TUNING GENETIC PROGRAMMING PERFORMANCE VIA BLOATING CONTROL AND A DYNAMIC FITNESS FUNCTION APPROACH

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2013

By
Geng Li
School of Computer Science

Contents

Abstract 9

Declaration 10

Copyright 11

Acknowledgements 12

1 Introduction 13

1.1 Genetic Programming and Bloating . . . 14
1.2 Objectives . . . 16
1.3 Contributions . . . 17
1.4 Thesis Overview . . . 18

2 Genetic Programming 20

2.1 Variable Length Representation . . . 20
2.2 Genetic Programming Inputs . . . 22
2.3 Genetic Programming Workflow . . . 24
2.3.1 Initial Population Generation . . . 25
2.3.2 Fitness Evaluation . . . 28
2.3.2.1 Fitness Value . . . 28
2.3.2.2 Evaluation Performance . . . 30
2.3.3 Parent Selection . . . 31
2.3.4 Breeding Operators . . . 35
2.4 Benchmark Problems . . . 38
2.4.1 Multiplexer . . . 39
2.4.2 Symbolic Regression . . . 39
2.4.3 Parity . . . 41

2.4.4 Artificial Ant 41

3 Bloating Theory and Control Techniques 43

3.1 Theoretical Code Growth Models 43

3.1.1 Intron Theory and Defense against Crossover 44

3.1.2 Defense against Crossover Variations 46

3.2 Anti-Defense against Crossover Evidences 47

3.3 Other Models 48

3.4 Bloating Control Techniques 49

3.4.1 Modifications of Crossover 50

3.4.2 Modifications of Fitness Function 53

3.4.3 Modifications of Selection 55

3.4.4 Modifications of GP Flow 55

3.5 Conclusion 57

4 Theoretical Analysis of Bloating Effects 58

4.1 Introduction 58

4.2 Bottom-up Tree Evaluation 59

4.2.1 Motivation 59

4.2.2 Theoretical Performance Analysis 63

4.2.3 Experiments 67

4.3 Estimating Activation Rate 70

4.4 Experimenting with Activation Rate Estimation 73

4.5 Activation Rate and Fitness 75

4.6 Activation Rate and Tree Size 77

4.7 Activation Rate and Crossover 81

4.7.1 Crossover Effects 81

4.7.2 Semi-Intron Crossover 85

4.8 Conclusion 89

5 Removal Bias & Depth-Constraint Crossover 90

5.1 Introduction 90

5.2 Background and Related Work 91

5.3 A New Quantitative Model 94

5.4 Depth Constraint Crossover 98

5.5 Depth Difference Hypothesis 109

5.6 Conclusion 113

6 Norm-Referenced Fitness Evaluation 115

6.1 Introduction 115

6.2 Motivation 116

6.3 Internal Fitness Measure 118

6.4 Norm-referenced Fitness 122

6.4.1 Initial Experiments 124

6.4.2 Analysis of Selection Intensity 126

6.5 Implicit Bias Theory 127

6.5.1 Further Experiment in Even Parity 5 Domain 132

6.5.2 Experiment in Other Problem Domains 133

6.6 Partial Norm-referenced Fitness 139

6.7 Conclusion 143

7 Conclusion and Further Work 146

7.1 Contributions 147

7.2 Further works 149

Bibliography 151

Word Count: 38,451

List of Tables

4.1 Top-down Full and Bottom-up Evaluation Performance (7000-trees) 68
4.2 Top-down and Bottom-up Evaluation Performance (7000-trees) . . 69
4.3 Theoretical Estimation of Activation Rate in Multiplexer11 74
4.4 Observed Activation Rates in 1500, 5000, 15000 Trees Experiments 74
4.5 Comparison between Observed and Estimated Activation Rate . . 75
4.6 Tree Depth and Tree Activation Rate 77
4.7 Crossover Effects Distribution 83
4.8 Distribution of Constructive and Destructive Crossover based on Definition 6 84
4.9 Distribution of Constructive and Destructive Crossover Grouped by Crossover Point's Activation Rate based on Definition 6 84

5.1 Removal Bias at Generation 49 using No Bloating Control Methods 95
5.2 Correlation between Amount of Removal Bias and Average Depth of the Generation 97
5.3 Removal Bias at Generation 49 with Depth-Limiting Method . . . 97
5.4 Correlation between Amount of Removal Bias and Average Depth of the Generation In Experiment 2 98
5.5 Summary of Experiment Results Comparing Koza-Style Depth Limiting Method and Depth Constraint Crossover 101
5.6 Removal Bias at Generation 49 with Depth Constraint Crossover . 104
5.7 Summary of Experiment Results Comparing Koza-Style Depth Limiting Method and Depth Constraint Crossover combined with Depth Limiting Method 106
5.8 Summary of Experiment Results of Depth Constraint Crossover with Real Number Threshold Parameter ϵ 108
5.9 Summary of Experiment Results Comparing Depth Constraint Crossover with Other Bloating Control Methods 112

6.1 Fitness Values of 4 Individuals in P 119
6.2 Best Fitness at Gen 50 in Even Parity 5 Problem 125
6.3 Number of GP Runs when Norm-referenced Fitness Function Performs Better, Equal, or Worse Compared to the According Original Fitness function with Same Initialization 125
6.4 Experiment Results for 10 Initializations Running 300 Generations 133
6.5 Best Fitness at Gen 50 in Artificial Ant Problem 135
6.6 Best Fitness at Gen 50 in Sextic Problem 137
6.7 OneMax Problem Performance for N in 20, 50, 100, 150 and 200 . 139
6.8 Best Raw Fitness at Generation 50, 100, 200, 300 using Partial Norm-referenced, Norm-referenced and Original Fitness Function . 142

List of Figures

2.1 An Example GP Tree and its Lisp S-expression 21

4.1 Viewing test case inputs as permutation in Multiplexer 11 problem 60
4.2 Size of Individual as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$ 79
4.3 Number of Nodes needs to be Evaluated as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$ 80
4.4 Number of Nodes needs to be Evaluated as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$ (Depth up to 50) . 80
4.5 Average Fitness Changes for ϵ from 0 to 2048 in Steps of 10 . . . 87
4.6 Average Tree Depth Changes for ϵ from 0 to 2048 in Steps of 10 . 87
4.7 Average Tree Size Changes for ϵ from 0 to 2048 in Steps of 10 . . 88

5.1 Example of a General Crossover 92
5.2 Amount of Removal Bias over Generations without Bloating Control 96
5.3 Amount of Removal Bias over Generations with Depth-Limiting Method 98
5.4 Size of Run Changes over 50 Generations for Depth Constraint Crossover and Koza-style Depth Limiting 102
5.5 Size of Run Changes over 50 Generations for Depth Constraint Crossover combined with Depth Limiting and Koza-style Depth Limiting 105

6.1 The Selection Intensity of Tournament Selection for the Original Fitness Function and Norm-referenced Fitness Functions with Different λ Settings 127
6.2 $\lambda_{original}$ for a GP Run in Even Parity 5 Problem using Original Fitness Function 129

6.3 Comparison of Best Fitness of Generation Changes over Generations Between Original Fitness Function and Norm-referenced Fitness Functions with Different λ Settings 132
6.4 Best Fitness by Generation for Norm-referenced and Original Fitness Function in Multiplexer 11 Problem 134
6.5 Average Population Fitness Change over Generations for Norm-referenced and Original Fitness Function in OneMax Problem . . 138

Abstract

Inspired by Darwin's natural selection, genetic programming is an evolutionary computation technique which searches for computer programs best solving an optimization problem. The ability of GP to perform structural optimization at the same time of parameter optimization makes it uniquely suitable to solve more complex optimization problems, in which the structure of the solution is not known a priori. But, as GP is applied to increasingly difficult problems, the efficiency of the algorithm has been severely limited by bloating. Previous studies of bloating suggest that bloating can be resolved either directly by delaying the growth in depth and size, or indirectly by making GP to find optimal solutions faster. This thesis explores both options in order to improve the scalability and the capacity of GP algorithm. It tackles the former by firstly systematically analyzing the effect of bloating using a mathematical tool developed called activation rate. It then proposes depth difference hypothesis as a new cause of bloating and investigates depth constraint crossover as a new bloating control method, which is able to give very competitive control over bloating without affecting the exploration of fitter individuals. This thesis explores the second option by developing norm-referenced fitness function, which dynamically determines the individual's fitness based on not only how well it performs, but also the population's average performance as well. It is shown both theoretically and empirically that, norm-referenced fitness is able to significantly improve GP performance over the standard GP setup.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

Acknowledgements

Writing this last part of my PhD thesis fills my mind with gratitude, and fond memories of the first night I arrived in Manchester back in 2004, when I started my undergraduate study. The past 8 years have scripted the most wonderful experience of my life, the finale being my PhD.

First I would like to express my deep sense of gratitude to Dr Xiao-Jun Zeng for his able supervision, strong encouragement, constant availability and effective guidance. The discussions have always been constructive, pleasant and fruitful, and I owe him a lot for giving me plenty of freedom throughout my PhD study. It was truly a pleasure and honour for me to be able to work with him for 5 years, and my best wishes are always with him and his family.

I am deeply indebted to my best friend Dongdong Li for his constant financial support throughout my PhD study. Without his support, this PhD work would not have been possible. My warmest wishes to him and his wife Youran Huang. I would also like to thank my friends Mingyang Zhao, Shihai Wang, Yun Yang, Ruofei He, Naikuo Yang and Xinkai Wang for all the valuable discussions we had, and more importantly, for making my life in Manchester more enjoyable.

Special thanks to my colleagues in UBS, especially to Joseph Rosario Bastine Joseph, Valvit Kurti, and Kieron Wall, for their understanding and unstinted support during my thesis-writing period. Also, special thanks to Soundharya Pradha, for her support during my thesis revision time.

Last, but definitely not the least. To my parents, my thankfulness cannot be expressed enough. For always being there, for their everlasting patience and encouragement, for inspiring me, for their thoughts of wisdom, and for pushing me farther than I thought I could go. *Xie Xie*. I only wish they could realize that I am now old enough to not worry about anymore.

Chapter 1

Introduction

Artificial Intelligence is a branch of computer science which aims to study and design "intelligent agents". Intelligent agent is defined as a system that perceives its environment and takes actions which maximize its chances of success [RN02]. Given this definition, a large share of Artificial Intelligence problems can be resolved in theory by intelligently searching through all possible solutions. In another word, many different problems in Artificial Intelligence can be converted into searching of optimal solution within problem space. For example, logical deduction can be viewed as searching for a path from premises to conclusions. Robotics controlling algorithms use search to find optimal strategies in configuration space.

The simplest searching algorithm is blind exhaustive search, in which every single solution within the searching space is checked for its quality. We get the optimal solution after the search space is exhausted. Blind exhaustive search is easy to understand and simple to perform. But it is only feasible to apply it to simple problem domains in which the searching space is very small, and it is rarely sufficient for real world problems in which the searching space is always very large.

An improvement over exhaustive search is hill climbing. Hill climbing is based on optimization in Mathematics. In hill climbing, search normally starts from a blind guess and then refines the guess incrementally until no more refinements can be made. The refinement is usually based on the first and second order derivative of the target function. One limitation of hill climbing search is that it is not stable. The choice of initial guess affects both the quality of the final solution and the speed the algorithm converges to the solution, especially for

problems with complex search landscape, in which hill climbing is very likely to converge at a local optimal rather than the global optimal.

Hill climbing is one example of a more general category of searching called heuristic search. In heuristic search, heuristic is used to suggest to the searching with "best estimates" of which part the optimal solution lies in the searching space. Generally, heuristic embeds the domain specific information tied to the problem. In the case of hill climbing, first order derivative is used as heuristic. With heuristic, the searching program can eliminate parts of search space that are unlikely to have optimal solution such that the efficiency of the search is improved.

This thesis concentrates on a very different kind of optimization algorithm, *Evolutionary Computation*. Evolutionary computation is a population based searching algorithm inspired by Darwin's natural selection and Genetics in Biology. Evolutionary computation searches for "optimal" solution by repeatedly evaluating large number of candidate solutions (i.e. called a population), selecting "fitter" ones, modifying them, and producing new candidate solutions until an optimal solution is found or resources are exhausted. Common evolution computation techniques include Genetic Algorithms, Evolutionary Programming, Evolution Strategies and more recently Genetic Programming. This thesis's focus is primarily Genetic Programming, although a number of findings in this thesis have a wider applicability, and can be applied to other methods as well.

1.1 Genetic Programming and Bloating

Genetic Programming (GP) is a dialect of Genetic Algorithm developed in 1990s. In genetic algorithm, a population of abstract representations of candidate solutions to an optimization problem is evolved. Candidate solutions are typically encoded in binary strings of fixed size. The evolution starts from a population of randomly generated individuals (i.e. the initial generation). In each generation, the *fitness* of each individual is evaluated first. Then multiple individuals are stochastically selected (based on fitness) and modified (commonly either recombined through crossover or randomly mutated through mutation) to from a new generation. This process is repeated until an optimal solution is found, or the evolution reaches the maximum number of generations specified. One advantage of genetic algorithm is that only the genetic representation of the solution (i.e.

candidate encoding scheme) and a fitness function need to be defined, both of which require relatively little domain specific knowledge. As a result, it is possible to solve a huge number of complex problems, in which heuristics are very hard to be deducted because of complexity of the search space landscape, using genetic algorithms and also the dialect Genetic Programming.

Genetic programming, developed mainly by John R. Koza in his pioneering book [Koz92], is a specialization of genetic algorithm, in which each individual within population under evolution is a computer program (commonly represented as an arbitrary-length tree). In [Koz92], Koza showed that "A wide variety of seemingly different problems from many different fields can be reformulated as problems of program induction". Compared with genetic algorithm, using computer programs as individual gives GP a unique advantage. In genetic algorithm, the structure of the individual (the encoding scheme) needs to be defined explicitly and is fixed in evolution process. For complex problems, defining the structure itself can be a very hard task even for domain experts. On the other hand, computer programs evolved in GP are flexible with variable shapes. This allows GP to perform structural optimization (i.e. the structure of individual can also be evolved) at the same time of the parameter optimization. In recent years, GP has been successfully applied to a variety of engineering problems including symbolic regression, classification control, robotics, game playing, cellular automata programming and many others. Good surveys of GP applications can be found in [WHM+97], [BBSW03] and [RV10].

The usage of arbitrary-length tree representation in GP promotes the applicability of GP to more complex problems. But it has an unforeseen side effect: the size of individuals in the population under evolution tends to grow uncontrolled without improvements in the quality of the solution. This phenomenon, called *bloat* or *bloating*, has the four negative effects to GP. Firstly, bloating slows down the search process. In the central of GP lies the fitness evaluation which is computational expensive and resource demanding. The larger the individuals within population, the slower the whole GP searching process would be. Secondly, bloating demands more memory space. As the memory price gets much cheaper in recent years, this seems not be a huge problem. But when applying GP to more complicated problems, it is natural to program GP with distributed architecture. Population synchronization would become an issue if the individual

were very big. Thirdly, bloating results in decease in solution quality. For example, hardware topology synthesis is one of the most important application areas for GP. As the GP generated solution will be eventually used to produce hardware chip, larger solutions result in higher manufacturing cost and higher power consumption, neither of which are desirable. Last but not least, bloating places a time limit to GP. In GA where fixed-length encoding is used, the algorithm can always run as long as needed. This is however not the case in GP. Because of bloating, available computation resources may be exhausted long before the desired solution is found, especially in complex problems. As a result, in GP, there is a race between finding the optimal solution and resources exhausted caused by bloating. These negative effects of bloating represent challenges when scaling up GP to more complex problem domains.

1.2 Objectives

The objective of this thesis is to explore modifications of standard GP in order to improve GP algorithm's general performance, such that GP algorithm can be applied to more complex problem domains. This thesis full fills this objective by trying to answer the following three questions:

1. How bad is the bloating?
2. How can bloating be controlled?
3. Apart from bloating control, how can we improve the GP performance in general?

The purpose of the first question is to develop a better understanding of the problem of bloating itself. Bloating has been cited as a severe problem which would limit the scalability of GP for a long time. Most of the works in literature have been concentrating on finding the root cause of bloating and how to eliminate it. But, there is very little effort spending on bloating itself. The author feels that it would be more appropriate to develop a deeper understanding of bloating itself before trying to attack it.

The purpose of the second question is to explore alternative bloating control methods. Since the development of GP, there are a good number of effective bloating control methods have been developed. Are there any missing dynamics can be leveraged to develop simple and effective bloating control methods?

The purpose of the third question is to explore alternative approaches to improve GP performance. This is because if the ultimate purpose of the bloating control is to allow GP to be applied to more complex problems, in theory the same effect could also be achieved by improving GP performance by itself rather than fighting bloating.

1.3 Contributions

This thesis presents three main contributions to GP research work:

1. It develops the concept of *Activation Rate*. Activation rate quantitatively models the importance of a node within a tree. Using activation rate, the thesis qualitatively analyzes how bloating effects the computation effort (number of nodes) required to evaluate a program tree. It also analyzes the crossover effect in a more detailed manner separating the crossover selection from subtree swapping using activation rate. Finally, it develops a new crossover called semi-intron crossover, based on activation rate. Part of this work has been published in [LZ10a];

2. It implements a new bloating control method called *Depth Constraint Crossover*. Depth constraint crossover is motivated by a quantitative analysis of removal bias bloating theory. The thesis quantitatively defines removal bias and empirically studies the correlation between the removal bias defined and bloating. It then theorizes a new cause of bloating called *Depth Difference Hypothesis*. This thesis also presents experiment results showing the effectiveness of depth constraint crossover in controlling bloating by comparing it with several existing bloating control methods. Part of this work has been published in [LZ10b];

3. It develops a new fitness function scheme called *Norm-referenced fitness function*, which is motivated by norm-referenced test in education. The new fitness function scheme defines fitness considering both individual's raw fitness and the population's average fitness. Empirical studies show that, norm-referenced fitness function is capable of improving the overall performance of GP system. In additional, theoretical analysis of norm-referenced fitness function reveals that the original fitness function is actually a special case of norm-referenced fitness function. Norm-referenced fitness function is

able to explicitly address one weakness of the original fitness function, which is defined as the *curse of evolution*. Part of this work has been published in [LZ11].

1.4 Thesis Overview

The rest of this thesis is organized as follows. The following two chapters introduce genetic programming in detail, and provide a review of GP literature. Chapter 2 gives an introduction to genetic programming, and benchmark problems. Chapter 3 gives a review of GP literature mainly on bloating theory and bloating control. These two chapters serve as background for the rest of this thesis.

In chapter 4, we address the question "How bad is the bloating?". The concept of activation rate is developed and studied in detail. The power of activation rate is that it is possible to estimate the activation rate and use it for theoretical analysis of GP. Here, using activation rate, more precisely the estimated activation rate, the computation effort required to evaluate trees are formally studied. Also, a number of interesting experiment results about crossover are also discussed.

Chapter 5 tries to answer "How can bloating be controlled?". It firstly reviews a well established bloating control theory, the removal bias. Then it extends the removal bias theory using a new quantitative model and discovers depth difference hypothesis as a new drive behind bloating. It then develops depth constraint crossover which explicitly address the issue raised by depth difference hypothesis and gives a number of empirical evidences to support the effectiveness of depth constraint crossover.

Chapter 6 presents norm-referenced fitness function, which extends the original fitness function to not only taking into account the raw fitness achieved by the individual, but also how well the rest of the individuals perform within the same population, in order to answer the question "How can we improve the GP performance in general?". It further shows that the norm-referenced fitness function is a general case of the original fitness function. The original fitness function suffers from the limitation of curse of evolution. By adjusting the control parameter λ appropriately, norm-referenced fitness function is able to overcome the limitation of the original fitness function and significantly improve GP performance

across all benchmark problems. In addition, a modification of tournament selection called partial norm-referenced fitness function is also developed to address one potential runtime limitation of norm-referenced fitness function when used in combination with tournament selection.

Chapter 7 concludes this thesis with a summary of the contributions made and a detailed discussion of the further works.

Chapter 2

Genetic Programming

Inspired by Darwin's natural selection, genetic programming (GP) is a computer simulated evolution environment evolving computer programs. In the last chapter, we gave a general description of how GP works. In this chapter, we examines genetic programming workflow in more detail. This information serves as background knowledge for discussions in subsequent chapters. This chapter is organized as follows. Firstly, we give an introduction of the representations of computer programs in GP. We then show how GP works and discuss each GP phase in detail. Finally, we conclude this chapter by describing the four most popular GP benchmark problems.

2.1 Variable Length Representation

As we mentioned in Section 1.1, one of the most important features which distinguishes genetic programming from genetic algorithm is that, the individuals under evolution in GP are computer programs. Computer programs can be represented in many different ways. The most widely used representation is tree. Tree representation of a computer program follows the functional programming paradigm of defining programs. In functional programming, computation is treated as the evaluation of mathematical functions (functions which do not have side effects) or a combination of functions in the form of high-order functions. In this way, the notion of state and mutable data are avoided. In [Koz92], Koza uses Lisp S-expression to represent computer programs and tree to visualize it, as shown in Figure 2.1. In a typical tree, there are two kinds of nodes: leaf nodes and non-leaf nodes. Leaf nodes represent inputs into the program while non-leaf nodes

represent functions. As a non-fixed representation, individual program trees may

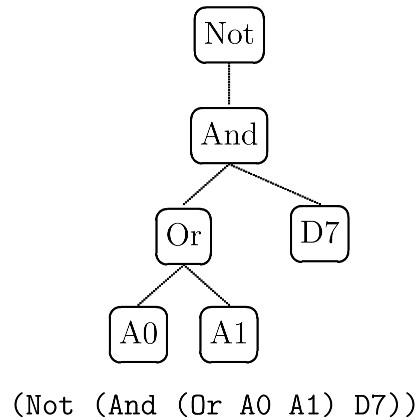


Figure 2.1: An Example GP Tree and its Lisp S-expression

vary in depth, number of nodes (size), and also shape. This differs from the fixed representation scheme used in genetic algorithms in which every individual has the same format. Another significant difference is that in classic genetic algorithm, every bit in the individual has the same weight if not explicitly changed. In another word, the string representation is a non-hierarchical structure. But in the case of GP, trees are hierarchical. Each node in a tree has a weight implicitly associated. It is well known in a GP tree that, nodes towards the root of the tree are more important than nodes deep down in the same tree. The importance of the node not only depends on itself but also depends on the surroundings it stays in. Program trees (also known as Tree-based GP) are the primary focus in the thesis. Chapter 4 greatly extends the discussion of tree structure.

