AASF	UNIXO-SDC	125.68 ($\pm$29.11)	-
	UNIXO	**106.38 ($\pm$24.27)**	**0.00**
CUBIC	UNIXO-SDC	**133.80 ($\pm$48.09)**	**0.00**
	UNIXO	137.43 ($\pm$46.52)	-
QUART	UNIXO-SDC	133.67 ($\pm$46.24)	-
	UNIXO	**129.81 ($\pm$44.30)**	**0.00**

Table 6.3: Table showing depth comparison using standard and semantically driven crossover. Problem indicates the test problem being studied. Experiment is the type of crossover being used. Mean Program Length is the mean number of nodes present in the trees in all generations. PT is the P-Value result of a Paired T-test. In the case of statistical significance the P-Value is aligned with the best result.

Problem	Experiment	Mean Program Length	PT
4PAR	KOZAXO-SDC	**151.28 ($\pm$44.52)**	**0.00**
	KOZAXO	239.22 ($\pm$86.77)	-
5MAJ	KOZAXO-SDC	**104.59 ($\pm$38.92)**	**0.00**
	KOZAXO	181.64 ($\pm$83.13)	-
6MUX	KOZAXO-SDC	**79.35 ($\pm$29.81)**	**0.00**
	KOZAXO	117.88 ($\pm$56.19)	-
7PAR	KOZAXO-SDC	**362.02 ($\pm$151.61)**	**0.00**
	KOZAXO	432.03 ($\pm$180.23)	-
9MAJ	KOZAXO-SDC	**351.40 ($\pm$212.41)**	**0.019**
	KOZAXO	356.25 ($\pm$204.41)	-
11MUX	KOZAXO-SDC	**101.74 ($\pm$47.72)**	**0.00**
	KOZAXO	128.95 ($\pm$75.20)	-
AASF	KOZAXO-SDC	148.04 ($\pm$38.44)	-
	KOZAXO	**127.81 ($\pm$33.92)**	**0.00**
CUBIC	KOZAXO-SDC	163.56 ($\pm$60.70)	-
	KOZAXO	**161.87 ($\pm$62.33)**	**0.01**
QUART	KOZAXO-SDC	180.71 ($\pm$72.64)	-
	KOZAXO	**168.51 ($\pm$62.69)**	**0.00**

Table 6.4: Table showing depth comparison using standard and semantically driven crossover. Problem indicates the test problem being studied. Experiment is the type of crossover being used. Mean Program Length is the mean number of nodes present in the trees in all generations. PT is the P-Value result of a Paired T-test. In the case of statistical significance the P-Value is aligned with the best result.

Tables 6.3 and 6.4 show a comparison of the size of programs produced when using SDC with either uniform swap point crossover or biased swap point crossover. The results dictate that SDC consistently produces smaller programs in the Boolean domain problems, but in the ant and symbolic regression domains 5 out of 6 experiments produce significantly shorter programs using original crossover. These conflicting results are examined in more detail in the discussion section.

Figures 6.1 — 6.5 show varying levels and trends of the numbers of crossovers which are rejected for parent child equivalence. In the majority of the experiments there is an upward trend in the numbers of rejected crossovers, which in some cases is nearly the same size as the population. This demonstrates that the SDC algorithm is having to do more work later in the runs to enforce semantic diversity during evolution. The two experiments which contrast this trend are the CUBIC and QUART (figure 6.5) which both show a decreasing trend to very small numbers of rejected crossover (if considered as a percentage of the total population)

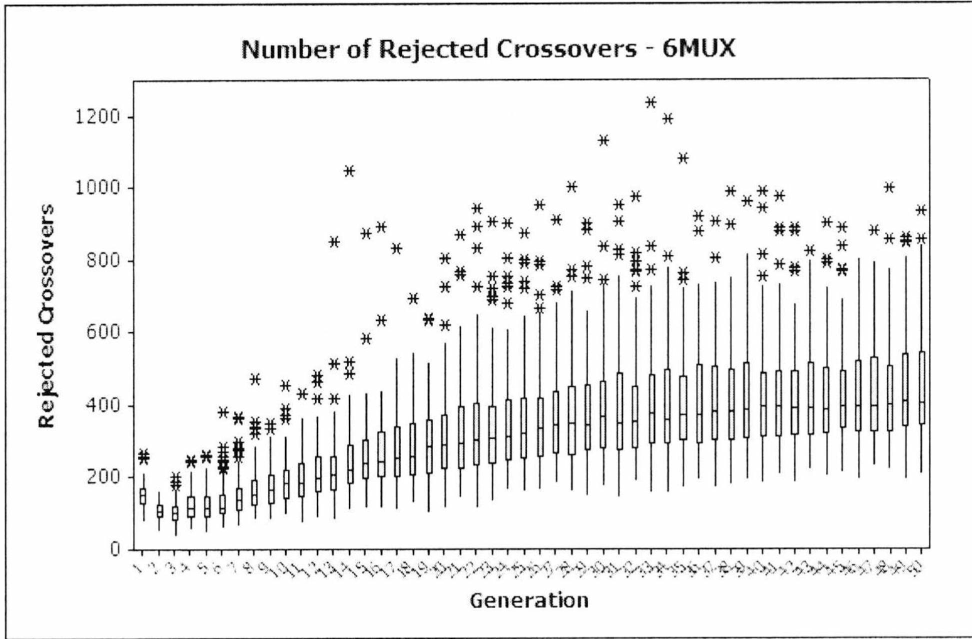
6.1.4 Discussion

The results presented in tables 6.1, 6.2, 6.3 and 6.4 highlight two separate effects of using semantically driven crossover. These are the improvement of fitness and the problem dependant reduction in bloat.

Improvement in Fitness

The motivation for researching this crossover technique was to test the hypothesis that increasing semantic diversity in a population of programs would result in the increased performance of GP. In all but one of these experiments, semantically driven crossover has resulted in statistically significant improvements in the performance of GP. In the case where performance was not better than existing crossover, the fitness was statistically equivalent. The fact that maintaining semantic diversity in GP populations increases performance is important as it demonstrates that directly relating operators to the semantic search space does have advantages in evolution.

Whilst critics of this operator will compare semantically driven crossover to a brood operator (an operator that produces more children than are required and selects children against a criteria to be inserted into the new population Zhang et al. [2006]) saying that it has more



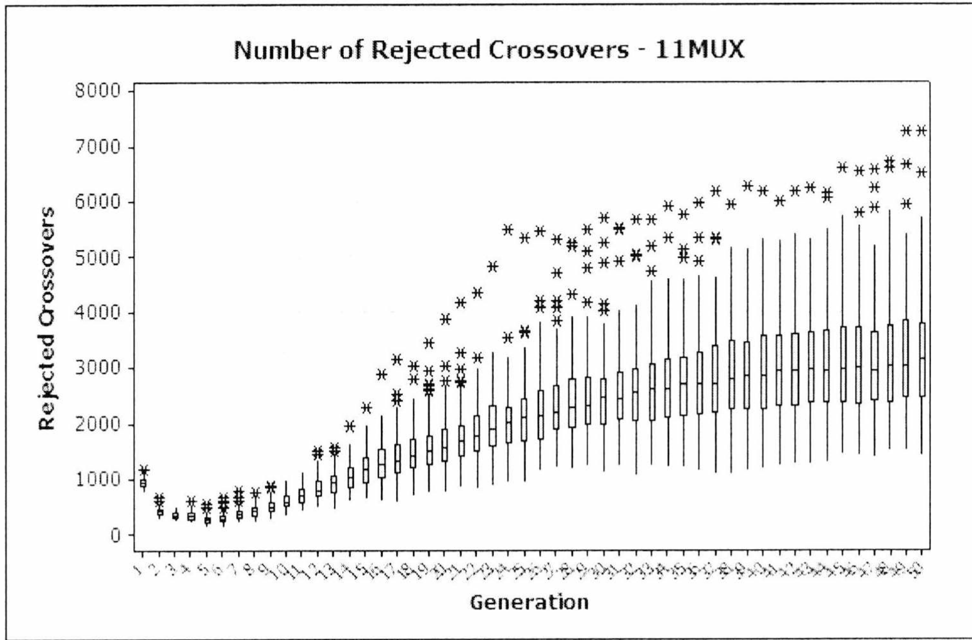
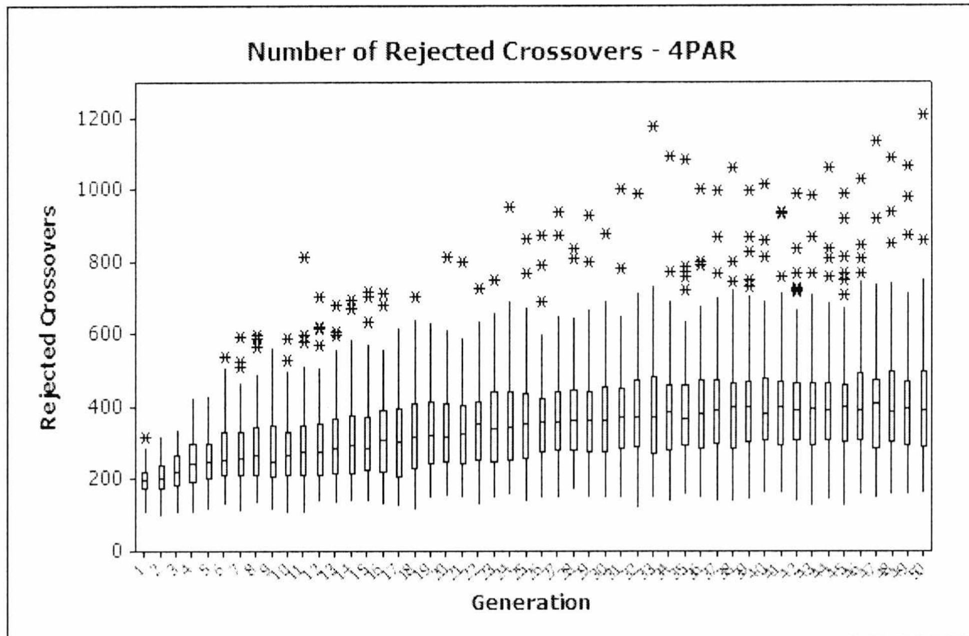


Figure 6.1: The figures show the numbers of rejected crossovers in box plot form, with outliers marked as * using the KOZAXO-SDC variant of SDC for the multiplexer experiments. The results presented are averaged over 100 runs.



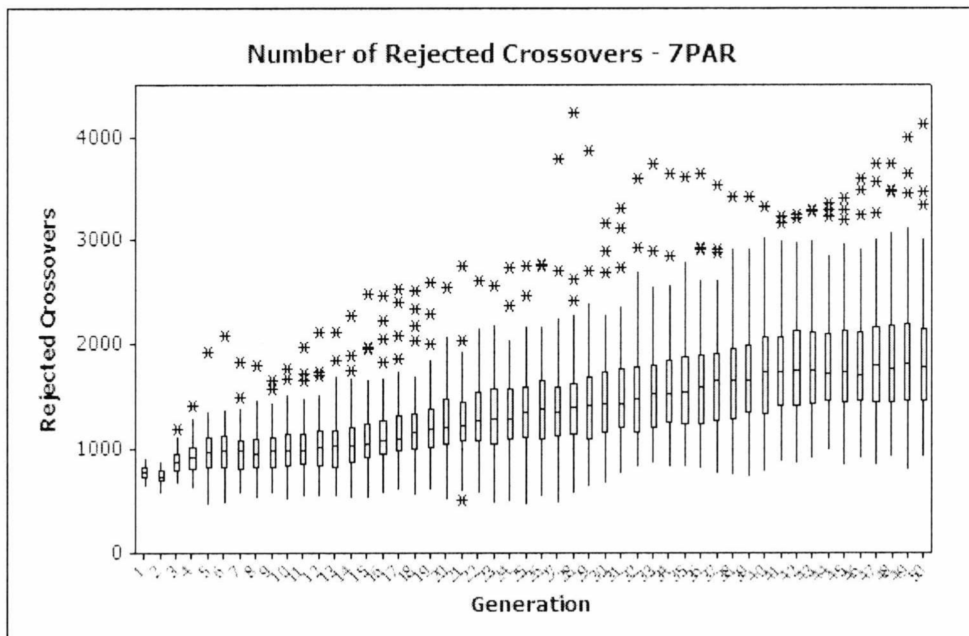
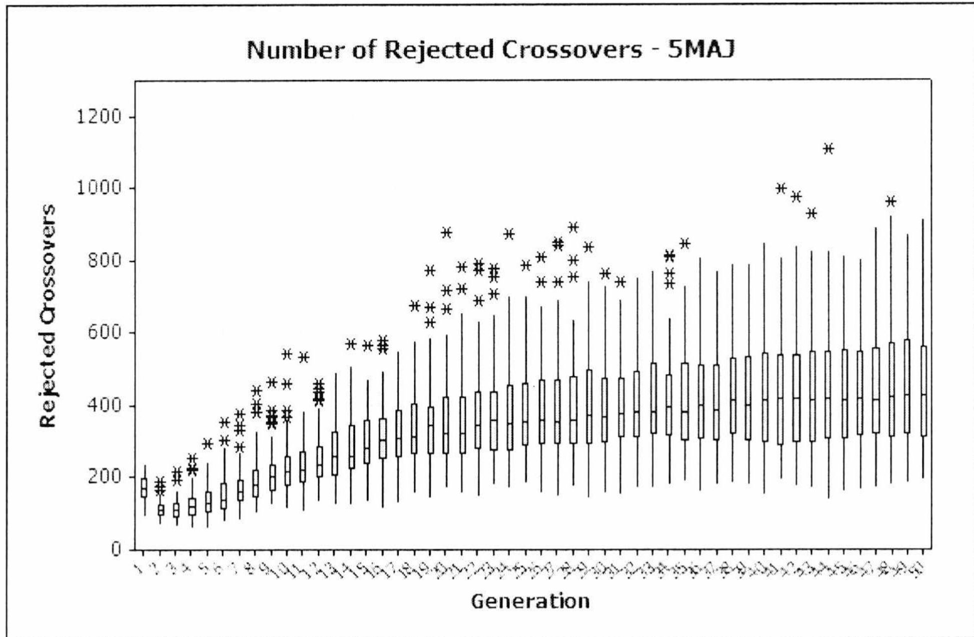


Figure 6.2: The figures show the numbers of rejected crossovers in box plot form, with outliers marked as * using the KOZAXO-SDC variant of SDC for the even parity experiments. The results presented are averaged over 100 runs.



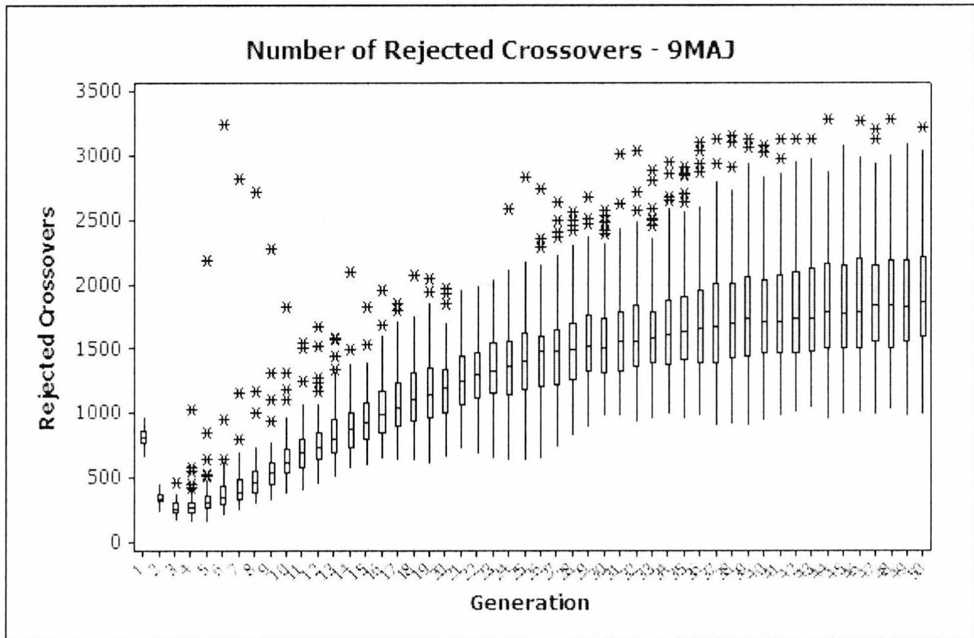


Figure 6.3: The figures show the numbers of rejected crossovers in box plot form, with outliers marked as * using the KOZAXO-SDC variant of SDC for the majority experiments. The results presented are averaged over 100 runs.

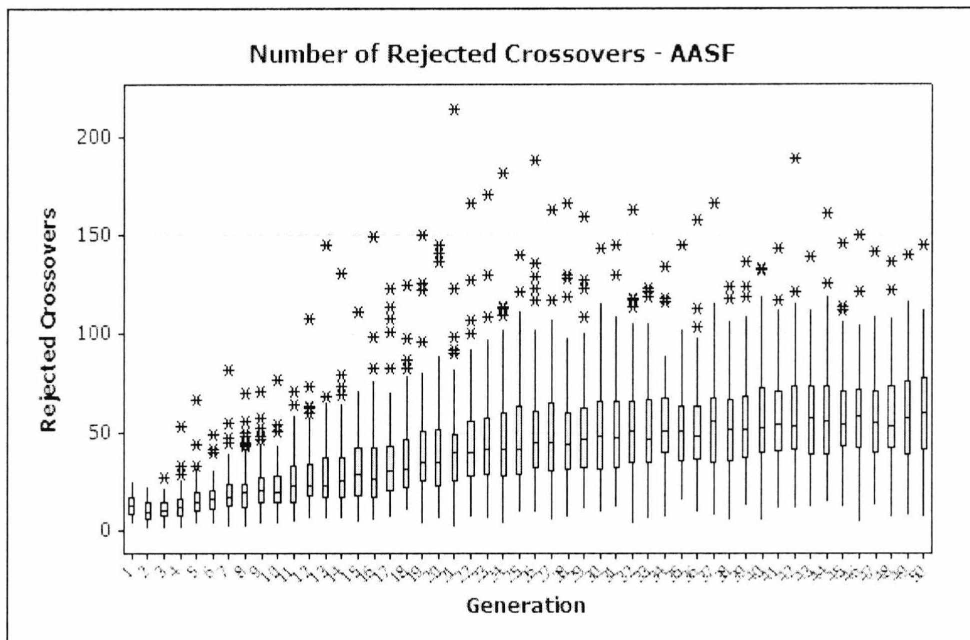


Figure 6.4: The figure shows the numbers of rejected crossovers in box plot form, with outliers marked as * using the KOZAXO-SDC variant of SDC for the artificial ant experiment. The results presented are averaged over 100 runs.

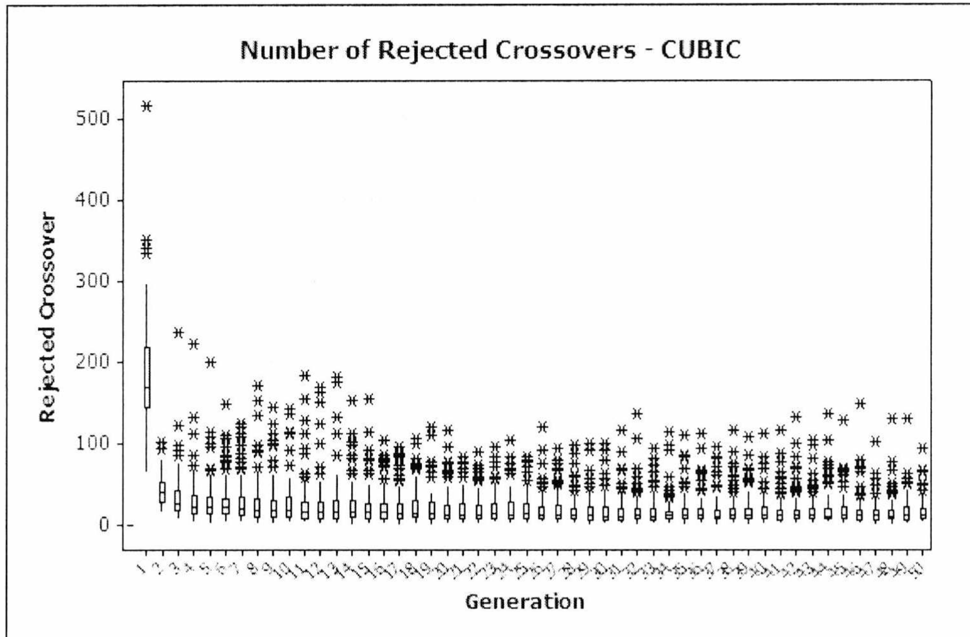
chances to pick a desirable child program, one can argue that semantically driven crossover does not examine any particular feature of a child program except its behavioural originality which may be a positive or negative result.

Moreover, this operator may offer improvements in GP scalability. If one considers a situation where a fitness function is computationally expensive to execute (for example, a situation with a large number of input-output scenarios where behavioural representation comparison will operate relatively quickly compared to standard fitness assessment), using abstraction to enable crossovers to produce novel children more often will reduce the number of fitness neutral and semantically equivalent programs that are assessed. Essentially, this operator may give the GP practitioner the opportunity to do a more efficient evolution compared to performing more evolution (either by increasing population size or generations).

Reduction of Bloat

As described in section 3.2, there are several theories as to the cause of bloat. The results presented in tables 6.3 and 6.4 demonstrate that semantically driven crossover can have a significant impact on bloat both positively and negatively. In the experiments presented, all

KENT TEMPLEMAN LIBRARY UNIVERSITY



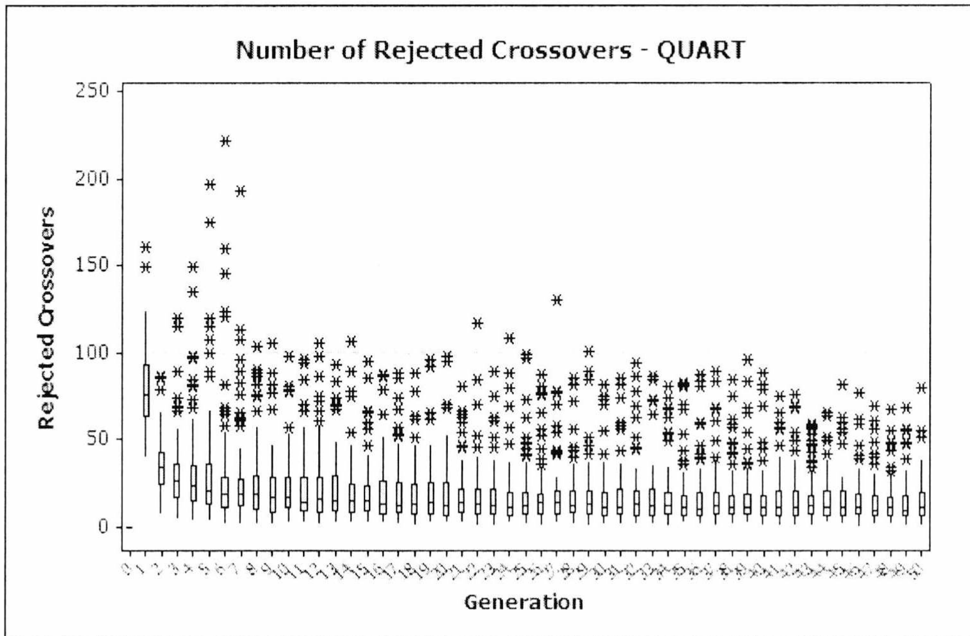


Figure 6.5: The figures show the numbers of rejected crossovers in box plot form, with outliers marked as * using the KOZAXO-SDC variant of SDC for the symbolic regression experiments. The results presented are averaged over 100 runs.

of the Boolean domain problems experiences reduced bloat when using SDC, whereas all but one of the other experiments experiences increased bloat when using the SDC.

Figures 6.1 — 6.5 showed varying trends of rejected crossover. In the experiments presented, all of the Boolean problems showed substantial levels of rejected crossover with an increasing trend. A possible reason for this is that, as introns build up within programs, crossover finds it harder to chose swap points which will effect the overall behaviour of the candidate program. This would lead to higher numbers of crossovers being rejected as time increments. The artificial ant and the symbolic regression domain, generally saw a relatively low level of crossovers being rejected. In the symbolic regression domain, there was a decreasing trend in the level of rejected crossovers. This may be as a result of the SDC forcing diverse evolution paths in the early generations of the run and after this point candidate programs retained that diversity. In this situation the number of behaviours in the search space which are attainable with larger programs becomes important. The symbolic regression domain will have many more behaviours than the discrete domains when programs are large and this would contribute to the lower number of reverted crossovers in the symbolic regression domain.

An interesting note at this point is the way in which introns are defined. The contrast between the Boolean and symbolic regression domains is that IF statements in the Boolean domain render areas of code unreachable whereas in the symbolic regression domain, invalidators (such as multiply by 0) render code redundant. Changes to redundant code may result in the code becoming active again, whereas changes in unreachable code can never have an effect. The lack of unreachable code could be a reason for fewer rejected crossovers in the symbolic regression domain.

Given the varying results, the impact of semantically driven crossover is compared to existing theory. When considering the protection from deletion theory, the SDC algorithm would magnify the effect of destructive crossover as it only checks whether a crossover is behaviourally neutral rather than whether it is constructive or destructive. As a result, if introns are prevented from being added to programs at the point of crossover, it would be expected that programs would perform worse in terms of overall score if the protection from deletion theory were applied. Given that the results presented do not show any degradation of performance using semantically driven crossover, it would indicate that introns are not

required to achieve good performance. Furthermore, given that in the majority of the experiments presented, there is increased performance as well as reduced program size, it would suggest that programs do not need to be protected from having valuable subtrees deleted.

A further problematic aspect of this theory is the idea that to achieve high scores GP evolution is dependent on an inefficient one to many mapping between behaviour and syntax respectively. The system proposed in this section has taken a small step towards a one to one relationship between syntax and behaviour and outperforms the standard GP technique. This would indicate that it may be better to aspire to a one to one mapping between syntax and behaviour to achieve better performance.

When considering the fitness causes growth theory, two aspects need to be considered. The first of these is the concept that programs may need to grow to attain higher scores as they may be missing essential parts. The second is the idea that a population will drift to a region of the search space where they are more likely to attain a higher score and multiply resulting in the same standard behaviour. Both of these factors are plausible under the system presented and although the SDC mean program size is substantially smaller than the standard GP, it does increase with improved performance.

One interesting difference between standard crossover and the SDC algorithm is the fact that whilst the search may move to areas of semantically equivalent programs, the lineage will be different because each crossover forces the creation of a child program which will make a new behavioural step in relation to the parent programs. Therefore, whereas in standard crossover, code may drift to semantically equivalent and fitness equivalent areas of the search space, in SDC the population be more diverse.

Removal bias hinges on the idea that in the event of a fitness neutral crossover, it is more probable that a small sub tree will be swapped for a larger one during crossover. Behaviourally neutral crossovers are subsumed by fitness neutral crossovers, however, figures 6.1 — 6.5 show that a substantial proportion of behaviourally neutral crossovers are rejected. As a result of these experiments, the SDC algorithm serves to dampen the effect of removal bias, as it will prevent a substantial proportion of fitness neutral crossovers from occurring. Whilst it is possible for fitness neutral crossovers to occur under this system, the removal bias theory remains a plausible explanation for the code growth present in our experiments.

Semantically driven crossover would have no noticeable effect on modification point depth

providing the swaps near the leaves of the tree were not in intron areas. If the swaps were in intron areas near the leaves of the tree, then semantically driven crossover would force modification point depth to be more aggressive at increasing the program size. Considering that semantically driven crossover hampers the build up of introns, it is less likely that swaps in the leaves of the program tree will be in intron areas, therefore, the effect of modification point depth in this situation is unchanged.

When considering crossover bias, it is suggested that smaller programs are more frequently sampled and removed by the fitness function due to poor fitness. Smaller programs have more simplistic behaviours and repeated sampling of these simplistic behaviours, in relation to the parent programs, would be prevented by semantically driven crossover. Therefore, semantically driven crossover, may slightly reduce the effect of crossover bias. A counter argument to this is that, there are many simplistic behaviours and that semantically driven crossover only compares child behaviour to the parent. The result of this would be that it is perfectly plausible to sample more smaller programs despite semantically driven crossover.

6.1.5 Conclusions

Semantically driven crossover can significantly increase performance of GP in the majority of problems presented and decrease code bloat in a selection of the problems presented. It has been demonstrated that the addition of behaviourally neutral code can be eliminated at the point of crossover using abstract representation. The higher performance level is unsurprising as the semantically driven crossover algorithm effectively forces programs to make more movements around the behavioural search space. Using the semantically driven crossover algorithm, probability would dictate that the search is wider which is supported by our results.

One factor that remains interesting is the high number of behaviourally neutral crossovers that occur under the standard GP process. This represents a substantial waste in terms of computation and reduces the power of the GP search. Despite the theory that introns protect valuable code from destruction in crossover, the combination of two factors would indicate that introns are not required to achieve high performance in GP. These factors are increased performance and in some cases smaller program sizes. Based on these two factors, and the

added computational burden of bigger programs, it can be argued that unreachable introns are undesirable in GP.

6.1.6 Future Work

There are several areas of interest following on from this work. One of the key aspects of whether a crossover will result in a behavioural change is how the code transplanted from the swap partner will work together with the rest of the program. A further understanding of linkage and context may yield clues as to how to make crossover more effective in terms of increasing the probability of changing the behaviour of a program with each crossover. With greater analysis, it may even be possible to intelligently guide crossover to be a more effective operator.

These experiments have demonstrated the benefit of a wider, behaviourally driven search to GP. It would be useful to conduct an experiment which forced the GP to move into new areas of the search space as well as quantify how many times behaviours are produced to analyse and behavioural bias. To do this GP could be hybridised with a semantic based Tabu search.

Finally, if the ability to measure the difference between two behaviours became available, it may be possible to enumerate the search properties of different crossover operators and as a result intelligently choose between crossover operators to aid in the GP search. For example, if GP prematurely converged, it may be possible to choose an operator that on average causes a high level of semantic change in order to continue searching the search space.

6.2 Semantically Driven Mutation

In this section, the semantically driven mutation (SDM) algorithm is described and evaluated. The motivation to create this algorithm is similar to that of the motivation for developing semantically driven crossover, in that forcing each mutation to take a new semantic step in relation to its parent program in the search space should improve performance as a larger search is conducted. Similarly to semantically driven crossover, the SDM algorithm works to improve performance by not allowing mutated programs to be produced when they are beha-

viourally equivalent to the original program. The aim of this is to avoid returning to sections of the search space that have already been traversed. The SDM algorithm is compared to standard sub tree mutation over the problems described in the test suite in section 4.1 and performance, program size and rejected mutations results are evaluated.

In a similar fashion to the semantically driven crossover algorithm, SDM is an extension to the development of different mutation algorithms (described in detail in section 2.7) and acts as a wrapper around existing mutation techniques using pre and post evaluation of the operator. The abstraction techniques used are set out in section 4.3. Whilst mutation initially was not favoured by GP practitioners (Koza [1992]) later reviews by Luke and Spector [1997, 1998] have shown mutation can be as effective as crossover during evolution.

In section 6.2.1, the experiment specific parameters are described. Section 6.2.2 presents the performance, program size and rejected mutation results and section 6.2.3 presents a discussion of the results. In sections 6.2.4 and 6.2.5, conclusions and suggestions for future work related to the use of the SDM are presented respectively.

6.2.1 Methods and Algorithms

The aim of this work is to demonstrate the positive effects of redesigning the mutation operator so that instead of merely altering syntax, it will attempt to alter the behaviour of programs. The SDM process is set out in algorithm 6.2.

The SDM algorithm will try to mutate a program into a new behavioural state. The process involves performing a standard sub tree mutation; however, after each mutation attempt, the algorithm checks to ascertain that each mutated program is semantically different to the original program. In some cases it may not be possible to semantically mutate a program (for example, if the program contained substantial inviable code) and as a result a counter system has been applied such that the mutation operator will have at most five attempts to behaviourally mutate a program, after which the original program is returned. Despite initial fears that this algorithm would be slow due to the creation of the representations of behaviour, run time appears roughly comparable to standard mutation.

The experiment suite used is as set out in section 4.1 with the following modifications. No crossover has been used in order to isolate this variable from the experiment. Mutation is applied with 0.9 probability and reproduction with 0.1 probability. Standard sub tree mutation

Algorithm 6.2 Semantically Driven Mutation Algorithm

```
for each program in population
  if random_no < prob_mutation
    counter = 0
    while counter < 5
      generate semantic_representation1 of program
      select mutation_point (uniform)
      generate sub_tree using grow (depth 4)
      insert sub_tree at mutation_point
      generate semantic_representation2 of mutated_program
      if semantic_representation1 ≡ semantic_representation2 {
        revert mutated_program back to program
      else
        break
      end if
      counter++
    end while
  end if
end for each
```

is used and a uniformly selected node is replaced with a new sub tree. The new sub tree is generated using the GROW depth 4 algorithm. Whilst Luke [2000a] highlighted a weakness the of GROW algorithm, a consistent method was required to generate new subtrees for experimental comparison and FULL and Ramped Half and Half were not suitable for the purposes of generating individual subtrees. FULL would generate new subtrees all of the same depth and RHH operates over a number of programs changing program depth and initialisation type which is not suitable for randomly generating one subtree.

6.2.2 Results

Table 6.5 shows that the SDM algorithm significantly improves the performance of GP runs in 8 out of 9 experiments. In the CUBIC experiment the overall maximum scores favoured traditional mutation. When comparing the scores at generation 50, the SDM produced better performing results in 4 out of 9 experiments. The other 5 results were statistically similar.

Table 6.6 shows that the SDM has no clear effect on program size. Two experiments reported shorter programs using the SDM, 4 with standard sub tree mutation and 3 with no statistical difference. Further to this, unlike semantically driven crossover, there appeared to be mixed program length results across individual domains.

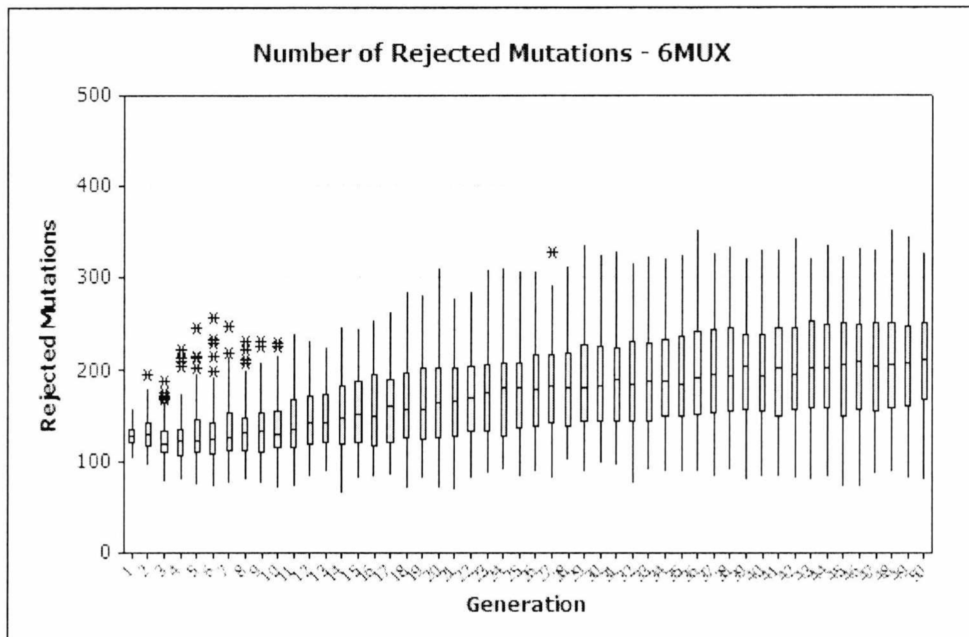
Figures 6.6 — 6.10 show varying trends of the rate at which mutations are rejected over

Problem	Experiment	Overall Max (StDev)	PT	G50 Max (StDev)	2T	Success
4PAR	SDM	**0.0768 (±0.0774)**	**0.00**	**0.0231 (±0.0423)**	**0.009**	74% (G9)
	MUT	0.0970 (±0.0749)	-	0.0400 (±0.0483)	-	52% (G6)
5MAJ	SDM	**0.0389 (±0.0426)**	**0.00**	**0.0088 (±0.0154)**	**0.00**	74% (G7)
	MUT	0.0527 (±0.0395)	-	0.0219 (±0.0241)	-	47% (G8)
6MUX	SDM	**0.0831 (±0.0553)**	**0.00**	0.0406 (±0.0478)	0.141	47% (G7)
	MUT	0.0946 (±0.0520)	-	0.0511 (±0.0520)	0.141	42% (G8)
7PAR	SDM	**0.3511 (±0.0583)**	**0.00**	**0.2680 (±0.0373)**	**0.009**	0% –
	MUT	0.3584 (±0.0549)	-	0.2813 (±0.0339)	-	0% –
9MAJ	SDM	**0.1576 (±0.0344)**	**0.00**	0.1204 (±0.0098)	0.224	0% –
	MUT	0.1595 (±0.0340)	-	0.1221 (±0.0104)	0.224	0% –
11MUX	SDM	**0.1942 (±0.0703)**	**0.00**	**0.1117 (±0.0380)**	**0.00**	0% –
	MUT	0.2070 (±0.0620)	-	0.1334 (±0.0452)	-	0% –
AASF	SDM	**0.3411 (±0.0733)**	**0.00**	0.2787 (±0.1315)	0.139	10% (G8)
	MUT	0.3685 (±0.0719)	-	0.3045 (±0.1202)	0.139	5% (G13)
CUBIC	SDM	989.18 (±209.76)	-	780.81 (±156.61)	0.963	0% –
	MUT	**984.24 (±212.18)**	**0.002**	779.80 (±150.75)	0.963	0% –
QUART	SDM	**1554.88 (±483.73)**	**0.00**	1088.59 (±385.14)	0.81	0% –
	MUT	1605.38 (±492.15)	-	1100.94 (±340.11)	0.81	0% –

Table 6.5: The table shows results using sub tree and semantically driven mutation. Problem is the problem being examined. Experiment is the type of mutation being used where MUT is standard mutation and SDM is semantically driven mutation. Overall max is the mean of the maximum scores over all generations and the StDev is the standard deviation of the maximum scores. The scores are presented in standardised fitness (lower = better). PT is the statistical result of paired T tests comparing the maximum scores over the generations. G50 Max is a mean of the maximum scores at generation 50 and StDev is the standard deviation of these maximum scores. The scores are represented in standardised fitness. 2T is the statistical result based on 2 sample T test of the maximum scores at generation 50. Success is the number of runs which reached full score and the bracket figures represent the first generation in any run at which full score was achieved. Scores for the discrete fitness functions (Boolean and Ant domains) have been scaled to be comparable. The scores quoted for the regression domain are raw absolute error.

Problem	Experiment	Overall Length (StDev)	PT
4PAR	SDM	**144.62 (±29.45)**	**0.00**
	MUT	159.08 (±39.87)	-
5MAJ	SDM	**82.27 (±21.75)**	**0.00**
	MUT	90.32 (±28.82)	-
6MUX	SDM	52.29 (±12.85)	0.60
	MUT	51.95 (±16.08)	0.60
7PAR	SDM	215.71 (±47.86)	-
	MUT	**206.04 (±44.67)**	**0.00**
9MAJ	SDM	86.78 (±25.30)	-
	MUT	**80.34 (±23.19)**	**0.00**
11MUX	SDM	41.02 (±8.87)	-
	MUT	**38.92 (±9.64)**	**0.00**
AASF	SDM	121.14 (±28.66)	-
	MUT	**111.65 (±26.02)**	**0.00**
CUBIC	SDM	70.00 (±11.15)	0.494
	MUT	70.19 (±10.76)	0.494
QUART	SDM	68.78 (±11.05)	0.222
	MUT	69.29 (±10.00)	0.222

Table 6.6: The table compares program lengths using sub tree mutation and semantically driven mutation. Problem is the problem being examined. Experiment is the type of mutation being used where MUT is standard mutation, SDM is semantically driven mutation. Overall Length is an average of the average program lengths for all generations and StDev represents the standard deviations of these mean lengths. PT indicates the result of a Paired T-test analysed at the 95% confidence level.



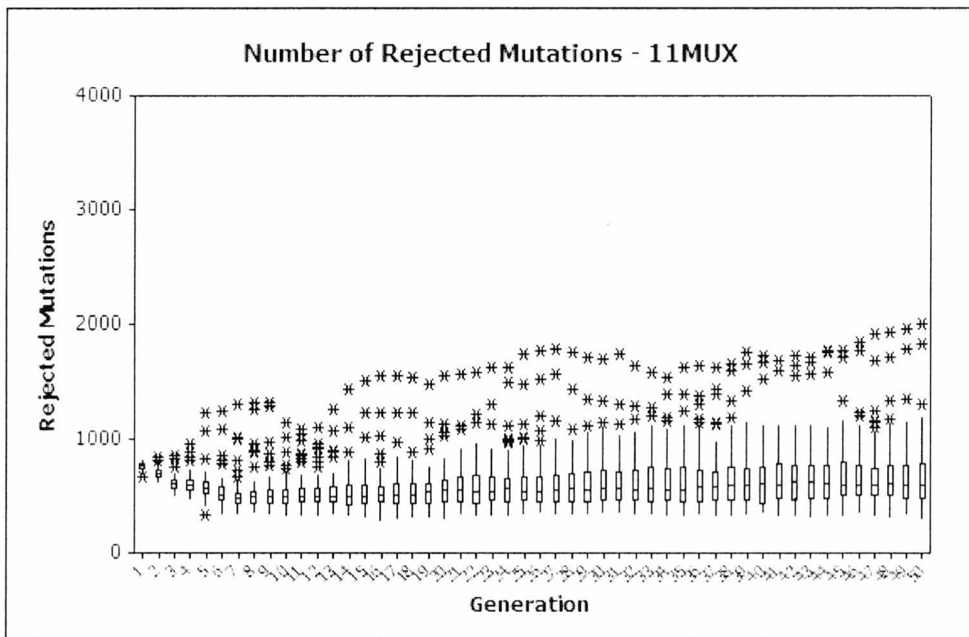
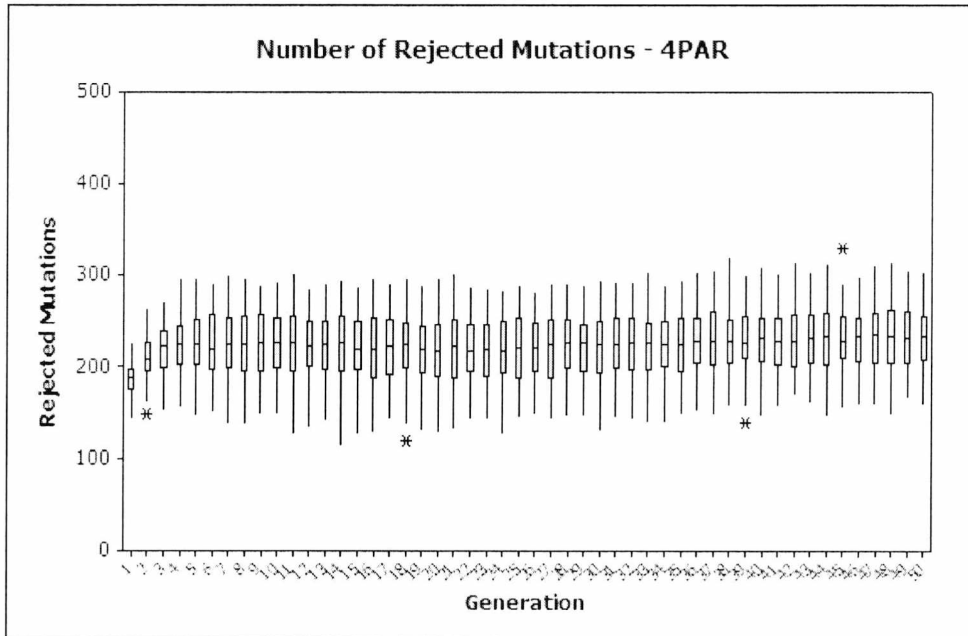


Figure 6.6: The graphs indicate the number of mutations that were rejected (or disallowed) by the SDM algorithm when considering the multiplexer problems. The Y axis have been scaled to represent the size of the population. All results are averaged over 100 runs.