In addition to trees, other computer program representations also available in GP including Linear GP [Bra04], Parallel Distributed GP [Pol97], Cartesian GP [MH08] and so on. Linear GP defines a computer program following the imperative programming paradigm. Imperative programming paradigm is the most widely used programming paradigm in modern programming languages. Imperative programming describes computation in terms of instructions that change program states. In linear GP, computer programs are linear sequences of instructions which execute sequentially. Instructions are able to read program states from, and store results back to registers and memory. One motivation for the development of Linear GP is that imperative programming paradigm fits into

modern computer architecture more neutrally compared to declarative programming paradigm, such as functional programming. The other arguably more important motivation is performance. GP is a computational demanding algorithm especially when the population evolved is large. Most of the computation efforts required in GP are spent on evaluating individuals' fitness. While functional programs such as Lisp S-expressions cannot be executed directly without interpreters which convert higher-level code into machine code at runtime, Linear GP instructions which usually are machine codes or higher-level language constructs however can be executed directly without this translation process. This makes it possible to evaluate population of similar size several orders of magnitude faster. Parallel Distributed GP and Cartesian GP are two examples of Graph-based GP. The development of Graph-based GP is motivated by the fact that tree is actually a special kind of graph. Graph-based GP is suitable for evolving highly parallel programs by reusing partial results, i.e. repeated evaluation of the same subtree can be avoided. But special genetic operators are required in breeding phase in order to ensure offsprings produced having the correct syntax. A much more comprehensive introduction of these GP variations can be found in [PLM08].

2.2 Genetic Programming Inputs

Before applying genetic programming to a specific problem domain, the following domain specific inputs need to be defined: *function set*, *terminal set*, *fitness function*, and a number of *control parameters*. Function set is the set of all available functions for a problem. Terminal set is the set of all inputs for a problem. When a program tree is constructed, the tree's non-leaf nodes are always drawn from the function set while leaf nodes are drawn from the terminal set. For example, in Figure 2.1, Not, And, and Or are from function set, while A0, A1, and D7 are from terminal set. Similar to functions defined in programming languages, functions in function set usually have a number of input arguments and a single output. This is regarded using the parent/child relationship and is visualized as edges connecting tree nodes. For example, in Figure 2.1, we say that D7 is a child of And and And is the parent of node D7.

The function set and terminal set together form the primitives available for GP to explore the problem searching space. The completeness and sufficiency of the terminal and function set are critical when applying GP. If too many values

are put into either function set or terminal set, the searching space of GP increases exponentially. However, if the function set or terminal set have too few values, they may not be sufficient to construct the optimal solution. For example, in a symbolic regression problem to approximate a damping function, the problem becomes much easier when trigonometric functions are available in the function set. Otherwise, GP needs to discover Taylor expansion dynamically to be able to correctly approximate the objective function with only polynomials, which is much harder. In complex problems, determining function set and terminal set itself is not a trivial exercise. For example, to simulate stock price movement using GP, is historical price sufficient as terminal set? Should moving average also be part of terminal set? Are primitive arithmetic operators sufficient to form the function set? Or, higher order functions such as first order derivative or domain specific functions such as Black-Scholes model [BS73] also should be part of the function set? These questions can only be asked by domain experts on a problem by problem basis.On one hand, in most cases when GP behavior is studied using benchmark problems, the function and terminal set are considered to be well defined, on the other hand, when applying GP to solve practical problems, the choice of function and terminal sets needs to be carefully justified.

Fitness function is used to judge the quality of the individual candidate solution within the population evolved. The quality is usually also referred as *fitness* or *raw fitness*. The exact form of fitness function may vary from problem to problem. Typically, a fitness function consists of a number of test cases. Each test case includes a number of inputs and a desired output. A program tree is evaluated using the inputs and the actual output is compared with the target output. If the actual output matches the target output (or within a predefined error margin), the program tree is considered to be correct for this test case. Because each program tree needs to be evaluated using all test cases, the more test cases defined in fitness function, the longer it takes to evaluate. The choice of test cases can either be a permutation of all input values, or a sample of values. Fitness plays a central role in GP because, raw fitness or a value derived from raw fitness, guides the selection of fitter individuals to produce offsprings. Section 2.3.2.1 gives a more detailed discussion about fitness.

Control parameters also play an important role in genetic programming. Common control parameters include population size, maximum number of generations, crossover rate, mutation rate and termination condition. Generally, in GP, the

population size used is usually bigger than what is used in genetic algorithm. The most classic setting from Koza in [Koz92] is 4000. Some later researches use 2000 or 500. In Koza's classic settings, the maximum number of generations is 50, crossover rate is 90% and mutation rate is 10%. The termination condition is either the maximum number of generations reached or the optimal solution is found before. Those settings remain the most popular choices in most later research.

2.3 Genetic Programming Workflow

Despite some minor variations, genetic programming generally works as described in Algorithm 1:

Algorithm 1 General Genetic Programming Flow

1: **function** apply-genetic-programming(terminal set $\boldsymbol{ts}$, function set $\boldsymbol{fs}$, maxGeneration mg, fitness function f, Control Params $params$) : best Individual
2: population $\boldsymbol{p}$ $\leftarrow$ build-initial-population($\boldsymbol{ts}$, $\boldsymbol{fs}$, $params.popSize$, $params.maxInitDepth$)
3: currentGen $cg \leftarrow 0$
4: **while** $cg < mg$ **do**
5: fitness $\boldsymbol{f} \leftarrow \{\}$
6: **for all** individual i in p **do**
7: $\boldsymbol{f} \leftarrow \boldsymbol{f} \cup f(i)$
8: **end for**
9: **if** reachBest($\boldsymbol{p}$, $\boldsymbol{f}$) **then**
10: **return** best($\boldsymbol{p}$)
11: **else**
12: $\boldsymbol{p} \leftarrow$ breed-new-population($\boldsymbol{p}$, $\boldsymbol{f}$, $params$)
13: **end if**
14: $cg \leftarrow cg + 1$
15: **end while**
16: **return** best($\boldsymbol{p}$)
17: **end function**

The key components/procedures/operations in GP include:

1. Generation of the initial population, which creates the first generation of population randomly. The fist generation or generation 0 serves as the starting point for evolution;

2. Fitness evaluation, which evaluates individual's raw fitness based on fitness function defined. The fitness is used to guide the selection of parents in breeding phase;

3. Breeding for new generation, which creates new population from the current population. Breeding phase contains two steps: parent selection and breeding.

2.3.1 Initial Population Generation

There are three commonly used methods to generate the initial population, introduced by Koza in [Koz92], Full, Grow and ramp-half-and half. Full method always chooses node from function set if the depth of the node is smaller than the predefined limit. When predefined maximum depth is reached, it selects from terminal set to complete the tree. The full method can be summarized using the recursive code in Algorithm 2.

Algorithm 2 Full Method to Randomly Generate Tree

```
1: function build-tree-using-full-method(terminal set ts, function set fs,
   maxDepth md, currentDepth cd) : rootNode rn
2: if cd < md then
3:    rn ← select random f from fs
4:    for all childNode cn ∈ rn.inputs do
5:       cn ← build-tree-using-full-method(ts, fs, md, cd + 1)
6:    end for
7: else
8:    rn ← select random t from ts
9: end if
10: return rn
11: end function
```

With full method, trees generated are guaranteed that every branch has the maximum depth specified. In grow method, the root node is randomly selected from the function set. Then, child nodes whose depths are smaller than maximum allowed depth are selected randomly from not only function set but also terminal set. As a result, grow method is able to generate trees with different shapes, containing branches of various depths. The grow method can be summarized using the recursive code in Algorithm 3.

Algorithm 3 Grow Method to Randomly Generate Tree

```
1: function build-tree-using-grow-method(terminal set ts, function set fs,
   maxDepth md, currentDepth cd) : rootNode rn
2: if cd = 1 then
3:    rn ← select random f from fs
4: else
5:    if cd < md then
6:       rn ← select random n from fs ∪ ts
7:    end if
8: else
9:    rn ← select random t from ts
10: end if
11: if rn ∈ fs then
12:    for all childNode cn ∈ rn.inputs do
13:       cn ← build-tree-using-grow-method(ts, fs, md, cd + 1)
14:    end for
15: end if
16: return rn
17: end function
```

Unlike full and grow methods, ramp-half-and-half method is not a new method to generate a single program tree, rather a systematic approach used in combination with grow and full methods to enhance the diversity when generating a population of program trees, in terms of tree shape, size, and depth. Let d be the maximum allowed tree depth and n be the number of program trees to generate, ramp-half-and-half method firstly divides n into d groups of depth $1, 2, \ldots, d-1, d$. Then for each depth group which contains $\frac{n}{d}$ individuals, half of the individuals are generated using full method, the other half are generated using grow method.

Although full and grow method are primarily designed to generate trees for initial population, these two methods, especially grow method, are also used to generate random sub-trees in mutation operation. Given the simplicity, these two methods together with ramp-half-and-half method are the most widely implemented methods to initialize population in GP. But, these classic methods do have some drawbacks. For example, one limitation of grow method is that, even through the maximum allowed depth is defined, there is very little guarantee that the program tree generated has at least one branch which reaches maximum allowed depth. This is because the actual depth of the tree generated using grow method highly depends on the size of function set compared to terminal set. If

the terminal set is considerably larger compared to function set, the tree generated tends to be very small due to the fact that terminals are more likely to be selected during the generation of the non-leaf nodes. In [Iba96], Iba suggests that these methods are not necessarily a uniform sampling of the searching space. Langdon and Poli also suggest that ramped-half-and-half method has a bias and this partially results in GP performs poorly in ant problems [LP98a]. In addition, constraint on the maximum allowed depth may be inadequate for certain problem domains. An example is symbolic regression problem where trees generated using ramp-half-and-half method with maximum allowed depth set to 6 tend to be too small and too simple. As an alternative to tree depth, Chellapilla in [Che97] uses tree size (number of nodes) as the constraint in stead of the depth and reports positive results. Moreover, recent researches also find that initialization methods may affect bloating. In [Luk00b], Luke points out that grow method used in mutation operator can produce substantial amount of bloat even in the absence of selection pressure. In [DP07], Dignum and Poli also suggest that initialization methods which tend to produce smaller trees may speed up bloating.

There are a number of alternative initialization methods available. In [Iba96], Iba introduces RAND_tree algorithm which generates trees uniformly based on a bijection method, which improves the GP performance. In [Luk00c], Luke introduces two tree-creation algorithms Probabilistic Tree-Creation (PTC) 1 and 2. PTC1 allows user to define generated tree size and specify the user-defined probabilities over the appearance of functions. PTC2 further extends PTC1 to allow user to define a probability distribution over generated tree sizes. Both methods give user better control over tree creation enabling more rigorous control over expected tree size with a very low computational complexity. Seeding is another method which allows user to input hand-crafted trees into the initial population. These hand-crafted trees can be created by domain experts who can suggest a good starting point of search. The initial populations can be created by mutating a handful number of hand-crafted trees. When applying seeding, it is important to maintain the population's diversity since the tree generation is no longer random. A good introduction on seeding can be found in [PLM08]. There are also researches in Grammar-based GP related to population generation. Grammar-based GP [Whi96] is a branch of GP which allows Grammar based constraints to be applied to program trees under evolution. Special tree initialization algorithm is required to ensure the tree created follows the grammar constraints. [BGS96]

and [GAMRRP07] develop algorithms to uniformly generate trees in the context of Grammar-based GP.

2.3.2 Fitness Evaluation

Fitness evaluation is the most important procedure in GP. This is not only because it is the most computationally expensive step, but also the result from fitness evaluation, *fitness*, is used to determine whether the solution has been found, and to guide the selection of individuals to produce the subsequent generation.

2.3.2.1 Fitness Value

In the fitness evaluation step, the fitness function is applied to individuals within population to measure the quality of the individual, called *raw fitness*. According to Koza, raw fitness is defined as "the measurement of fitness that is stated in the natural terminology of the problem itself" [Koz92]. The raw fitness value usually is a single aggregated value over raw results from a number of test cases. When fitness is used in breeding phase to select "fitter" individuals, raw fitness can be used directly. But more commonly, a transformation of raw fitness value as an alternative measurement of fitness is used. In [Koz92], for example, Koza also defines *standardised fitness*, *adjusted fitness* and *normalised fitness*. The standardised fitness is defined to be non-negative and smaller values represent better individuals. It is customary, but not absolutely necessary to let the optimal individual to have standard fitness value 0. Standard fitness is good to tell whether one individual is better than another individual. Adjusted fitness is defined to easily tell how much better or worse an individual is compared with another individual. Normalised fitness is calculated by dividing adjusted fitness for the individual by the summation of the adjusted fitness for all individuals within the population. Both adjusted fitness and normalised fitness are useful when applying fitness proportionate selection (see Section 2.3.3), while rank based selection methods such as tournament selection can use any of these fitness measurements.

In addition to evolve a single objective, GP can also evolve multiple objectives at the same time. The existence of multiple objectives may be due to the nature of the problem, i.e. minimizing cost and maximizing performance, or come from

non-functional requirements such as parsimony, efficiency and so on. One natural way to solve a multi-objective problem is to convert it into a single-objective problem by combining the individual objective functions into a single composite function (using weighted sum for example). In [Koz92], Koza gives an example of fitness function combining correctness (with 75% as weight) and efficiency (with 25% as weight) to solve the block stacking problem. Another widely used example in GP is parametric parsimony pressure [LP06]. Parametric parsimony pressure is a bloating control method which modifies the fitness value to take into account individual program tree's size or depth, in additional to raw fitness. The motivation behind parametric parsimony pressure is to penalize individuals with the same raw fitness but have bigger program trees. A more detailed review on parametric parsimony pressure will be given in Section 3.4.2. One problem with the weighted summation approach is that proper selection of weights or utility functions are extremely hard even for domain experts. Moveover, the relative importance of these objectives may change over generations as well. Another problem is that it can be very difficult to scale multiple objectives properly. Another approach to solve multi-objective GP problems is to adapt the related techniques from other evolutionary algorithms such as genetic algorithm. The main idea behind multi-objective optimization is to find the Pareto optimal solution set (Pareto front) [KCS06]. A Pareto optimal set is a set of solutions that are non-dominated by each other. A candidate solution is said to dominate another candidate solution, if and only if the former is not inferior to the later in all objectives and there is at least one objective in which the former is better. A very good tutorial about multi-objective in the context of genetic algorithm can be found in [KCS06].

In most of the cases, fitness function gives a numeric value as individual's fitness. This is desired rather than necessary and in some special cases, it is not even required. Since the ultimate purpose of fitness is to serve the selection procedure, different selection methods demand the fitness information in different forms. For example, fitness proportionate selection requires fitness values in a very high precision. Tournament selection, on the other hand, only requires fitness ranking information. In a more extreme example, Tettamanzi [Tet96] presents competitive selection, with which the notion of fitness is never directly used.

2.3.2.2 Evaluation Performance

Evolutionary computation is well known to be a relatively slow optimization method compared to other methods such as linear programming, neural network and so on. This is partially because evolutionary computation is evolving a population of candidate solutions, while other methods mainly work on a single one. In the case of GP, managing performance becomes even more challenging because of bloating. As we briefly introduced in Section 1.1, bloating is a well observed phenomenon in GP that the depth and size of program trees increase without improvements in individuals' fitness. One of the problems caused by bloating is that it takes longer and longer to evaluate the population as the population evolves. When optimizing GP runtime performance, fitness evaluation is always the primary concern as fitness evaluation is the most time consuming and computation intensive operations in GP.

In this section, we briefly review a number of different techniques proposed in literature to optimize fitness evaluation. These techniques can be divided into three categories. The fist category of methods directly speeds up fitness evaluation. One of the motivation behind the development of Linear GP is to be able to evolve real machine codes, which are much faster to execute compared to logical program trees. Distributed GP is able to distribute the evaluation workload over a network of computation units. An example of such distributed GP system is PGPS proposed in [OCPT97]. PGPS uses a master-slave model in which there is a master instance performing selection, breeding and evaluation of individuals are distributed to a number of slave nodes. Another example is island model adopted from other evolutionary computation methods which divides population into multiple sub-populations and evolves sub-populations independently on distributed nodes. Other works fall into this category include: in [KM94], Keith and Martin discuss GP implementation issues in detail, comparing several different usage of data structures and their impact to runtime performance. In [Kei04], Keijzer examines a number of subtree caching mechanisms that are capable of improving the runtime efficiency of GP system. In [Lan09], Langdon develops an implementation of GP using CUDA. CUDA is a programming API for NVIDIA graphic card. It enables scientific programming or general purpose programming to leverage the power of GPU.

The second category of techniques reduces the number of fitness test cases need to be evaluated. For example, in [Alt94b, Alt94a, Tac94], Altenberg and

Tackett develop brood recombination crossover which uses only a small fraction of test cases to evaluate offsprings produced. In brood recombination, rather than only producing one pair of offsprings, each pair of parent produces a number of offsprings. But only one pair of offspring survives based on the *culling function*, a function derived from the fitness function which is much less computationally costly. By using culling function, even through overall there are more offsprings to be evaluated because of the brood recombination, the overall computational cost is still managed. In [GTV02], a statistical method is used to select only a fraction of test cases in fitness evaluation.

The third category of methods reduces the number of individuals that need to be evaluated in fitness evaluation. For example, in [XZA06b], a population clustering method is proposed to decrease the total number of individuals that need to be evaluated. In [Koz92], Koza avoids re-evaluation of individuals who are created by reproduction. In [Jac05], Jackson finds a special kind of crossover, the fitness-preserving crossover, in which offsprings produced have the same fitness as their parent. Jackson explicitly used this property to avoid fitness evaluation on those offsprings created by fitness-preserving crossover. In [LBP03], an adaptive parameter technique is developed to gradually decrease population size in GP rather than using a fixed-size population.

In additional to these methods, most of bloating control methods are able to improve evaluation performance by reducing the depth and size of program trees. These methods will be reviewed in detail in Chapter 3. An example of these methods is Tarpeian method. In [Pol03], Poli develops Tarpeian method which marks randomly selected individuals whose size (number of nodes) are above population's average individual size to lowest possible fitness value without evaluating them. Although it is primarily designed to control bloating, it is also able to improve evaluation performance since these large individuals marked no longer require to be evaluated.

2.3.3 Parent Selection

Based on Darwin's natural selection principle, GP evolves the population of candidate solutions by selecting fitter individuals in the current generation to create offsprings to form the next generation. Fitter individuals have higher chance to be selected as parents to produce offsprings. This simulates the idea of "the fitter survives". We split the discussion of breeding phase in GP into two sections. In

this section, we discuss a variety of selection methods, and in the next section, we review a number of common breeding operators.

Before any selection methods are reviewed, we firstly discuss several measurements of selection methods. The key concept behind selection is *selection pressure*. There are a number of definitions of selection pressure can be found in literature. In the simplest form, selection pressure measures how much more likely a fitter individual will be selected over an average individual. The higher the selection pressure is, the faster the system converges. Fast convergence speed reduces the overall running time for GP, but may leads to what is known as premature convergence i.e. system converges to a local minimum. On the other hand, lower convergence speed decreases the chance of premature convergence, but may results in GP failing to find optimal solution because of either bloating or reaching maximum number of generations specified. Selection pressure is not only affected by selection method, it is also subject to the distribution of fitness in the population [BT96]. A number of measurement of selection pressure have been developed by Blickle and Thiele in [BT96], including average fitness, fitness variance, reproduction rate, loss of diversity and selection intensity. Also, in [GD91], growth ratio and takeover time are introduced. Although these measurements are developed in the context of genetic algorithm, they can be directly applied to genetic programming. Here, we only give a brief review of selection intensity, which will be used in later chapters. More information of other measurements can be found in [BT96]. The selection intensity I of a selection method is defined as:

$$I = \frac{\bar{M}^{*} - \bar{M}}{\bar{\sigma}}$$

where $\bar{M}^{*}$ is the expected mean fitness after selection, $\bar{M}$ is the expected mean fitness before selection, and $\bar{\sigma}$ is the mean fitness variance before selection.

In GP, most of the common selection methods are adapted from genetic algorithm. The most popular selection methods include fitness proportionate selection, ranking selection and tournament selection. Fitness proportionate selection is firstly developed by Holland in [Hol92], and firstly used in GP in [Koz92] by Koza. In fitness proportionate selection, the probability that an individual is selected is proportional to the fitness of that individual. Formally, given a population P with n individuals and individual i's fitness is f_i, the probability

individual i being selected is:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j},$$

Fitness proportionate selection has several limitations. The first one is that fitness proportionate selection depends on the scale of fitness function [BT96]. If the differences between high fit and low fit are small, the selection method tends to have very small selection pressure. On the other hand, if the differences are large, fitter individuals would dominate in the selection process. Another problem is that the selection pressure varies in different stage of evolution. In early stage of evolution for example, the selection pressure tends to be very small [BT96] due to the fact that randomly generated individuals tend to have similar fitness. To overcome this problem, an improvement over proportionate selection, greedy over-selection is introduced by Koza in [Koz92] to improve the selection intensity, and hence improve the performance of GP. In greedy over-selection, the population is divided into two groups, Group I and Group II, based on the individuals' fitness value. Group I's individuals are fitter than Group II's. When an individual is selected, individual is selected from Group I with 80% probability. A final problem is that the common implementation of fitness proportionate selection, the weighted roulette wheel, is quite computational expensive with time complexity of $O(n^2)$ [GD91]. A pre-sorting may reduce the lookup complexity to $O(nlogn)$. But sorting itself can be computational expensive as well, especially when the population is large. Moreover, the selection can only be performed after the whole population have been evaluated.

Another commonly used selection method is tournament selection. Unlike fitness proportionate selection in which the fitness value is directly used, tournament selection uses ranking information. It simulates real world selection scenarios, for example, when two bulls fight over the right to mate with a given cow. In tournament selection, a fixed number of individuals (common tournament size in GA is 2, in GP is more commonly to use 5 or 7) are randomly selected to form a tournament. Then the fittest in the tournament is selected. One advantage of tournament selection is that, the selection pressure can be easily tuned by changing tournament size. Generally speaking, given a population, bigger tournament size results in higher selection pressure. A more detailed study of tournament selection pressure can be found in [BT96]. Another advantage of tournament

selection is that it is very fast with time complexity $O(n)$, without requiring to pre-sort the population. In addition, evaluation of individuals can be delayed to when the individual is selected as part of a tournament. With the "not-sampled issue" for tournament selection [XZ12] (i.e. not all individuals are guaranteed to be selected at least once as part of a tournament), fitness evaluation of individuals not being sampled can be avoided. Moreover, each tournament can be formed in parallel, which makes tournament selection a natural choice in parallel implementation of GP. Two issues with tournament selection are multi-sampled issue and not-sampled issue, which are studied in detail in [XZ12]. Multi-sampled issue refers to the same individual sampled multiple times in a tournament, while not-sampled issue refers to individual not being selected for any tournaments. In [XZ12], Xie and Zhang clarify that resolving multi-sampled issue and not-sampled issue do not improve the tournament selection in a statistical significant manner. Researches also have been performed to get a finer level selection pressure control over tournament selection. For example, in [GD91], Goldberg and Deb present a modification of tournament selection introducing an extra probability p to give a finer level of control when tournament size is set to 2 in the context of genetic algorithm. Although the primary motivation of population clustering algorithm is to improve fitness evaluation performance, it is demonstrated in [XZA06a] that the algorithm is also able to dynamically adjust selection pressure along with evolution.