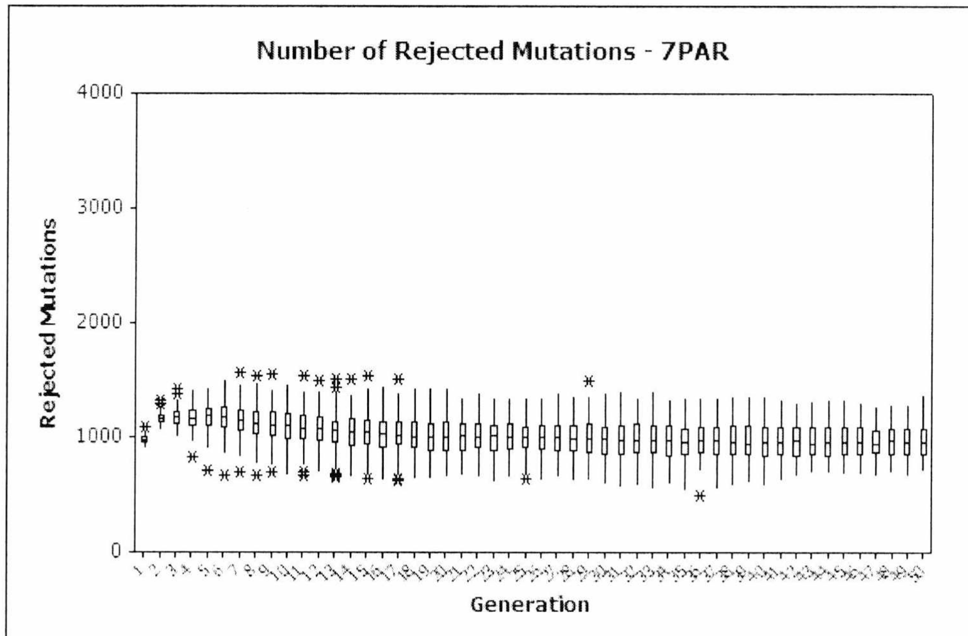
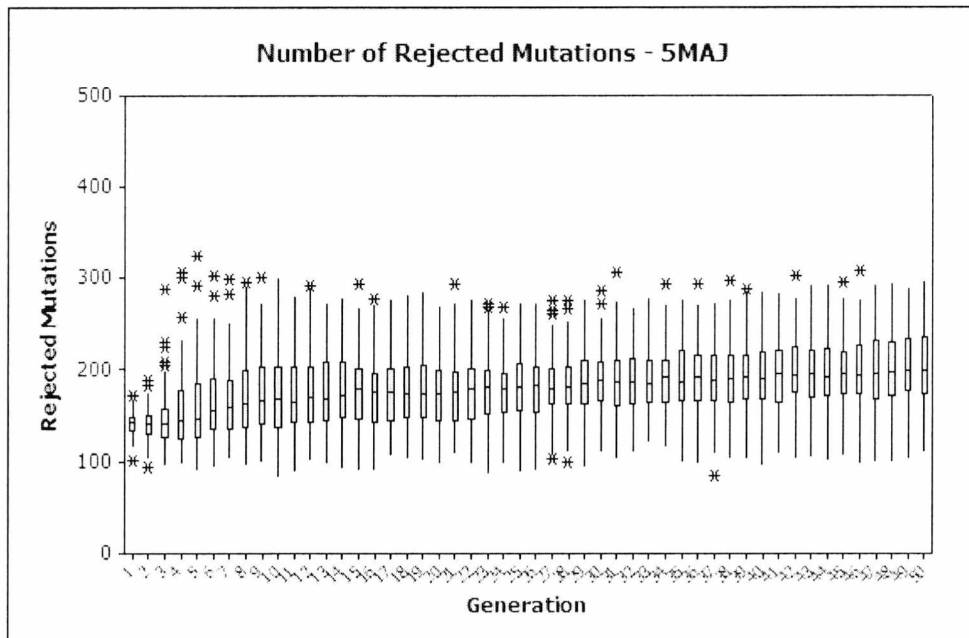


Figure 6.7: The graphs indicate the number of mutations that were rejected (or disallowed) by the SDM algorithm when considering the even parity problems. The Y axis have been scaled to represent the size of the population. All results are averaged over 100 runs.



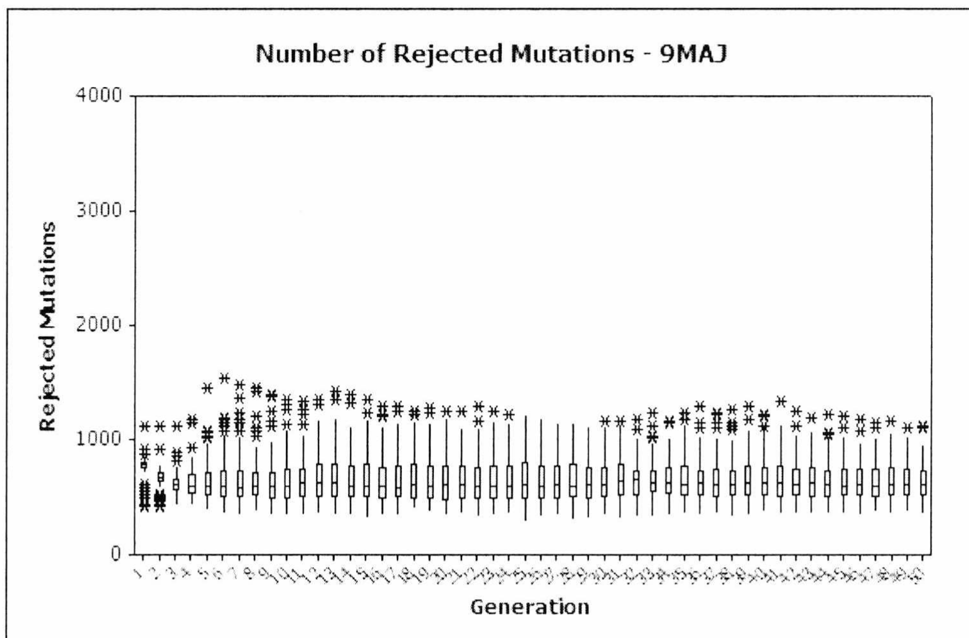


Figure 6.8: The graphs indicate the number of mutations that were rejected (or disallowed) by the SDM algorithm when considering the majority problems. The Y axis have been scaled to represent the size of the population. All results are averaged over 100 runs.

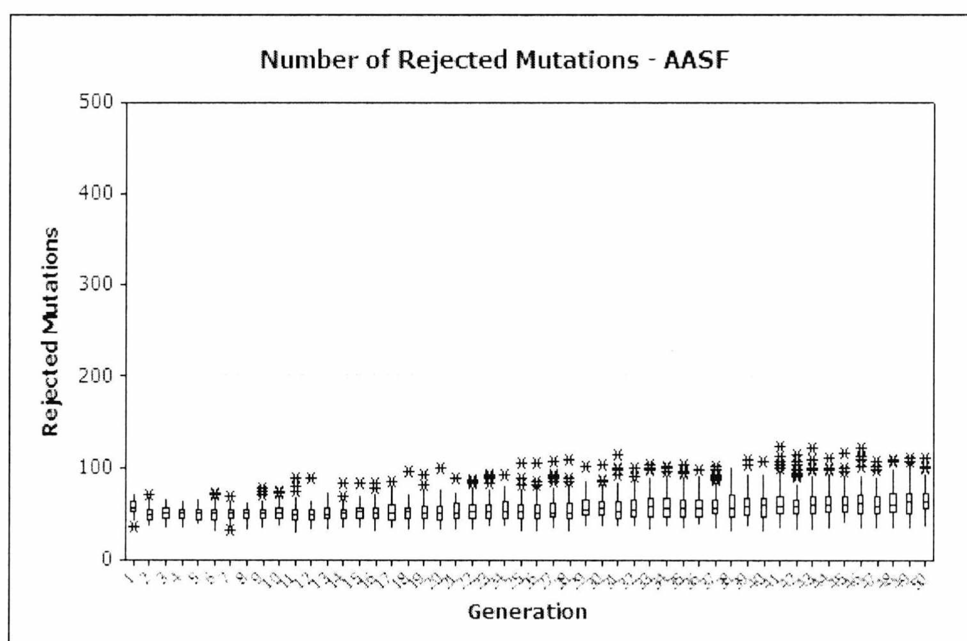
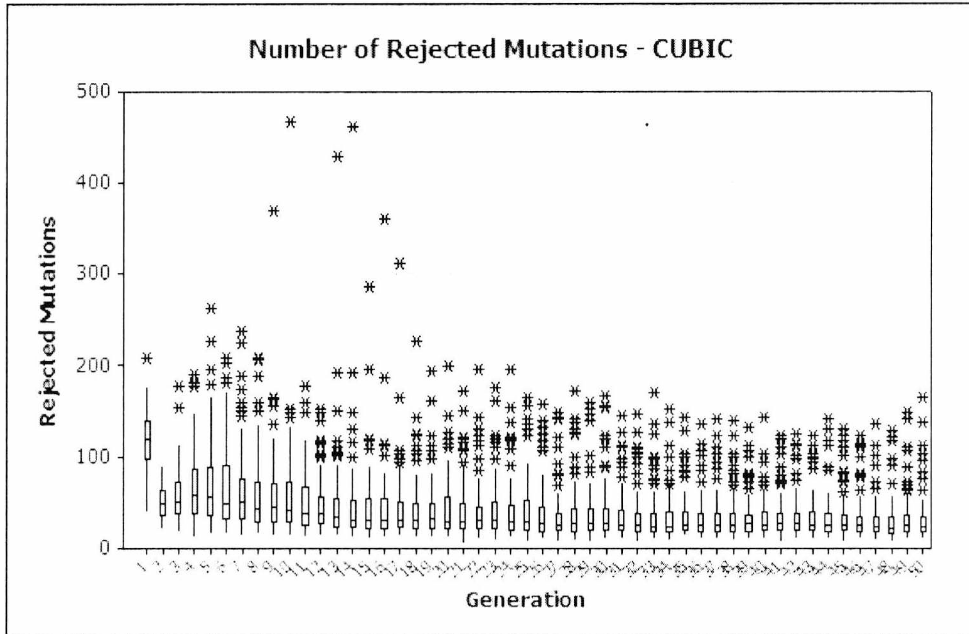


Figure 6.9: The graphs indicate the number of mutations that were rejected (or disallowed) by the SDM algorithm when considering the artificial ant problem. The Y axis have been scaled to represent the size of the population. All results are averaged over 100 runs.

the generations. One rejection counts when the first child produced by the SDM algorithm is equivalent to the original program. Further attempts to mutate the program (up to the five allowed) are not counted as rejected mutations. When considering the multiplexer experiments (figure 6.6), the 6MUX show a clear increasing trend in the number of mutations being rejected between 20% and 40% of all mutations as time goes on. The 11MUX shows a constant level of approximate 600 rejections over each generation. The parity experiments (figure 6.7) both show a constant level of rejections with over 40% for the 4PAR and approximately 25% for the 7PAR. The majority experiments (figure 6.8) show varying trends. The 5MAJ increases the level of rejected mutation to nearly 40% of the population. The 9MAJ has a constant level of approximately 600 rejections for each generation. The AASF (figure 6.9) shows very few rejections in comparison to some of the other problems featuring a constant level of approximately 50 rejections. The regression models (figure 6.10) present a contrast to similar results from other problems. Both CUBIC and QUART present a decreasing trend in the numbers of mutations being rejected over time.



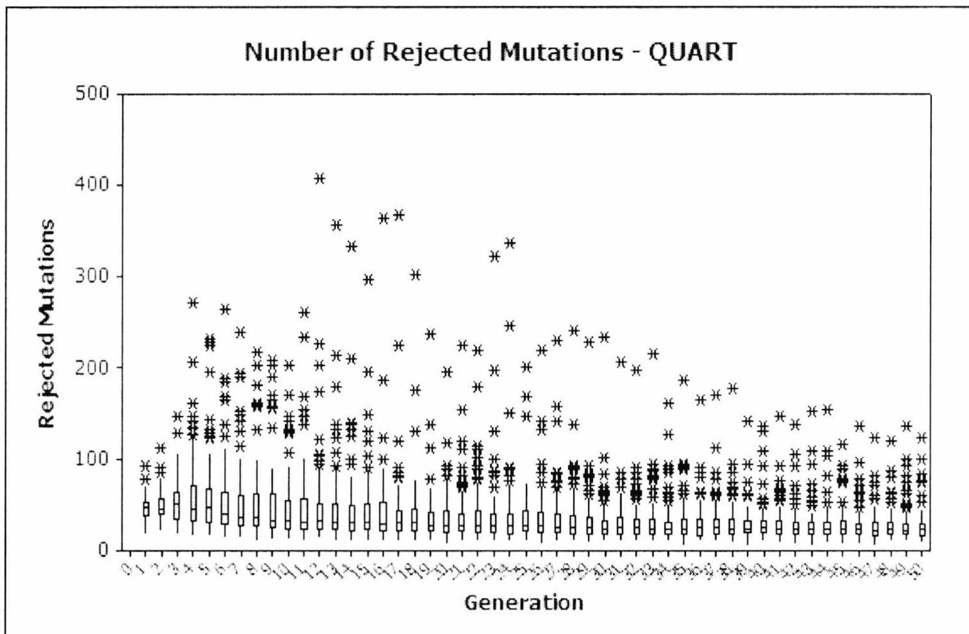


Figure 6.10: The graphs indicate the number of mutations that were rejected (or disallowed) by the SDM algorithm when considering the symbolic regression problems. The Y axis have been scaled to represent the size of the population. All results are averaged over 100 runs.

6.2.3 Discussion

The results presented indicate that the one to many relationship between behavioural and syntactic representations constitutes an inefficiency in performance. Using standard mutation, many fitness evaluations represent wasted computational effort, since the mutated program is semantically equivalent to the parent program. In all but one experiment, the overall comparisons demonstrated that the semantically driven mutation must force increased levels of movement (as evidenced by figures 6.6 — 6.10) around the search space. The result is that better solutions are found quicker compared to traditional sub tree mutation and this is evidenced in the performance of SDM (table 6.5).

A speculative explanation for the varying program size results is the possibility that the increased search forced programs to move to different regions of the search space which required different numbers of nodes in the trees. This effect would be problem specific and as such, varying results can be observed in the program sizes produced by the SDM algorithm when compared to standard sub tree mutation.

In addition to the GP performance and program size results, the percentage of programs being rejected was examined. Figures 6.6 — 6.10 show varying trends of rejection rates. It is difficult to predict the numbers of rejections expected to be present and different trends may be the result of different explanations. Considering 6MUX and 5MAJ, clear positive trends could be a symptom of increased intron areas being present as mutations take place causing many to be rejected due to swap point positioning. In a search context, the decreasing trends present in CUBIC and QUART may indicate that at first the SDM had to work hard to enforce diversity in the early generations, but once a diverse population became present, it was less likely that programs would converge again.

A final point of note is that the SDM has only been applied to a simple version of sub tree mutation. As mentioned in section 2.7 there are several other mutation techniques. Whilst each technique has a different mutation process, the SDM concept could be applied over the top of each of these different mutation processes. One fact to be drawn from increasing semantic diversity in standard sub tree mutation is the increase in overall performance noted in all but one of our experiments, and there is no reason to assume that applying the SDM concept on other mutation processes cannot demonstrate a similar increase in GP performance.

6.2.4 Conclusions

In conclusion, there are two key points to draw from the results generated using semantically driven mutation. Firstly, the SDM algorithm significantly increased the performance of GP in all but one experiment. This indicates that SDM is a valuable mutation operator because of its ability to increase performance in the majority of experiments.

Secondly, the variable rejections rates indicate that an explanation of the effect for using the SDM is non trivial. Constant, increasing and decreasing trends are observed and despite attempting to correlate these trends with program size, there appears to be no link. All that can be provided is speculative reasons for the level of rejected mutation present when using the SDM algorithm.

6.2.5 Future Work

There are two key areas of future work the use of the SDM lead to. Firstly, the simple path is to use the SDM with other types of mutation operator (for example, hoist, point or even size fair homologous). The mechanics of this are relatively simple to implement, however, given that increasing diversity in both crossover and mutation appears to have a positive effect on GP performance, speculatively, one could predict that this will simply increase performance for each of the operators separately.

The second area worth examining is to study the level of semantic change caused by mutation operators. This information could be used in two ways. Firstly, it could be used by practitioners to make intelligent choices as to which operators to use. Secondly, this semantic change information could be compared with similar information relating to the crossover algorithm to examine whether crossover and mutation exhibit different search properties (Angeline [1997]).

6.3 Semantic Pruning

This section presents an algorithm to reduce program bloat during evolution using a semantics based technique. The objective is to completely remove both unreachable and redundant introns from candidate programs during evolution. The reason for performing these experiments is to examine the effects on evolution of removing introns through pruning. By remo-

Algorithm 6.3 Semantic Pruning Algorithm

for each *candidate_program* in *population*
  translate *candidate_program* to *representation*
  if *representation* is a *constant_behaviour*
    swap *candidate_program* for a *random_terminal*
  else
    translate *representation* back to *reduced_candidate_program*
    swap *candidate_program* for *reduced_candidate_program*
  end if
end for each

ving all introns from the candidate programs, it becomes possible to examine how introns may or may not be valuable to evolution.

In order to achieve this goal, a semantic pruning technique, which reduces programs via generating a behavioural representation of syntax and then translates the behaviour back to a reduced syntax tree is presented. Semantic pruning represents a novel step, as unlike other bloat control techniques (descriptions in section 3.2.2), and more specifically unlike program reduction or editing, semantic pruning completely rebuilds the program tree from a canonical representation of behaviour. This characteristic results in semantic pruning falling into the category of *code reconstruction* which is equivalent to refactoring a program to reduce its size.

As a method to reduce program size, semantic pruning achieves a dramatic reduction in bloat, in some cases rendering resultant programs human readable. The price of the dramatic level of bloat reduction is varying performance results when compared to a control experiment.

6.3.1 The Semantic Pruning Method

The pruning algorithm is executed after crossover and mutation operators and immediately prior to selection, and consists of translating each member of the population into a canonical semantic form, then reconstructing a syntax tree from that representation. This process removes all introns from the code.

Algorithm 6.3 sets out the semantic pruning process. Programs are reduced unless they are classed as a constant behaviour (as set out in section 4.3.4). When programs are constant behaviours, they are swapped for a random terminal. The reason for this is that constant behaviours do not represent a behaviour that uses any of the input behaviours and

Problem	Experiment	Max Score	PT	Max Score G50	2T	Success (Gen)
4PAR	Pruned	0.1514 (±0.0030)	-	0.1499 (±0.0160)	-	0%
	Control	**0.0949 (±00735)**	**0.00**	**0.0425 (±0.0494)**	**0.00**	50% (G9)
5MAJ	Pruned	0.1280 (±0.0097)	-	0.1253 (±0.0031)	-	0%
	Control	**0.0406 (±0.0396)**	**0.00**	**0.0159 (±0.0245)**	**0.00**	63% (G9)
6MUX	Pruned	**0.0255 (±0.0522)**	**0.00**	**0.0088 (±0.0321)**	**0.00**	93% (G3)
	Control	0.0763 (±0.0498)	-	0.0431 (±0.0529)	-	51% (G6)
7PAR	Pruned	**0.2585 (±0.1121)**	**0.00**	**0.0951 (±0.0417)**	**0.00**	2% (G40)
	Control	0.3286 (±0.0685)	-	0.2333 (±0.0259)	-	0% –
9MAJ	Pruned	0.1018 (±0.0356)	-	0.0548 (±0.0079)	-	0% –
	Control	**0.0912 (±0.0362)**	**0.00**	**0.0496 (±0.0089)**	**0.00**	0% –
11MUX	Pruned	**0.0472 (±0.0936)**	**0.00**	**0.0019 (±0.0107)**	**0.00**	97% (G10)
	Control	0.1019 (±0.0906)	-	0.0339 (±0.0401)	-	46% (G14)
AASF	Pruned	0.2802 (±0.0628)	0.161	0.2655 (±0.0990)	-	6% (G3)
	Control	0.2828 (±0.0722)	0.161	**0.2300 (±0.1160)**	**0.02**	11% (G5)
CUBIC	Pruned	1046.11 (±146.25)	-	956.76 (±178.29)	-	0% –
	Control	**816.26 (±287.25)**	**0.00**	**498.64 (±204.69)**	**0.00**	0% –
QUART	Pruned	1694.87 (±448.78)	-	1336.42 (±473.88)	-	0% –
	Control	**1258.54 (±596.09)**	**0.00**	**711.89 (±311.47)**	**0.00**	0% –

Table 6.7: The table shows results comparing the performance of pruned algorithm to traditional GP runs. Experiment shows which are the control GP experiment and the pruned GP experiment. Max Score indicates an average of the maximum scores (± the standard deviation). Standardised fitness is used (all results normalised to fall between 0 and 1, except continuous regression domain), so lowest values indicate best performance. PT indicates the P-Value obtained from doing a Paired T-test comparing the pruned and control experiments. The result is aligned with the best performing result. Max Score G50 is the average of the maximum scores at generation 50 (± the standard deviation). 2T shows the P-Value of a 2 sample T-test comparing the maximum scores at generation 50. The value is aligned with the best result. Success is the number of runs that reach full score and (Gen) is the first generation in which a full score was obtained from any run.

as a result cannot be back translated to a valid syntax. This process iterates over the whole of the candidate program population.

The experimental parameters are as set out in section 4.2 and experiments are conducted on the experiment suite outlined in section 4.1. The semantic pruning process is performed before fitness is assessed and selection takes place.

Problem	Experiment	Mean Program Length	PT
4PAR	Pruned	**16.76 ($\pm$2.92)**	**0.00**
	Control	248.88 ($\pm$89.61)	-
5MAJ	Pruned	**14.79 ($\pm$2.61)**	**0.00**
	Control	178.44 ($\pm$79.54)	-
6MUX	Pruned	**8.87 ($\pm$3.05)**	**0.00**
	Control	112.62 ($\pm$50.95)	-
7PAR	Pruned	**141.58 ($\pm$41.68)**	**0.00**
	Control	527.81 ($\pm$232.21)	-
9MAJ	Pruned	**138.65 ($\pm$58.64)**	**0.00**
	Control	351.43 ($\pm$206.38)	-
11MUX	Pruned	**23.55 ($\pm$6.49)**	**0.00**
	Control	136.08 ($\pm$78.85)	-
AASF	Pruned	117.62 ($\pm$25.45)	0.581
	Control	118.18 ($\pm$29.16)	0.581
CUBIC	Pruned	**15.23 ($\pm$5.48)**	**0.00**
	Control	161.87 ($\pm$62.33)	-
QUART	Pruned	**20.76 ($\pm$8.01)**	**0.00**
	Control	168.51 ($\pm$62.69)	-

Table 6.8: The table shows a comparison of program lengths using the pruned algorithm and traditional GP runs. Problem indicates the test problem being studied. Experiment indicates whether pruning is being used. Mean Program Length is the mean number of nodes present in the trees in all generations. PT is the P-Value result of a Paired T-test. In the case of statistical significance the P-Value is aligned with the best result.

6.3.2 Results

Table 6.7 shows that semantic pruning can dramatically effect performance in GP. Overall, there appears to be no clear pattern as to whether semantic pruning is a benefit or a hindrance. When comparing overall performance, 5 experiments favour the control runs, 3 favour the pruned runs and 1 is statistically equivalent. At generation 50 a similar mix of performance is visible. In 6 experiments the control runs are better performing and in 3 experiments the pruned runs produce better performance. Whilst some domains clearly produce better performing runs for the control runs (for example, the symbolic regression experiments), and others for the pruned runs (for example, the multiplexer experiments) some outcomes are less obvious. The even parity results are contrasting in that runs perform better for the 4PAR using control parameters and better for the 7PAR using pruning.

Table 6.8 shows that semantic pruning dramatically decreases program sizes in all cases

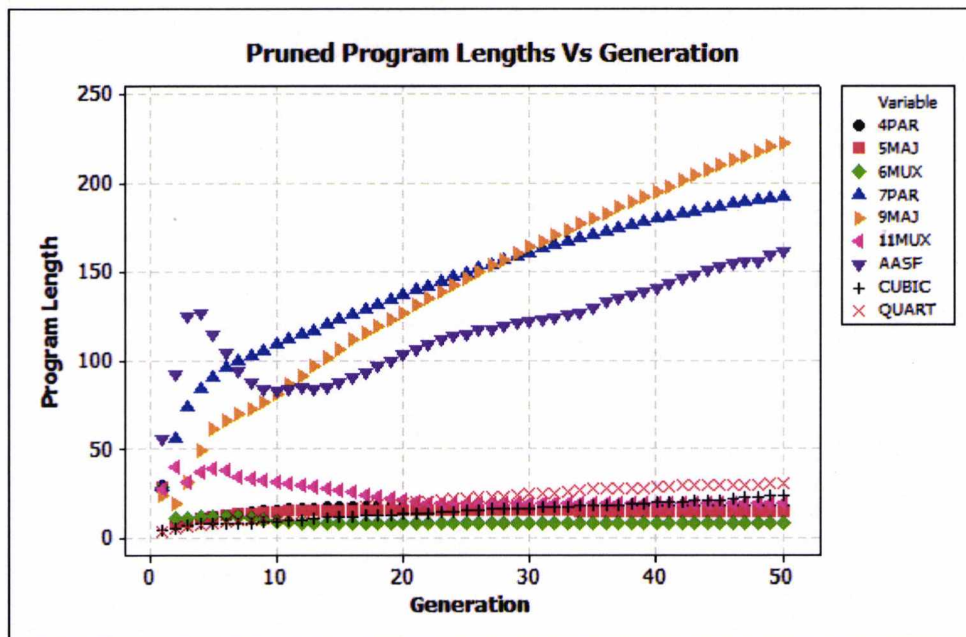


Figure 6.11: The figure shows the length of the pruned programs over the generations. All data is averaged over 100 runs.

but the artificial ant. In addition to vastly reduced program sizes, many of the programs produced by the pruning algorithm are human understandable. For example, a common resultant program output for the 6MUX problem was:

```
Program 1 Score = 0.0
IF A0 ( IF A1 D0 D1 ) ( IF A1 D2 D3 )
```

Whilst the above example of output is easy to note down for humans (as an answer for the 6MUX), it is difficult for GP to generate a minimal candidate solution such as this. Using semantic pruning, simplified and accurate statements such as the above are common output programs.

Figure 6.11 shows how the average lengths of the pruned programs change over the course of a run. Despite the ability of semantic pruning to control program size, programs in a selection of the experiments continue to grow in size as the generations pass.

6.3.3 Discussion

It is clear from the results in table 6.8 that semantic pruning is a very effective mechanism for removing introns from program trees. Combined with this, it is clear from the performance

results presented in table 6.7, that this reduction can have both positive and negative effects on the evolvability of programs. As a result of this, an investigation into the role of program structure in evolution may yield clues as to the evolvability of programs. The analysis of program structure will be developed in greater detail in chapter 7.

Whilst on the subject of program shape, the contrast in results brings into question the way in which programs are translated from behavioural representation to syntactic form (as outlined in section 4.3). For the Boolean experiment, the ROBDDs only provide one obvious route in which to translate the behaviour to syntax. However, this reconstruction of syntax can be affected by variable ordering of the ROBDD (Downing [2006b]). Other domains are less clear, and the statistical outlier in table 6.7 was the result for the artificial ant problem.

The artificial ant, in this situation, has produced statistically equivalent overall performance results and statistically similar tree lengths. One difference, when compared to the Boolean back translation mechanism (behaviour to code) is that the ant back translation mechanism produces full trees rather than deep thin trees. It is intriguing that this has resulted in equivalent performance between the two experiments, as it suggests that standard initialisation (Ramped Half and Half) operates more towards producing bushy trees. This is in agreement with Luke [2000a] analysis of the GROW algorithm. Furthermore, it is also in agreement with the conclusion of Langdon [2000], in that the initialisation process would have a notable effect on the initial structure of the population as well as the continuing population structure throughout evolution.

With the mixed results presented, the question remains as to the value of introns in GP evolution. In terms of performance, they are not needed for the multiplexer problems or the 7PAR problem. Theoretically, introns are not needed to succeed at any of the other problems, so the question remains as to why such poor performance is obtained in several of the experiments presented? One speculative answer, could be that introns are required purely for "padding" to make high performing tree shapes more readily acceptable to evolutionary search. In a highly multimodal landscape GP could easily do worse than random search and neutrality can help to improve the situation. This would tie together the works of Daida et al. [2003] with other works involving research into neutrality in GP, for example, Downing [2005], Galvan Lopez et al. [2005].

Despite the semantic pruning process, figure 6.11 shows that in some experiments pro-

grams continue to grow. The only difference in this situation is that the code growth is valuable exon code and as a result, candidate solutions should be moving towards an optimum, possibly global, but maybe local. A possible explanation for the mixed effects of pruning on program growth could be the concept of structural steps as a form of fitness step. Traps in the form of visual cliffs in the 3 dimensional search space are not uncommon topics, however, a structural step in this case may be slightly different. Consider the possibility that a search agent is very close to a solution in the search space; however, in order to change its behaviour (phenotype) to take the next small search step, the candidate program would have to substantially, or improbably modify itself at the syntactic level in order to achieve the change in behaviour. This is similar to the concept of locality (Rothlauf [2006]) and in this situation, problem difficulty would be complimented by a structural difficulty. Referring back to figure 6.11, this may explain why some problems are able to grow (with 100% effective code) towards a solution while others stagnate at a particular size (or achieve full fitness) as the structural steps for different problems may be at different program sizes.

Introns would allow neutral evolutions (Downing [2005, 2006a]) which may help smooth out the fitness steps, both in terms of problem hardness and structural evolutions. This would be in direct contrast to the aggressive nature of the semantic pruning procedure. An alternative approach to alleviate this problem would be to combine semantic pruning with search operators that can force behavioural change and examine the effects. This set of ideas is studied in further detail in section 6.4.

When considering how the results presented relate to existing theories of bloat, the fact that introns are completely removed (with the exception of the ant with simulated introns) does impact upon some of the explanations of bloat. The semantic pruning implementation by design will not allow protection from deletion to function effectively, and as such, we should see marked decrease in performance, however, our results are conflicting on this issue. Removal bias is also effectively removed because there are no smaller inactive subtrees near the leaves of a program tree to be exchanged for larger inactive subtrees. As part of the fitness causing bloat explanation, this system would prevent groups of larger programs with similar fitness values occurring, given that semantic pruning can reduce redundant but executable code. As a counter argument, the concept of programs favouring moving towards a particular program size potentially links with program shape in that the fitness function is

causing programs to favour a particular structure. Not allowing this movement to take place could be a partial explanation for the variations in performance presented in these results.

6.3.4 Conclusions

In the context of the reduction of bloat, semantic pruning is effective at reducing program size, but it also demonstrates that pruning might not necessarily be an ideal process to use during the course of a GP run, due to its dramatic effect on performance in a non consistent manner. The mixed results produced by semantic pruning demonstrate that whilst a formal reduction mechanism is being used to improve GP efficiency, it is not a simple task to reduce programs without reducing evolutionary potential. It has been shown that semantic pruning can be either, very successful, a complete failure or produce statistically indistinguishable results and this problem dependence indicates that other forces are at work, namely the role of program structure and the locality of program structures during evolution.

In line with the work of Daida et al. [2003] and Langdon [2000], it is clear that program structure (or the lack of evolvability of program structure) has effected these results in the Boolean domain. Whilst the artificial ant mechanism uses a different reduced tree construction method (producing full trees), it acts as a counter example supporting the importance of the shape of the program trees during evolution.

Program structure, being one of or an additional causes of either GP hardness or problem dependence, would add further complexity for the GP practitioner. On one level of difficulty, the practitioner must design an algorithm that is capable of finding an optimal solution in a search space of undulating fitness, and on a second level the GP practitioner needs to take care to ensure that a representation does not need to be optimised itself (forming a particular shape) in order to be able to provide the best possible solution to all GP problems.

6.3.5 Future Work

Based on this work, there are two major avenues for future work. Firstly, there needs to be more research into representation in GP. This work highlights the potential for weaknesses of the tree structure and the potential additional complexity through shape optimisation that it causes during the course of a GP run. Whilst other authors have utilised other representations (Teller and Veloso [1996], Banzhaf et al. [1998]), there is scope for additional

theoretically backed work (such as Rothlauf [2006]), but more specifically for GP.

Secondly, if GP practitioners are to continue using the tree representation, then there is a need for further research in controlling program shape. One way of doing this could be implemented from initialisation, for example, using modified versions of Luke's probabilistic tree creation algorithms Luke [2000a] to bias the shapes of initialised programs. A second approach may be to enforce a variety of structures in the population and control the values of the nodes using probability such as Salustowicz and Schmidhuber [1997] probabilistic incremental program evolution. A final approach for controlling size and shape distributions or programs is using crossover equalisation (Dignum and Poli [2008a]). This would control the size and shape of programs used in crossover, therefore influencing the structure of programs during evolution.

6.4 Intron Free Genetic Programming

This section combines all of the work presented so far in chapter 6 in order to create intron free GP with behavioural search. The motivation for this is to increase the efficiency of GP by moving program syntax a step closer to program behaviour. This can be harnessed to increase the power of search operators (sections 6.1 and 6.2) and to reconstruct programs in their most efficient form by removing all introns (section 6.3).

A secondary motivation for this particular set of experiments is that the role of introns in GP has remained unclear and several of the experiments in the previous sections and chapters have demonstrated that introns may not be required in order to achieve good performance in GP. The ability to explicitly remove all introns from every point in GP evolution provides the opportunity to formally test whether the presence of introns is beneficial or detrimental to GP.

There are two potential criticisms of these procedures. The first question that may be asked is "Why not just use a canonical representation for GP?". The answer to this lies in two parts. Firstly, it is extremely difficult to create a representation that will remain canonical during the evolution process. For example, performing GP using ROBDDs as the representation would not result in perfectly reduced binary decision diagrams after each operation. The same problem occurs for all of the abstraction mechanisms contained in this thesis and the

abstraction representations used all have a series of reduction rules that need to be applied in order to achieve a canonical form. Secondly, if GP is to expand as a main stream form of code generation then it would be helpful for GP to be able to operate on main stream forms of code. Whether its Java, C++ or another language, it is relatively easy to achieve the same behaviour using different syntax.

The second criticism is to do with the semantic methods being more computationally expensive that traditional GP. The response is that this work focuses on performing GP correctly and efficiently rather than just using larger populations and longer runs. Problems with larger numbers of input-output combinations would be the first to benefit as a larger number of input-output combinations makes the fitness function more computationally expensive. This is compounded by repeatedly assessing the fitness of semantically equivalent programs. Forcing new search steps in relation to parent programs means that less computational time is spent assessing the fitness of equivalent programs. Whilst there is some trade off in the computational effort required to build representations of behaviour, in theory, GP would need to perform less fitness evaluations to find good candidate solutions.

6.4.1 Methodology

The parameters used on the experiment suite are aimed at eliminating introns at every stage of GP. With the exception of the parameters mentioned below, all other parameters are as specified in section 4.2.

For initialisation, hybrid semantically driven initialisation has been used in order to remove introns in the starting population. Chapter 5 demonstrated the difficulty in constructing a behaviourally diverse starting population and the hybrid semantically driven initialisation has been chosen, not because it was the best, but because it was never the worst performing technique in the experiments. The other initialisation methods presented extreme results from problem to problem.

A probability of 0.45 is used for semantically driven crossover in order to ensure behaviourally novel child programs are produced. A probability of 0.45 is used for semantically driven mutation, again to ensure the production of behaviourally novel children. A probability of 0.1 is used for reproduction. After this, should any programs be produced that have introns through linkage issues with new subtrees, semantic pruning will be applied to remove these

Problem	Experiment	Max Overall (±StDev)	PT	Max G50 (±StDev)	2T	Success (Gen)
4PAR	Intron Free	**0.0510 (±0.0754)**	**0.00**	**0.0056 (±0.0253)**	**0.001**	95% (G6)
	Control	0.0765 (±0.0778)	-	0.0231 (±0.0423)	-	72% (G5)
5MAJ	Intron Free	**0.0119 (±0.0220)**	**0.00**	**0.0009 (±0.0053)**	**0.00**	97% (G5)
	Control	0.0235 (±0.0222)	-	0.0122 (±0.0166)	-	63% (G5)
6MUX	Intron Free	**0.0165 (±0.0525)**	**0.00**	**0.0000 (±0.0000)**	**0.002**	100% (G3)
	Control	0.0417 (±0.0605)	-	0.0069 (±0.0211)	-	89% (G3)
7PAR	Intron Free	**0.2779 (±0.1035)**	**0.00**	**0.1243 (±0.0430)**	**0.00**	0% –
	Control	0.3311 (±0.0696)	-	0.2330 (±0.0411)	-	0% –
9MAJ	Intron Free	0.1028 (±0.0324)	-	0.0592 (±0.0074)	-	0% –
	Control	**0.0944 (±0.0355)**	**0.00**	**0.0534 (±0.0081)**	**0.00**	0% –
11MUX	Intron Free	**0.0563 (±0.1024)**	**0.00**	**0.0000 (±0.0000)**	**0.00**	100% (G12)
	Control	0.0791 (±0.1001)	-	0.0195 (±0.0241)	-	79% (G13)
AASF	Intron Free	**0.3015 (±0.0596)**	**0.00**	**0.2587 (±0.0631)**	**0.03**	0% –
	Control	0.3354 (±0.0747)	-	0.2797 (±0.0724)	-	0% –
CUBIC	Intron Free	952.16 (±132.56)	-	841.30 (±205.95)	-	0% –
	Control	**862.63 (±271.55)**	**0.00**	**563.43 (±212.81)**	**0.00**	0% –
QUART	Intron Free	1283.39 (±408.23)	0.11	991.47 (±280.31)	-	0% –
	Control	1330.51 (±587.96)	0.11	**755.52 (±313.15)**	**0.00**	0% –

Table 6.9: The table shows the performance of GP when comparing runs containing no introns to a control experiment. Problem indicates which problem is being tackled and Exp indicates whether the run was a control run or an intron free run. Max overall indicates the best performing score overall ± the standard deviation (in discrete domains scores are scaled against total score for comparison). PT indicates the result of a Paired T-test comparing the maximum overall scores by generation. The P-Value is aligned with the best performing result. Max G50 indicates the average maximum score at generation 50 (in discrete domains scores are scaled against total score for comparison). 2T indicates the result of a 2 sample T-test comparing the maximum scores at G50. The P-Value is aligned with the best performing result. Success shows the number of runs to reach full score and generation indicates the earliest any run reached full score. All results are averaged over 100 runs.

introns from the code.

6.4.2 Results

Table 6.9 shows the performance results comparing intron free GP with a control experiment. Overall, intron free GP performed best in 6 out of 9 experiments and the control in 2 out of 9 experiments. The QUART experiment is statistically similar overall. At generation 50, intron free GP performed best in 6 experiments and the control performed best in 3 experiments.

Table 6.10 shows a comparison of the programs lengths generated by the intron free

Problem	Experiment	Mean Length	PT
4PAR	Intron Free	19.05 ($\pm$2.97)	0.00
	Control	233.40 ($\pm$82.30)	-
5MAJ	Intron Free	17.12 ($\pm$3.18)	0.00
	Control	143.12 ($\pm$61.46)	-
6MUX	Intron Free	10.79 ($\pm$3.16)	0.00
	Control	105.83 ($\pm$48.56)	-
7PAR	Intron Free	132.00 ($\pm$38.65)	0.00
	Control	409.44 ($\pm$165.64)	-
9MAJ	Intron Free	134.21 ($\pm$48.47)	0.00
	Control	299.43 ($\pm$171.93)	-
11MUX	Intron Free	39.70 ($\pm$26.16)	0.00
	Control	113.20 (64.31)	-
AASF	Intron Free	83.78 ($\pm$28.91)	0.00
	Control	106.11 ($\pm$30.87)	-
CUBIC	Intron Free	18.55 ($\pm$7.61)	0.00
	Control	130.68 ($\pm$43.46)	-
QUART	Intron Free	27.91 ($\pm$9.75)	0.00
	Control	138.32 ($\pm$46.17)	-

Table 6.10: The table shows the program length results comparing no intron GP to a control GP experiment. Problem indicates the problem being tacked. Experiment indicates whether the results are for the intron free or control experiment. Mean length indicates the mean length of the programs and PT indicates the results of a Paired T-test comparing the mean lengths. The P-Value is aligned with the smallest programs. All results are averaged over 100 runs.

Problem	Rejected Crossovers	Rejected Mutations
4PAR	128.19 ($\pm$46.15)	17.86 ($\pm$2.62)
5MAJ	174.03 ($\pm$44.52)	14.88 ($\pm$2.21)
6MUX	191.00 ($\pm$73.1)	11.88 ($\pm$1.74)
7PAR	323.90 ($\pm$130.90)	79.13 ($\pm$11.71)
9MAJ	270.90 ($\pm$99.00)	64.96 ($\pm$9.56)
11MUX	478.10 ($\pm$316.6)	53.78 ($\pm$7.97)
AASF	3.84 ($\pm$0.82)	15.54 ($\pm$2.26)
CUBIC	20.29 ($\pm$4.36)	3.28 ($\pm$0.51)
QUART	13.90 ($\pm$3.82)	3.40 ($\pm$0.53)

Table 6.11: The table shows the rejection levels of semantically driven crossover and semantically driven mutation. Problem indicates the test problem being considered. Rejected crossovers indicates the mean number of crossovers rejected ($\pm$ the standard deviation) and rejected mutations indicates the mean number of mutations rejected ($\pm$ the standard deviation). All results are averaged over all generations and 100 runs.

experiments and the control experiments. Unsurprisingly, the program reduction ability of semantic pruning prevails and in every experiment and shorter programs are produced for the intron free runs. The artificial ant result is interesting because when the problem was assessed with pruning only, statistically similar program sizes were recorded, yet the program size is statistically less in the no intron experiment.

Table 6.11 compares the mean rejection rates of semantically driven crossover and semantically driven mutation for each of the problems tackled. It is difficult to directly compare the numbers of rejected crossovers and mutation due to the different way in which they are implemented (SDC taking new parents each time and SDM trying five times to mutate a program); however, there are obvious discrepancies between experiments such as mutation attaining more rejections that crossover in the artificial ant problem.

6.4.3 Discussion

Despite many theories concerning introns (Luke [2000b], Soule and Foster [1998], Tackett [1994], Banzhaf et al. [1998], Nordin et al. [1995]), the experiments presented show no clear and consistent pattern as to the role of introns. Whilst the results presented in table 6.7 show that the complete removal of introns through pruning alone on balance adversely effects performance, it is not a consistent result. Furthermore, with the application of semantically dri-

ven search operators in order to enable the search to behaviourally move around the search space and overcome structural steps, the performance of intron free GP suddenly improves notably to being an on balance positive effect on evolution in the experiments presented.

Table 6.9 shows that whilst performance is improved compared to semantic pruning alone, performance across domains is inconsistent. This inconsistency is troubling, especially when theories such as hitchhiking, protection from deletion and removal bias are built on the presence and importance of introns. The results presented simply do not support any consistent value of either unreachable or redundant introns and as such the importance of theory based upon the inconsistent value of introns should be cautioned against.

Studying the results in table 6.9, it is reasonable that an explanation of the purpose of introns may be proposed in the form of a problem specific concept. This leads on to a consideration of the fitness landscape of individual problems. As mentioned in the sections 6.3 and briefly in section 2.1, characteristics of the search space such as fitness steps may be responsible for the perceived value of introns.

As an example explanation for this problem, consider the contrast between the Boolean and symbolic regression domains. In the Boolean domain, discrete levels of fitness and as such behaviour are present. In contrast, the symbolic regression domain is continuous and so the fitness steps could be very small. The continuous domain with very small steps between different fitness values would allow a number of similar but not equivalent candidate programs to be present. This can be evidenced by the low rejection rates for the symbolic regression experiments as shown in table 6.11. In comparison, problems in the Boolean domain show relatively large numbers of rejection rates indicating that evolution could be struggling to move to a new fitness value. One possible reason for this is that there is less chance of identical functionality in the continuous symbolic regression domain. There are also two other concepts to consider. Firstly, there is the idea of a taking fitness step which is changing a program such that it can move to a different fitness value. Secondly, there is the idea of a structural step, which is how much of the genotype needs to be changed in order to cause a new fitness value of the program. It may be the case that small structural steps in the symbolic regression domain are enough to cause small fitness changes and that in the Boolean domain, larger structural steps are required in order to cause a fitness change.

Whilst semantically driven crossover and mutation will help overcome fitness and structu-

ral steps, they are potentially not a perfect solution. Consider that a candidate program has reached a fitness step. Consider also that the fitness steps around the candidate program are of different sizes. The iterative approach of the SDC and SDM algorithms and the probability of a major structural alteration being relatively low, due to the nature of a tree would result in the SDC and SDM favouring the more probable structural steps compared to the less probable ones when presented with a choice.

Both large structural steps and small structural steps could potentially cause problems for the semantically driven operators. Small steps on a continuous domain result in a difficulty trying to leverage semantic power to the search as evidenced by the symbolic regression continuous fitness landscape. Large steps may cause a bias to more easily accessible areas of the search space due to the probability of representation change through the search operator. In contrast traditional search operators may take more search steps to achieve the same goal as they would require the build up of introns in the representational structure in order to be able to move over a difficult structural step.

This further questions the way search is conducted in GP. Poli and Langdon [1998] showed that different operators presented different search properties. Based only on the different numbers of rejected crossover and mutations in table 6.11, it would be reasonable to suggest that semantically driven operators result in different search properties in comparison to traditional operators and that these properties may even be domain specific. Comparison of the numbers of rejections for SDC and SDM algorithms indicate a difference in search properties between crossover and mutation in this analysis.

It may be the case that introns are required to smooth out structural steps in the search space and as such aid the search operators available to GP practitioners on difficult fitness landscapes. The tree representation of GP does not directly correlate (in that the size of a change to the genotype may not correspond to the size of a change of a phenotype) to the fitness landscape in terms of low locality and as such introns may be required to bridge large gaps in the representation of the genotype in comparison to a small change in the phenotype between two fitness states.

6.4.4 Conclusions

In theory, intron free GP should increase the power of the GP search and in the majority of the experiments presented, intron free GP is very effective. In practice, introns may be required problem dependently in order to allow larger stuctural changes to genotypes to occur and as a result allow the creation of a different phenotype. This calls into question the way in which search operators function and the representations upon which they function. In order to search freely, search operators need to be able to navigate the fitness landscape without the hindrance of having to bridge large gaps between genotypes to enable movement between behavioural states. Ideally, there would be high locality present between genotype and phenotype and in theory, this would result in the behaviour of search operators being more predictable in comparison to the current situation.

6.4.5 Future Work

This section leads to three suggestions of areas for future research. Firstly, further investigation needs to take place (beyond that of Poli and Langdon [1998] and considering mutation as well as crossover) into the search properties of different search operators over a range of domains (Dignum and Poli [2007], Dignum [2008], Dignum and Poli [2008a]). A deeper understanding of this feature may enable the intelligent choice of search operators or even a theoretically based design of new search operators for GP.

Secondly, an evaluation of representation and comparison of different representation types could be conducted to enable a deeper understanding of representational concepts such as locality and its effects on GP. An evaluation similar to that of Rothlauf [2006], but specifically for GP would be valuable for the community.