According to Blickle and Thiele in [BT96], ranking selection is firstly developed in [GB89] in the context of genetic algorithm to resolve the problems of fitness proportionate selection. In ranking selection, individuals are firstly sorted based on fitness values. If there are N individuals within population, the worst individual has rank 1 and the best individual has rank N. Then, the probability for a individual with rank i to be selected is determined by a function. This function can be either linear, exponential [BT96], or even polynomial [HH08]. In linear rank selection, let i be the rank of the individual, N be the population size and η be the control parameter, the probability for individual with rank i being selected is determined by the following function:

$$p_i = \frac{1}{N}(1 - \eta + (2\eta - 2)\frac{i-1}{N-1}),$$

in which, the real valued control parameter η can be used to tune selection pressure and $1 \leq \eta \leq 2$. When $\eta = 1$, $p_i = \frac{1}{N}$, i.e. each rank is selected uniformly. When $\eta = 2$, ranking selection gives highest selection pressure. In exponential ranking selection, the probability is determined using:

$$p_i = \frac{c^{N-i}}{\sum_{j=1}^{N} c^{N-j}}$$

where the exponent base parameter $0 < c < 1$, can be used to control the selection pressure. In polynomial rank selection, the probability for an individual with rank i being selected is:

$$p_i = \sum_{j=1}^{d+1} a_j k^{j-1}$$

where a_j are control parameters and d is the degree of the polynomial. One limitation of rank selection is that, similar to fitness proportionate selection, the whole population still needs to be sorted before individuals are selected.

There are a number of other selection methods that have been developed but not widely used in GP including truncation selection, fitness uniform selection, and so on. [BT96] gives a good review of these methods. In addition, there are also a number of modifications of standard selection methods developed for bloating control, which will be reviewed in Chapter 3.

2.3.4 Breeding Operators

There are three most widely used breeding operators in GP, namely *crossover*, *mutation* and *reproduction*. The most widely used operator is crossover. Crossover mimics the process of sexual reproduction. In the simplest form, two individuals are firstly selected using selection methods discussed in previous section from the population. Then one node is randomly selected from each individual and subtrees under two selected nodes are swapped to form a pair of offsprings. The node selected is called crossover point. Even through this standard crossover remains the most widely used crossover, it is not without problems. The biggest problem is that the crossover point selection is purely random without considering the context of the parent program. This is very different from real world biology crossover which operates in a highly constrained and controlled context. As a result, crossover in GP is generally considered to be destructive. In [NB95], Nordin

and Banzhaf quantitatively study the effect of crossover through a number of experiments. They define the change in fitness $\Delta f_{percent}$ from parent's fitness to child's fitness as:

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100$$

They analyze a large number of individuals in early generations of the symbolic regression problem and find that there is a high probability that the function of the program is severely damaged, resulting in a fitness decrease for the individual. A similar experiment with a much more complex experiment setup is performed in thesis in Section 4.7. To address this issue, a number of alternative crossover operators towards a more homologous crossover have been developed. The brood recombination [Tac94] is inspired by the fact that in real world, animals usually produce more offsprings and not all of them can live. In brood recombination, instead of creating only two offsprings, a much bigger number of offsprings are created and only the best two survive and put into the new generation. In [HS97], Harries and Smith develop a hill climbing based crossover. Hill climbing based crossover performs crossover between one program and an identical copy of itself with no bias of selection of crossover point. This results in a smaller change on the original program and this simulates a hill-climbing like local search. Similarly, in [Ang97], another local search method, the headless chicken crossover is developed, in which a parent is selected and crossover is performed between the selected tree and a randomly generated tree. Headless chicken crossover can also be considered as a kind of mutation since it exhibits a number of features from mutation. In [PL98], Poli and Langdon introduce *uniform crossover* which constructs offsprings in a bitwise manner, similar to crossover in GA. In uniform crossover, instead of swapping the entire subtrees, tree nodes within the similar structure in both parents are swapped individually.

Another problem with standard crossover is that there is an implicit bias towards deeper nodes due to the nature of tree structure. A random selected crossover point has higher chance to be at a bigger depth simply due to the fact that the number of nodes of a certain depth increases as the depth increases. This implicit bias results in smaller subtrees being swapped and hence smaller effects on crossover. Considering leaf nodes (deepest nodes) as a special case, in [Koz92], Koza defines that 90% of time the crossover point is selected at non-leaf nodes, while 10% of time the crossover point is selected at terminal nodes. A much more comprehensive study is conducted by Angeline in [Ang96] in which four fixed

leaf node selection frequencies and two adaptive leaf node selection methods are experimented. Results of experiments show that there is not a universal setting which is optimal for every problem. To address the more general depth bias issue, in [HS97], Harries and Smith develop a depth-based crossover, in which the depth of crossover point is selected with equal probability. In [IIS99], depth-dependent crossover further expands depth-based crossover by allowing a pre-defined probability or self-adapted probability assigned to selection of crossover point depth.

Unlike crossover, mutation is a genetic operator which works on a single parent. In mutation, similar to crossover point, one node called mutation point is randomly selected from the parent, and then the subtree under the mutation point is replaced with a randomly generated subtree. Typically, grow method discussed in Section 2.3.1 is used to generate the subtree. This form of mutation is also called subtree mutation and it is the most common form of mutation. A more comprehensive list of mutation variations can be found in [PS06] and [PLM08]. Here, we only go through a number of typical ones. Hoist mutation used in [Kin93] creates the new individual by randomly selecting a subtree from the selected parents. This results in a smaller individual created compared to the parent. Point mutation used in [PL97a] randomly selects and changes one node rather than the entire subtree. If a function node is selected, function with the same arity is replaced to preserve the structure of the tree. In [Ang96], Angeline uses a non-standard genetic programming algorithm, in which offspring reproduction is done firstly by crossover. Then, offsprings created are mutated before putting into new population. Five different mutations are used in the algorithm: grow, shrink, cycle, switch and numerical terminal mutation. Grow mutation randomly selects a leaf node and replaces it with a randomly generated tree up to depth 7. This increases mutated tree size. The shrink mutation on the other hand reduces tree size by replacing a random subtree with a random terminal. Cycle mutation works the same as hoist mutation with the limitation that only function nodes are selected. Switch mutation swaps the subtrees from a random parent node. The parent node needs to be a function which is not commutative. Lastly, numerical terminal mutation assign a random numeric value draw from a Gaussian distribution to a random leaf numeric node, which is similar to ephemeral random constants (ERCs) used in [Koz92]. In [VTCC03], inflate and deflate mutation are developed to study fitness-distance correlation.

Uniform subtree mutation is developed in [BA02] as an alternative to standard subtree mutation to control bloating. In [BJ09], Beadle and Johnson extend their research on semantically driven crossover [BJ08] to mutation and develop semantically driven mutation (SDM). SDM performs a number of standard sub-tree mutation attempts and only accept the one which mutate the parent into a new behavioral state.

Even through mutation is actively used in genetic algorithm, mutation is generally considered as destructive and is not widely used in GP. This is partially because in [Koz92] Koza's classic breeding setting, crossover is used 90% of time, reproduction is used 10% of time and mutation is not used at all. When mutation is used in early GP works, the mutation to crossover ratio tends to be very low as well. Most of researches afterwards follow the same configuration. But there are a number of researches latter show that mutation can be beneficial. For example, in [LS97] and later in [LS98], Luke and Spector perform extensive amount of experiments to study the effect of crossover and mutation with a number of different parameter setups. They conclude that with correct parameter setting, crossover is more effective compared to mutation. But in most of the cases, the difference between these two operators are not significant.

Finally, the reproduction operator [Koz92], instead of creating new individuals based on selected individuals, directly injects individuals with high fitness value into the new population. This approach mimics the idea of "the fittest survives". Reproduction is usually used along with either crossover or mutation or both to form the breeding strategy. One benefit of using reproduction is that individuals created via reproduction do not need to be re-evaluated. This reduces the fitness evaluation cost of GP.

2.4 Benchmark Problems

There are four testing problems firstly formulated and used in [Koz92], *multiplexer*, *symbolic regression*, *parity* and *artificial ant*. These four problems then become the most widely used benchmark problems in GP. Most of researches in GP use these problems to test and compare algorithms' efficiency.

2.4.1 Multiplexer

Multiplexer is a boolean circuit design problem. The input to the Multiplexer N circuit consists of k address bits a_i and 2^k data bits d_i, such that $N = k + 2^k$. So, the input of Multiplexer circuit is:

$$a_{k-1}, \ldots, a_0, d_{2^k-1}, \ldots, d_1, d_0.$$

The output of the Multiplexer circuit is the boolean value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. The commonly used Multiplexer problem is Multiplexer 6 ($k = 2$) and Multiplexer 11 ($k = 3$).

The Terminal set of Multiplexer problem is the set of inputs. For example, in Multiplexer 11, the Terminal set is:

$$T = \{A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7\}.$$

The function set of Multiplexer problem consists of basic boolean gates. The most commonly used function set is:

$$F = \{AND, OR, NOT, IF\}.$$

Since each terminal has value either 0 or 1, the permutation of inputs has 2^N distinct combinations. Each combination in the permutation with the expected result forms a test case. As a result, for example, in Multiplexer 11, there are $2^{11} = 2048$ test cases. The fitness evaluation function for Multiplexer problem is the number of test cases in which an individual is correct. Thus, an optimal solution gives correct result for all possible inputs.

2.4.2 Symbolic Regression

Symbolic Regression is the process of determining a summary Mathematical expression for a set of data points. The most commonly used symbolic regression is linear regression. In linear regression, one is given a set of values of various independent variables(s) and the corresponding values for the dependent variable(s). The goal is to discover a set of numerical coefficients for a linear combination of the independent variable(s) that minimizes some measure of error between the given values and computed values of the dependent variable(s). Similarly, in

quadratic regression, the goal is to discover a quadratic equation which minimizes some measure of error. In Fourier regression, the goal is to discover a sine and cosine function which minimizes error.

The process of discovering the coefficients in a formula is given in symbolic regression algorithm. But it is left to the user to define which type of function should be used. But in most complex problems, choosing the right base function (e.g. either linear or quadratic) by itself is an issue. In another word, common regression methods only takes care of how to find the coefficients of a determined formula form. But the discovery of the formula form itself is left to the user. However, in genetic programming, both the formula form and coefficients in formula can be dynamically determined and evolved.

The most commonly used benchmark problem in Symbolic Regression settings are as follows. In benchmark problem, the equation we tried to discover is $y = x^4 + x^3 + x^2 + x + 1$. The terminal set consists only the independent variable x:

$$T = \{x\}.$$

The function set consists commonly used mathematical operators:

$$F = \{+, -, *, \%, SIN, COS, EXP, LOG\}.$$

The test cases are 20 points randomly generated using target formula $y = x^4 + x^3 + x^2 + x + 1$. The fitness function is the sum of absolute value differences between value of the dependent variable produced by individual program tree and the target value of dependent variable in test cases.

Symbolic regression problem domain shows the advantage of GP using variable length representation. In complex problems, the user hardly knows the exact representation form which best fit the problem. With variable length representation, when population is evolved, the representation of each individual is also evolved. This frees user from defining the form of the representation explicitly prior to GP runs. As a result, Genetic Programming is uniquely suitable for those kinds of complex problems.

2.4.3 Parity

Like multiplexer, parity is also a boolean problem. Parity problem family has two variations, even parity and odd parity. In even parity with n boolean inputs, the expected program outputs 1 if the number of 1's in n inputs is even. The program returns 0 if the number of 1's in n inputs is odd. In odd parity with n boolean inputs, the expected program outputs 1 if the number of 1's in n inputs is odd. Commonly used number n for parity is 3, 5, 6, and 11. For a N parity problem, the terminal set contains all inputs:

$$T = \{I_0, I_1, \ldots, I_{N-2}, I_{N-1}\}$$

The function set of the parity problem consists of boolean functions:

$$F = \{AND, NAND, OR, NOR\}.$$

Usually, the permutation of inputs is used as test cases. For a N parity problem, there are 2^N test cases. Similar to multiplexer problem, given an individual, the fitness function returns the number of test cases in which the individual's output is correct.

2.4.4 Artificial Ant

Finally, artificial ant problem attempts to generate a robotic navigation program which controls an ant to move within a grid to "eat" pellets. The navigation program can move the ant forward, turn the ant left and turn the ant right. The goal is to find a program which controls the ant to eat the most number of food pellets within 400 time steps. There is no inputs required in artificial ant problem. As a result, the terminal set is empty. Function set of ant problem includes:

$$\{\textbf{progn3}, \textbf{progn2}, \textbf{if-food-ahead}, \textbf{move}, \textbf{left}, \textbf{right}\}$$

progn3 accepts three inputs and executes them sequentially. **progn2** accepts two inputs and executes them sequentially. **if-food-ahead** accepts two inputs and executes the fist one if there is a food pellet in front of the ant, otherwise, it executes the second input. **move** moves the ant forward one cell in grid towards the direction ant is heading, eating the food pellet if there is one. **left** turns the ant 90 degrees to the left in the same cell. **right** turns the ant 90 degrees to the

right in the same cell. One interesting feature of **move**, **left** and **right** is that they produce side effects. As a result, **progn3**, **progn2** and **if-food-ahead** are not commutative. For example, (**progn2 move left**) and (**progn2 left move**) gives completely different result.

Unlike the other three problems, the fitness function of artificial ant problem does not contain any test cases. Instead, the individual is evaluated by simulating in a predefined grid with food pellets. An ant is placed in the top left corner of the grid facing right and uses the program to move the ant for a maximum of 400 steps and the fitness of the program is the number of food pellets ate by the ant. The most widely used pellets topology is "Santa Fe Trail" introduced by Koza in [Koz92].

Chapter 3

Bloating Theory and Control Techniques

In Chapter 2, we gave a detailed introduction of genetic programming. We mentioned several times that, *bloat* or *bloating* is a phenomenon that the program trees evolved tend to grow in size and depth without improvements in quality of solution. Bloating is an undesired side effect of using arbitrary length representation in genetic programming. Because bloating severely limits the scalability of GP, as a result, a lion share of genetic programming literatures studies the cause of bloating and proposes a good number of bloating control techniques to tackle bloating since the very beginning stage of GP research. The most recent good review of bloating theory and bloating control methods can be found in [da 08]. In this chapter, we survey these researches to give a clear map of previous achievements. This review serves as background information for the rest of this thesis, especially for Chapter 5, in which a new bloating control method called depth constraint crossover is proposed. This chapter is organized as follows. Firstly, theoretical analysis of code growth in literature are surveyed. Then, survey of various bloating control techniques is provided.

3.1 Theoretical Code Growth Models

Over last few years, there are a number of theories proposed to explain why bloating occurs. These theories include *defense against crossover*, *redundancy theory* [Bli96], *replication accuracy theory* [MM95], *removal bias* [SF98b], *modification point depth theory* [Luk00a], *hitchhiking* [Tac94], *fitness causes bloat* [LP97b], and

crossover bias [PLD07]. These theories are able to explain bloating from different perspectives. But, a single comprehensive theory has not been concluded yet. This suggests that the mechanism which drives bloating is far more complex than previously thought and it is more likely a handful of causes which contribute to it.

3.1.1 Intron Theory and Defense against Crossover

Early research of bloating concentrates on the existence of introns in GP program trees. The name "intron" originates from Biology Genetics. In Genetics, DNA is considered as the genetic material that is propagated from one generation to the next. This is because DNA contains the instruction on how to build proteins, which are the building blocks of life. Although DNA is the carrier of genetic material, however it is not directly involved in protein synthesis process. It only directs protein synthesis indirectly by sending instructions in the form of RNA. A number of researches reveal that not all DNA sections are included in the RNA. In fact, surprisingly only around 3% of DNA martials exist in the RNA. DNA segments which are coded into the "final" RNA are called gene (also called *exon*). Those DNA segments which are not involved in the coding of RNA are called non-coding DNA. One kind of non-coding DNA is *intron*. In RNA creation process, introns are firstly coded into RNA with the rest of gene, but they are later removed in the final RNA product. This intron-exon structure, especially the existence of introns, remains to be fully studied and understood in Genetics. On one hand, introns do not contribute to functionality of genes since they are striped off from the final RNA. On the other hand, if introns do not make contribution to the synthesis process, it would have long been eliminated by natural selection. A good survey of intron research in Genetics can be found in [WL96].

In the context of evolution computation, intron is usually defined as "non-functional" codes or sub-programs within an individual, which does not affect the fitness of the individual. Introns are not all in the same form. The formation and structure of one intron can greatly differ from another one. The first discussion on different types of introns can be found in [NB95]. A number of different taxonomies of introns have also been developed in [SH98, Ang98, LSPF99, SF98b, BT94, SH02]. In [Luk00b], Luke gives a more comprehensive bestiary of introns, based on the structure and the formation of the introns. Two broad categories of introns defined by Luke are *Inviable* and *Unoptimized* code. Inviable codes

are subtrees which cannot be replaced by anything “that can possibly change the individual’s operation”. Unoptimized code are subtrees which “perform no function in the individual, but can be replaced with subtrees which do perform a function”. Apart from certain specific intron types such as inviable code in certain problem domains, identifying other different types of introns remains to be a complex manual process which cannot be fully automated.

Although intron seems to be useless and redundant at phenotype level, intron can be beneficial to the overall evolution process. For example, in the context of genetic algorithm, in [Lev91], Levenick shows that introducing explicitly defined introns into bit strings can lead to dramatic improvements in success rate as much as a factor of 10. Similarly, Nordin et al. [NFB96] introduce explicitly defined introns in GP which improves fitness, generalization and algorithm efficiency. In [Ang94], Ageline classifies the emergence of introns is an “innate” advantage of GP. Ageline further describes the usefulness of introns as a protector of program semantics from crossover: if a crossover operator is performed within intron codes, the semantics of the offspring are preserved.

Ageline’s view is the first reference of the bloating theory *defense against crossover*. The origin of the theory can not be traced in literature, but this does not affect defense against crossover being the most widely cited bloating theory. Defense against crossover is based on the general perception that crossover is destructive. Empirical results of the destructiveness of crossover can be found in [NB95]. The offsprings created by a crossover operation may have better fitness, the same fitness or worse fitness compared to their parents. However, the probability of these three cases are not even. Nordin and Banzhaf show in [NB95] that there is a high probability that the function of the program is severely damaged, resulting in a fitness decrease for the individual. Given the destructive nature of crossover, if the crossover point is selected within a building blocks (exons), there is a high probability that this building block will be damaged, resulting decrease in offspring’s fitness. On the other hand, if the crossover interferes an intron, then there would be no harmful effects to the fitness of the offspring produced. As a result, individuals with more percentage of introns are more likely to survive within the population, since the selection of crossover point is more likely to fall into intron region. In another word, GP favors individuals with higher ratio of introns. In [JF97], Soule and Fostoer develop a constructive crossover in which only crossover results in better offspring fitness compared to parent is preserved. Their

experiments show that using only constructive crossover, a substantial amount of bloating can be reduced. More experimental evidence to support defense against crossover can be found in [BL02] and [SH02].

3.1.2 Defense against Crossover Variations

There are several variations of defense against crossover. In [BT94] and later in [Bli96], Blickle proposes *redundancy theory* which has a more formal mathematical model to explain bloating caused by destructiveness of crossover. In Blickle's mathematical model of GP, redundancy is defined as a subtree which does not contribute to the functionality of the tree. Using the mathematical model, he proves that the redundancy in the trees of a certain fitness value will increase over generation, because trees with more redundancy are more likely to survive from crossover. Blickle concludes that the bloating is driven by redundancy because for program trees with a given fitness value, increasing tree size is the only way to achieve increase in redundancy, since some minimal size is required for exons to allow the individual to reach that fitness level. One problem with Bickle's definition of redundancy is that, it is so generic that it is sometime hard or even impractical to compute the actual redundancy in experiments. In [BT94], Blickle gives a way to compute redundancy for boolean problems. A subtree is redundant if the subtree does not participate in evaluation. We will introduce this operation, *marking*, later in the next section.

In [MM95], McPhee and Miller suggest that there is a replication accuracy force in GP evolution which results in the development of a collection of semantically equivalent correct individuals. Given a fitness level, larger trees are more likely to produce structurally different but semantically equivalent offsprings in crossover, as the crossover is more likely to happen within semantically irrelevant regions. As a result, given a certain fitness level, larger individuals' offsprings are more likely to survive and this results in an increase in the average population depth and size.

Removal bias [SF98b] is another widely cited bloating theory developed on top of the defense against crossover. In the selection of crossover point, the subtree is more likely to be within intron region if a deeper crossover point is selected. As a result, due to the destructiveness of crossover, offspring created in crossover by removing a smaller subtree and attaching a bigger subtree (i.e. the growing offspring), is more likely to survive compared with offspring created by removing

a bigger subtree and receiving a smaller subtree (i.e. the shrinking offspring). In other words, the destructiveness of crossover results in GP system favoring the growing offspring. Empirical experiments to support removal bias theory can be found in [SH02, Sou98]. Later in this thesis, the removal bias is extended and studied in detail in Section 5.3.

Modification point depth [Luk00a, Luk03], also known as *depth-correlation* theory [Str03], is an abstraction over removal bias. Modification point depth theory proposes that there is a more general mechanism behind tree growth which is the bias towards the selection of deeper crossover point. As reported in [GR96] and [IC99], the existence of this bias is due to that, there is a strong inverse correlation between the depth of the crossover point and the crossover effect on fitness. The preference over deeper crossover point results in GP favoring larger tree and smaller subtree swapped in crossover, both of which result in bloating. Modification point depth theory has been further advanced by Streeter in [Str03]. Streeter argues that "the key idea behind both the intron theory and the depth correlation theory is that the receiving parents who produce offspring more similar to themselves will tend to be large, i.e. that it is large trees which will be most resilient in the face of crossover". Streeter further define the concept of resilience and show that the buildup of resilience is essential for code growth.

3.2 Anti-Defense against Crossover Evidences

There are also several anti-defense against crossover experiment evidences exist in the literature. The first evidence can be traced back to [Tac94]. In [Tac94], Tackett argues that defense against crossover implies that code growth is driven by selection pressure. But the brood recombination which is less destructive does not results in less bloating. He further suggests that bloating may relate to a phenomenon known as *hitchhiking*, a product of recombination interacting with selection. The phenomenon of hitchhiking is firstly reported in GA by Holland et al. when they tried to use *royal road function* to demonstrate the superiority of GAs over local search methods [MFH92]. *Building-block hypothesis* [Hol92], arguably the theoretical foundation of GA, states that GA works well when short, low-order, high-fit schemata (building blocks) recombine to form even more high-fit, high-order schemata. Based on *building-block hypothesis*, when an individual of high fitness is discovered, its high fitness allows the schemata to spread within

the population. But not only the schemata which contributes to fitness is spread out, a portion of non-functional code (introns) is also accompanied as hitchers. This hitchhiking effect slows down the discovery of schemata in other positions. Tackett adapts the hitchhiking theory from GA into GP and provide two experiment results to support the theory.

Another experiment evidence can be found in [BB03], in which, Brameier and Banzhaf analyze how different types of variations affect code growth in Linear GP. They define neutral variation as genetic operations which result in no change at the phenotype level, and destructive variation as genetic operations which result in worse fitness at phenotype level. They argue that experiments performed in [JF97], in which offsprings created by constructive crossover are preserved, even though result in less bloating, not necessarily imply that destructive variation is the main force of bloating. This is because the experiment also rejects neutral variation. Brameier and Banzhaf further develop a number of experiments to analyze how neutral and destructive variations affect code growth. Their experiment results show that the influence of non-neutral variations is considerably smaller than expected, and the neutral variations, in contrast to destructive variations, drive code growth. Brameier and Banzhaf's experiment result, on one hand confirms the emergence of introns which contributes to the code growth, but on the other hand, rejects the destructiveness of crossover as the driving factor behind the propagation of introns.

A further evidence is presented by Luke in [Luk00a, Luk00b], in which Luke uses marking operator to identify inviable code with a program tree and refuses crossover points selected in intron region. In this way, the inviable codes are removed and all crossovers can only happen at exon region. Luke's experiments using this version of crossover show that, the code growth still happens even when inviable code crossover is restricted, due to the propagation of unoptimized code or pseudo-inviable code.

3.3 Other Models

In addition to intron based models which consider the propagation of intron and the destructiveness of crossover as the root cause of bloating, *fitness causes bloat* [LP97b, LP98b, LSPF99, LP06], as the name suggests, proposes that fitness and

selection pressure drive code growth. This theory is also known as *solution distribution* [Sou98], *diffusion theory* [Luk00b, Str03], *drift theory* [BB03, SH02] and *nature of program search spaces theory* [Pol03] in literature. In [LP97b], Langdon and Poli study and compare artificial ant with and without fitness selection pressure. They theorize that the fitness function drives searching of optimal solution to converge to the searching of candidate solutions with the same fitness but different genotype representations. This is because in GP, finding improved solutions is relatively easy initially but becomes increasingly more difficult [Lan96b]. As a result, the selection gradually favors representations which have the same fitness as their ancestors. Since there are many more longer representations of a given fitness, as a result, the searching is drifted towards searching for larger trees which results in increase in tree size. One significance of fitness causes bloat theory is that, if it was fitness, the foundation of evolution, which causes code growth, bloating would become a building block of GP, although undesired, rather than an unforeseen side effect. This would also mean that we may never be able to remove bloating without harming the usefulness of GP.

Crossover bias is a relative recent bloating theory which states that "bloat is simply caused by the sampling of short, unfit programs" [PLD07, DP07, PMV08]. In [PLD07], Poli et al. show that crossover pushes the population towards a particular distribution of program sizes, called *Lagrange distribution of the second kind*, in which smaller programs are much more frequently sampled than larger ones. However, these short programs are generally unfit and less likely to survive. Hence, larger programs in population are more likely to be selected to breed more frequently.

3.4 Bloating Control Techniques

In addition to theoretical works discussed in previous section which try to explain why bloating happens, there are a lot more research works have been done to tackle bloating. In general, these bloating control methods modify different aspects of GP process to achieve less code growth at genotype level in the hope of not sacrificing GP performance in phenotype level. In this section, a survey of these works is presented. There are different taxonomies available in literatures [Zha97, LP06, da 08]. In this thesis, we feel that a more neutral taxonomy would be based on the nature of the modifications. As a result, the rest of this section

reviews various bloating control methods and groups them based on where the modification of GP lies in.

3.4.1 Modifications of Crossover

Since the most widely accepted bloating theory is defense against crossover, there is no doubt that crossover becomes the primary target for bloating control researches. A biggest share of literatures attempt to modify almost every aspect of crossover in order to control bloating. This includes restricting selection of crossover point, experimenting alternative subtree swapping techniques, introducing acceptance test over offsprings produced and so on.

The first and the most widely used bloating control method is to place a constraint on the maximum depth of the program tree used by Koza in [Koz92]. In Koza's depth limiting crossover, offsprings whose depth is bigger than 17 is rejected and the parent is put into the new population instead. Because of the simplicity of the method and the fact that it is the first bloating control method, this method has been programmed into every GP package and it is the most widely used bloating control method. The choice of 17 as the threshold is rather accidental without a formal justification, but it has now became the common practice in GP community. A variant of depth limiting method is *size limiting*, where the number of nodes in program tree is used as the limiting criterion rather than tree depth [KABK99, LSPF99, LP97b]. Despite the simplicity, depth limiting is surprisingly effective. Luke showed that a number of later developed control methods cannot outperform depth limiting by themselves [LP02a]. But a combination of these methods with depth limiting always outperforms plain depth limiting [LP06]. Depth limiting is also referred to as *capping* in [Sou98], because it essentially places an absolute upper limit to the program tree depth under evolution. In [SC04, SC05a], Silva develops *dynamic limits*, in which rather than using a fixed threshold on tree depth or size, the limit, either depth or size, can be dynamically raised or lowered based on the best solution found so far.

The biggest limitation of depth limiting method is that the method does not have any effect until the individuals' depth reaches the upper limit. In fact, in [DP08], Dignum and Poli point out that size limit can result in over-sampling of smaller size programs, which based on crossover bias theory, could speed up bloating rushing the population to reach the absolute upper limit. Once the absolute limit is reached, depth limiting method then tends to have a negative

effect to GP performance. This is because once the depth limit is reached, the offspring in crossover which receives bigger subtree and gives up smaller subtree, originally favored because of the destructiveness of crossover, is more likely to be rejected because of its depth and being replaced by its parent. The other offspring which receives smaller subtree and gives up bigger subtree, even though is less likely to be rejected by depth limit, but is also less likely to be favored in fitness selection. The combined effect results in a larger amount of parents being injected into the next generation, which may slow down the whole evolution process. Experiments performed in [GR96, LP97a] show that crossover in combination with depth limiting results in premature convergence in MAX problems.

Marking is a bloating control method introduced by Blickle and Thiele in [BT94] motivated by redundancy theory. Blickle defines redundancy as subtrees which do not contribute to the functionality of the individual. Based on redundancy theory, individuals within population tend to increase in size over generations to increase their redundancy such that they are more likely to survive from destructiveness of crossover. The idea behind marking is to avoid crossover at redundant edges by firstly identifying all redundant edges and then restrict crossover point selection to non-redundant edges. Even through the definition of redundancy is quite generic, the identification of redundant edge in practice is not that straightforward and can be quite computational expensive. In [BT94], Blickle gives an implementation of marking in binary problems by marking every node in a tree when the node is evaluated. After fitness evaluation, all nodes without marks are redundant. One thing to note is that Blickle's implementation of marking does not strictly follows his definition of redundancy, and marking is not able to identify all possible redundant subtrees. In fact, marking is only able to identify inviable code defined by Luke in [Luk00b]. In [Bli96], Blickle further advances marking by introducing *delete crossover*, which swaps subtrees which contains all redundant nodes with a single node to directly reduce redundancy. Positive results for marking and delete crossover have been reported in 6-multiplexer problem.

In addition to marking, there are a number of crossover modifications developed based on defense against crossover. The *hill-climbing crossover* [Sou98, SF98b], also called *pseudo-hillclimbing* [LP02a], rejects offsprings whose fitness is worse than their parents. A stricter version of hill-climbing crossover, the

"upward-mobility" selection [Alt94a], only accepts offsprings whose fitness is better than their parents. Another variation, the *improved fitness selection* [SH98], only accepts offsprings whose fitness is different from their parents, without the preference between better or worse fitness. The theoretical foundation of these methods can also be traced back to Brameier and Banzhaf's observations in [BB03], which show that natural variations drive code growth.

There are several other modifications of crossover motivated by *fitness causes bloating*. In [Lan00b], Langdon develops *size fair crossover*, in which, the choice of the crossover point in the second parent is guided by the size of subtree to be deleted from the first parent. Any crossover point selected from the second parent which results in subtree to be deleted from the second parent bigger than twice as big as the subtree to be deleted from the first parent will be rejected. Size fair crossover is motivated by the development of *size fair mutation* [Lan98, LSPF99], in which the size of random subtree generated depends on the size of the subtree to be replaced. Langdon also further extends the size fair crossover to create *homologous crossover*. Homologous crossover, in order to preserve the context in which the subtrees are swapped, in addition to have the restriction as defined in size fair crossover, requires the subtree deleted from the second parent to be as similar as possible to the subtree deleted from the first parent. This is motivated by the fact that the worth of the code not only depends on its quality, but also depends on the context in which the code stays.

In [PL97b], *one-point crossover* is developed which selects a "common" crossover point in both parents. In [HS97, SH98], Harries and Smith develop *same depths crossover* which firstly selects a depth at random from the the smaller parent, then it selects crossover points randomly from nodes appearing at that depth in each of the parents. The theoretical basis for these methods can be traced back to modification point depth theory. One-point crossover and same depths crossover both promote crossover points further away from leaf nodes to be selected and bigger subtrees to be swapped and thus reduces bloating, since according to modification point depth theory, deeper modification points in crossover promote bloating.

Finally, there are a number of relatively ad-hoc modifications. The *waiting room* introduced in [PL04] creates a waiting queue. Newly created individuals are placed into the queue rather than directly into population. The queue is sorted according to the size of individuals. The larger the individual, the longer

it remains in the queue. *Prune and Plant* [ACEAS+08], which is inspired by a strategy of the same name used in Agriculture, shows positive results improving best fitness achived across a number of problem domains.

3.4.2 Modifications of Fitness Function

In addition to the modifications of crossover, another natural approach to control bloating is to incorporate program size as part of the objectives in the fitness function. This approach is also called *parsimony pressure* or *parametric parsimony pressure* [LP06]. The general idea behind this approach is to penalize individuals with bigger size in fitness. Since GP is capable of evolving the population based on fitness function, the ambition of this approach is that, once the program size becomes part of the fitness function, there is then no need to explicitly control bloating and GP should be able to evolve individuals with better fitness and smaller size at the same time. Early references of using parsimony pressure can be traced back to [Koz92] and [KEK93], although the primary concern is not bloating. In [KEK93], Kinnear reports that "adding inverse size to the fitness measure along with correctness not only decreases the size of the resulting evolved algorithms, but also dramatically increases the effectiveness of the evolution process". However, Soule and Foster [SF98a] show that parsimony pressure can produce poorer performance and the effects of parsimony pressure on an evolving population is more than limiting the code growth.

In the simplest form, the *linear parametric parsimony pressure*, program size is used as a linear factor in fitness function. Formally, let i be an individual, $P(i)$ be the raw fitness function, and s_i be the size of individual i. Then the fitness function $f(i)$ is defined as:

$$f(i) = P(i) - \alpha s_i$$

where α is the parsimony coefficient. Other variations of parsimony pressure can be found in [IdGS94], [KM99] and [CC99]. Because it is applied at a fine granted quantity level rather than rank level, the parsimony pressure essentially establishes a quantitative trade off between fitness and size. However, accurately defining this trade off is hard if not entirely impossible, given the complexity of fitness landscape and program size distribution. In addition, this trade off expectation may change over evolution process. For example, in the early stage of GP when individuals are generally unfit and small, an individual which is fitter

and bigger may be preferred, while the same individual may not be preferred in later stage of GP. To address these problems, Zhang and Mühlenbein [ZM95] propose a dynamic approach which dynamically adapts the parsimony coefficient parameter based on the fitness and size of the best individual in the generation. Poli and McPhee [PM08] also develop a mathematical model to derive parsimony coefficient which allows user to accurately control the effect of parsimony pressure and even switch the bloating control off or on at different stages of evolution.

Tarpeian method [Pol03] is another variation of parsimony pressure method, in which before any individuals are evaluated for fitness, a number of individuals whose size is bigger than population's average size are randomly selected and marked to the lowest possible fitness value available. This effectively denies these individuals to be selected as parent even if they are fitter compared to other individuals. Experiments in [Pol03] show that in the presence of Tarpeian method, population tends to grow to "a much less extent", i.e. at least one order of magnitude smaller than population evolved without bloat control. However, in [LP06], Luke and Panait argue that Tarpeian method can be "overly aggressive" and tends to be sensitive to parameters.

Instead of building a single combined fitness function of both fitness and size, another approach is to consider size and fitness as two separated and independent objectives, changing the fitness-based selection process to be a multi-objective selection process. Pareto-based methods developed to control bloating using Pareto-dominance based multi-objective selection can be found in [EN01] (Non-domination Tournament), [dWP01] (FOCUS), [BBTZ01] (SPEA2) and [PL04] (Biased Multiobjective Parsimony). A more detailed review of these methods can be found in [LP06] and [BBZ].

More recently, in [TNM13], inspired by fitness causes bloat theory, Trujillo et al. remove the notion of fitness, and use the novelty instead. In novelty search, instead of using the quality as selection pressure, uniqueness is used. Searching is biased to individuals which introduces novelty into the search with respect to the rest of the population. Experiment results on several classification problems show that, novelty search is able to reduce the mean size of the evolved population.

3.4.3 Modifications of Selection

Modifications of selection have been considered as an alternative approach to apply parsimony pressure. These methods are also called *non-parametric parsimony pressure* methods [LP06]. In general, these methods modify standard GP selection methods (mostly tournament selection) to take into account parsimony pressure. In [LP02a], Luke and Panait develop *double tournament* and *proportional tournament*. Double tournament, as the name suggests, performs two tournaments, the qualifying tournament and the final tournament. Fitness is used in qualifying tournament as the criterion and program size is used in the final tournament or vice versa. In proportional tournament, some portion of tournaments selects individuals based on fitness and others are based on parsimony. In [LP06], a detailed comparison of performance of different bloating control methods is performed by Luke, and it shows that the performance of double tournament is very competitive.

In [LP02b], Luke and Panait propose *lexicographic parsimony pressure* which treats fitness as the the primary objective and tree size as a secondary objective in a lexicographic order. This makes selection favoring smaller individuals when the fitness of the individuals is the same. They further extend the idea by grouping individuals with similar fitness into a group and treat them as having the same fitness. Two grouping methods are developed called *direct bucketing* and *ratio bucketing*. Experiments of direct lexicographic parsimony pressure, direct bucketing and ratio bucketing can be found in [LP02b] and [LP06]. Later in this thesis, we also compare the newly proposed bloating control method, depth constraint crossover, with double tournament, direct lexicographic parsimony pressure, proportional tournament, ratio bucket and also Tarpeian method.

3.4.4 Modifications of GP Flow

In contrast to bloating control methods discussed above which modify a concrete GP operator or component, the following bloating control methods tend to modify standard GP in a more fundamental way by either introducing a new operator, component or global constraints.

The development of *explicitly defined introns* is mainly motivated by researches of explicitly inserted introns in GA [Lev91]. Explicitly defined intron

(EDI) is a non-functional instruction segment that is “intentionally” inserted between two building blocks, which act as an intron. In [NFB96, SH98], it is shown that by manually inserting EDIs into individuals in GP, the propagation of introns can be controlled and it also results in better GP performance in terms of evolution speed.

Code editing is proposed by Koza in [Koz92]. Code editing is originally designed as a secondary genetic operator [Koz92] which simplifies optimal program tree GP finds at the end of the evolution. However, it is capable of reducing the program tree size and hence reduce bloating. Code editing can be applied in different ways [da 08]. In [SFD96], positive performance result is reported. However, Haynes reports that editing (he refer as repairing) may leads to premature convergence [Hay98].

Automatically defined functions, also introduced by Koza [Koz94], is a modularization technique which reduces structural complexity of solutions. Automatically defined functions, although not primarily designed to control bloating, results in simplified code. This represents another way to control bloat (i.e. increasing the efficiency of GP process). Similar modularization techniques include *automatically defined macros* [Spe96] and so on. A complete list can be found in [da 08].

In [dVGPG04], a well-established method for parallel evolutionary computing, *island model*, is used to combat bloating. Theoretical analysis is developed to show that increasing the number of sub-populations results in smaller overall program size. Then empirical experiments are performed in [dVGPG04] using one well-known benchmark, even parity five.

Resource-limited GP, introduced by Silva in [SC05a, SC05b], uses a single limit imposed on the total amount of tree nodes that the entire population can use. This threshold can be regarded as resources that each individual within population competes in order to survive. Similar to depth limiting methods which place a cap on individual level, resource-limited GP also places a cap, but more generally at the population level. This allows GP to implicitly balance the trade-off of having better fitness and smaller size at individual program level.

In [SV09], [Sil11] and later in [SDV12], Silva et al. develop operator equalisation inspired by the crossover bias theory. The idea behind operator equalisation is to prevent offsprings which are either too small to be useful or bigger than

needed entering the new population. In operator equalisation, before any offspring is created, a pre-defined target distribution of tree size is firstly initialized. Then, after a new individual is created and before it is put into the new population, the new offspring is only accepted by the new population if its size fits into the target distribution or its fitness is better than any existing individual. Experiment results of operator equalisation shows plausible outcome in controlling bloating. But a side effect of operator equalisation is that much more number of additional evaluations (sometimes by an order of magnitude) is required to find offsprings full-filling the target distribution criteria [Har12]. To address this performance issue, an improvement over the operator equalisation can be found at [GGP11].

3.5 Conclusion

In this chapter, we give a relatively comprehensive review of GP researches related to bloating. As the primary obsolete which limits the GP's ability to scale up in order to solve more complex problems, bloating control has been under active research since the very beginning stage of GP. Although the truth behind bloating and the holy grail to combat bloating are still hidden from us, we have seen a substantial development in literature towards the ultimate solution.

Chapter 4

Theoretical Analysis of Bloating Effects

4.1 Introduction

In genetic programming (GP), bloating is an unforeseen side effect of using variable length representation such as trees. Bloating is a phenomenon that the size of individual program in population tends to grow as GP runs without improvement (or very little improvement) of the individual's fitness. Despite other problems, the most serious problem caused by bloating is that it slows down fitness evaluation. Since fitness evaluation uses most of the computation effort and time in GP, by slowing down fitness evaluation, bloating severely limits the feasibility of GP for more resource demanding complex problems. In the battle against bloating, quantitative measurements of tree node importance can be used as guidelines for modifications of GP operators. For example, in [SCZ09], a distance-based node contribution measurement using the minimum difference between the input and the output of the node as measurement is proposed. It is shown in [SCZ09] that a bloating strategy based on this measurement can significantly reduce bloating.

In this chapter, we develop an alternative mathematical model in the context of boolean GP problems, called *activation rate*, to understand and measure how effective and important each tree node is in fitness evaluation and what cost or contribution each node is making to the individual's overall performance. Activation rate is a development over a previous bloating control method, marking [BT94]. Activation rate uses the frequency that tree nodes are invoked (evaluated)

in fitness evaluation as a quantitative measurement of the node's importance. Using this new model, it is possible to perform analysis of GP dynamics from a new perspective. For example, in Section 4.6, we find using estimated activation rate that apart from extreme tree shapes, for most of the other tree shapes, there exists an upper bound for the negative effects of bloating slowing down evaluation. In Section 4.7, we give a more thorough study of crossover effects and also give an anti-modification point depth theory experiment evidence.

The rest of this chapter is organized as follows. In the next section, we introduce a new tree evaluation algorithm called bottom-up tree evaluation. Bottom-up tree evaluation is developed to explicitly address one limitation of top-down tree evaluation. The concept of activation rate is developed as part of the theoretical analysis of bottom-up tree evaluation algorithm. This work has previously been published in [LZ10a]. Then, in Section 4.3 and 4.4, we further extend the concept of activation rate by developing a method to estimate the activation rate. Experiment performed has shown that the estimation of activation rate is very accurate. The advantage of estimated activation rate is that, with it, it is possible to perform a number of theoretical analysis of GP. We conclude this chapter with several experiments of GP using activation rate in Section 4.6 and 4.7 which give a number of interesting observations.

4.2 Bottom-up Tree Evaluation

4.2.1 Motivation

The most general form of fitness function contains a number of test cases. Each individual program in the population needs to be evaluated with inputs from each test case. Then the actual output is compared with expected output defined in the fitness function. The raw fitness of an individual usually is defined as the distance between the actual output and the desired output. Standard tree evaluation evaluates a program tree top-down recursively starting from the root node (see Algorithm 4). The sequence that nodes evaluate is the same as a depth-first traverse of the program tree.

In some GP problems, for example multiplexer problem and parity problem, the test cases are permutation of input values. For example, in multiplexer 11 problem, there are 11 inputs (3 address bits and 8 data bits). Since each input is

Algorithm 4 Pseudo Code for Top-down Evaluation of Program Tree

```
1: function top-down-evaluate(TreeNode node) : evaluatedResults
2: if node is Terminal then
3:    evaluatedResults ← value-of(node)
4:    return output
5: else
6:    for input ∈ node's input do
7:       top-down-evaluate(input)
8:    end for
9:    evaluatedResults ← evaluate(node)
10:   return evaluatedResults
11: end if
12: end function
```

binary (can either be 0 or 1), the permutation of inputs has $2^{11} = 2048$ different cases. Although every value in this permutation is distinct, if we iterate through this permutation, we can find that not *every* input changes every time. For

```
Inputs:  A0 A1 A2 D0 D1 D2 D3 D4 D5 D6 D7
Values:  0  0  0  0  0  0  0  0  0  0  0   Case 1
         0  0  0  0  0  0  0  0  0  0  1   Case 2
         0  0  0  0  0  0  0  0  0  1  0   Case 3
         0  0  0  0  0  0  0  0  0  1  1   Case 4
         0  0  0  0  0  0  0  0  1  0  0   Case 5
         .  .  .  .  .  .  .  .  .  .  .
         .  .  .  .  .  .  .  .  .  .  .
         .  .  .  .  .  .  .  .  .  .  .
         1  1  1  1  1  1  1  1  1  0  0   Case 2045
         1  1  1  1  1  1  1  1  1  0  1   Case 2046
         1  1  1  1  1  1  1  1  1  1  0   Case 2047
         1  1  1  1  1  1  1  1  1  1  1   Case 2048
```

Figure 4.1: Viewing test case inputs as permutation in Multiplexer 11 problem

example, for the permutation in Figure 4.1, comparing Case 1 to Case 2, only input D7 is changed. Comparing Case 2 to Case 3 only D6 and D7 changes. If we let $C(x)$ be the number of changes in a permutation for an input x, then for a given input x whose index is n (index starts 1 from left to right) and the total number of inputs is N, then:

$$C(x) = \frac{2^N}{2^{N-n}} = 2^n \tag{4.1}$$

For example, for the permutation in Figure 4.1:

$$C(D7) = \frac{2048}{2^0} = 2048, C(A0) = \frac{2048}{2^{10}} = 2.$$

Given the above fact, now let's consider evaluating the following program tree using the above permutation as test case inputs:

Not(And(Or(A0, A1), D7))

And's left subtree Or(A0, A1)'s value changes at maximum 4 times ($C(A1) = 4$), while the right subtree D7's value changes 2048 times. Using the classical top-down evaluation algorithm, the left subtree evaluates 2048 times while most of the evaluations yield the same result. In fact, since the left subtree only changes at maximum 4 times, 4 evaluations are enough. Thus, for this subtree, $2048-4 = 2044$ evaluations are wasted. In Multiplexer 11 problem, this may not be a very big issue since functions like And, Or, Not, If are very fast to execute. But for problems whose function set consists of very complex functions, the top-down evaluation algorithm is not very efficient in the above scenario.

The above scenario gives an example when top-down evaluation fails to perform well. Going back to the example, if the node And has a single cache which stores the previous result of left subtree, then when And evaluates, it would be able to take the new value of D7 and then compute the output using the value stored in cache. In this way, then the left subtree of And does not need to be re-evaluated when both A0 and A1 are not changed, i.e. the above problem of top-down evaluation is solved.