Finally, considering the tree representation in current GP, an analysis of program shape through evolution may yield clues as to the requirements evolution makes of the tree representation. More specifically, to assess whether program structure itself influences the course of evolution.

Chapter 7

An Analysis of Program Structure in Genetic Programming

The aim of this chapter is to analyse program structure during the course of evolutionary runs using GP. The study presented in this chapter is motivated by the results obtained in section 5.2.4, which demonstrated that the shape of programs at initialisation could have an effect on overall GP performance. In section 6.3, code reconstruction using semantic pruning demonstrated varied effects on the performance of GP and even with the combination of semantically driven operators (in section 6.4), performance results were still varied in comparison to traditional GP.

The results generated using semantic pruning indicate that the structure of programs defined by the tree representation has at least some role to play in evolution; however, the theory of locality adds complexity to the study. In GP, low locality is an issue and as such it implies that changes in the phenotype of a program are disproportionate to changes in the genotype of the program. Despite the issue of locality, results in this thesis (section 5.2.4, 6.3 and 6.4) and research by other authors Daida et al. [2001, 2003], Daida [2003, 2004] indicate that shape may have a role to play in evolution.

The aim of this chapter is to evaluate the role of shape in evolution from a more detailed perspective than previously outlined in this thesis. In section 7.1, the methodology used in this chapter is outlined. In section 7.2, the shape profiles of the tree representation are profiled over time for the different problems in the test suite (section 4.1). In section 7.3, the changes in shape profiles are compared to changes in fitness in each of the test problems

in order to examine locality. Sections 7.4 and 7.5 discuss and conclude the findings of this chapter respectively and section 7.6 suggests areas for future research based on results presented in this chapter.

7.1 Methodology

The problems examined in this section are as described in section 4.1 and the general parameters are as described in section 4.2, unless a specific experiment is indicated to use different general parameters. In addition to a standard GP run, the numbers of nodes at each depth of each program tree have been recorded. This provides information which describes the composition and structure of the trees. This information is averaged over the population in each generation in order to create an average shape of a program tree for a population in a generation.

Averaging this information may be controversial as the differences between program structures could be substantial; however, it easy to demonstrate on a microscopic level (individual program level) that structural preferences and levels of locality can occur. See figure 2.4 for an example of locality at the microscopic level. It is more interesting to demonstrate the level of change of the structure of programs in a population during evolution. Once this level of change can be measured, it is possible to compare how the level of structural change in a population correlates to the level of fitness change in a population, during an evolutionary run.

All experiments have been run 100 times for averaging to ensure the effects of profiling are accurate when GP is run repeatedly.

7.2 Profiling Program Structure

Figures 7.1 — 7.9 show how the structure of programs evolve over generations. The figures present some interesting common effects and some problem specific effects.

A very noticeable common effect is the early evolution of an increase in the number of nodes at depths 5 and 6. This increase in the number of nodes at depths 5 and 6 usually occurs before generation 10. After generation 10, the programs appear to grow in size in

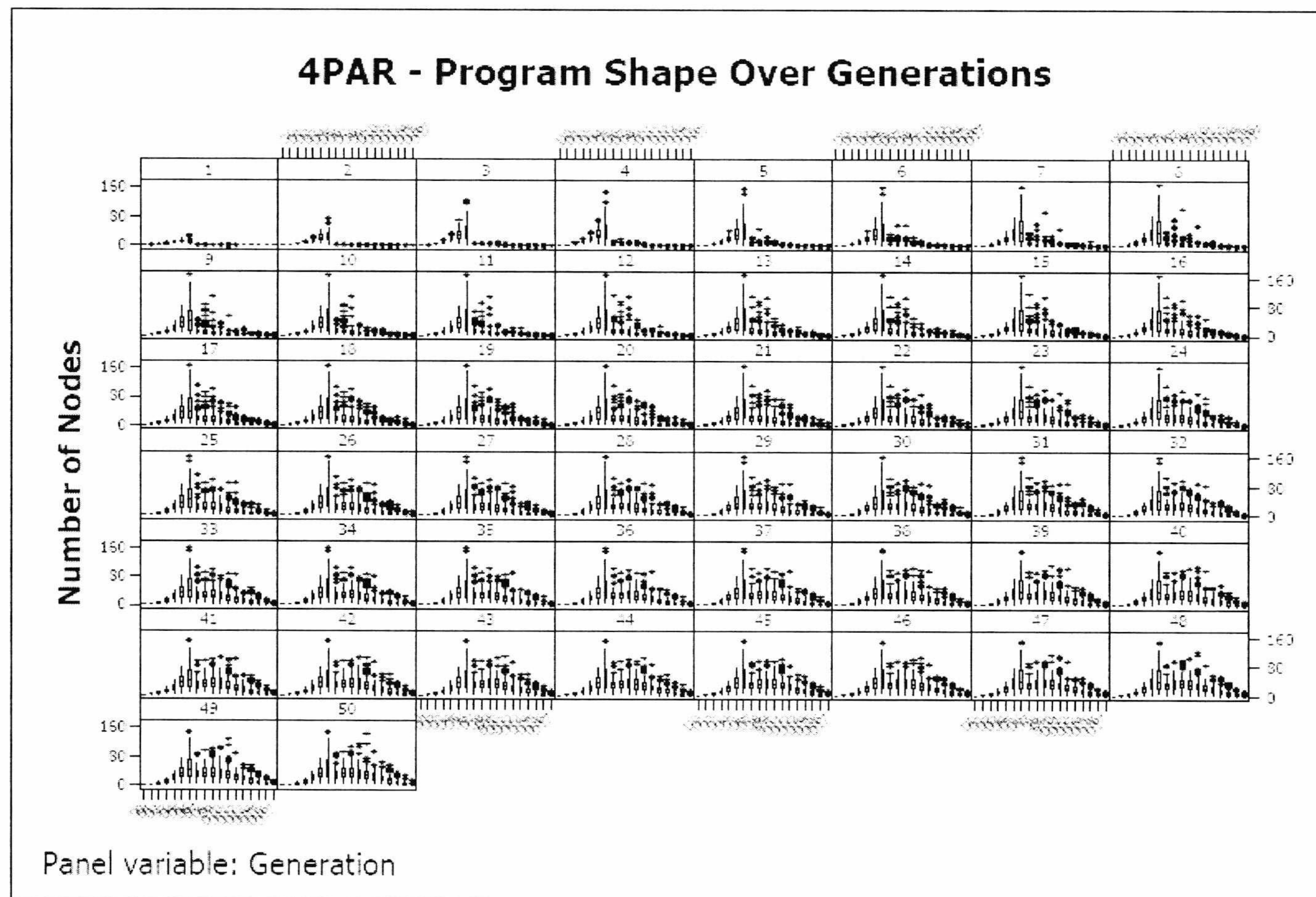


Figure 7.1: The figure shows a box plot representing the average program structure of a population over generations for 4PAR. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs.

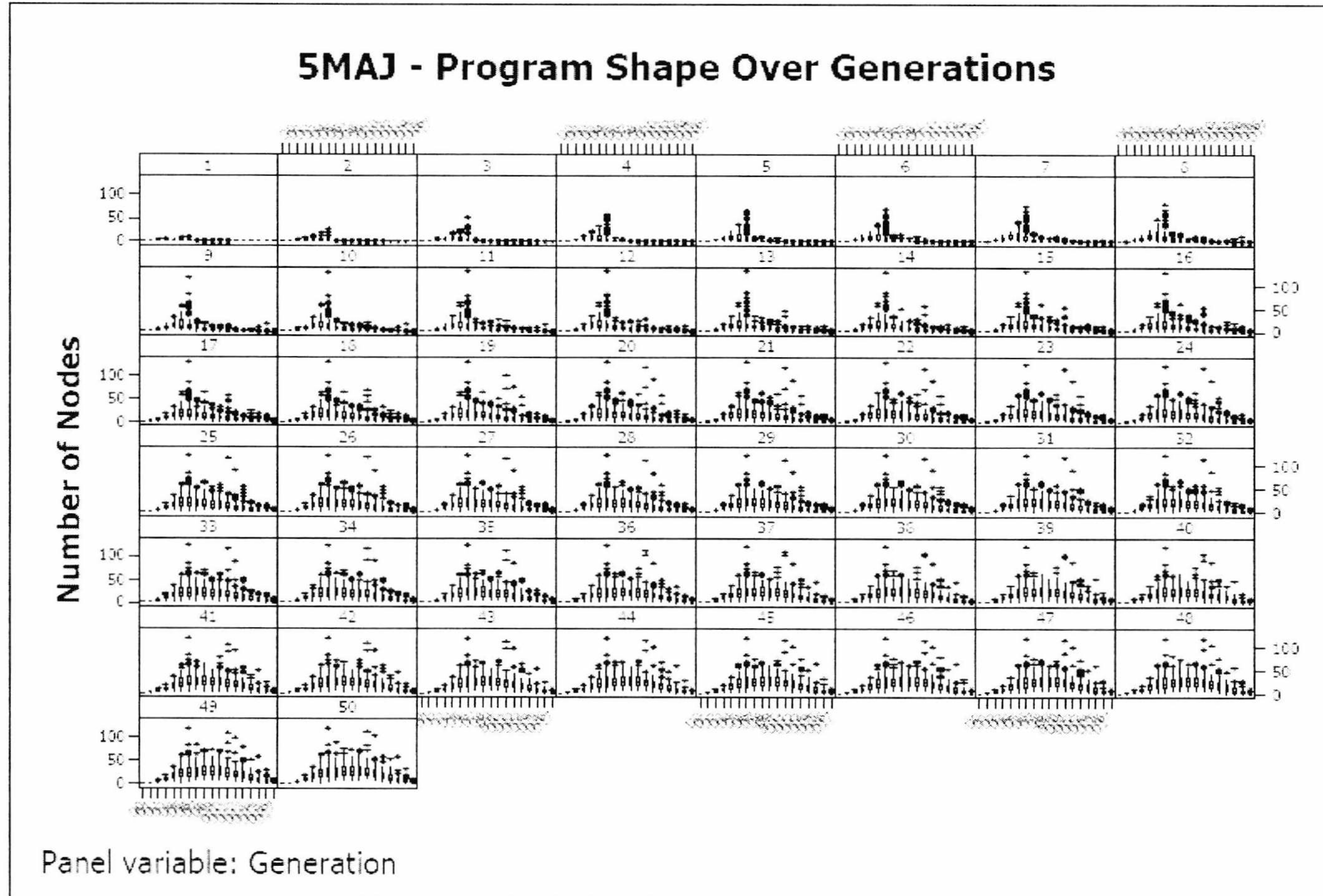


Figure 7.2: The figure shows a box plot representing the average program structure of a population over generations for 5MAJ. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

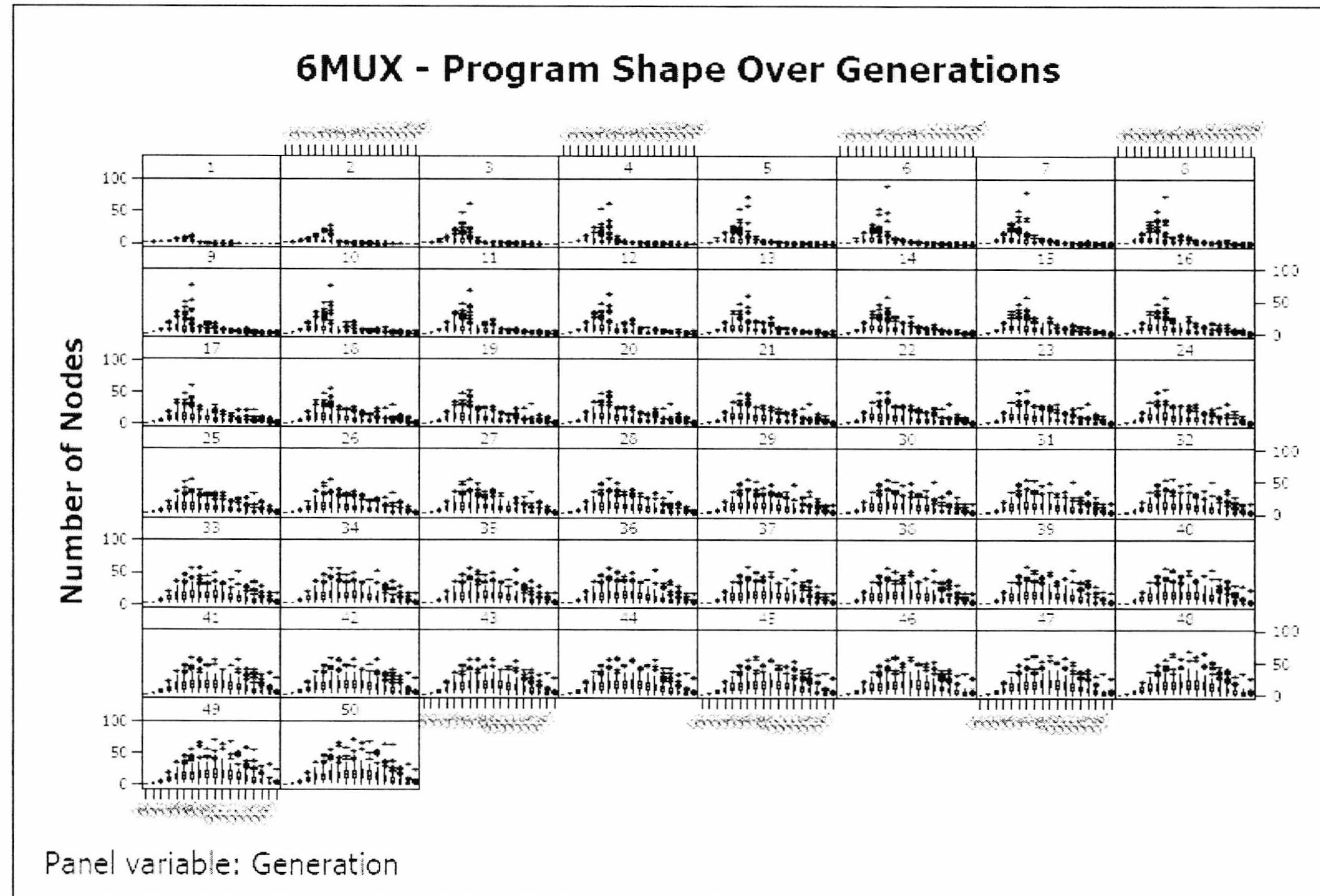


Figure 7.3: The figure shows a box plot representing the average program structure of a population over generations for 6MUX. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

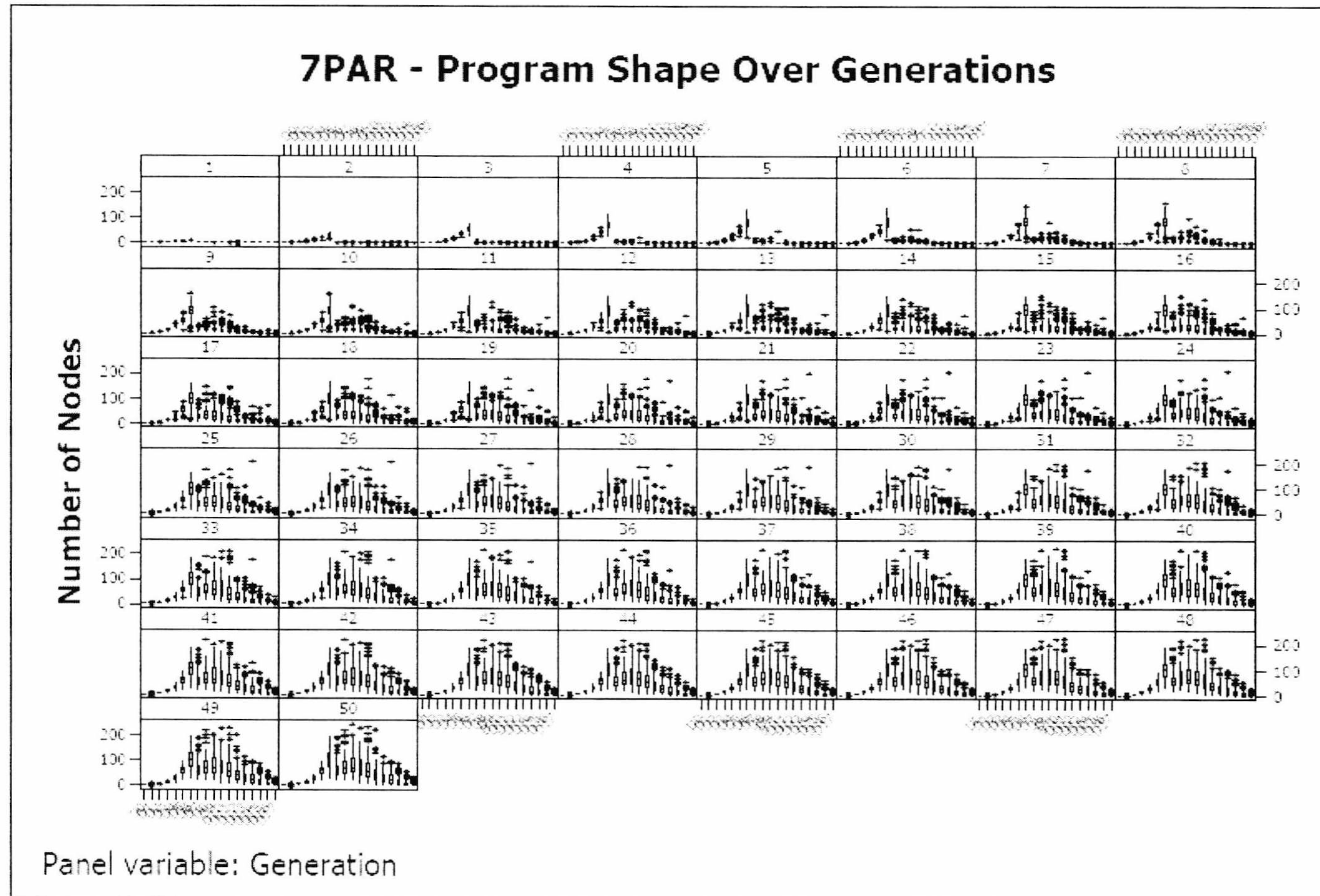


Figure 7.4: The figure shows a box plot representing the average program structure of a population over generations for 7PAR. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

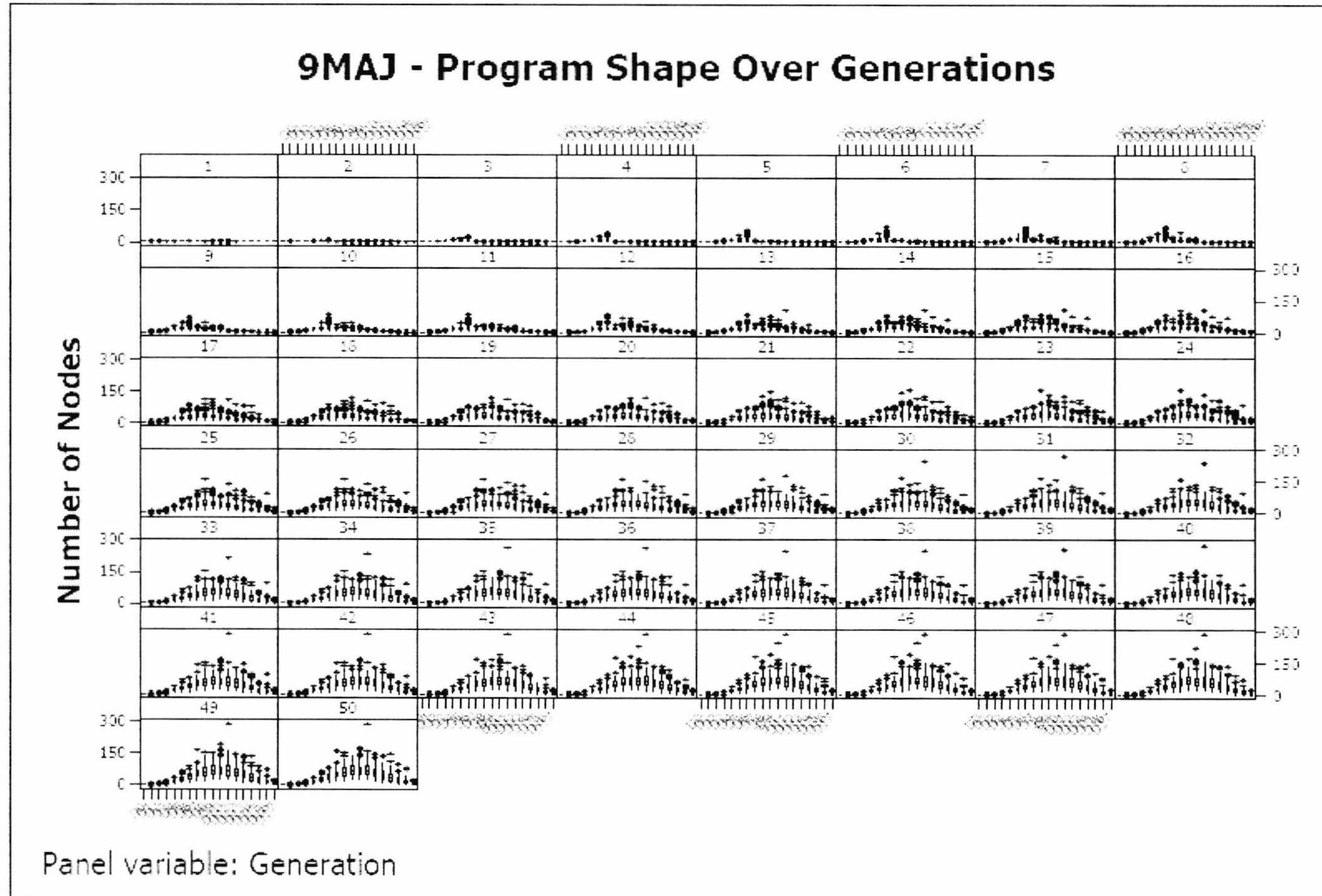


Figure 7.5: The figure shows a box plot representing the average program structure of a population over generations for 9MAJ. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