But how does node And know if A0 or A1 is changed or not without checking them explicitly? The bottom-up tree evaluation solves this problem. The bottom-up evaluation can be summarized as the pseudo code in Algorithm 5. In bottom-up tree evaluation, each node in the tree has a single cache which stores the previous result of this node. The value in the cache can be accessed by other nodes. There is also a unique queue needed for the evaluation process. The tree evaluation has three phases. In the preparation phase, each node within the tree is labeled with an index based on the depth-first traverse order. Then in the first phase (line 3 to line 10 in Algorithm 5), every terminal node is checked if the new value equals previous value stored in its cache or not. If the value has been changed, then the cache is updated and the parent of the node is enqueued into the central queue. In the second phase (line 12 to line 19 in Algorithm 5),

every node stored in the central queue is dequeued (line 14 in Algorithm 5). The node is re-evaluated and if the output of the node changes, the node's parent is enqueued. This process repeats until the queue is empty. Then the output of this evaluation is stored in the root node's cache (line 19 in Algorithm 5).

Algorithm 5 Pseudo Code for Bottom-up Evaluation of Program Tree

```
 1: function Bottom-up-evaluate(TreeNode root) : evaluatedResults
 2: label each node in the tree using depth-first index
 3: {Phase 1}
 4: init(queue)
 5: for terminal ∈ terminals of tree whose root is root do
 6:    if terminal's value changes then
 7:       update terminal's local cache
 8:       queue.enqueue(terminal.parent)
 9:    end if
10: end for
11:
12: {Phase 2}
13: while queue.count > 0 do
14:    node ← queue.dequeue()
15:    if node's value changes then
16:       update node's local cache
17:       queue.enqueue(node.parent)
18:    end if
19: end while
20: evaluatedResults ← root.cache.value
21: return evaluatedResults
22: end function
```

The queue initialised at line 4 in Algorithm 5 has the following 2 features:

1. each node can only be added once, i.e. each element of the queue is unique;
2. the order of the queue is the reverse order of the index labeled in preparation phase.

The first feature ensures each node in the tree only evaluates once. In another word, in the worst case scenario of bottom-up tree evaluation, all nodes in the tree will be evaluated once. The second feature of the queue ensures that a parent node in queue would only be evaluated after all its children nodes are evaluated. This is because nodes in a tree is labeled based on the depth-first traverse order. Please note that, labeling nodes in a tree based on the breath-first traverse order

also full-fills the requirement of the queue. But, when since depth-first traversal can be implemented recursively while breath-first traversal requires more memory, in the implementation of bottom-up tree evaluation, depth-first tree traversal is preferred.

Intuitively, bottom-up tree evaluation outperforms top-down evaluation because in bottom-up evaluation, only "necessary" evaluations are performed, redundant unnecessary evaluations in top-down evaluations which yield the same output are eliminated. The percentage of saving depends on the particular permutation of test cases. This is because for a given tree, the performance of bottom-up evaluation varies for different permutation of test cases. In the rest of this section, we perform detailed analysis of the performance of bottom-up evaluation algorithm.

4.2.2 Theoretical Performance Analysis

The bottom-up evaluation adds local caches to each node and a central queue into the tree evaluation process in order to eliminate wasted function evaluations in top-down evaluation. It is assumed that function node evaluation is complex and the overheads introduced by queue operations can be neglected compared with savings. Based on this assumption, the performance of both top-down and bottom-up evaluation are related to the number of function calls on non-terminal nodes in the tree. Using the number of function calls as the comparison criterion, the performance of both algorithms can be theoretically analyzed.

Assuming each function's arity is two (each function has two inputs) and there are N different inputs (terminals). Considering a full tree T of depth d, the total number of nodes in T is:

$$Num(T) = 2^d - 1$$

Since in a full tree, all nodes of depth smaller than d are function nodes, the number of non-terminal nodes is:

$$NonNum(T) = 2^{d-1} - 1$$

Let the total number of test cases be $N_{testcases}$, so the number of function calls for top-down tree evaluation (i.e. the performance of top-down evaluation) is:

$$P_{Topdown}(T) = (2^{d-1} - 1) \cdot N_{testcases} \tag{4.2}$$

(4.2) only applies to some problem domains. This is because (4.2) assumes that the tree is traversed (every node is evaluated) while using top-down evaluation. This is true for problem domains such as symbolic regression. In symbolic regression problem, all functions in function set are mathematical operators. In this case, all children nodes of the function node need to be evaluated.

But in problem domains such as multiplexer or parity, the tree is not traversed in top-down evaluation. In multiplexer domain for example, there are four functions: And, Or, If and Not. For function Not, the child node always evaluates. For function If, only two children nodes evaluate every time (the condition node and the according action node). And and Or function can be implemented in two different ways: bitwise or short-circuit. In bitwise implementation, both children nodes of And and Or function evaluate. In short-circuit implementation of And, the first child of And always evaluates, the second child evaluates only when the first child evaluates true. Similarly, in short-circuit Or, the second child evaluates only when the first child is false. Because this nature of multiplexer function set, when tree is top-down evaluated, only a fraction of nodes are actually evaluated, and hence (4.2) cannot be applied directly to problem domains such as multiplexer.

To calculate the number of function calls in domains like multiplexer, an explicit measurement to the above feature of the function set is required. This feature can be studied using the concept of activation rate, which is an extension of marking. Marking is a bloating control method which explicitly model this phenomenon. Marking [BT94] is developed by Blickle and Thiele inspired by their redundancy theory. The idea behind marking is to avoid crossover point selected at redundant edges. Although marking is not primarily designed for controlling bloating [Bli96], it provides a first algorithm to identify introns [LP06], more precisely inviable code defined by Luke in [Luk00b]. Since intron theories which blame the propagation of intron regions as the cause of bloating are the most widely cited bloating theory, this makes marking more valuable. For example, in [LP06], Luke uses marking to experiment, and presents a first experiment evidence against intron theory. In [Jac05], Jackson modifies fitness evaluation with marking information to speed up the execution of GP.

The concept of activation rate expands marking. In marking, nodes which are invoked are marked with a marking flag. Nodes which are not invoked then don't have the marking flag. Using this scheme, we can only differentiate between nodes

which have been invoked or not, and there is no difference between nodes which have been invoked once or 10 times. In another word, in marking, the scheme is binary (either flagged or not). In activation rate, rather than a binary flag as in marking, for each node, we place a counter counting the number of times the node has been invoked (we define this value as activation rate). Using this scheme, similar to marking, nodes which have never been invoked have activation rate equals to zero. Tree nodes which have been invoked have activation rate bigger than zero. The difference between activation rate and marking is that, nodes which have been invoked are treated differently. Formally, we define the activation of a node as:

Definition 1 (Activation) *Let x be a node in a tree T, in Top-down evaluation, given a set of test cases Tcs, the* Activation *of x, denoted as* $\Lambda_{Tcs}(x)$*, is total number of times the node x is evaluated.*

Using the definition of activation, we can further define the *Rate of Activation*:

Definition 2 (Activation Rate of Node) *Let x be a node in a tree T, Tcs be a test cases set which contains N test cases, the* Activation Rate *of node x,* $\Theta_{Tcs}(x)$ *is:*

$$\Theta_{Tcs}(x) = \frac{\Lambda_{Tcs}(x)}{N}$$

With both definitions above, we can define the *Activation Rate of Tree*:

Definition 3 (Activation Rate of Tree) *Let x be a node in a tree T, N(T) be the number of nodes in T, Tcs be a test cases set which contains n test cases, the* Activation Rate *of tree T,* $\Theta_{Tcs}(T)$ *is:*

$$\Theta_{Tcs}(T) = \frac{\sum_{x}^{x \in T} \Theta_{Tcs}(x)}{N(T)}$$

Using the concept of activation rate, the true function call counts of top-down evaluation in any domain C_{True} and the function call counts using top-down traverse evaluation $C_{Traverse}$ has the following relationship:

$$C_{True} = \Theta \cdot C_{Traverse} \tag{4.3}$$

where Θ is the average activation rate of all non-leaf nodes in a program tree. Clearly, we can see that (4.2) implicitly assumes that $\Theta = 1$. In problem domains like symbolic regression, program tree's activation rate $\Theta \equiv 1$, while in boolean problem domains such as multiplexer, $\Theta \leq 1$. So, for a full tree T, (4.2) can be generalized using (4.3):

$$\begin{aligned} P_{Topdown}(T) &= C_{True} \\ &= \Theta \cdot C_{Traverse} \\ &= \Theta \cdot P_{Traverse} \\ &= \Theta \cdot (2^{d-1} - 1) \cdot N_{testcases} \end{aligned} \tag{4.4}$$

In Bottom-up evaluation, let X(T) be the set of terminals in a tree T, then:

$$P_{Bottomup}(T) = \sum_{x}^{x \in X} C(x) E(X) \tag{4.5}$$

Where C(x) is defined in (4.1), and E(X) is the average number of nodes affected when some $x \in X$ changes. The exact form of E(X) is very hard to deduce because it is related to not only the specific terminal in X, but also how those terminals are connected in the tree. Without further assuming the tree structure, which leads to the loss of generality of analysis, it is impossible to estimate E(X). Similarly, since the value of Θ in (4.4) also depends on the shape of the tree, without further assuming the tree structure, it is impossible to estimate $P_{topdown}$ for problem domain like multiplexer.

But we can perform worst case scenario analysis for problem domains whose tree's activation rate $\Theta \equiv 1$. In the worst case scenario, every terminal x in X has index $n = N$. In this case, the cache has no effect at all because every input changes every time, i.e.:

$$E(X) = 2^{d-1} - 1 \tag{4.6}$$

Substituting C(x) and E(X) in (4.5) with (4.1) and (4.6), and also given the fact that the queue used in bottom-up tree valuation only allows unique nodes, we get:

$$P_{Bottomup}(T) = 2^n \cdot (2^{d-1} - 1)$$

Since $n = N$ and $2^n = N_{testcases}$, so we get:

$$P_{Bottomup}(T) = N_{testcases} \cdot (2^{d-1} - 1)$$

Which is the same as $P_{Topdown}(T)$. So bottom-up evaluation performs the same as top-down evaluation in the worst case for problem domains in which $\Theta \equiv 1$, i.e.

$$P_{Bottomup}(T) \leq P_{Topdown}(T).$$

Now considering the probability that worst case scenario happens. For a full tree T of depth d, the probability p:

$$p = (\frac{1}{N})^{2^{d-1}},$$

which is very small.

4.2.3 Experiments

Theoretical analysis in previous section qualitatively shows that bottom-up evaluation is better than top-down evaluation for problem domains in which $\Theta \equiv 1$. A quantitative analysis of how much better the bottom-up evaluation is however cannot be theoretically deducted. In addition, the performance of bottom-up evaluation for domains in which $\Theta \leq 1$ cannot be theoretically analyzed without further assuming the structure of the tree evaluated. To fill in both gaps, this section uses experiments in multiplexer 11 problem domain to quantitatively compare performance of the bottom-up evaluation and the top-down evaluation.

In the first experiment, we compare performance of bottom-up and top-down evaluation for problem domain in which $\Theta \equiv 1$. Although in Multiplexer 11 domain, the average activation rate of non-leaf nodes are generally smaller than 1, we can artificially converts it to 1. This is done by converting short-circuit And and Or to bitwise And and Or. For If, we evaluate both the true subtree and the false subtree before returning the output of the former. Using this approach, when a program tree is evaluated, every node is traversed. We call this traversing evaluation top-down full evaluation.

In this experiment, we compare top-down full evaluation and bottom-up evaluation. The experiment is designed as follows. 7000 trees are randomly generated using ramp-half-and-half method [Koz92]. The depth of generated tree ranges from 2 to 13. Duplicated trees are removed. We evaluate it twice firstly using top-down full method and then bottom-up approach. Then we compare the number of non-leaf node calls using each method. The experiment result can be found in Table 4.1. From the experiment data, we can find that a substantial amount

Depth	Size	Top-down Full	Bottom-up	Performance Enhance (%)
2	3.4571	2048	373	81.79
3	6.9253	5960	1062	82.18
4	13.7057	12935	2643	79.57
5	26.9746	26545	5748	78.35
6	53.4802	53880	12127	77.49
7	103.3013	104852	24029	77.08
8	195.7506	199130	47073	76.36
9	373.1875	380656	88593	76.73
10	258.6831	263774	65297	75.25
11	441.6177	450245	110474	75.46
12	683.4360	698935	172932	75.26
13	1102.4485	1127861	278861	75.28

Table 4.1: Top-down Full and Bottom-up Evaluation Performance (7000-trees)

of function calls ($\geq 75\%$) can be saved using bottom-up tree evaluation. This experiment result confirms the theoretical conclusion we deducted in the previous section: bottom-up tree evaluation outperforms top-down evaluation for problem domains in which $\Theta \equiv 1$.

In the second experiment, we compare performance of the bottom-up and top-down evaluation for domain in which $\Theta \leq 1$. The experiment is designed as follows. Using the same 7000 trees generated in the first experiment, we evaluate each tree twice firstly using top-down and then bottom-up evaluation algorithm. We also record the average activation rate of non-leaf nodes in top-down evaluation. The experiment result is summarized in table 4.2.

From experiment data in Table 4.2, we can find that bottom-up evaluation outperforms top-down evaluation when tree depth is small. But as the depth of the tree increases, the performance improvement becomes smaller. For very deep trees (depth bigger than 8 in Table 4.2), top-down tree evaluation outperforms bottom-up evaluation. This is mainly because as the depth of the tree increases, the average activation rate of non-leaf nodes (Θ) decreases.

If we use top-down full evaluation as the baseline, let P1 be the performance improvement of bottom-up evaluation compared with top-down full evaluation and P2 be the performance enhancement of top-down evaluation compared with top-down full evaluation. From the first experiment, we know that in multiplexer

Depth	Size	Top-down	Bottom-up	Performance Enhance (%)	Θ
2	3.4571	2048	373	81.79	1.0000
3	6.9253	4964	1062	78.61	0.8482
4	13.7057	8908	2643	70.33	0.7249
5	26.9746	14797	5748	61.15	0.6014
6	53.4802	23583	12127	48.58	0.4832
7	103.3013	35437	24029	32.19	0.3834
8	195.7506	51861	47073	9.23	0.3063
9	373.1875	77173	88593	-14.80	0.2341
10	258.6831	48792	65297	-33.83	0.2214
11	441.6177	62075	110474	-77.97	0.1633
12	683.4360	75248	172932	-129.82	0.1248
13	1102.4485	95206	278861	-192.90	0.0979

Table 4.2: Top-down and Bottom-up Evaluation Performance (7000-trees)

11 domain, $P1 \approx 0.75$. Since from (4.4), we know that:

$$P_{Topdown} = \Theta \cdot P_{TopdownFull}$$

So:

$$\begin{aligned} P2 &= \frac{P_{TopdownFull} - P_{Topdown}}{P_{TopdownFull}} \\ &= 1 - \Theta \end{aligned}$$

When the tree depth is small (smaller than 8 in table 4.2), $\Theta > 0.25$. So, $P1 > P2$. When the tree depth is bigger than 8 in table 4.2, the $\Theta < 0.25$, then $P1 < P2$. So, in general, the performance of bottom-up evaluation compared with top-down evaluation P:

$$\begin{aligned} P &= \frac{P_{Topdown} - P_{Bottomup}}{P_{Topdown}} \\ &= \frac{(P_{TopdownFull} - P_{Bottomup}) - (P_{TopdownFull} - P_{Topdown})}{\Theta \cdot P_{TopDownFull}} \\ &= \frac{1}{\Theta} \cdot \left(\frac{P_{TopdownFull} - P_{Bottomup}}{P_{TopdownFull}} - \frac{P_{TopdownFull} - P_{Topdown}}{P_{TopdownFull}}\right) \\ &= \frac{1}{\Theta} \cdot (P1 - P2) \\ &= \frac{P1 - 1 + \Theta}{\Theta} \end{aligned}$$

Because from the experiment, Θ inverse correlates to the depth of the tree, bottom-up tree evaluation outperforms the top-down evaluation algorithm when

the depth of the tree is small. This limitation may seem to be very strict for normal GP setup. However, with effective bloating control, which is able to significantly reduce the average depth of the population, we believe the bottom-up tree evaluation's overall performance would be better than top-down evaluation.

4.3 Estimating Activation Rate

In the last section, we developed the concept of activation rate to investigate the feasibility of the bottom-up tree evaluation algorithm. The activation rate of a tree node represents the frequency the node is evaluated for different test cases. Computer programs respond to different inputs with different flows of execution. In tree representation, those flows of execution are represented as different orders of node invocation. So the phenomenon that tree nodes' activation rates are different from each other reflect this nature of computer programs.

The activation rate can also be theoretically estimated. This is achieved by analyzing the function set of the problem domain. For example, if we consider the function set of multiplexer problem domain, there are four functions: And, Or, If and Not. As we mentioned earlier, for function Not, the child node of the function always evaluates. For function if, only two children nodes evaluate every time (the condition node and the according action node). Function And and Or can be implemented in short-handed manner. In short-handed And, the first child of And always evaluates, the second child only evaluates when the first child evaluation returns true. Similarly, in short-handed Or, the second child only evaluates when the first child evaluates false.

If And is a parent node and its activation rate $\Theta_{\mathbf{parent}}$ is 1, let's then consider the activation rate of And's child: intuitively, there is a 50% chance that the child is the first (left) child. In that case, the activation rate Θ of the child is also 1. There is also another 50% chance that the child is the second (right) child. In that case, the activation rate Θ is 0.5, because there is 50% chance that the first child evaluates true. So, the child of And has the activation rate:

$$\Theta_{\mathbf{child}}^{\mathbf{And}} = 0.5 \cdot 1 + 0.5 \cdot 0.5 = 0.5 + 0.25 = 0.75$$

More formally, let A be the event when the child node is the left child and $P(A) = 0.5$, B be the event when the left node is evaluates true and $P(B) = 0.5$.

Then:

$$\begin{aligned}
\Theta^{\textbf{And}}_{\textbf{child}} &= \Theta_{\textbf{parent}} \cdot P(A \cup (B \cap A^{\mathsf{c}})) \\
&= \Theta_{\textbf{parent}} \cdot (P(A) + P(B \cap A^{\mathsf{c}}) - P(A \cap B \cap A^{\mathsf{c}})) \\
&= 1 \cdot (0.5 + 0.5 \cdot 0.5 - 0) \\
&= 0.75
\end{aligned}$$

Similarly, using the same definition of event A and B:

$$\begin{aligned}
\Theta^{\textbf{Or}}_{\textbf{child}} &= \Theta_{\textbf{parent}} \cdot P(B \cup (A^{\mathsf{c}} \cap B^{\mathsf{c}})) \\
&= \Theta_{\textbf{parent}} \cdot (P(B) + P(A^{\mathsf{c}} \cap B^{\mathsf{c}})) - P(B \cap A^{\mathsf{c}} \cap B^{\mathsf{c}})) \\
&= \Theta_{\textbf{parent}} \cdot (P(B) + P((A \cup B)^{\mathsf{c}}) - 0) \\
&= \Theta_{\textbf{parent}} \cdot (P(B) + 1 - P(A \cup B)) \\
&= \Theta_{\textbf{parent}} \cdot (P(B) + 1 - P(A) - P(B) + P(A \cap B)) \\
&= 1 \cdot (0.5 + 1 - 0.5 - 0.5 + 0.5 \cdot 0.5) \\
&= 0.75
\end{aligned}$$

In the case of Not, because it has only one child, so:

$$\Theta^{\textbf{Not}}_{\textbf{child}} = 1$$

In the more complicated case of If, which has three child nodes: the condition branch, the true branch, the false branch, let C be the event when the child node is at the condition branch, CT be the event when the condition branch evaluates true, T be the event when the child node is at the true branch, and F be the event when the child node is at the false branch. Given this definition we have:

$$\begin{aligned}
P(C) &= \frac{1}{3} \\
P(CT) &= 0.5 \\
P(T) &= \frac{1}{3} \\
P(F) &= \frac{1}{3}
\end{aligned}$$

So:

$$\begin{aligned}
\Theta^{\textbf{If}}_{\textbf{child}} &= \Theta_{\textbf{parent}} \cdot P(C \cup (CT \cap T) \cup (CT^{\mathsf{c}} \cap F)) \\
&= \Theta_{\textbf{parent}} \cdot (P(C) + P(CT \cap T) - P(C \cap CT \cap T) + P(CT^{\mathsf{c}} \cap F) \\
&\quad - P((C \cap CT^{\mathsf{c}} \cap F) \cup (CT \cap T \cap CT^{\mathsf{c}} \cap F)))
\end{aligned}$$

Because $C \cap T = \varnothing$ and $C \cap F = \varnothing$, so:

$$\begin{aligned}
\Theta^{\mathbf{If}}_{\mathbf{child}} &= \Theta_{\mathbf{parent}} \cdot (P(C) + P(CT \cap T) + P(CT^{c} \cap F) \\
&= 1 \cdot (\frac{1}{3} + 0.5 \cdot \frac{1}{3} + (1 - 0.5) \cdot \frac{1}{3}) \\
&= \frac{2}{3}
\end{aligned}$$

Summarizing all four cases, let's consider a child node in Multiplexer domain.

$$\begin{aligned}
\Theta^{\mathbf{Multiplexer}}_{\mathbf{child}} &= \frac{1}{4} \cdot \Theta^{\mathbf{And}}_{\mathbf{child}} + \frac{1}{4} \cdot \Theta^{\mathbf{Or}}_{\mathbf{child}} + \frac{3}{8} \cdot \Theta^{\mathbf{If}}_{\mathbf{child}} + \frac{1}{8} \cdot \Theta^{\mathbf{Not}}_{\mathbf{child}} \\
&= \frac{1}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{3}{4} + \frac{3}{8} \cdot \frac{2}{3} + \frac{1}{8} \cdot 1 \\
&= 0.75
\end{aligned}$$

In the above calculation, If function has higher weight while Not function has lower weight. This is because If function has three children nodes, while Not function has only one child node. So, when If and Not function appears with the same probability, a child node is more likely to be a child of If function compared to Not function.

Similarly, we can perform the same calculation for parity problem, whose function set contains And, Nand, Or and Nor:

$$\begin{aligned}
\Theta^{\mathbf{Parity}}_{\mathbf{child}} &= \frac{1}{4} \cdot \Theta^{\mathbf{And}}_{\mathbf{child}} + \frac{1}{4} \cdot \Theta^{\mathbf{Nand}}_{\mathbf{child}} + \frac{1}{4} \cdot \Theta^{\mathbf{Or}}_{\mathbf{child}} + \frac{1}{4} \cdot \Theta^{\mathbf{Nor}}_{\mathbf{child}} \\
&= \frac{1}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{3}{4} \\
&= \frac{3}{4} = 0.75
\end{aligned}$$

In the case of parity problem, all functions' arity is 2. So, each function has a weight of 0.25.

This value $\Theta_{\mathbf{child}}$ represents how much the activation rate changes from a parent node to its child node. We formally define this value as *descent rate*:

Definition 4 (Descent Rate of Activation Rate) Descent Rate of Activation Rate $\bigtriangledown$ *of a problem domain P is the average change of activation rate from a parent node to a child node in program trees from domain P.*

From the definition, we can see that the value of descent rate only depends on the function set of the problem domain. It is independent to any program trees generated. Using descent rate, we can estimate the activation rate for nodes of

certain depth. Let x be a node in tree T whose depth is d, so x's estimated activation rate is:

$$\Theta_{est}(d) = \bigtriangledown^{d-1} \tag{4.7}$$

According to (4.7), we know that nodes deep down in the tree have smaller activation rates. Because as we discussed before, activation rate represents the maximum number of test cases may be affected if a change is made on that node, as a result, if a change happens at a node far away from the root, less number of test cases will be affected. On the other hand, if the change is made at a node near the root, more number of test cases are likely to be affected. As a result, activation rate models the relative importance of nodes within a tree. A similar concept to activation rate in literature is "block activation" briefly mentioned in [Ros95]. The block activation is defined as the number of times the root node of the block is executed. It is used for the selection of the useful block of code within the parents. Therefore, the block activation is a concept different from the activation rate and it serves for the different purpose. Comparing the activation rate and the distance-based contribution measurement proposed in [SCZ09], the predominate advantage of activation rate is that activation rate can be theoretically estimated. The estimation method makes activation rate can be used for theoretical analysis of GP dynamics.