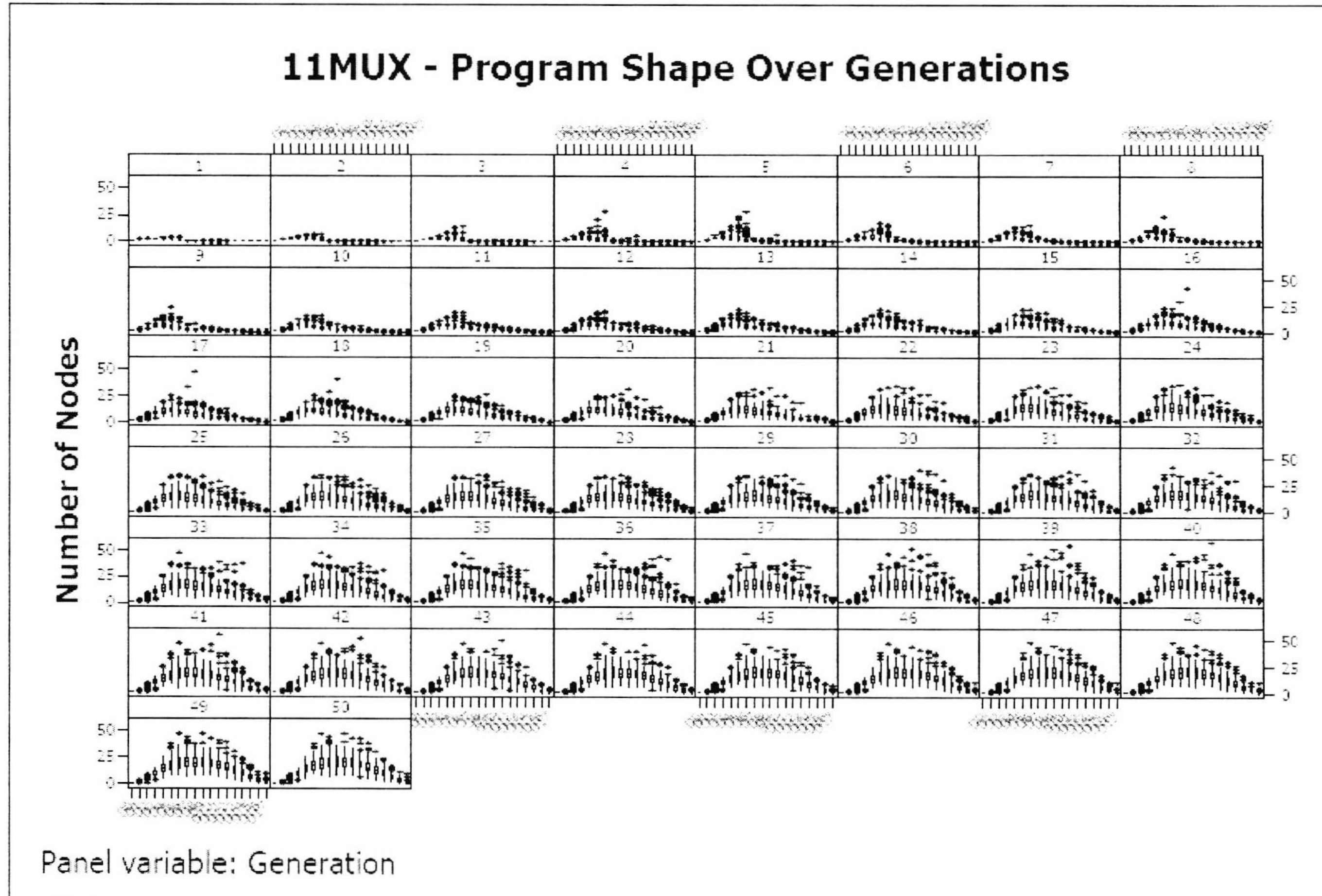


Figure 7.6: The figure shows a box plot representing the average program structure of a population over generations for 11MUX. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

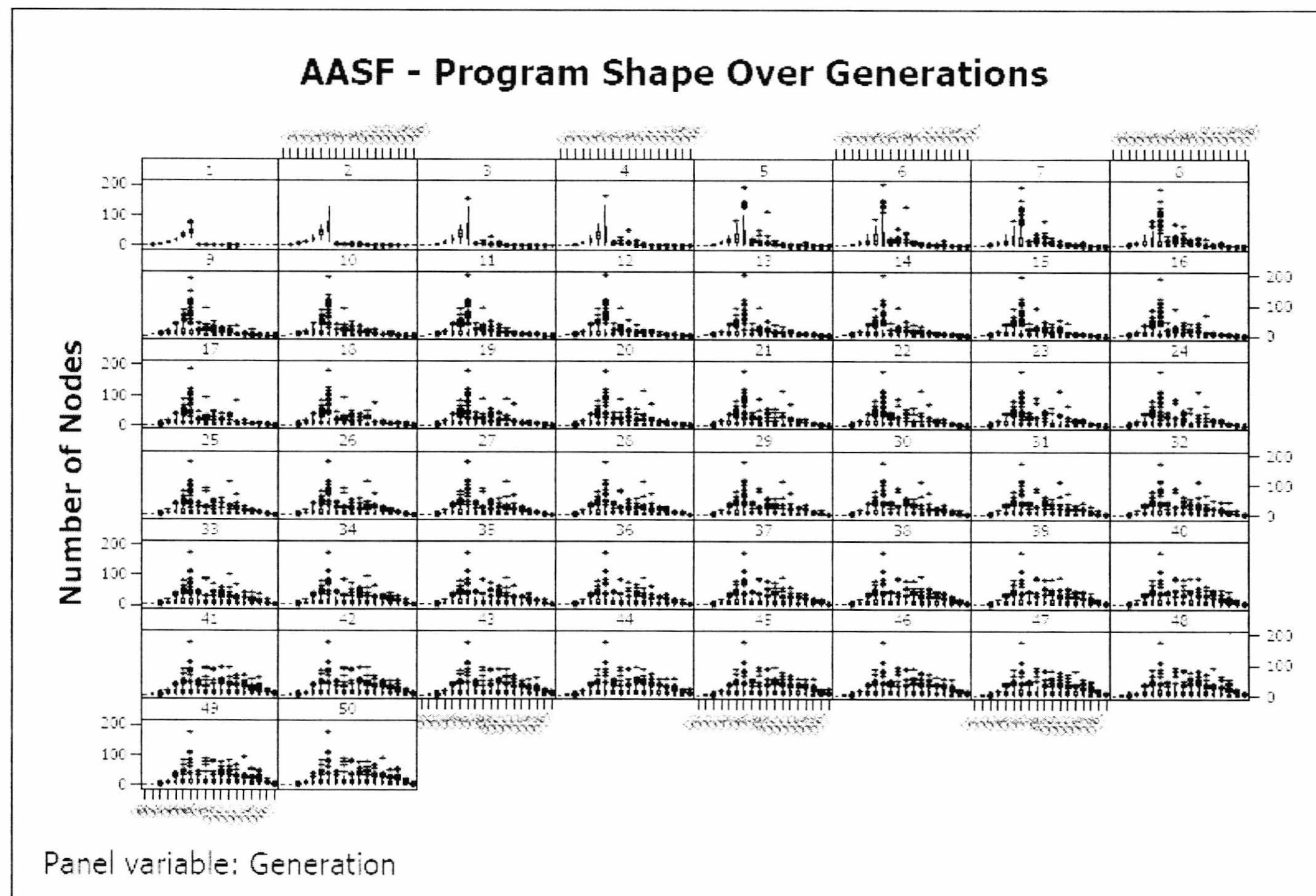


Figure 7.7: The figure shows a box plot representing the average program structure of a population over generations for AASF. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

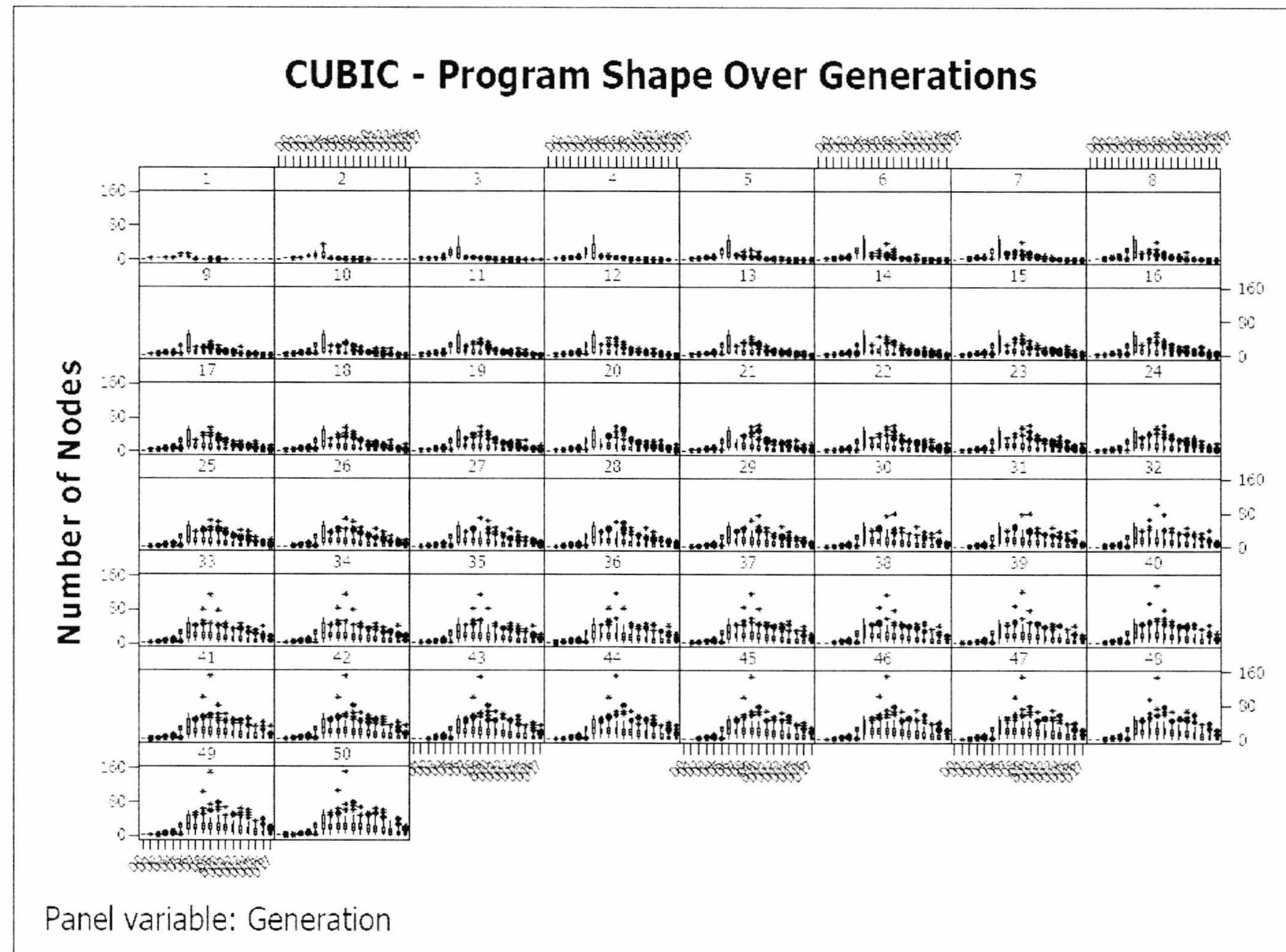


Figure 7.8: The figure shows a box plot representing the average program structure of a population over generations for CUBIC. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

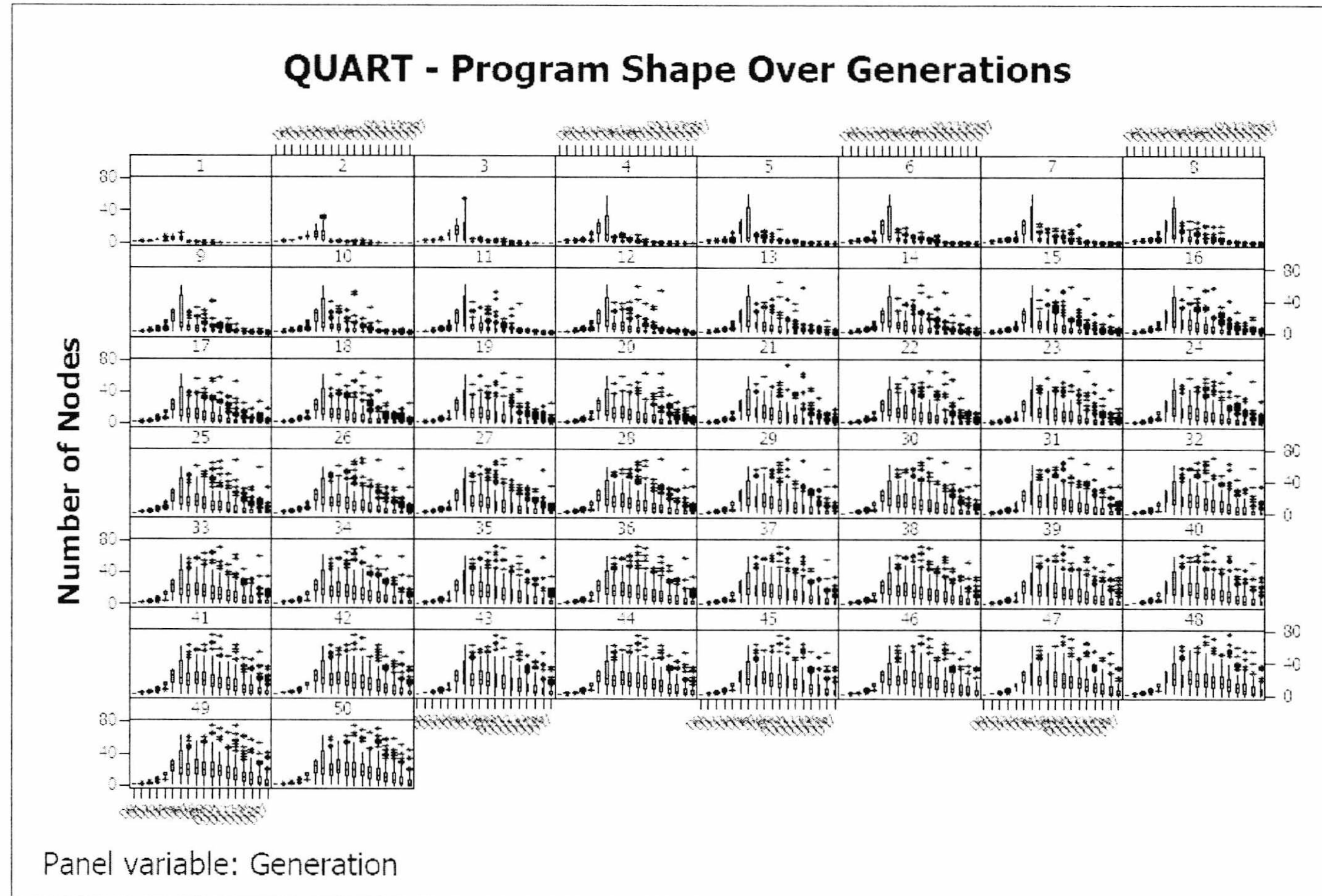


Figure 7.9: The figure shows a box plot representing the average program structure of a population over generations for QUART. For each box, the number of nodes at depth 0 is on the left side and depth 17 on the right side. All results are averaged over 100 runs

the lower depths (greater than depth 6). There appears to be a gradual padding out of the number of nodes at depth 7 and greater throughout the remainder of the generations after generation 10. A final noticeable effect is that the majority of outliers are all greater that the mean of the box plot. This would imply that some runs substantially bloat in comparison to others, yet none appear to substantially shrink compared to other runs.

Problem specific effects are in the form of spikes and hills. In the 4PAR, 7PAR, AASF, CUBIC and QUART, the spike generated at depth 6 appeared to remain until the end of evolution. In the 5MAJ, 9MAJ, 6MUX and 11MUX, a smooth hill was generated which reached a greater depth than the initially evolved spike at depth 6.

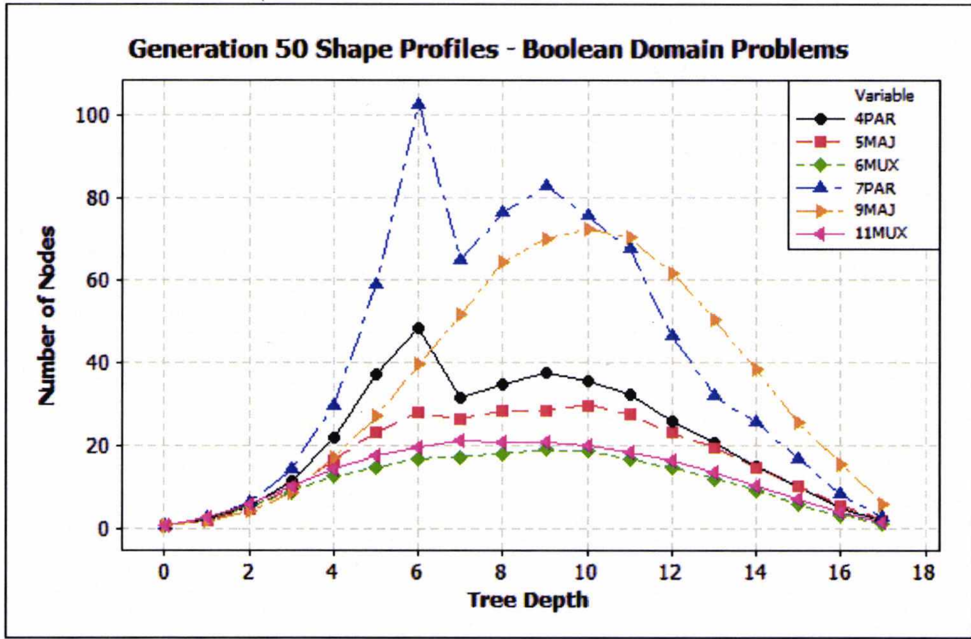
There are practical limits to the shapes evolved depending on the arity of the function sets. These limits determine an absolute minimum and maximum of the number of nodes at each depth in the tree. More specifically, at the early generations, the size and shape of trees is substantially limited by the arities of the function set. As depth increased the possible maximum limit at each depth increases exponentially.

Figure 7.10 shows the mean shape at generation 50 for each of the problems presented. The noticeable contrast is that 5 problems retain the spike at depth six where as 4 models present a smoother hill shape. Out of the five problems with spikes at depth 6, the 7PAR, AASF, CUBIC and QUART problems have resulted in relatively low success rates in the experimental results presented. The second point of note is that the experiments with the two lowest success rates during experimentation (7PAR and 9MAJ) have many more nodes at the majority of depths than the average shapes for other experiments.

This raises an interesting question in that, if GP is initialised using a fixed depth algorithm (such as Ramped Half and Half), how much more structural alteration work is having to take place to alter the structure of the program in order to make an optimal solution attainable with the collection of nodes present in the tree. In order to assess this, the following change metric has been developed:

$$\triangle S_t = \sum_{d=0}^{D} \mid n_{d,t} - n_{d,t-1} \mid \tag{7.1}$$

In equation 7.1, the structural change $\triangle S_t$ at time t is defined as the sum of the number of nodes difference at each depth between depth $d = 0$ and maximum depth D which in



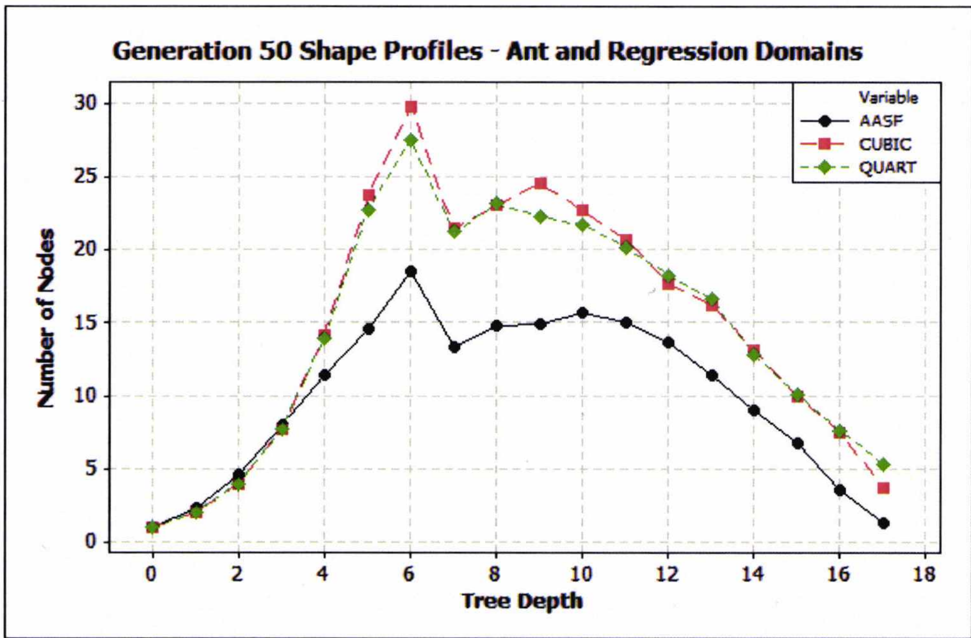


Figure 7.10: The figures show the shapes of the mean program tree at generation 50 for the different problems. These results are averaged over 100 runs.

Problem	G10	G10%	G11-50	G11-50%	Total
4PAR	129.18	37.54%	214.95	62.46%	344.13
5MAJ	76.52	26.56%	211.60	73.44%	288.12
6MUX	38.90	20.97%	146.56	79.03%	185.46
7PAR	255.66	30.87%	572.63	69.13%	828.29
9MAJ	110.31	15.92%	582.50	84.08%	692.81
11MUX	28.70	13.22%	188.34	86.78%	217.04
AASF	146.50	57.44%	108.56	42.56%	255.05
CUBIC	89.48	36.05%	158.77	63.95%	248.26
QUART	85.66	35.02%	158.91	64.98%	244.57

Table 7.1: The table shows the level of structural change taking place for each of the problem domains. Problem indicates the problem being evaluated. G10 indicates the total amount of structural change that has occurred up to and including G10. G10% indicates the total amount of structural change up to and including G10 as a percentage of total structural change. G11-50 indicates the total amount of structural change that occurs in generations 11 to 50. G11-G50% indicates the total amount of structural change that occurs in generations 11 to 50 as a percentage of total structural change. Total indicates the total level of structural change.

this case is 17. $n_{d,t}$ is the number of nodes at depth d in time period (or generation) t. The calculation is made using the average of the 100 runs produced during experimentation.

Table 7.1 shows the structural changes observed for the different test problems using the metric presented in equation 7.1. A statistical comparison of structural change over generations including all of the test problems using a one way ANOVA test revealed that the level of structural change in some experiments were statistically different (P-Value of 0.000). A further Tukey test revealed that the level of structural change for the 7PAR and 9MAJ were statistically different from the other experiments. This is unsurprising as 7PAR and 9MAJ typically have the most bloat in their runs. Two groups were formed revealing that the 7PAR and 9MAJ were statistically similar to each other and the other experiments were statistically similar to each other. Table 7.1 shows that 7PAR and 9MAJ feature larger levels of structural change compared to the other experiments.

The 4PAR, 7PAR, AASF, CUBIC and QUART experiments were grouped due to the fact that they feature a spike at depth 6 at generation 50. Table 7.1 reveals that a disproportionate amount of structural change occurred in the first 10 generations of evolution and figures 7.1, 7.4, 7.7, 7.8 and 7.9 show that this spike was formed within the first 5 generations of a run.

This early structural change and depth 6 spike effect is a result of crossover bias. In the early generations, smaller programs, being typically less fit than the larger programs (depths 5 and 6) are less likely to be selected resulting in more programs in the population being depth 6 in early generations (up to the 10th generation).

A final point of note is the high correlation between the height of the shape profiles in figure 7.10 and the mean length of programs in the population. Linear regression analysis indicates that with a formula of $Height = -5.99 + 0.208Length$, an R^2 of 99.5% is achieved, demonstrating that the largest number of nodes at any depth of a program tree can be explained with a high degree of accuracy using the program length as a predictor.

7.3 Program Structure and Problem Fitness

7.3.1 Structural Locality

Following on from the structural analysis presented in section 7.2, the structural change metric presented in equation 7.1 is compared with changes in mean fitness during the course of evolution. This study is performed on the averaged population structure in an average run, which is a run developed by averaging together all 100 runs by generation. In this comparison, the change in fitness is treated as a response variable in a regression equation and the structural change metric is applied as a predictor variable. The coefficient of determination (R^2) is used to show how much of the data is explained by the regression formula created. The adjusted (R^2adj) is also used to show how much of the data is explained by the regression model created. The R^2adj differs from the R^2 in that it only increases if the explanatory terms improve the model more than by chance.

Table 7.2 shows the level of correlation between changes in program structure and program fitness. In section 2.3, the concept of locality was discussed and it was predicted that in GP, there would be a low level of locality due to the fact it is relatively easy to demonstrate individual examples of low locality between genotypes and phenotypes. As a result of expected low levels of locality, the explanatory power of models predicting fitness changes using information derived from structural changes should be relatively low.

Table 7.2 shows that in all but one of the models, the R^2 and R^2adj are below 50% which indicates that the link between changes in structure and changes in fitness is relatively poorly

Problem	R^2	R^2adj
4PAR	43.4%	42.2%
5MAJ	11.9%	10.1%
6MUX	12.0%	10.2%
7PAR	30.3%	28.8%
9MAJ	3.9%	1.9%
11MUX	12.5%	10.7%
AASF	79.6%	79.2%
CUBIC	3.6%	1.6%
QUART	3.13%	1.3%

Table 7.2: The table shows the explanatory power of the regression models when trying to assess the correlation between structural changes and changes in the mean fitness of a run. R^2 indicates the coefficient of determination with a higher number indicating that the regression model has a greater ability to explain the data points present. R^2adj indicates the adjusted coefficient of determination. A higher number indicates that the regression model has a greater ability to explain the data points present.

explained by the model. This is in alignment with the theory that GP has the characteristic of low levels of locality. The surprising result is the R^2 and R^2adj values for the AASF problem which are nearly 80%. This implies that 80% of the data is explained by the model which indicates that there are much higher levels of structural locality in the AASF problem in comparison to the other problems.

This result enables explanation of some outlying results produced by the artificial ant experiments. Firstly, the AASF problem was the only problem to produce a statistically similar result when applying semantically driven crossover. Semantically driven crossover is specifically designed to bridge the gap between genotype and phenotype as part of the search operation. With high levels of structural locality present, a lot of the leverage of this search operator is lost because as long as the crossover alters the structure of the program slightly, there is a high probability it will alter the fitness as well. This is also supported by the relatively low level of crossover rejections for the AASF problem (figure 6.4).

When considering the pruning experiments, the AASF problem was the only experiment to produce no statistical change in fitness or program size. A possible reason for this is that as selection takes place, the fitter program are selected and then attempted to be reduced. If any different shaped programs are produced, the probability is that the fitness will change

and may be lower. This will result in these programs being filtered out at selection and the average population size being statistically similar whether pruning is applied or not.

When considering problems other than the AASF, one obvious question is, if changes in structure and changes in fitness are not linked, why is structure important to GP runs? The answer to this lies in behavioural bias. This bias is of the kind presented in section 5.2.3 where programs produced with specific size and shape limits produced repeated behaviours despite the fact they may be subject to low levels of locality. Furthermore, theories such as fitness causes bloat where programs group in areas of the search space in order to survive selection also support the idea of bias, due to the fact that they require bias and redundancy in representation to make fitness causes bloat applicable as a theory. As a result of these points and the experiments presented in sections 5.2.6 and 6.3, and the work of other authors (for example, Daida [2004]), it is clear that the structure of programs does influence evolution.

7.3.2 Changing Program Shape

Table 7.2 demonstrated that different problems presented different levels of structural locality. Following on from this fact, this section examines the effects of applying different operators presented in this thesis to assess whether there are changes in the levels of structural locality for the different problems.

Seven experiments are run on each of the nine problems (presented in section 4.1). The experiments are standard GP runs (as set out in section 4.2) with the following modifications. The KOZAXO experiment represents a run using the standard crossover with the 90% bias on functions and 10% bias on terminals at crossover swap points. KOZAXOSDC is an experiment using the Koza style standard crossover in conjunction with semantically driven crossover. PRUNED is a standard run with the semantic pruning algorithm switched on. SDI is a otherwise standard run were semantically driven initialisation has been applied to generate the starting population of programs. SDM is a run using semantically driven mutation with a probability of 0.9 and no crossover. MUT is a run using sub tree mutation (GROW depth 4 generation of sub trees) with a probability of 0.9 and no crossover. UNIXO is a standard run using crossover with uniform choice of swap points.

The results of these experiments are compared in two ways. Firstly, the shape profiles are compared using a one way ANOVA with a Tukey test to separate which shapes profile are

Shape Comparison - 4PAR (ANOVA) P = 0.009
PRUNED
MUT, KOZAXOSDC, SDI, SDM
KOZAXO, UNIXO

4PAR	R^2	R^2adj
KOZAXO	43.4%	42.2%
KOZAXOSDC	68.1%	67.4%
PRUNED	77.8%	77.4%
SDI	31.4%	30.0%
SDM	54.6%	53.6%
MUT	47.3%	46.2%
UNIXO	42.2%	41.0%

Table 7.3: 4PAR shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

different to each other. Secondly, a regression model is constructed to try to model changes in mean fitness based on the predictor of structural change as outlined in equation 7.1.

Tables 7.3 — 7.11 show several noteworthy features. The first areas to discuss are points that are general to all of the experiments presented.

Firstly, standard sub tree mutation presents a slightly higher level of structural locality than standard crossover (with biased swap points) (MUT Vs KOZAXO) in every experiment. This effect could be a result of sub tree mutation changing fewer nodes in comparison to crossover. Secondly, standard sub tree mutation presents slightly higher structural locality in all but one experiment compared to standard crossover with uniform swap points. As a result, in the majority of experiments presented mutation is more likely to result in fitness change when the structure of a tree is modified.

When evaluating tree shape, PRUNED, MUT and SDM consistently produce different and smaller tree shapes compared to the crossover mechanism; indicating, in conjunction with the previous paragraph, that there may be some fundamental difference between the way crossover and mutation function during evolution.

In all but the 7PAR and CUBIC problems, using the KOZAXOSDC in comparison to the KOZAXO increases structural locality in relative terms. This may be one of the reasons semantically driven crossover appears to be successful on a range of problems. When considering the mechanism behind KOZAXOSDC, the phenotypic link of semantically dri-

Shape Comparison - 5MAJ (ANOVA) P = 0.000
PRUNED
SDM, MUT,
KOZAXOSDC, SDI, UNIXO
KOZAXO

5MAJ	R^2	R^2adj
KOZAXO	11.9%	10.1%
KOZAXOSDC	20.8%	19.2%
PRUNED	66.0%	65.3%
SDI	0.8%	0.0%
SDM	41.1%	39.9%
MUT	47.6%	46.5%
UNIXO	40.8%	39.6%

Table 7.4: 5MAJ shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - 6MUX (ANOVA) P = 0.000
PRUNED
SDM, MUT,
KOZAXOSDC, SDI, UNIXO
KOZAXO

6MUX	R^2	R^2adj
KOZAXO	12.0%	10.2%
KOZAXOSDC	34.3%	32.9%
PRUNED	93.7%	93.6%
SDI	40.0%	38.8%
SDM	81.2%	80.8%
MUT	82.8%	82.5%
UNIXO	27.5%	26.0%

Table 7.5: 6MUX shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - 7PAR (ANOVA) P = 0.015
PRUNED
SDM, MUT, SDI, UNIXO, KOZAXOSDC
KOZAXO

7PAR	R^2	R^2adj
KOZAXO	30.3%	28.8%
KOZAXOSDC	30.1%	28.7%
PRUNED	3.4%	1.4%
SDI	59.4%	58.6%
SDM	32.8%	31.4%
MUT	32.6%	31.2%
UNIXO	28.6%	27.1%

Table 7.6: 7PAR shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - 9MAJ (ANOVA) P = 0.000
SDM, MUT
PRUNED
UNIXO
SDI
KOZAXO, KOZAXOSDC

9MAJ	R^2	R^2adj
KOZAXO	3.9%	1.9%
KOZAXOSDC	11.0%	9.2%
PRUNED	30.5%	29.1%
SDI	74.4%	73.9%
SDM	38.8%	37.5%
MUT	28.0%	26.5%
UNIXO	0.1%	0.0%

Table 7.7: 9MAJ shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - 11MUX (ANOVA) P = 0.000
MUT, SDM, PRUNED
UNIXO, KOZAXOSDC
KOZAXO
SDI

11MUX	R^2	R^2adj
KOZAXO	12.5%	10.7%
KOZAXOSDC	44.4%	43.3%
PRUNED	60.7%	59.9%
SDI	19.5%	17.8%
SDM	64.5%	63.8%
MUT	46.3%	45.2%
UNIXO	19.4%	17.7%

Table 7.8: 11MUX shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - AASF (ANOVA) P = 0.617
No difference between samples

AASF	R^2	R^2adj
KOZAXO	79.6	79.2
KOZAXOSDC	80.4	80.0
PRUNED	51.4	50.4
SDI	38.5	37.2
SDM	87.2	86.9
MUT	82.5	82.2
UNIXO	87.3	87.1

Table 7.9: AASF shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - CUBIC (ANOVA) P = 0.000
PRUNED
MUT, SDM
SDI
UNIXO, KOZAXO
KOZAXOSDC

CUBIC	R^2	R^2adj
KOZAXO	3.6%	1.6%
KOZAXOSDC	0.0%	0.0%
PRUNED	0.1%	0.0%
SDI	1.1%	0.0%
SDM	2.2%	0.2%
MUT	6.2%	4.3%
UNIXO	1.2%	0.0%

Table 7.10: CUBIC shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

Shape Comparison - QUART (ANOVA) P = 0.000
PRUNED
SDM, MUT, KOZAXOSDC
SDI, UNIXO, KOZAXO

QUART	R^2	R^2adj
KOZAXO	3.3%	1.3%
KOZAXOSDC	6.5%	4.6%
PRUNED	0.2%	0.0%
SDI	73.4%	72.9%
SDM	2.8%	0.8%
MUT	15.3%	13.6%
UNIXO	1.5%	0.0%

Table 7.11: QUART shape and structural locality comparison. The table on the left indicates the differences in shape analysed using a one way ANOVA and post hoc Tukey test to compare individual mean program shapes. Experiments are ordered with smallest at the top. The vertical lines indicate which shape profiles are statistically similar by group. The table on the right indicates a comparison of the explanatory power of the regression model predicting changes in mean fitness using structural changes.

ven crossover would speculatively result in more change in structure compared to standard crossover. The result of this appears to be increased changes in fitness and more movement around the search space would probabilistically result in a better chance of finding a global optimum.

When considering results that are specific to domains, the PRUNING experiments increase structural locality greatly for all of the Boolean problems (except 7PAR). Increased structural locality should make problems easier to solve, however, varied results using semantic pruning disagree with the idea that pruning alone makes all problems less complex and thus easier to solve. One of the arguments presented at the end of section 6.4 was that introns are required to traverse fitness steps. In this case local genotypic representations may be redundant and as such pruning reduces them back to the original behaviour and back translates them to a particular syntax tree. This would result in introns being required to allow an otherwise improbable modification to the genotype and as a result make a new step in the phenotypic space.

A second problem specific factor is the contrasting results in the symbolic regression domain. The results presented in tables 7.10 and 7.11 present substantial differences both in terms of shape groupings and structural locality results. It may be the case that the symbolic regression domain presents different representational properties for different target solutions.

The artificial ant domain, in comparison to the other problems, presents high structural locality which in theory should reduce the complexity of the problem at the representational level. A second aspect which causes the artificial ant domain to stand out is that out of the seven techniques presented in table 7.9, none managed to change the mean shape profile (at a statistically significant level) of evolved trees in the ant domain. It seems that in the ant domain, changes in tree structure are correlated to changes in fitness.

A final consideration is the role of semantically driven initialisation. Semantically driven initialisation appears to randomly reduce and increase structural locality from problem to problem. Results presented in table 5.7 showed the variable performance of semantically driven initialisation. It seems that not only is it difficult to construct a consistently good initialisation algorithm in terms of performance, but also in terms of producing a GP run with high locality in order to reduce the complexity of a problem.

7.4 Discussion

Structural Profiling

The structural profiling data presented in figures 7.1 — 7.9 showed one noticeable feature. This was the spike of nodes generated at depth 6 during the first few evolutions. This feature appears as a type of initialisation bias, based on crossover bias (Dignum and Poli [2007]). As crossover bias makes it less likely that shorter less fit programs will be selected, the average depths of programs in the population in early runs will increase to the predefined maximum depth limit of the initialisation algorithm, which in the case of RHH is six.

A question that arises from this research is, can GP practitioners use structural profiling in order to aid pruning or even bloat control as a whole? When considering pruning, shape profiling knowledge could be used to know at which depths to apply pruning. For example, in the 4PAR experiment (figure 7.1), there was still a spike at depth 6 at the end of the run. It may be possible to apply pruning at depths 7 and below in order to remove the worst of the bloat. In terms of bloat control in total, a system that uses a shape distribution to control preferable program structures (with some similarity to the system presented in Silva and Dignum [2009] controlling program lengths) may help to reduce bloat. It is clear from the evolved shape profiles and the further results presented in section 7.3.2 that both problem and operators used during evolution result in different shape profiles being produced. This information could be leveraged in order to control bloat with a self adaptive shape distribution.

Structure and Fitness

When comparing structural locality against changes in mean fitness, there are two observations to be drawn from the results in table 7.2. It is clear from table 7.2 that different problems present different levels of locality and this affects the complexity of problems (Rothlauf [2006]). Problems with high locality have been demonstrated to be less complex to solve than problems with low locality. One could theorise that the semantic operators presented in chapter 6 take steps to combat the issue of low locality. If one considers, that the semantic operators force change in the phenotypic space, then it is reasonable to suggest that the semantically driven operators help to combat complexity in situations where low levels of locality are present.

Considering the crossover operation (studying tables 7.3 — 7.11), the level of structural locality (or explanatory power when assessing the correlation between fitness change and structural change) is increased in all but two problems when using semantically driven crossover. A second argument to support this effect using crossover is that the ant domain appears to present relatively high structural locality in comparison to the other problems (table 7.2) presented. The results presented in section 6.1, table 6.2 indicate that it was the artificial ant experiment that was the only experiment to produce an equivalent fitness result overall. This is further corroborated in section 6.3, table 6.7 when studying the results obtained using semantic pruning, where again the artificial produces a unique result in that it is statistically similar to the control experiment both in terms of performance and program length.

A second point of note shown in the results presented in tables 7.3 — 7.11 is that standard sub tree mutation appears to encourage a higher structural locality when compared to standard crossover with biases swap points (KOZAXO). In addition to this, in several of the experiments the mean shape profile produced by sub tree mutation (MUT) is statistically different to the mean shape profile produced by standard crossover (KOZAXO) with biased swap points. This would indicate that on a structural level, standard crossover with biased swap points and standard sub tree mutation appears to function differently during evolution.

When comparing standard crossover with biased swap points (KOZAXO) to crossover with uniform choice of swap points (UNIXO), the results are less clear. In some cases KOZAXO presents higher structural locality and in others UNIXO presented higher structural locality. The important point to draw from this is that the choice of operators may increase or decrease problem complexity.

Further examination of the effect of semantic pruning indicates that in most cases (with exception to the symbolic regression problems) pruning increased structural locality substantially which in theory should make problems less complex. This is in direct contrast to the results presented in table 6.7 where the majority of experiments reported a drop in performance using semantic pruning. Semantic pruning has a second effect in that it also limits the level of redundancy because of the mechanism behind back translation. This could result in evolution being limited as the search operators would have to make improbable changes to the syntax in order to change the phenotype. Whilst this increases structural locality, it re-

moves redundancy which may enable higher probabilities of genotypic change between two phenotypic states. A suggested improvement to the pruning algorithm would be to prune, yet retain the context of the original program where possible. This may make it easier for the search operators to effect genotypic changes which will allow different phenotypic changes, thus improving the search ability of GP.

A further point highlighted by the results presented in tables 7.3 — 7.11 is that not only do the problems feature different levels of locality, the operators chosen to solve the problems can increase and decrease the representational complexities of a particular problem. A recommendation to be taken from this is that when practitioners want to test theories, they should not only consider problems from different domains, but also, where possible, different combinations of search operators in order to validate their experiments as widely as possible.

7.5 Conclusions

In conclusion, it appears that tree structure has at least some role to play in the evolution of programs. Results presented in this chapter have shown graphically and statistically that structure can vary by problem and choice of operators. Furthermore, the link between the change in structure and change in mean fitness of a run can give an indication of the level of structural locality present in a problem. The level of structural locality present can give an indication of the relative complexities of each problem from a representational point of view.

The results presented in this chapter at least start to explain one of the possible reasons for the success or failure of semantically driven operators. These results demonstrate that, whilst semantically driven operators were designed to improve diversity, when considering representational theory, semantically driven operators can help to reduce problem complexity by directly linking the genotypic space with the phenotypic space during a GP run.

Experimentation has shown that the effects of different search operators can also impact structural locality suggesting that not only the choice of problem will influence problem complexity but also the choice of operator may influence complexity during a GP run. This indicates that GP practitioners attempting to validate theories need to not only use problems from different domains, but also, where possible, apply different search operators to problems in order to study theoretical effects.

7.6 Future Work

Future ideas for research derived from the results presented in this chapter can be divided into three areas.

Firstly, the issue of initialisation has been evaluated in some detail in this thesis and the conclusion is that it is very difficult to construct an initialisation algorithm which consistently provides strong performance. One factor identified by the shape analyses in this chapter was the early depth 6 spike in nodes present to a degree in every problem presented. One possible area of research is to see how initialising programs in a band of depths that are less susceptible to crossover bias will affect a GP run. A simple example could be to use RHH with a depth range of 5 to 9 or to use a narrower range of depths in order to prevent crossover bias having such a dominant effect on the distribution of program sizes.

A second area of research would be to help augment an existing method of bloat control. Silva and Dignum [2009] presented an process known as a *fitness based self adaptive length distribution*. Having demonstrated that different shape profiles are present, it would be interesting to examine whether a *fitness based self adaptive shape distribution* can be developed in order to control bloat and possibly to increase performance.

Finally, a theoretical analysis of the representation concepts presented by Rothlauf [2006] (building block scaling, redundancy and locality) specifically for GP may present a greater understanding of representational issues which could be leveraged by the GP community.

Chapter 8

Conclusions

The methods, algorithms, experiments and discussions presented in this thesis have been used to both evaluate existing techniques used by GP practitioners and attempt to improve upon them. In some cases, experimentation has demonstrated that increasing the link between the genotypic and phenotypic search spaces by using behavioural representations has resulted in significant benefits to the GP. In other cases, the results are varied and lead to further discussions regarding theory and ways to improve the techniques presented in this thesis.

8.1 Contributions

There are three general contributions to be drawn from the investigations presented in this thesis.

Firstly, after an extensive empirical evaluation of diversity, bias and program structure during initialisation, it is clear that two different factors affect the performance of GP runs using the different initialisation algorithms presented. The factors are diversity in the starting population and the structure of the trees generated at initialisation. Empirical analysis reveals that both of these factors can influence the performance of GP run at statistically significant levels. A secondary issue is that of the initialisation of programs with no introns, where again empirical analysis reveals that performance results are varied. Having identified that shape and increasing phenotypic diversity can both influence initialisation, one hopes that practitioners developing new initialisation algorithms will take these factors into account.

The second contribution made by this thesis is that of the semantically driven operators; including semantically driven crossover, semantically driven mutation and semantic pruning.

In the experiments presented, semantically driven crossover consistently and significantly outperformed both variants of standard crossover (with uniform and biased choice of swap points). Whilst this algorithm was designed purely to increase diversity, in several cases it appeared to more closely link changes in the structure of programs to changes in the fitness of programs, which may indicate that it actually reduces the representational complexity of a GP problem by increasing structural locality. This algorithm has been one of the big successes of this thesis, and given that further research in the area is starting to appear (Nguyen et al. [2009]), the author hopes that this technique may be further adopted by the community.

Based on the same principles as semantically driven crossover, it is clear from experimentation that semantically driven mutation can significantly increase the performance of GP runs in the majority of the experiments presented. Further experimentation in chapter 7 indicates that mutation generally causes a stronger link between changes in structure and changes in fitness when compared to crossover, and results in significantly different and smaller program structures that crossover. Mutation in GP has been demonstrated to be as effective as crossover by Luke and Spector [1997, 1998]; however, it was not initially favoured in GP as a search operator by Koza [1992]. Preliminary evidence proposed in chapter 7 indicates that mutation might present different search opportunities and this combined with semantically driven mutation may enhance the search power of mutation in GP.

Whilst the use of semantic pruning (code reconstruction) in itself in the majority of cases was a negative result in terms of performance, semantic pruning is dramatically effective at reducing the size of programs. There are two points to be drawn from the lack of performance. Firstly, semantic pruning does not preserve the context of the original program due to the back translation mechanism changing all syntax to a common form. Because of this factor, semantic pruning will change the functionality of the genotypic search space because behaviourally equivalent, but different context programs are removed effectively causing cracks in the genotypic search space. As a result, whilst locality increases, the gaps between genotypes increase. This concept is validated further by the GP runs using the semantically driven operators together. Semantically driven crossover and mutation cause a stronger link between the genotypic and phenotypic search spaces, possibly increasing lo-

cality, and as such reducing representational complexity. The result in better performance in the majority of GP runs with human readable candidate programs. This result in itself is important because it is a large step towards increasing the efficiency of GP through controlling bloat and increasing diversity using the search operators. This result also suggests that a possible reason for the existence of introns is to pave over the cracks in the genotypic space, allowing access to new areas in the phenotypic space due to allowing a larger number of different combinations of syntax to be produced through search operations.

Finally, a structural analysis revealed previously unseen traits and links between structural changes and fitness changes. These links have added some explanatory power as to the level at which semantically driven operators may be effective. Two interesting points to note from the analysis of structure is that structure is affected not only by problem domain (which is unsurprising), but structure can also be affected by operator choice; notably in some cases a difference in structure using either crossover or mutation. Further investigation of this feature might provide more clues as to which operators to use to different probabilistic levels.

8.2 Future Directions of Research

Building upon this research, there are numerous avenues for future work. Several of these avenues are presented here.

The overwhelming point to note from the analysis of initialisation is that it is very difficult to design an initialisation algorithm which will consistently provide strong performance. Further understanding of initialisation is key because the results presented in chapter 5 indicate that changes in the choice of methods of initialisation can cause dramatic changes in the performance of GP runs. It would be desirable that further theoretical and practical work took place to understand the sensitive balance between diversity and shape and whether introns are required during evolution and produce initialisation algorithms that can control these factors. Once further initialisation algorithms have been developed to take account of these factors, it would be important to the community to conduct an up to date empirical comparison of initialisation techniques such as the comparison presented by Luke and Panait [2001].