4.4 Experimenting with Activation Rate Estimation

Definition 2 not only defines the activation rate, but also gives a practical algorithm to calculate rate of activation for each node in trees in GP population. (4.7) gives another method to estimate the activation rate. In this section, we study through experiments how accurate this estimation is.

The experiment is designed as follows. We randomly generate program trees in multiplexer 11 domain using ramp-half-and-half method. For each tree, we calculate each node's activation rate. Then, we group these trees generated by tree depth. For each group, we calculate the average activation rate for tree nodes' at each depth. This data is then compared with the theoretical estimation calculated using (4.7) also shown in Table 4.3.

Three separated experiments have been performed. In each experiment, the

Depth	Estimated Activation Rate	Estimated Activation
2	0.75	1536
3	0.5625	1152
4	0.421875	864
5	0.31640625	648
6	0.237304688	486
7	0.177978516	364
8	0.133483887	273
9	0.100112915	205
10	0.075084686	154
11	0.056313515	115
12	0.042235136	86
13	0.031676352	65

Table 4.3: Theoretical Estimation of Activation Rate in Multiplexer11

depth of tree generated is randomly chosen from 2 to 13. In the first experiment, we test 1500 trees generated. In the second experiment, 5000 trees are examined. In the third experiment, 15000 trees are tested. In each experiment, duplicated trees are removed before calculating the average activation rate. The experiment results are summarized in column 1500-Trees, 5000-Trees and 15000-Trees in Table 4.4. Since the searching space of all possible trees up to depth 13 is

Depth	1500-Trees	5000-Trees	15000-Trees	Diff-1	Diff-2	Weighted Avg
2	0.7405	0.7419	0.7386	0.26%	0.45%	0.7395
3	0.5594	0.5581	0.5589	0.09%	-0.13%	0.5587
4	0.4209	0.4184	0.4198	0.27%	-0.33%	0.4195
5	0.3164	0.3138	0.3163	0.05%	-0.77%	0.3157
6	0.2383	0.2358	0.2368	0.64%	-0.38%	0.2366
7	0.1792	0.1760	0.1782	0.53%	-1.24%	0.1778
8	0.1335	0.1315	0.1339	-0.33%	-1.78%	0.1333
9	0.0997	0.0988	0.1008	-1.07%	-1.96%	0.1002
10	0.0755	0.0737	0.0754	0.18%	-2.20%	0.0750
11	0.0578	0.0548	0.0567	1.97%	-3.26%	0.0563
12	0.0455	0.0398	0.0420	7.99%	-5.38%	0.0418
13	0.0345	0.0309	0.0314	9.86%	-1.60%	0.0315

Table 4.4: Observed Activation Rates in 1500, 5000, 15000 Trees Experiments

almost infinite, the experiment data are only samples of the searching space. As a result, we need to show that the average observed activation rate we collected is statistical representative before comparing it with estimated activation rate.

In Table 4.4, column Diff-1 shows percentage differences between 1500-trees and 15000-trees. Column Diff-2 shows percentage differences between 5000-trees and 15000-trees. We can see from table that these three data sets collected have very similar data. The percentage differences are below 2% in most cases. This shows that the average activate rate for nodes at certain depth is stable even when the sample size is small. In depth 12 and 13, the percentage differences are relatively big. This is because there are relatively very small number of trees have nodes at depth 12 and 13. We use the weighted average from depth 2 to 11 in table 4.4 to compare with estimated data in table 4.3.

Depth	Observed	Estimated	Diff	Diff%
2	0.739488739	0.75	0.010511261	1.40%
3	0.558744568	0.5625	0.003755432	0.67%
4	0.419514166	0.421875	0.002360834	0.56%
5	0.315700922	0.31640625	0.000705328	0.22%
6	0.236648017	0.237304688	0.00065667	0.28%
7	0.177774732	0.177978516	0.000203784	0.11%
8	0.133331426	0.133483887	0.000152461	0.11%
9	0.100218643	0.100112915	-0.000105728	-0.11%
10	0.075019345	0.075084686	6.53413E-05	0.09%
11	0.056308822	0.056313515	4.69221E-06	0.01%

Table 4.5: Comparison between Observed and Estimated Activation Rate

Table 4.5 compares the observed weighted average activation rate with the theoretical estimated activation rate. We can see that estimated activation rate is almost as accurate as the observed activation rate.The percentage differences are no bigger than 1.5%.

4.5 Activation Rate and Fitness

As defined in Section 4.2.2, an activation of a node is when the node is evaluated. It is a fact that a node must be evaluated to contribute the output of the program tree. This contribution does not have to be positive. In fact, the positive or negative effect is not only determined by the node itself but also depends on the "environment" this node stays. In fact, the "environment", i.e. the context, for the node plays an even more important role, such as passing correct parameters,

to ensure the node behaves properly. As a result, the effect for a function contributing to final result is very hard to quantify. But we are sure for the opposite situation: a node does not have any contribution if it has not even been called. In another word, a node at *least* needs to be called to be useful.

With this idea, we can see that the concept of activation rate of a node actually represents the maximum number of test cases that may be impacted when there is a change in the node. Activation rate expresses the importance of the node quantitatively using this impact. For example, if in a tree of multiplexer 11 problem, one node A's activation rate is 0.59375, and the other node B's activation rate is 0.2822265625. We say that A is about twice as important as B. This is because a change in node A may affect at most $0.59375 \times 2048 = 1216$ test cases, while a change in node B may only affect at most $0.2822265625 \times 2048 = 578$ test cases. In table 4.3, we show the estimated activation rate of nodes in different depth. The third column is calculated by the estimated activation rate times 2048. This gives the maximum number of test cases that may be affected. Our experiments and analysis of estimated activation rate previously suggests that there is a relatively strong inverse relationship between the depth and the activation rate of the node. We can use the activation rate to quantitatively expresses one general observation in GP: "nodes deeper in tree are less important than nodes near the root."

In addition to node's activation rate, we also define tree's activation rate in Definition 3. A program tree's activation rate represents the percentage of nodes in the tree which are evaluated. If we let $Re(T) = 1 - \Theta(T)$, then $Re(T)$ represents the percentage of nodes in the tree which are not evaluated (i.e. non-functional codes). Similar to Redundancy defined in [BT94], we can define Tree Redundancy using tree's activation rate:

Definition 5 (Redundancy of Tree) *Redundancy of Tree T, Re(T), is the percentage of non-functional codes in T:*

$$Re(T) = 1 - \Theta(T).$$

Because the tree's activation rate can be practically calculated (see Definition 3) or theoretically estimated (using (4.8)), tree redundancy can also be calculated in the above two ways.

The concept of tree redundancy gives a new way to analyze how tree growth affects the computation effort required to evaluate the tree. The computation effort required to evaluate a tree is proportional to the number of nodes needed to be evaluated in the tree, i.e.

$$\text{Effort}(T) \propto \sum_{i=1}^{d} f(i)$$

Where $f(d)$ is a node distribution function which returns the number of nodes for input depth. Because the existence of redundancy (i.e. non-functional codes), the above formula should be revised as:

$$\text{Effort}(T) \propto (1 - Re(T)) \cdot \sum_{i=1}^{d} f(i)$$

Since $Re(T) = 1 - \Theta(T)$, we have:

$$\text{Effort}(T) \propto \Theta(T) \cdot \sum_{i=1}^{d} f(i)$$

4.6 Activation Rate and Tree Size

Using experiment data from Section 4.4, we can find that the tree's activation rate decreases as the tree's depth increases, as in Table 4.6. Estimating a program

Tree Depth	1500-Trees	5000-Trees	15000-Trees	Weighted Average
2	0.821678	0.80597	0.784351	0.791982884
3	0.723303	0.709011	0.698499	0.702674163
4	0.603315	0.596777	0.589675	0.592278256
5	0.470983	0.481705	0.488888	0.485968349
6	0.383928	0.392998	0.38573	0.387294512
7	0.303356	0.304654	0.30542	0.30509786
8	0.241572	0.233286	0.236439	0.23606386
9	0.17956	0.183495	0.183299	0.183083721
10	0.175305	0.166293	0.168486	0.168451744
11	0.117756	0.127852	0.132266	0.130227163
12	0.101429	0.090494	0.100998	0.098585279
13	0.085222	0.071941	0.077086	0.076457116

Table 4.6: Tree Depth and Tree Activation Rate

tree's activation rate is very hard. This is because the tree's activation rate depends on the distribution of the number of nodes over tree depth, and the shape of a tree may vary greatly. For example, the tree size growth can be constant (i.e. in the case of list), linear, polynomial or exponentially (i.e. in the case of full tree). Formally, let $f(d)$ be a node distribution function which returns the number of nodes for input depth, then the estimated activation rate of a tree T can be calculated as follows:

$$\begin{aligned} \Theta_{estimate}(T,d) &= \frac{1 + \triangledown \cdot f(2) + \triangledown^2 \cdot f(3) + \ldots + \triangledown^{d-1} \cdot f(d)}{\sum_{j=1}^{d} f(j)} \\ &= \frac{\sum_{i=0}^{d-1} \triangledown^i \cdot f(i+1)}{\sum_{j=1}^{d} f(j)} \end{aligned} \tag{4.8}$$

$f(d)$ effectively defines the shape of the program tree and it is very hard to estimate in general. But once we get a concrete program tree, $f(d)$ is then known. We can then use (4.8) to estimate that tree's activation rate. In addition, using (4.8), the computation effort required to evaluate a tree Effort(T):

$$\begin{aligned} \text{Effort}(T) &\propto \Theta(T) \cdot \textstyle\sum_{i=1}^{d} f(i) \\ &\propto 1 + \triangledown \cdot f(2) + \triangledown^2 \cdot f(3) + \ldots + \triangledown^{d-1} \cdot f(d) \end{aligned} \tag{4.9}$$

Since the descent rate $\triangledown$ is fixed for a problem domain, the computation effort required to evaluate a tree relates to the distribution of nodes over depth.

In the next, we use (4.8) and (4.9) to analyze how the number of nodes actually need to be evaluated (the computation effort required) increases when the depth of the tree increases. As we discussed, without knowing the exact form of node distribution function $f(d)$, it is impossible to calculate $\Theta_{estimate}(T,d)$. As a result, we firstly work with theoretical models and define two node distribution functions as follows:

$$\begin{aligned} f_1(d) &= 2d^2 + 3d + 1 \\ f_2(d) &= 2^{d-1} \end{aligned}$$

Node distribution function $f_1(d)$ represents a quadratic growth model, and $f_2(d)$ represents an exponential tree growth model similar to a full binary tree. As we discussed early, the shape of tree varies for a given size. For example, for a binary tree of size l, the most compact tree has a depth of $\log_2 l + 1$, while the tallest one had depth of $\frac{l+1}{2}$. In [FO82], Flajolet and Oldyzko show that, between these two cases, the most common tree depth is near $l^{0.63}$, and the average height

converges slowly to $2\sqrt{\dfrac{\pi(l-1)}{2}} + O(l^{0.25})$ as l increases. Later, in [LSPF99], [Lan00b] and later in [Lan00a], Langdon et al. use a number of experiments to show that, there is a strong indication that, the average depth of binary trees in a population grows linearly at about one per generation. Using the result from Flajolet and Oldyzko, Langdon et al. further conclude that the growth in size is $O(\mathbf{generations}^{1.6})$ for reasonable sized programs, and it rises to a limit of $O(\mathbf{generations}^{2})$ for trees of more than 32,000 nodes. Based on this result from Langdon et al., we can find that, among our four theoretical models, $f_1(d)$ represents a common and average scenario, while $f_2(d)$ models a more extreme worst case.

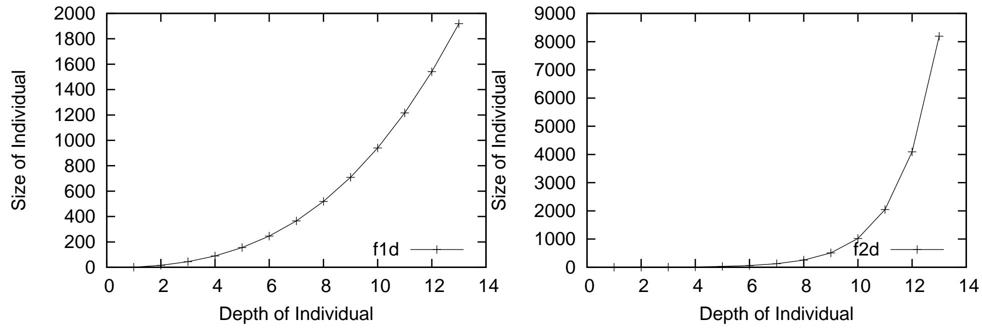


Figure 4.2: Size of Individual as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$

Given the two node distribution functions, Figure 4.2 visualizes how the size of individual increases as the depth of the individual increases based on the node distribution function $f_1(d)$ and $f_2(d)$ defined. Now, with the definition of $f(d)$, using (4.8), we can calculate $\Theta_{estimate}(T, d)$ and then the number of nodes need to be evaluated. Let $\bigtriangledown = 0.75$, Figure 4.3 summarizes how the number of tree nodes needs to be evaluated increases as the depth of the tree increases given the node distribution function $f_1(d)$, $f_2(d)$ and (4.8). Comparing Figure 4.2 and Figure 4.3, we can find out that, when the descent rate is smaller than 1, the actual number of nodes needs to be evaluated increases in a much slower manner compared to the increase in the actual size of the individual when the depth of the individual increases. Assuming GP uses a fixed amount of time to evaluate every tree node, then Figure 4.3 effectively visualizes quantitatively how Effort(T) (defined in (4.9)) increases as the depth of the tree increases. If we further expands the tree depth in Figure 4.3 from 13 to 50, as shown in Figure

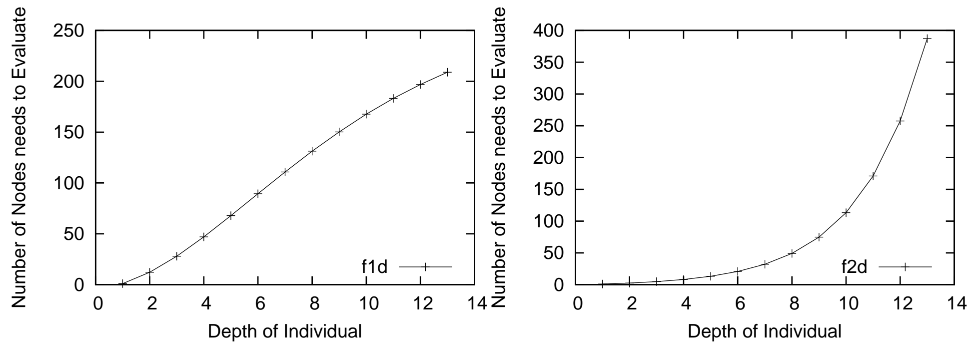


Figure 4.3: Number of Nodes needs to be Evaluated as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$

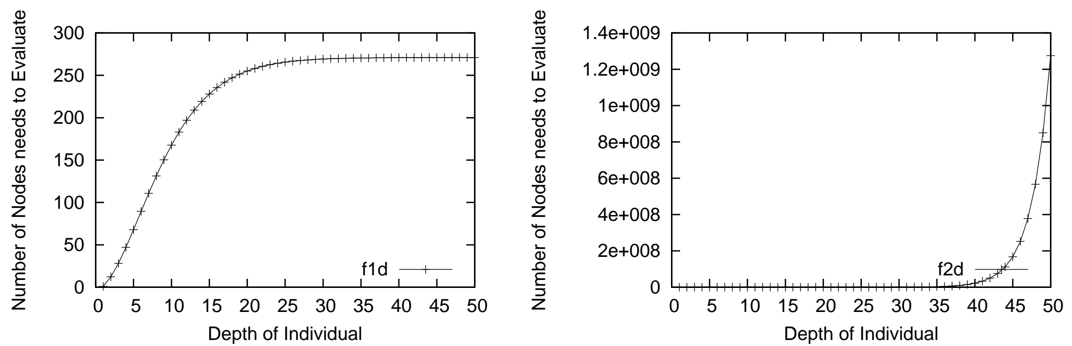


Figure 4.4: Number of Nodes needs to be Evaluated as Depth Increases based on Node Distribution Function $f_1(d)$ and $f_2(d)$ (Depth up to 50)

4.4, we can find that, interestingly, in the case of $f_1(d)$, the value of Effort(T) converges, and does not increase anymore even if the tree size further increases.

In conclusion, the above experiment results suggest two very interesting new observations regarding bloating for problem domains whose activation rate is smaller than one:

1. as the depth of the program tree increases, the computation effort required to evaluate the tree increases, but in a slower manner compared to increase in the size of the tree;

2. there potentially exists a theoretical upper bound for the computation effort required to evaluate trees for certain tree shapes.

4.7 Activation Rate and Crossover

In the previous section, we use activation rate to analyze how bloating effects the runtime tree evaluation time. In this section, we use activation rate to perform two analyze of crossover.

4.7.1 Crossover Effects

In [NB95], Nordin and Banzhaf study the effect of crossover. They define the change in fitness $\Delta f_{percent}$ from parent's fitness to child's fitness as:

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100 \tag{4.10}$$

They analyze a large number of individuals in early generations of the symbolic regression problem and find that there is a high probability that the function of the program is severely damaged, resulting in a fitness decrease for the individual. They also find that the second most common effect of crossover is that nothing happens. Based on this experiment, Nordin and Banzhaf conclude that crossover is generally destructive or neutral. Constructive crossover is rare.

This idea of destructive nature of crossover is widely accepted and serves as the basis for bloating theory "defense against crossover". In symbolic regression problem, the descent rate is 1. In this section, we perform a similar analysis in multiplexer 11 domain in which the descent rate is not 1, with a much more complex configuration setup. The purpose of this analysis is to explore if there is any relationship between the destructiveness of crossover and the activation rate.

The experiment of crossover effect in multiplexer 11 problem is designed as follows. We use four individual experiments to simulate different crossover scenarios. For all four experiments, we firstly generate a tree as a parent for crossover and evaluate its fitness. Then we perform crossover with another randomly generated tree to produce a child. We then evaluate the child's fitness and compare it with the parent tree using (4.10).

In the first experiment, denoted as Experiment 1, the parent tree is randomly generated using grow method and the tree depth is from 3 to 15. Because the raw fitness of randomly generated trees in multiplexer 11 problem typically ranges from 900 to 1100, Experiment 1 simulates early run of GP in multiplexer 11, i.e. crossover with low fitness individuals. In Experiment 1, we compare 30,000

trees. In the second experiment, denoted as Experiment 2, the parent tree's fitness ranges from 1300 to 1600. This simulates crossover with medium fitness individuals. These trees are collected from other GP runs. The second parent is a randomly generated tree using grow method. In Experiment 2, we compare 50,000 trees. In the third experiment, denoted as Experiment 3, the parent tree's fitness ranges from 1800 to 2000. This simulates crossover with high fitness individuals. The second parent is a randomly generated tree using grow method. In Experiment 3, we compare 50,000 trees. In the last experiment, denoted as Experiment 4, the parent tree's fitness ranges from 1800 to 2000. But unlike in Experiment 3, the other parent is not a randomly generated tree, instead, the second parent's fitness is also limited to the range from 1800 to 2000. Because randomly generated trees tend to have very low fitness (typically range from 900 to 1100), the purpose this experiment is to show whether the fitness of the second parent affects the effect of crossover or not. In Experiment 4, we compare 50,000 trees.

In [NB95], the effect of crossover is defined as constructive, destructive or neutral. Constructive crossover is a crossover event which makes fitness of the child at least 2.5% better compared with parent. Destructive crossover is a crossover event which make the child's fitness at least 2.5% worse compared with the parent. Neutral crossover is a crossover event which changes child's fitness within 2.5% compared with its parent. We found that the choice of 2.5% is very arbitrary and this rejects a small change of fitness. So, in this experiment, our analysis uses the following definition of crossover effect.

Definition 6 (Crossover Effect) *A* Strictly Constructive Crossover *is a crossover event which increases the fitness of child compared with its parent.*

A Strictly Neutral Crossover *is a crossover event which does not change the child fitness compared with its parent.*

A Strictly Destructive Crossover *is a crossover event which decreases the fitness of child compared with its parent.*

Using the above definition, the distribution of crossover effects can be summarized in Table 4.7. We can see from Table 4.7 that the observation that crossover is mostly destructive in [NB95] is correct in multiplexer domain in Experiment 2, 3 and 4. But in Experiment 1, there is no preferences between destructive and constructive crossover. Since in [NB95], the experiment is also performed in early runs of GP in symbolic regression, the result we got is quite different from what

Experiment	Destructive	%	Neutral	%	Constructive	%
1	6603	22.01%	16572	55.24%	6825	22.75%
2	34013	68.03%	12107	24.21%	3880	7.76%
3	31292	62.58%	18358	36.72%	350	0.70%
4	28972	57.94%	20494	40.10%	534	1.07%

Table 4.7: Crossover Effects Distribution

is observed in [NB95]. This suggests that the crossover effect in early runs of GP is problem domain dependent.

We can further study the effect of crossover using the above experiment data and activation rate. Crossover is a two phase process. Firstly, crossover points are selected and then, the subtrees are swapped on selected crossover points. Both the selection of crossover point and the subtree swapping contribute to the effect of crossover.

In order to analyze the effect of subtree swapping only, we modify (4.10) as follows:

$$\Delta f'_{percent} = \begin{cases} \dfrac{f_{before} - f_{after}}{N \cdot \Theta} \cdot 100 & \Theta \neq 0 \\ 0 & \Theta = 0, \end{cases} \tag{4.11}$$

where N is the number of test cases and Θ is the activation rate of the crossover point. In (4.11), $f_{after} - f_{before}$ is the change in fitness after crossover. Please note that, f_{after} and f_{before} in (4.11) can only be the raw fitness (see Section 2.3.2.1). Based on the definition of activation rate, $N \cdot \Theta$ represents the maximum number of test cases can be affected, when a change happens on the cross point. So, by dividing $f_{after} - f_{before}$ by $N \cdot \Theta$, $\Delta f'_{percent}$ represents the effect of swapping subtree in crossover solely without the need to consider the effect of selection of crossover point, as it is already considered in $N \cdot \Theta$.

Using (4.11), the distribution of crossover effects can be summarized as in Table 4.8. We can see that the effect of subtree swapping is relatively stable (the variance is relatively small in all cases). Initially, in Experiment 1, the effect of swapping are almost the same for constructive and destructive crossover. As the fitness of the parent increases, both effects increase but the destructive effect increases much faster. So, the constructive subtree swapping effect is almost neutral to the changes of parent fitness, while the destructive subtree swapping effect is positive related to parent fitness.

Experiment	Crossover Type	Mean	Variance	Standard Deviation
1	Constructive	0.0650	0.0090	0.0947
	Destructive	0.0653	0.0095	0.0975
2	Constructive	0.0773	0.0098	0.0990
	Destructive	0.1671	0.0193	0.1389
3	Constructive	0.1460	0.0263	0.1621
	Destructive	0.3100	0.0281	0.1676
4	Constructive	0.1689	0.0324	0.1799
	Destructive	0.3116	0.0294	0.1716

Table 4.8: Distribution of Constructive and Destructive Crossover based on Definition 6

The effect for the selection of crossover point can also be studied using activation rate. Because Experiment 1, 3 and 4 simulate relatively extreme cases, in this analysis, we only use data from Experiment 2. Table 4.9 summarizes the dis-

Θ Range	Total	Constructive	Destructive	Constructive %
$0.1 > \Theta \geq 0.0$	2974	661	2313	22.23
$0.2 > \Theta \geq 0.1$	6541	726	5815	11.10
$0.3 > \Theta \geq 0.2$	7611	822	6789	10.80
$0.4 > \Theta \geq 0.3$	3236	309	2927	9.55
$0.5 > \Theta \geq 0.4$	1106	113	993	10.22
$0.6 > \Theta \geq 0.5$	7465	699	6766	9.36
$0.7 > \Theta \geq 0.6$	869	82	787	9.44
$0.8 > \Theta \geq 0.7$	1652	105	1547	6.36
$0.9 > \Theta \geq 0.8$	601	46	555	7.65
$1.0 > \Theta \geq 0.9$	142	4	138	2.82
$\Theta = 1$	5696	313	5383	5.50

Table 4.9: Distribution of Constructive and Destructive Crossover Grouped by Crossover Point's Activation Rate based on Definition 6

tribution of constructive and destructive crossover based on the crossover point's activation rate. For example, the first row in the table says for crossover point with activation rate $\Theta \in [0.0, 0.1)$, there are in total 2974 non-neutral crossover happens, in which 22.23% of them are constructive crossover. From Table 4.9, we can clearly see that there is a trend for the number of constructive crossover to decrease as the crossover point's activation rate increases. Since the tree node's activation rate is negatively correlated to the depth of the node, we can conclude that deeper crossover point promotes constructive crossover.

In conclusion, there are the following four interesting findings in our experiment of crossover effects using multiplexer 11 domain:

1. In initial GP runs of multiplexer problem (Experiment 1), there is no preference between constructive and destructive crossover.

2. As the parent's fitness increases, the crossover becomes more destructive (Experiment 2 and 3).

3. The effect of subtree swapping is very stable. The constructive subtree swapping effect is almost neutral to the changes of parent fitness, while the destructive subtree swapping effect is positive related to parent fitness.

4. The depth of crossover point do affect crossover effect. Deeper crossover point promotes constructive crossover.

4.7.2 Semi-Intron Crossover

As we discussed previously, marking can be used to define and find a special kind of intron, the inviable code. Because activation rate is an extension to marking, we can also define this kind of intron using activation rate:

Definition 7 (Introns) *Introns are subtree in T whose root node x's activation rate is zero.*

Because activation rate marks non-introns differently, it gives a lot more flexibility in finding introns. For example, a node may have just been called once in 11 Multiplexer domain for all 2048 test cases. This node, while using marking, is classified as non-intron. But the effect of this node is almost the same as intron. With the concept of activation, we can cope with this situation by defining semi-intron:

Definition 8 (Semi-Introns) *Semi-Introns are subtree in T whose root node x's activation is smaller than a predefined value ϵ.*

Semi-intron is the same as pseudoinvible code mentioned by Luke in [Luk03]. But our definition provides a quantitative measurement and a practical algorithm to identify semi-introns. Semi-introns are nodes which can be viewed as introns. By adjusting the value of ϵ, we can ignore a number of nodes which only affect very small number of test cases by classifying them as introns. So, similar to marking,

we can also develop semi-intron crossover which avoids crossover points selected at nodes which are semi-introns.

Semi-intron crossover works in the following way. While program trees are executed in fitness evaluation, the activation rate for each node is calculated. This is achieved by adding an accumulator of type integer for each node. When a node is invoked, the accumulator increases by 1. Similar to marking, the additional overheads in terms of both memory and execution time for calculating activation rate is very small. After tree evaluation, with activation rate information, only nodes whose activation rate are bigger than or equal to the parameter ϵ can be selected as crossover points in crossover.

Using semi-intron crossover with different settings for parameter ϵ, we can analyze how crossover point selection affects the GP performance and bloating. We use multiplexer 11 problem in this experiment. The experiment is designed as follows. The parameter ϵ in semi-intron crossover ranges from 0 to 2048. For each ϵ value, 50 independent GP runs are performed to collect information including average fitness increases, average tree size (number of nodes) increases, and average tree depth increases. We then compare fitness, tree depth and size changes for different ϵ values. As a result, there are 102,400 independent GP runs are examined. Fitness change is used as a measurement of GP performance, while tree depth and size changes are used as measurements of bloating.

Parameter settings for each independent GP run are as follows. The population size is 500. Fitness proportionate selection is used. In breeding, there is 90% chance semi-intron crossover is used and there is 10% chance that reproduction is used. GP runs for 300 generations. Ramp-half-and-half method [Koz92] is used for initial population generation. One thing to note is that when parameter ϵ equals to 0, the semi-intron crossover behaves very similar to the normal classical crossover. When parameter ϵ equals to 1, the semi-intron crossover is equivalent to marking.

The experiment results can be viewed in Figure 4.5, 4.6 and 4.7. When the parameter ϵ of semi-intron crossover increases from 0 to 2048, the selection of crossover point becomes more and more restricted to nodes near the root. In another word, the average depth of crossover point becomes smaller and smaller. This is because activation rate is negatively correlated to node's depth (from (4.7)). As a result, semi-intron crossover restricts the depth of crossover point using phenotype information (fitness), rather than based on genotype information

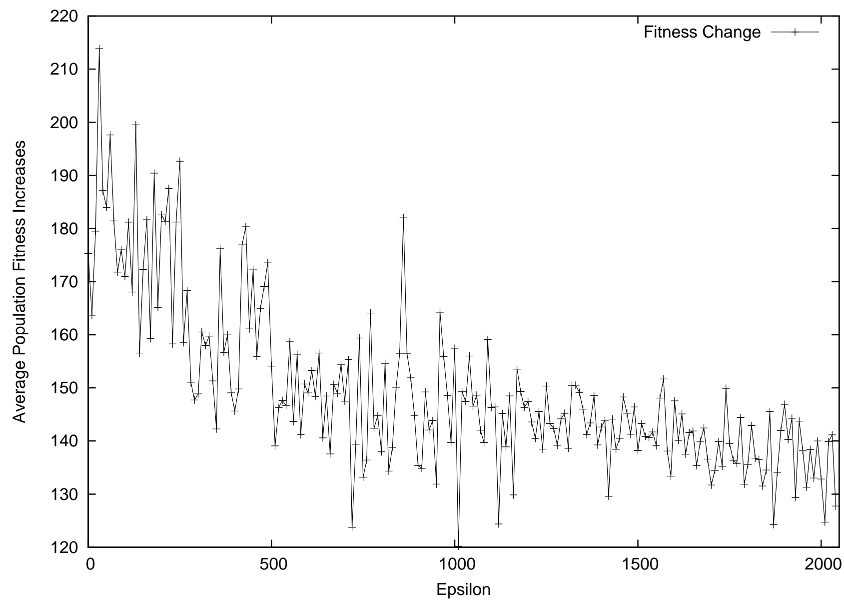


Figure 4.5: Average Fitness Changes for ϵ from 0 to 2048 in Steps of 10

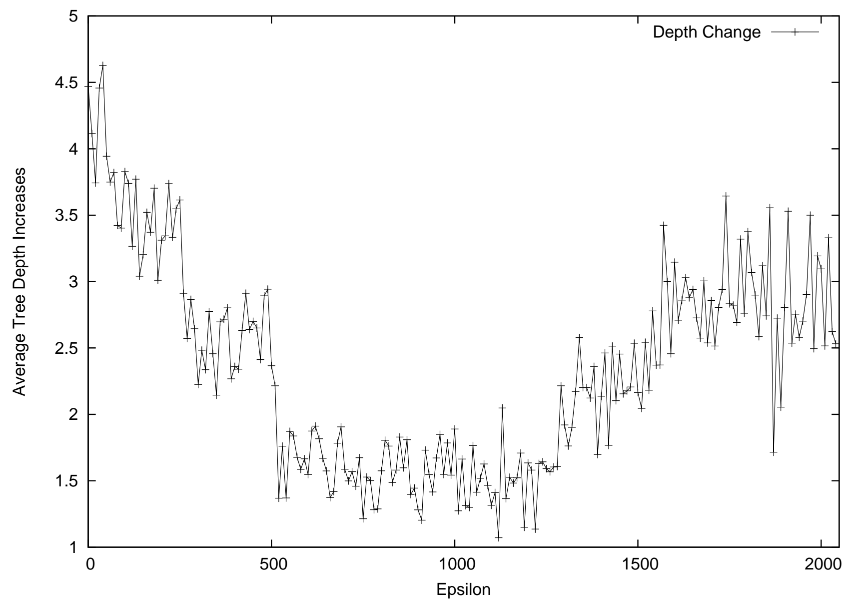


Figure 4.6: Average Tree Depth Changes for ϵ from 0 to 2048 in Steps of 10

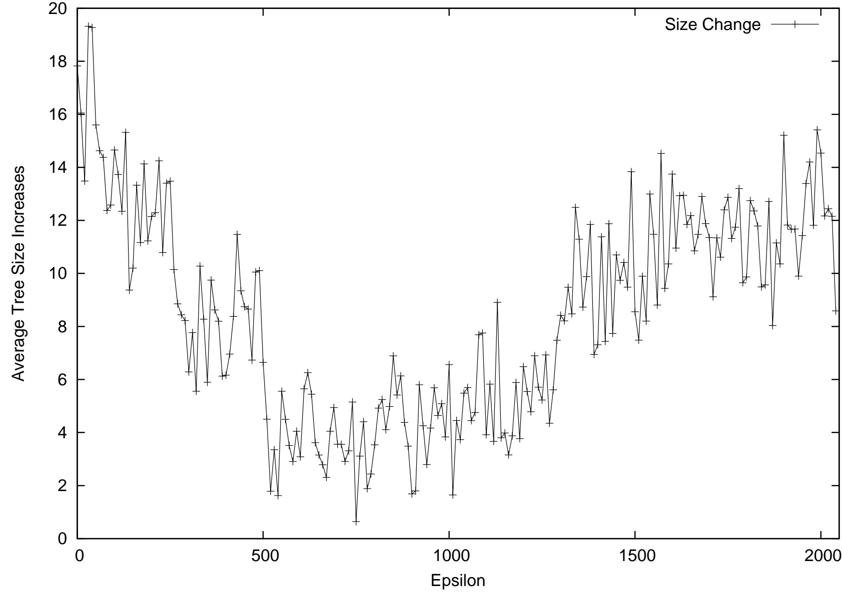


Figure 4.7: Average Tree Size Changes for ϵ from 0 to 2048 in Steps of 10

(node depth) directly.

From Figure 4.5, we can find that as ϵ increases, the fitness performance generally decreases. This phenomenon is reasonable and can be explained using activation rate. Because crossover is generally destructive [NFB96], crossover points with higher activation rate result in more decrease in offsprings' fitness. As a result, the average population fitness decreases as ϵ increases. Another explanation to this phenomenon is that by increasing ϵ, we limit the ability of semi-intron crossover by restricting the freedom of crossover point selection. Since selection and crossover contribute to the fitness performance, limiting the effect of crossover reduces the fitness buildup.

The second observation from Figure 4.6 and 4.7 is that, as ϵ increases, the average tree depth and size firstly decrease and then slowly increase. The decreasing phase (ϵ ranges from 0 to around 500) can be explained using modification point depth theory [Luk03]. In a nutshell, modification point depth theory says that deep crossover points contributes to code growth and bloating. In our experiment, as ϵ increases and crossover point depth decreases, both average tree depth and size decrease (i.e. bloating is reduced). This supports modification point depth theory. But in the second phase (ϵ ranges from 500 to 2048), as ϵ further increases, the average depth and size should keep decreasing, according to modification point depth theory. But as we can see from Figure 4.6 and 4.7, the

average tree depth and size increase. This observation provides an experiment evidence against the modification point depth theory.

The finally observation from experiment data is that there is generally a trade off between GP performance (fitness in this case) and bloating control. Reducing bloating through limiting crossover point depth decreases GP performance. But the relationship between fitness and bloating is very complex.

4.8 Conclusion

In this chapter, we develop activation rate as a new quantitative model of tree structure. The development of activation rate is motivated by analyzing bottom-up tree evaluation algorithm. Activation rate models the importance of a node within a tree using how many times the node has been evaluated as the weight. This is because a node needs to be at least evaluated in order to contribute to the fitness of the individual. One advantage of activation rate is that it can be theoretically estimated. Experiment results show that, the estimated activation rate can be very accurate. The development of the activation rate, especially the estimated activation rate, has enabled us to analyze GP dynamics from a new perspective.

Using activation rate, we perform a very detailed analysis about how the computation effort required to evaluate a tree increases as the depth and size of the tree increases. This analysis is important because it provides a quantitative method to model by how much bloating slows down the GP tree evaluation. We find that the computation efforts required increases slower compared to the size of the tree when the depth of the tree increases. Moreover, there exists an upper bound for the computer effort required for certain tree shapes. These two observations give a new insight into the effect of bloating. Also, with activation rate, we greatly extend the analysis of crossover effect in literature and find that the effect of crossover is more complex than we previously thought. Finally, we develop a new crossover called semi-intron crossover based activation rate. Experiments of semi-intron crossover gives interesting evidence contrary to modification point depth theory.

Chapter 5

Removal Bias & Depth-Constraint Crossover

5.1 Introduction

In the last chapter, we developed *activation rate* and use it to quantitatively study how bloating affects the computation effort required to evaluate trees in boolean problems. As we reviewed in Chapter 3, bloating theory and bloating control have been and remain to be one of the hottest research areas in GP. Over past few years, a good number of bloating theories and control methods have been proposed and developed. While bloating control methods developed form the front line of defense against bloating, the importance of bloating theories cannot be neglected. Even though a single, comprehensive theory to explain bloating has not been concluded yet, bloating theories developed over past few years explain bloating from different perspectives and they are more complementary than substitutes to each other. These theories also guide or serve as foundations for the development of bloating control methods.

In this chapter, we revisit one of the bloating theories, the *removal bias*. We extend the removal bias theory by developing a quantitative definition of removal bias and show using experiments that the amount of removal bias defined is positively correlated to code growth. This experiment result gives another empirical evidence to support removal bias. In addition, using this experiment result, we develop the *depth difference hypothesis* which acts as a more general abstraction over, or extension to removal bias theory. Depth difference hypothesis, simply speaking, blames the depth difference of the subtrees swapped in crossover

as the root cause of bloating. This depth difference in turn is driven by the depth difference of parents selected in crossover. As a direct application of depth difference hypothesis, we also develop *depth constraint crossover* as a new bloating control method motivated by depth difference hypothesis. Empirical results show that the newly developed bloating control technique, despite the simplicity, is very effective in controlling bloat without sacrificing fitness, especially when used in combination with Koza-style depth limiting. The work presented in this chapter significantly extends the discussion of depth constraint crossover in [LZ10b].

The rest of this chapter is organized as follows. In the next section, we firstly expand the discussion of removal bias from Section 3.1.2. Then, we define a quantitative model of removal bias and present experiment results to show the strong correlation between our definition of removal bias and bloating. After this, we introduce the depth constraint crossover in detail and compare it with a number of existing bloating control methods. Finally, we conclude this chapter with a more detailed discussion of depth difference hypothesis proposed.

5.2 Background and Related Work

Before start discussing about removal bias, we use a crossover example to establish the naming convention we are going to use in the rest of this chapter. Considering crossover in general as shown in Figure 5.1, the inputs into crossover are two individuals and we call them *Parent A* and *Parent B* respectively. The output are two offsprings and we call them *Offspring A* and *Offspring B* respectively. The crossover firstly selects one crossover point from each parent. We call the crossover point selected from parent A the *Crossover Point A*. We call the subtree in parent A whose root is crossover point A the *Subtree A*. The subtree A is the subtree which is going to be deleted from parent A. Similarly, we call the crossover point selected from parent B the *Crossover Point B*. We call the subtree in parent B whose root is crossover point B the *Subtree B*. In the example in Figure 5.1, subtree B and parent B are the same because crossover point B happens to be the root node of parent B. After crossover point A and B are selected, offspring A and B are created by swapping subtree A and B. Offspring A is created by Parent A giving up Subtree A and receiving Subtree B. Offspring B is created by Parent B giving up Subtree B and receiving Subtree A.

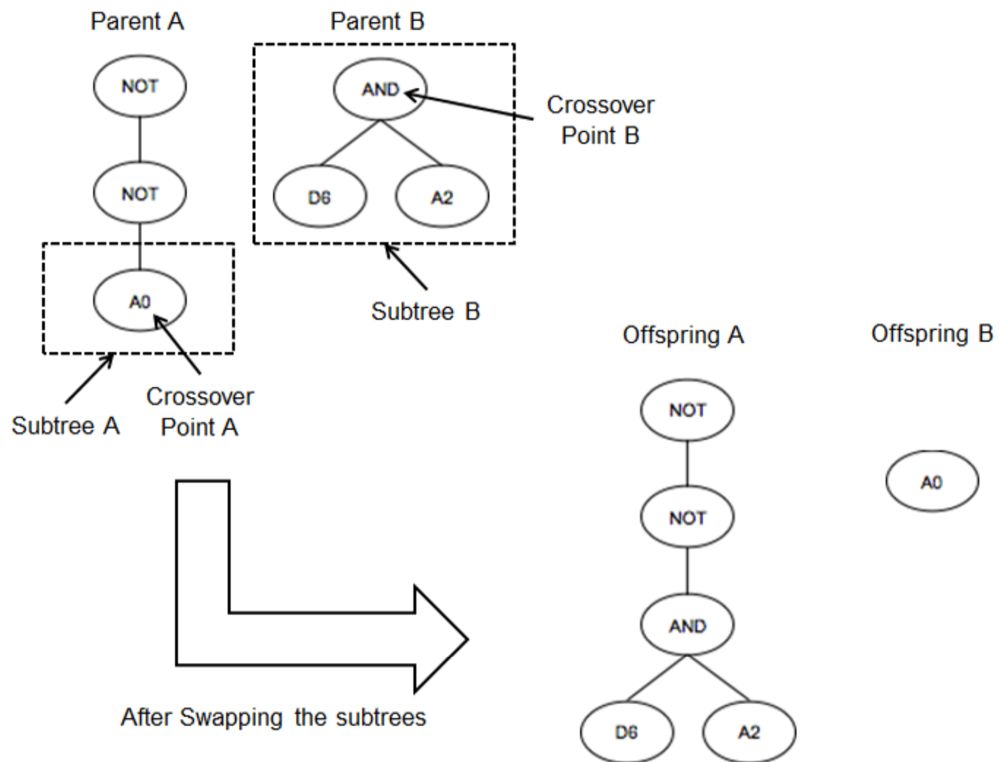


Figure 5.1: Example of a General Crossover

The theory of removal bias is among the oldest bloating theories. The theory is firstly introduced by Soule and Foster in [SF98b]. The foundations of removal bias theory are intron theory and defensive against crossover (categorized as "protective hypothesis" in [SF98b]), in which the propagation of intron (non-functional code) which defends the individual against the destructiveness of crossover is considered as the root cause of bloating. In [SF98b], Soule and Foster point out that if the root node of any subtree is intron, all descendants within the subtree are also introns. For example, if crossover point A selected is intron, then all nodes in subtree A are introns as well. Subtrees selected to be swapped in crossover are more likely to be within intron regions when the crossover points selected are deeper. As a result, if we consider two parents in crossover A and B, if the depth of crossover point A is bigger compared to the depth of crossover point B, resulting subtree A to be smaller (in terms of both depth and size) compared to subtree B, offspring A which is produced by parent A giving up smaller subtree A and receiving bigger subtree B (which is more likely to grow in size and depth), is more likely to inherit the fitness from parent A. This is because the

change (crossover point A) is more likely to be within intron region and changes in introns do not effect fitness. On the other hand, offspring B which is produced by parent B giving up bigger subtree B and receiving smaller subtree A is more likely to have inferior fitness. This is because the change (crossover point B) is more likely to be within exon region and changes in functional codes usually results in worse fitness due to the destructiveness of crossover. Given the above analysis, Soule and Foster further ague that even though the crossover does not necessarily increase population size, in the presence of selection pressure, offspring A would be favored over offspring B from phenotype perspective of view. The accumulated effects of this bias favoring smaller subtrees to be deleted contribute to bloating in parallel to the protective hypothesis.

Soule and Foster provide two pieces of experiment evidence to support the theory in [SF98b]. In the first experiment, the size and fitness change between parent and offspring during crossover is examined. Soule and Foster show that for offspring whose fitness equals or exceeds their parents, there is usually an increase in offsprings' size compared to their parent, especially in early generation of GP. Another experiment is performed using non-destructive crossover, in which crossover only produces offspring whose fitness exceeds or equals to their parents' fitness, with which the code growth contributed from protective hypothesis no longer applies. Experimental results show that, even with non-destructive crossover, the code growth can still be observed without protective hypothesis because the existence of removal bias. But the amount of bloating is less compared to normal crossover in which both protective hypothesis and removal bias apply. This result also confirms their view that there are multiple causes of bloating. Removal bias and protective hypothesis are complementary and explain bloating from two different perspectives. Further experiment evidence to support removal bias theory can be found in [SH02, Luk00a]. In [SH02], Soule and Heckendorn show that "there is a strong inverse relationship between the size of the removed branch during crossover and the resulting change in fitness caused by that crossover", however, "the size of the added branch is negatively correlated to the fitness change". In [Luk00a] and further in [Luk03], Luke argues that there is a more general bias favoring deeper crossover point and this causes bloating, i.e. the modification point depth theory, and favoring smaller removed subtree (suggested by removal bias) can be thought as a special case of the modification point depth theory.

5.3 A New Quantitative Model

Despite the long history of removal bias, we don't think the study of the theory is as well established as other theories such as defense against crossover. This is partially due to the widely accepted proposition made by Luke in [Luk00a], in which he suggests that the modification point depth theory is an abstraction over removal bias theory. We agree that under Luke's interpretation of removal bias theory, it is naturally a special case of the modification point depth theory. However, we believe that the removal bias theory does have room for another interpretation which cannot be explained using the modification point depth theory. In this section, we explore a new interpretation of the removal bias theory using a new quantitative model.

The center of removal bias theory lies the depth of crossover point and the size of subtree swapped. These two factors, the depth of crossover point and the size of subtree, are interrelated, i.e. deeper crossover point results in smaller subtree. Based on Luke's interpretation [Luk00a], it is favoring deeper crossover point which results in bloating. However, Soule and Heckendorn argue in [SH02] that, favoring deeper crossover point *in removed branch* is an essential condition rather than optional. Here, we propose that not only the smaller *removed* branch but also the bigger *added* branch play an important role in the formation of bloating. In fact, we theorize that it is the depth difference between the two subtrees swapped in crossover which causes the bloating. We call this, the *depth difference hypothesis*.