Rothlauf [2006] examines in detail how different representations can change the com-

plexity of a problem. Semantically driven operators not only work to increase diversity but preliminary results indicate they may help to change the level of structural locality in a problem which would influence representational complexity. A detailed analysis of the representational issues presented by Rothlauf [2006] specifically for GP could result in more search operators which are theoretically motivated to reduce representational complexity, and as such should result in the increased performance of GP runs.

Results obtained using semantic pruning indicate that code reconstruction as a process has a less than ideal effect on evolution (of performance) although, in this case, it does help to provide an explanation for the presence of introns. This algorithm could be improved to retain the context of the original program, which should increase the probability of genotypic changes. This may lead to a situation where the intron free GP results would improve as there is more opportunity for taking new steps in the genotypic search space.

A second experiment using semantic pruning could be conducted to assess the validity of the hitchhiking bloat theory. The sub trees that are swapped during crossover could be reduced using semantic pruning to not include introns, and the resulting performance and program sizes compared and evaluated.

A possible application of the research in chapter 7 concerning shape profiling could be to build a method similar to that of Dignum and Poli [2008a], Silva and Dignum [2009], but rather than use length distributions, consider shape distributions. This may result in additional power to reduce bloat in specific areas of a program whilst retaining high performance.

One final thought is for the day that GP is routinely capable of evolving complete programs coded in main stream programming languages such as Java or C++. The concepts and theories presented in this thesis will still be useful as there are many ways to abstract and analyse the behaviour of programs (Nielson et al. [1999]). In addition, execution of many programs to assess fitness will be slow, and rather than increasing population sizes or the number of generations, it will be preferable to GP practitioners to make evolution more efficient. The concepts presented in this thesis are motivated specifically to perform GP more efficiently rather than increasing the amount of evolution.

Bibliography

Lee Altenberg. Emergent phenomena in genetic programming. In Anthony V. Sebald and Lawrence J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241, San Diego, CA, USA, 24-26 February 1994a. World Scientific Publishing. ISBN 981-02-1810-9. URL http://dynamics.org/~altenber/PAPERS/EPIGP/.

Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994b. URL http://dynamics.org/~altenber/PAPERS/EEGP/.

Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press. URL http://www.natural-selection.com/Library/1996/gp96.zip.

Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann. URL http://www.natural-selection.com/Library/1997/gp97a.zipbroken.

Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994. URL http://citeseer.ist.psu.edu/187189.html.

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Pro-*

gramming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco, CA, USA, January 1998. ISBN 1-55860-510-X. URL `http://www.elsevier.com/wps/find/bookdescription.cws_home/677869/description#description`.

Lawrence Beadle and Tom Castle. Epoch X - Genetic Programming Analysis Software. http://www.epochx.org, 2007.

Lawrence Beadle and Colin Johnson. Semantically driven crossover in genetic programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 111–116, Hong Kong, 1-6 June 2008. IEEE.

Lawrence Beadle and Colin G Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, September 2009a. ISSN 1389-2576. doi: 10.1007/s10710-009-9082-5. URL `http://www.cs.kent.ac.uk/pubs/2009/2947`.

Lawrence Beadle and Colin G Johnson. Semantically driven mutation in genetic programming. In *IEEE Proceedings of the Congress on Evolutionary Computation*, pages 1336–1342. IEEE, IEEE, 2009b.

Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241). URL `http://www.tik.ee.ethz.ch/~tec/publications/bt94/GPandRedundancy.ps.gz`.

Walter Bohm and Andreas Geyer-Schulz. Exact uniform initialization for genetic programming. In Richard K. Belew and Michael Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, University of San Diego, CA, USA, 3–5 August 1996. Morgan Kaufmann. ISBN 1-55860-460-X.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. URL `citeseer.ist.psu.edu/bryant86graphbased.html`.

Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004. URL http://www.cs.nott.ac.uk/~smg/research/publications/gustafson-ieee2004-preprint.pdf.

Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997. ISSN 1089-778X. doi: doi:10.1109/4235.661552.

Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985. URL http://www.sover.net/~nichael/nlc-publications/icga85/index.html.

Sara Guilherme Oliveira da Silva. *Controlling Bloat: Individual and Population Based Approaches in Genetic Programming*. PhD thesis, Coimbra University, Portugal, April 2008. URL http://cisuc.dei.uc.pt/ecos/dlfile.php?fn=1749_pub_phdsara.pdf.

Jason Daida. Considering the roles of structure in problem solving by a computer. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 5, pages 67–86. Springer, Ann Arbor, 13-15 May 2004. ISBN 0-387-23253-2.

Jason M. Daida. What makes a problem GP-hard? A look at how structure affects content. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, chapter 7, pages 99–118. Kluwer, 2003.

Jason M. Daida and Adam M. Hilss. Identifying structural mechanisms in standard genetic programming. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1639–1651, Chicago, 12-16 July 2003. Springer-Verlag. ISBN 3-540-40603-4. URL http://sitemaker.umich.edu/daida/files/LNCS2724lattice.pdf.

Jason M. Daida, Robert R. Bertram, Stephen A. Stanhope, Jonathan C. Khoo, Shahbaz A. Chaudhary, Omer A. Chaudhri, and John A. Polito II. What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2(2):165–191, June 2001. ISSN 1389-2576. doi: doi:10.1023/A:1011504414730.

Jason M. Daida, Hsiaolei Li, Ricky Tang, and Adam M. Hilss. What makes a problem GP-hard? validating a hypothesis of structural causes. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1665–1677, Chicago, 12-16 July 2003. Springer-Verlag. ISBN 3-540-40603-4.

Kenneth A DeJong. *Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975. URL http://www.cs.gmu.edu/~eclab/kdj_thesis.html.

Stephen Dignum. An analysis of genetic programming operator bias regarding the sampling of program size with potential applications. In Jano van Hemert, Mario Giacobini, and Cecilia Di Chio, editors, *EvoPhD 2008*, Naples, March 2008.

Stephen Dignum and Riccardo Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1588.pdf.

Stephen Dignum and Riccardo Poli. Operator equalisation and bloat free GP. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe

De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 110–121, Naples, 26-28 March 2008a. Springer. doi: doi:10.1007/978-3-540-78671-9_10.

Stephen Dignum and Riccardo Poli. Crossover, sampling, bloat and the harmful effects of size limits. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 158–169, Naples, 26-28 March 2008b. Springer. doi: doi:10.1007/978-3-540-78671-9_14.

Richard M. Downing. Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 615–622, Vancouver, 6-21 July 2006a. IEEE Press. ISBN 0-7803-9487-9. URL http://www.cs.bham.ac.uk/~rmd/pubs/gradualism.pdf.

Richard M. Downing. Evolving binary decision diagrams with emergent variable orderings. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund Burke, Juan J. Merelo-Guervos, L. Darrell Whitley, and Xin Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 798–807, Reykjavik, Iceland, 9-13 September 2006b. Springer-Verlag. ISBN 3-540-38990-3. doi: doi:10.1007/11844297_81. URL http://www.cs.bham.ac.uk/~rmd/pubs/ppsn06.pdf.

Richard Mark Downing. Evolving binary decision diagrams using implicit neutrality. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2107–2113, Edinburgh, UK, 2-5 September 2005. IEEE Press. ISBN 0-7803-9363-5. URL http://www.cs.bham.ac.uk/~rmd/pubs/evolvingbddsCEC2005.pdf.

Jeroen Eggermont, Joost N. Kok, and Walter A. Kosters. Detecting and pruning introns for

faster decision tree evolution. In Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 1071–1080, Birmingham, UK, 18-22 September 2004. Springer-Verlag. ISBN 3-540-23092-0. doi: doi:10.1007/b100601. URL http://www.liacs.nl/~kosters/ppsn8/ppsn2004.pdf.

Aniko Ekart and S. Z. Nemeth. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, March 2001. ISSN 1389-2576. doi: doi:10.1023/A:1010070616149.

Edgar Galvan Lopez, Katya Rodriguez Vazquez, and Riccardo Poli. Beneficial aspects of neutrality in GP. In Franz Rothlauf, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2005)*, Washington, D.C., USA, 25-29 June 2005. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005lbp/papers/45-lopez.pdf.

Alma Lilia Garcia-Almanza and Edward P. K. Tsang. Simplifying decision trees learned by genetic programming. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 7906–7912, Vancouver, 6-21 July 2006. IEEE Press. ISBN 0-7803-9487-9. URL http://privatewww.essex.ac.uk/~algarc/Publications/WCCI2006.pdf.

Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press. URL http://citeseer.ist.psu.edu/153919.html.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989. ISBN 0201157675. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201157675.

Steven Gustafson. *An Analysis of Diversity in Genetic Programming.* PhD thesis, School of Computer Science and Information Technology, University of Nottingham, Nottingham, England, February 2004. URL http://www.cs.nott.ac.uk/~smg/research/publications/phdthesis-gustafson.pdf.

Steven Gustafson, Edmund K. Burke, and Graham Kendall. Sampling of unique structures and behaviours in genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 279–288, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag. ISBN 3-540-21346-5. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3003&spage=279.

Steven Gustafson, Edmund K. Burke, and Natalio Krasnogor. The tree-string problem: An artificial domain for structure and content search. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 215–226, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6. URL http://www.cs.nott.ac.uk/~smg/research/publications/eurogp2005-gustafson-etal.ps.

Thomas Haynes. Phenotypical building blocks for genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 26–33, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann. ISBN 1-55860-487-1. URL http://citeseer.ist.psu.edu/cache/papers/cs/2284/http:zSzzSzadept.cs.twsu.eduzSz~thomaszSzpbb_gp.pdf/haynes97phenotypical.pdf.

Thomas Haynes. Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338, Winter 1998. doi: doi:10.1162/evco.1998.6.4.311. URL http://www.mitpressjournals.org/doi/pdfplus/10.1162/evco.1998.6.4.311.

John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-58111-6.

Hitoshi Iba. Random tree generation for genetic programming. Technical Report ETL-TR-95-35, ElectroTechnical Laboratory (ETL), 1-1-4 Umezono, Tsukuba-city, Ibaraki, 305, Japan, 14 November 1995.

David Jackson. Behavioural diversity and filtering in GP navigation problems. In Leonardo

Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 256–267, Tuebingen, April 15-17 2009. Springer. doi: doi: 10.1007/978-3-642-01181-8_22.

Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, 28 March-1 April 1993. IEEE Press. ISBN 0-7803-0999-5. doi: doi: 10.1109/ICNN.1993.298674. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.icnn93.ps.Z.

Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press. ISBN 0-7803-1899-4. doi: doi:10.1109/ICEC.1994.350026. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.wcci.ps.Z.

Vecchi M. P. Kirkpatrick S., Gelatt Jr. C. D. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994. ISBN 0-262-11189-6.

John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003. ISBN 1-4020-7446-8. URL http://www.genetic-programming.org/gpbook4toc.html.

W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press. doi: doi:10.1109/ICEC.1998.700102. URL http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.wcci98_bloat.pdf.

W. B. Langdon. Size fair tree crossovers. In Eric Postma and Marc Gyssen, editors, *Proceedings of the Eleventh Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'99)*, pages 255–256, Kasteel Vaeshartelt, Maastricht, Holland, 3-4 November 1999. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/fairxo_bnaic99.ps.gz.

W. B. Langdon and W. Banzhaf. Genetic programming bloat without semantics. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, volume 1917 of *LNCS*, pages 201–210, Paris, France, 16-20 September 2000. Springer Verlag. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_ppsn2000.pdf.

W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997. ISBN 3-540-76214-0. URL http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.bloat_wsc2.ps.gz.

W. B. Langdon and R. Poli. Why ants are hard. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann. ISBN 1-55860-548-7. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.antspace_gp98.pdf.

W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. ISBN 3-540-42451-2. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/.

William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000. ISSN 1389-2576. doi: doi:10.1023/A:1010024515191. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL_fairxo.pdf.

William B. Langdon and Wolfgang Banzhaf. Repeated patterns in tree genetic programming. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco To-

massini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 190–202, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6. doi: doi:10.1007/b107383. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_egp2005.pdf.

William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999. ISBN 0-262-19423-6. URL http://www.cs.bham.ac.uk/~wbl/aigp3/ch08.pdf.

Xiang Li and Vic Ciesielski. An analysis of explicit loops in genetic programming. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2522–2529, Edinburgh, UK, 2-5 September 2005. IEEE Press. ISBN 0-7803-9363-5.

Moshe Looks. On the behavioral diversity of random programs. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1636–1642, London, 7-11 July 2007. ACM Press. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1636.pdf.

Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000a. URL http://ieeexplore.ieee.org/iel5/4235/18897/00873237.pdf.

Sean Luke. Code growth is not caused by introns. In Darrell Whitley, editor, *Late Breaking*

Papers at the 2000 Genetic and Evolutionary Computation Conference, pages 228–235, Las Vegas, Nevada, USA, 8 July 2000b. URL http://www.cs.gmu.edu/~sean/papers/intronpaper.pdf.

Sean Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.* PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA, 2000c. URL http://www.cs.gmu.edu/~sean/papers/thesis2p.pdf.

Sean Luke. Modification point depth and genome growth in genetic programming. *Evolutionary Computation*, 11(1):67–106, Spring 2003. doi: doi:10.1162/106365603321829014.

Sean Luke and Liviu Panait. A survey and comparison of tree generation algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. ISBN 1-55860-774-9. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2001/d01.pdf.

Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, Fall 2006. ISSN 1063-6560. doi: doi:10.1162/evco.2006.14.3.309.

Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann. URL http://www.cs.gmu.edu/~sean/papers/comparison/comparison.pdf.

Sean Luke and Lee Spector. A revised comparison of crossover and mutation in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*,

pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann. ISBN 1-55860-548-7. URL `http://www.cs.gmu.edu/~sean/papers/revisedgp98.pdf`.

Hammad Majeed and Conor Ryan. Context-aware mutation: a modular, context aware mutation operator for genetic programming. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1651–1658, London, 7-11 July 2007. ACM Press. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1651.pdf`.

S. R. Maxwell. Why might some problems be difficult for genetic programming to find solutions? In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 125–128, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore. ISBN 0-18-201031-7.

Ben McKay, Mark J. Willis, and Geoffrey W. Barton. Using a tree structured genetic algorithm to perform symbolic regression. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 487–492, Sheffield, UK, 12-14 September 1995. IEE. ISBN 0-85296-650-4. doi: doi:10.1049/cp:19951096. URL `http://scitation.aip.org/getpdf/servlet/GetPDFServlet?filetype=pdf&id=IEECPS0019950CP414000487000001&idtype=cvips&prog=normal`.

Nicholas Freitag McPhee and Nicholas J. Hopper. Analysis of genetic diversity through population history. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. ISBN 1-55860-611-4. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-421.pdf`.

Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 134–145, Naples, 26-28 March 2008. Springer. doi: doi:10.1007/978-3-540-78671-9_12.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996. ISBN 0-262-13316-4. URL http://www-mitpress.mit.edu/mitp/recent-books/cog/mitnh.html.

Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. Semantic aware crossover for genetic programming: the case for real-valued function regression. In Leonardo Vanneschi, Steven Gustafson, and Marc Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, Tuebingen, April 15-17 2009. Springer. Paper 37 Forthcoming.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995. URL http://citeseer.ist.psu.edu/nordin95explicitly.html.

Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem, 9-14 October 1994a. Springer-Verlag. ISBN 3-540-58484-6. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/ppsn-94.ps.gz.

Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose,

editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994b. Morgan Kaufmann. ISBN 1-55860-356-5. URL http://citeseer.ist.psu.edu/cache/papers/cs/163/http:zSzzSzwww.ai.mit.eduzSzpeoplezSzunamayzSzpaperszSzfoga.pdf/oreilly92troubling.pdf. Published 1995.

Una-May O'Reilly and Franz Oppacher. Hybridized crossover-based search techniques for program discovery. In *Proceedings of the 1995 World Conference on Evolutionary Computation*, volume 2, pages 573–578, Perth, Australia, 29 November - 1 December 1995. IEEE Press. ISBN 0-7803-2759-4. URL http://ieeexplore.ieee.org/iel2/3507/10438/00487447.pdf.

J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag. ISBN 3-540-65899-8. URL http://www.cs.essex.ac.uk/staff/poli/papers/Page-EUROGP1999.pdf.

Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2): 123–163, June 2001. ISSN 1389-2576. doi: doi:10.1023/A:1011552313821. URL http://ipsapp009.lwwonline.com/content/getfile/4723/5/4/fulltext.pdf.

Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 204–217, Essex, 14-16 April 2003. Springer-Verlag. ISBN 3-540-00971-X. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2610&spage=204.

Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming*

1997: Proceedings of the Second Annual Conference, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann. URL http://citeseer.ist.psu.edu/327495.html.

Riccardo Poli and William B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann. ISBN 1-55860-548-7. URL http://www.cs.essex.ac.uk/staff/poli/papers/Poli-GP1998.pdf.

Riccardo Poli and Nicholas McPhee. Parsimony pressure made easy. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA, 12-16 July 2008. ACM Press. forthcoming.

Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, March 2003a. doi: doi:10.1162/106365603321829005. URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partI.pdf.

Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003b. doi: doi:10.1162/106365603766646825. URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partII.pdf.

Riccardo Poli, William B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain, 11 - 13 April 2007. Springer. ISBN 3-540-71602-5. doi: doi:10.1007/978-3-540-71605-1_18.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming.* Published via http://lulu.com and freely available at

http://www.gp-field-guide.org.uk, 2008. URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza).

Riccardo Poli, Mario Graff, and Nicholas Freitag McPhee. Free lunches for function and program induction. In *FOGA '09: Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*, pages 183–194, Orlando, Florida, USA, 9-11 January 2009. ACM. ISBN 3-540-27237-2. doi: doi:10.1145/1527125.1527148.

William F. Punch, Douglas Zongker, and Erik D. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-01158-1.

Justinian P. Rosca and Dana H. Ballard. Rooted-tree schemata in genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 11, pages 243–271. MIT Press, Cambridge, MA, USA, June 1999. ISBN 0-262-19423-6. URL http://www.cs.bham.ac.uk/~wbl/aigp3/ch11.pdf.

Franz Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer-Verlag, pub-SV:adr, second edition, 2006. ISBN 3-540-25059-X. URL http://download-ebook.org/index.php?target=desc&ebookid=5771. First published 2002, 2nd edition available electronically.

Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14-15 April 1998. Springer-Verlag. ISBN 3-540-64360-5. URL http://www.lania.mx/~ccoello/eurogp98.ps.gz.

R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997. doi: doi:10.1162/evco.1997.5.2.123. URL ftp://ftp.idsia.ch/pub/rafal/PIPE.ps.gz.

Sara Silva and Jonas Almeida. Dynamic maximum tree depth. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wil-

son, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1776–1787, Chicago, 12-16 July 2003. Springer-Verlag. ISBN 3-540-40603-4. URL http://cisuc.dei.uc.pt/ecos/dlfile.php?fn=109_pub_27241776.pdf.

Sara Silva and Ernesto Costa. Comparing tree depth limits and resource-limited GP. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 920–927, Edinburgh, UK, 2-5 September 2005. IEEE Press. ISBN 0-7803-9363-5.

Sara Silva and Stephen Dignum. Extending operator equalisation: Fitness based self adaptive length distribution for bloat free GP. In Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 159–170, Tuebingen, April 15-17 2009. Springer. doi: doi:10.1007/978-3-642-01181-8_14.

Sara Silva, Pedro J. N. Silva, and Ernesto Costa. Resource-limited genetic programming: Replacing tree depth limits. In Bernardete Ribeiro, Rudof F. Albrecht, Andrej Dobnikar, David W. Pearson, and Nigel C. Steele, editors, *Adaptive and Natural Computing Algorithms*, Springer Computer Series, pages 243–246, Coimbra, Portugal, 21-23 March 2005. Springer. ISBN 3-211-24934-6. URL http://www.lri.fr/~sebag/Examens/sara.icannga05.pdf.

F. Somenzi. Cudd: CU Decision Diagram Package release. http://vlsi.colorado.edu/ fabio/CUDD/, 1998. URL citeseer.ist.psu.edu/somenzi98cudd.html.

Terence Soule and James A. Foster. Code size and depth flows in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA, 13-16 July 1997.

Morgan Kaufmann. URL http://citeseer.ist.psu.edu/cache/papers/cs/14038/http:zSzzSzwww.stcloudstate.eduzSz~tsoulezSzpaperszSzspace4.pdf/soule97code.pdf.

Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press. URL http://citeseer.ist.psu.edu/313655.html.

Chris Stephens and Henri Waelbroeck. Schemata evolution and building blocks. *Evolutionary Computation*, 7:109–124, 1998.

C. R. Stevens and H. Waelbroeck. Effective degrees of freedom in genetic algorithms and the block hypothesis. In Thomas Back, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 34–40, East Lansing, 1997. Morgan Kaufmann.

Matthew J. Streeter. The root causes of code growth in genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 443–454, Essex, 14-16 April 2003. Springer-Verlag. ISBN 3-540-00971-X. URL http://www.cs.cmu.edu/~matts/Research/mstreeter_eurogp_2003.pdf.

Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/watphd.tar.Z.

Astro Teller and Manuela Veloso. PADO: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996. URL http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/PADO.ps.

John Whaley. JavaBDD. http://javabdd.sourceforge.net/, 2007. URL http://javabdd.sourceforge.net/.

P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995a. URL http://divcom.otago.ac.nz/sirc/Peterw/Publications/ml95.zip.

P. A. Whigham. Inductive bias and genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, Sheffield, UK, 12-14 September 1995b. IEE. ISBN 0-85296-650-4. URL http://divcom.otago.ac.nz/sirc/Peterw/Publications/galesia.zip.

P. A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 29 November - 1 December 1995c. IEEE Press. URL http://divcom.otago.ac.nz/sirc/Peterw/Publications/schema.zip.

P. A. Whigham. Search bias, language bias, and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA, 28–31 July 1996. MIT Press. URL ftp://www.cs.adfa.edu.au/pub/xin/whigham_gp96.ps.gz.

John R. Woodward and James R. Neil. No free lunch, program induction and combinatorial problems. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 475–484, Essex, 14-16 April 2003. Springer-Verlag. ISBN 3-540-00971-X. URL http://www.cs.bham.ac.uk/~jrw/publications/2003/NoFreeLunchProgramInductionandCombinatorialProblems/nfl.ps.

Masayuki Yanagiya. Efficient genetic programming based on binary decision diagrams. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 234–239, Perth, Australia, 29 November - 1 December 1995. IEEE Press. doi: doi:10.1109/ICEC.1995.489151.

Mengjie Zhang, Xiaoying Gao, Weijun Lou, and Dongping Qian. Investigation of brood size

in GP with brood recombination crossover for object recognition. In Qiang Yang and Geoffrey I. Webb, editors, *PRICAI 2006: Trends in Artificial Intelligence, Proceedings 9th Pacific Rim International Conference on Artificial Intelligence*, volume 4099 of *Lecture Notes in Computer Science*, pages 923–928, Guilin, China, August 7-11 2006. Springer. ISBN 3-540-36667-9. doi: doi:10.1007/11801603_107.