In order to verify the depth difference hypothesis, we firstly quantitatively define the depth difference as the *amount of removal bias*:

Definition 9 (Amount of Removal Bias) *Let Parent A and Parent B be the two parents selected in crossover, and Subtree A and Subtree B be the two subtrees selected to be swapped from Parent A and Parent B respectively, also let the function Depth(.) returns the depth of the input subtree. Then, the* amount of removal bias *denoted as Q_{bias} for this crossover operation is defined as:*

$$Q_{bias} = |Depth(Subtree\ A) - Depth(Subtree\ B)|$$

One thing to note is that, in our definition, the depth of subtree is used rather than size of subtree. Given a fixed tree shape, the size and depth of the subtree can be used interchangeably. We use depth rather than size because we think the

depth of subtree gives better control over size.

Problem	Q_{bias}	$\sigma_{Q_{\text{bias}}}$	Parent Diff	$\sigma_{\text{Parent Diff}}$	Diff2	σ_{Diff2}
Ant	5.83	10.59	17.63	17.33	0.19	26.19
11-Multiplexer	4.96	7.55	9.72	11.16	-0.19	16.48
Even-5	5.10	9.16	14.83	15.11	0.24	22.74
Regression	16.59	53.65	26.64	70.91	1.77	77.41

Table 5.1: Removal Bias at Generation 49 using No Bloating Control Methods

In the next, we perform experiments to show that the amount of removal bias can be widely observed. In the experiment, we perform GP runs with standard configuration settings and passively record the amount of removal bias based on the definition for each crossover operation. The experiment settings are as follows. We use four problem domains: Artificial Ant with Santa Fe food trail, 11-Multiplexer, Even-5 Parity and Symbolic Regression of $x^6 - 2x^4 + x^2$. The population size is 500 and GP runs for 50 generations. Tournament selection is used with tournament size 5. Only crossover and reproduction are used in breeding process with probability 90% and 10% respectively. In this experiment, we do not use any bloating control methods. Using this parameter setting, approximately 562,500 crossover operations are analyzed in each problem domain. The experiment is performed using ECJ [Luk09].

Table 5.1 summarizes the amount of removal bias at generation 49. Because in the experiment, GP runs for 50 generations, generation 49 is the last generation which performs crossover operations. In the table, the Parent Diff column represents the depth difference between two parents:

$$\text{Parent Diff} = |\text{Depth(Parent A)} - \text{Depth(Parent B)}|$$

Diff2 is calculated as:

$$\begin{aligned}\text{Diff2} &= [\text{Depth(Subtree A)} - \text{Depth(Subtree B)}] \\ &\quad -[\text{Depth(Parent A)} - \text{Depth(Parent B)}]\end{aligned}$$

At generation 49, for artificial ant, multiplexer-11 and parity-5, the amount of removal bias is around 5. However, we observes a much bigger value (16.59) for symbolic regression problem. We think this is due to the natural of symbolic

regression problem, which is a real numbered problem. Another interesting observation from Table 5.1 is that while both Q_{bias} and Parent Diff are quite big, the Diff2 on the other hand is quite close to 0, even though the standard deviation of Diff2 is quite big. This suggests that there is a weak correlation between the amount of removal bias and the depth difference between two parents.

Figure 5.2 shows how Q_{bias} changes over generations in the experiment. This

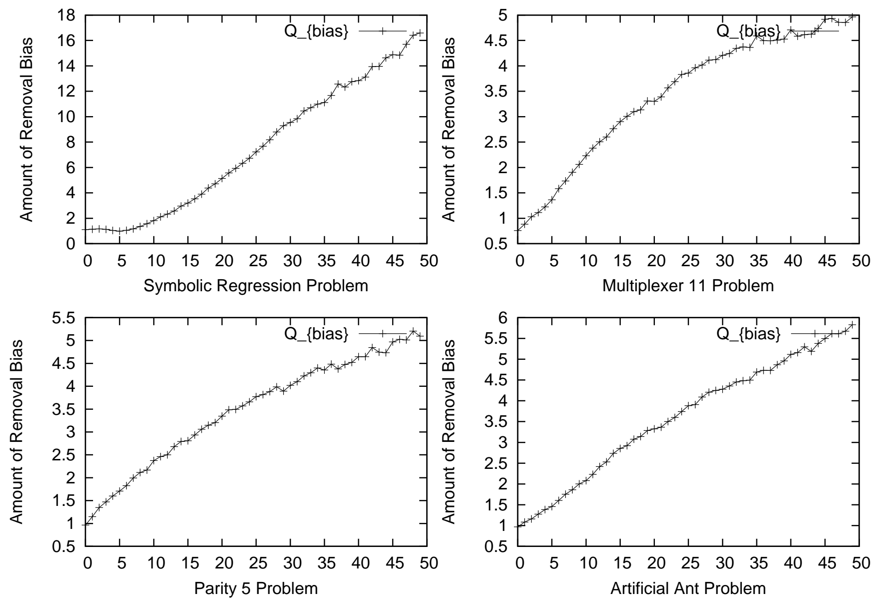


Figure 5.2: Amount of Removal Bias over Generations without Bloating Control

figure clearly shows the existence of the amount of removal bias and it grows over generations. The next step of our work is to analyze if the amount of removal bias can be linked back to bloating. To analyze the relationship between the amount of removal bias and bloating, we perform a correlation analysis between the amount of removal bias and the average depth of the population in each domain using the experiment data. In each problem domain and each generation, we calculate the average amount of removal bias from all crossover operations in that generation and combine it with the average population depth to form a data pair in correlation analysis. Since, for each problem domain, we run 50 tests and each test runs for 50 generations, this gives us in total approximately 2,500 data pairs in correlation analysis for each problem domain. The result of the correlation analysis is as shown in Table 5.2, in which, we can see that there is a strong correlation between the amount of removal bias and the average depth of the population across all four testing domains. Since the average depth of

	Ant	11-Multiplexer	Even-5	Regression
Correlation	0.9023	0.8852	0.9307	0.9519

Table 5.2: Correlation between Amount of Removal Bias and Average Depth of the Generation

population directly correlates to bloating, this result shows that there is a strong correlation between the amount of removal bias and the code bloating.

The first experiment shows that *there is a strong correlation between the amount of removal bias and the bloating*. In the next experiment, we verify whether this relationship still holds when there exists the parsimony pressure. In the second experiment, we use the same configuration settings as the first experiment but use Koza-style depth limiting to explicitly control bloating. In depth limiting method, the maximum depth allowed is set to 17. Similar to Table 5.1, Table 5.3 summarizes the amount of removal bias at generation 49 in the second experiment. From Table 5.3, we can see that, similar to results observed from the

Problem	Q_{bias}	$\sigma_{Q_{\text{bias}}}$	Parent Diff	$\sigma_{\text{Parent Diff}}$	Diff2	σ_{Diff2}
Ant	3.03	3.46	1.86	2.06	-0.04	5.27
11-Multiplexer	3.03	3.35	1.21	1.46	0.06	4.83
Even-5	2.82	3.23	0.84	1.09	0.05	4.47
Regression	3.05	3.63	0.74	1.14	-0.04	4.85

Table 5.3: Removal Bias at Generation 49 with Depth-Limiting Method

first experiment, Diff2 is quite close to 0. Comparing Table 5.1 and Table 5.3, we can find that, in the second experiment in which the bloating is controlled by Koza-Style depth limiting method, the amount of removal bias has been reduced considerably across all problem domains. In Figure 5.3, we plot how the amount of removal bias changes over generations along with the average depth of the population. From Figure 5.3, we can see that both the average amount of removal bias and the average depth of the population increase in the same trend. Because of the depth limiting method, the average depth of the population is controlled, and as a result, the average amount of removal bias has been reduced. Similar to Table 5.2, Table 5.4 gives the correlation analysis results between the average amount of removal bias from all crossover operations in one generation and the average population depth of the same generation across four problem domains in the second experiment. This quantitative result confirms the observation in Figure 5.3.

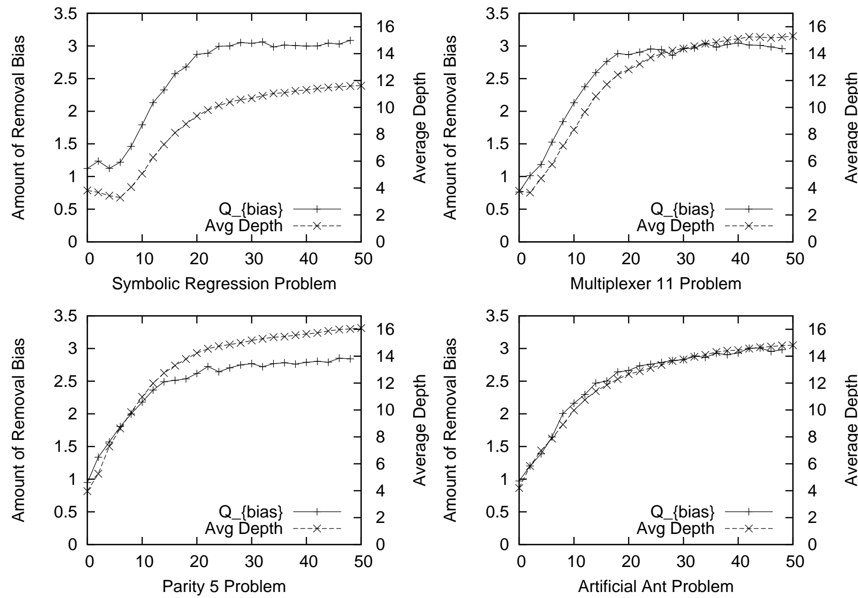


Figure 5.3: Amount of Removal Bias over Generations with Depth-Limiting Method

	Ant	11-Multiplexer	Even-5	Regression
Correlation	0.8205	0.8533	0.8797	0.9581

Table 5.4: Correlation between Amount of Removal Bias and Average Depth of the Generation In Experiment 2

In conclusion, the above two experiment results show that, our definition of the amount of removal bias:

1. can be widely observed in different GP problems;
2. increases over generations;
3. has a strong correlation to the average depth of the population.

It is also showed in the second experiment that, the amount of removal bias can be reduced by controlling the average depth of the population.

5.4 Depth Constraint Crossover

In the last section, we develop a quantitative model called the amount of removal bias. Empirical results in the last section demonstrate the strong correlation

between the amount of removal bias defined and the average depth of the population. It also shows that controlling bloating is able to reduce the amount of removal bias. Although this result is quite interesting, it is not sufficient enough to support the *depth difference hypothesis*. This is because, in depth difference hypothesis, we are more interested in whether the "inverse" relationship holds, i.e. whether *reducing the amount of the removal bias is able to reduce the amount of bloating*. In this section, we use the amount of removal bias to develop a depth constraint crossover to support the depth difference hypothesis.

Depth constraint crossover implements a very simple strategy to *directly* control the amount of removal bias by modifying the standard crossover. In depth constraint crossover, an additional constraint is implemented in the selection of crossover points such that the depth difference between subtree A and subtree B swapped must be no bigger than a predefined threshold ϵ, i.e:

$$Q_{\text{bias}} = |\text{Depth}(\text{Subtree A}) - \text{Depth}(\text{Subtree B})| \leq \epsilon$$

where $\epsilon \in \mathbb{Z}^*$.

The code implementation of depth constraint crossover is relatively simple as well. It can be implemented as in Algorithm 6. In the implementation, if depth of parent1 is smaller than parent2, parent1 and parent2 are swapped such that the while loop is guaranteed to be able to find suitable crossover point in parent2 full-filling the constraint. One thing to note is that, the selection of crossover point1 and point2 in the pseudo code not necessarily needs to be random. Any existing crossover point selection methods can be used as long as the selected pair of points full-fill the constraint. Another note is that this implementation could potentially slow down GP if too many retries are required to find a suitable crossover point in parent2 especially when both d1 and size of parent2 are big. However, our experiment shows that there is very little performance difference between depth constraint crossover and normal crossover in terms of running time. This is because, the depth constraint crossover is able to effectively limit the growth in size for the population such that it never needs to face big individuals. As a result, our implementation of depth constraint crossover follows the same logic as in Algorithm 6. Modifications of Algorithm 6 may be required if poor runtime performance is observed especially when the initial population contains big individuals.

Algorithm 6 Depth Constraint Crossover

1: **function** crossover(individual parent1, individual parent2, param ϵ) : offspring1, offspring2
2: **if** depth(parent1) < depth(parent2) **then**
3:    parent1 ↔ parent2
4: **end if**
5: point1 ← random-select(parent1)
6: point2 ← random-select(parent2)
7: d1 ← depth-subtree-whose-root-is(point1)
8: **while** d1 > depth(parent2) + ϵ **do**
9:    point1 ← random-select(parent1)
10:    d1 ← depth-subtree-whose-root-is(point1)
11: **end while**
12: d2 ← depth-subtree-whose-root-is(point2)
13: **while** $|d1 - d2| > \epsilon$ **do**
14:    point2 ← random-select(parent2)
15:    d2 ← depth-subtree-whose-root-is(point2)
16: **end while**
17: offspring1, offspring2 ↔ subtree-swap(parent1, parent2, point1, point2)
18: **end function**

Next, we perform experiments to compare the performance of depth constraint crossover with Koza-style depth limiting method. We choose Koza-style depth limiting method as the baseline to test depth constraint crossover performance because it is the most widely accepted bloating control method. In the first experiment, we compare depth constraint crossover against depth limiting using four problem domains: Artificial Ant with Santa Fe food trail, 11 Boolean Multiplexer, Even-5 Parity and Symbolic Regression of $x^6 - 2x^4 + x^2$. Similar to experiments in the previous section, the population size used is 500 and GP runs for 50 generations. Tournament selection of size 5 is used. Only crossover and reproduction operator are used in breeding phase with 90% and 10% probability respectively. For depth limiting method, the maximum depth threshold is set to 17. For depth constraint crossover, we experiment with threshold parameter ϵ ranges from 0 to 5 in step of 1.

The experiment results can be found in Table 5.5. In the table, each row represents experiment result for a specific parameter. For each experiment, the first row is the statistics at generation 0. The second row is the statistics at generation 50. For each experiment parameter setting, we collect the Size of Run, which is the average tree size for all individuals, the Depth of Run, which is the

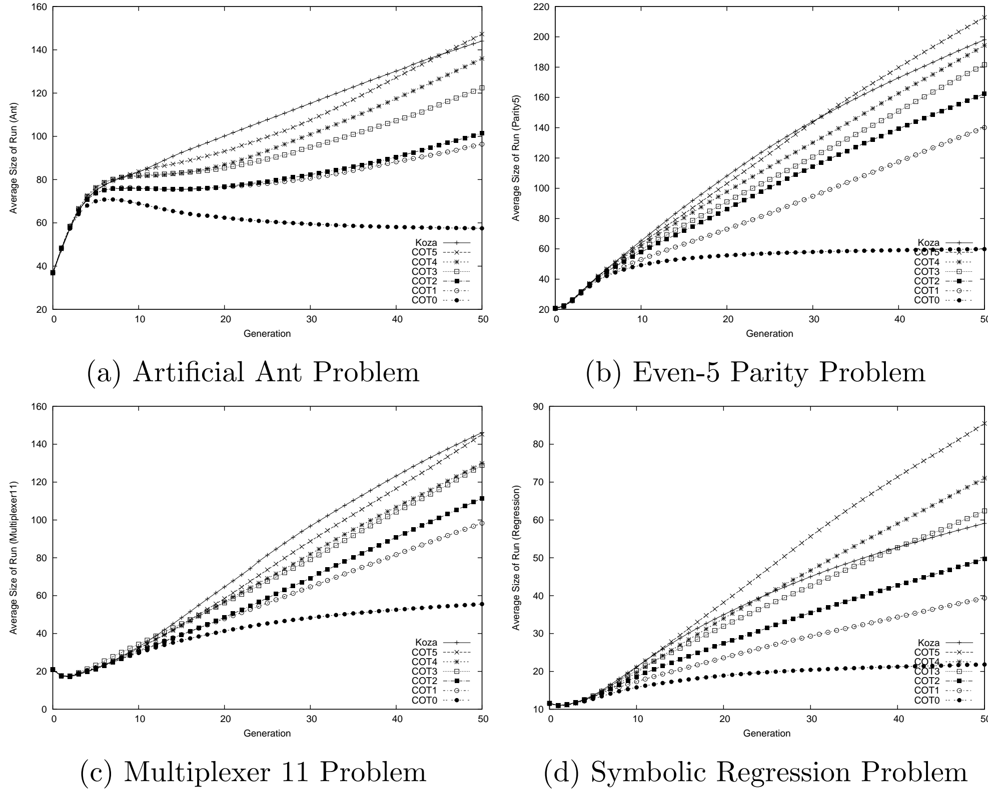
Artificial Ant Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
Koza	0	36.5072	2.1070	4.1998	0.0539	58.3000	7.0150
	50	144.0344	43.0130	12.1597	0.6649	29.1020	9.5110
$\epsilon = 5$	0	36.7016	1.9822	4.2029	0.0476	59.2000	5.0239
	50	147.2643 →	43.3230	13.9536	2.2482	27.1667 →	9.4237
$\epsilon = 4$	0	37.1428	1.3010	4.2172	0.0419	58.1200	8.8806
	50	135.9783 →	36.6117	13.0163	1.8166	27.8000 →	10.5527
$\epsilon = 3$	0	37.0173	1.7962	4.2124	0.0459	59.6600	8.5758
	50	122.4274 →	45.2742	11.9634	1.6146	27.2128 →	9.3875
$\epsilon = 2$	0	37.0212	1.6986	4.2073	0.0457	59.6600	7.1934
	50	101.4703 ↑	29.4275	10.2296	1.01850	28.7959 →	8.1365
$\epsilon = 1$	0	36.8406	1.7274	4.2100	0.0537	60.1800	5.9184
	50	96.3762 ↑	41.4020	8.4120	0.7659	29.5417 →	8.4260
$\epsilon = 0$	0	36.6874	1.8300	4.2010	0.0526	59.2600	7.3670
	50	57.4887 ↑	23.5606	5.3487	0.5224	31.6531 →	7.7947

Symbolic Regression Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
Koza	0	11.5321	0.2897	4.5721	0.0350	1.0706	0.3470
	50	59.0995	15.5746	12.5412	0.5956	0.0880	0.0646
$\epsilon = 5$	0	11.5594	0.3379	4.5715	0.0500	1.0836	0.3383
	50	85.4887 ↓	21.3514	17.8416	3.1305	0.0953 →	0.1005
$\epsilon = 4$	0	11.4974	0.2285	4.5677	0.0344	1.1188	0.3201
	50	71.0383 →	18.8290	16.2361	2.5738	0.0896 →	0.0723
$\epsilon = 3$	0	11.5854	0.3083	4.5760	0.0356	1.0544	0.3645
	50	62.3953 →	11.9895	14.3390	1.8280	0.0890 →	0.0862
$\epsilon = 2$	0	11.5956	0.2933	4.5719	0.0429	1.0928	0.3429
	50	49.7187 ↑	11.0970	11.8744	1.2432	0.0917 →	0.0976
$\epsilon = 1$	0	11.4577	0.3261	4.5636	0.0451	1.1120	0.3670
	50	39.3750 ↑	13.8156	9.1164	1.2259	0.0980 →	0.0884
$\epsilon = 0$	0	11.5370	0.3612	4.5710	0.0457	1.2310	0.3567
	50	21.8227 ↑	7.7429	5.3728	0.5544	0.2689 ↓	0.1808

Multiplexer 11 Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
Koza	0	20.8731	1.1156	3.7884	0.0549	766.3200	18.1035
	50	146.1525	35.4965	12.0101	0.6416	256.3878	85.8078
$\epsilon = 5$	0	21.0576	1.1638	3.8044	0.0514	769.9600	22.0708
	50	145.2237 →	37.8773	12.7697	1.5776	249.3400 →	90.1873
$\epsilon = 4$	0	20.9180	1.0904	3.7868	0.0637	768.0000	15.2630
	50	129.9460 →	40.5407	12.2724	1.5458	247.9600 →	77.9215
$\epsilon = 3$	0	20.9720	1.1065	3.7906	0.0534	770.1600	21.3311
	50	128.8608 →	36.8215	11.2843	1.1211	239.9600 →	80.3241
$\epsilon = 2$	0	20.7576	1.2816	3.7830	0.0676	765.9200	21.0939
	50	111.4878 ↑	33.1141	10.0520	1.0513	248.4082 →	71.6055
$\epsilon = 1$	0	20.9818	0.9284	3.7880	0.0441	770.1000	20.3590
	50	98.2923 ↑	31.7001	8.4215	0.6990	221.6000 →	87.5808
$\epsilon = 0$	0	20.8796	0.9276	3.7901	0.0575	770.6600	19.2433
	50	55.5466 ↑	24.6967	4.9741	0.5823	250.5600 →	97.7401

Even-5 Parity Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
Koza	0	20.6750	0.8614	3.9663	0.0502	13.9400	0.5444
	50	198.0517	34.8353	13.2706	0.4673	6.4600	1.7574
$\epsilon = 5$	0	20.6471	0.6698	3.9667	0.0450	14.0800	0.4400
	50	212.8723 →	51.8730	15.0923	1.6717	6.5400 →	1.8784
$\epsilon = 4$	0	20.7148	0.8290	3.9665	0.0479	13.8800	0.5154
	50	194.3445 →	30.3713	13.9462	1.4281	5.8000 →	1.3115
$\epsilon = 3$	0	20.7821	0.8588	3.9688	0.0523	14.0200	0.5828
	50	181.5988 →	42.1715	13.0648	1.3508	6.0400 →	1.5615
$\epsilon = 2$	0	20.9270	0.7650	3.9765	0.0517	14.1200	0.4308
	50	162.4306 ↑	34.0961	11.3054	0.8166	5.8000 →	1.9596
$\epsilon = 1$	0	20.5198	0.9599	3.9604	0.0551	14.1800	0.5546
	50	140.1154 ↑	28.1080	9.39379	0.5782	4.9400 ↑	2.0041
$\epsilon = 0$	0	20.5762	0.7667	3.9624	0.0426	14.1600	0.5783
	50	59.8392 ↑	0.8709	5.8739	0.0149	7.0400 →	1.3410

Table 5.5: Summary of Experiment Results Comparing Koza-Style Depth Limiting Method and Depth Constraint Crossover



(a) Artificial Ant Problem

(b) Even-5 Parity Problem

(c) Multiplexer 11 Problem

(d) Symbolic Regression Problem

Figure 5.4: Size of Run Changes over 50 Generations for Depth Constraint Crossover and Koza-style Depth Limiting

average depth for all individuals, and Raw Fitness, which is the best raw fitness achieved in the run. All these values in the table are an average of 50 independent runs. We use Size of Run as the criterion to compare the performance of depth constraint crossover and depth limiting method. Best raw fitness is presented to ensure that the depth constraint crossover does not harm GP performance when controlling bloating. For all values in the table, smaller values are better. To establish means for statistical significance, we perform T-test with a 99% confidence interval on size of run and raw fitness. In the table, $\uparrow$ represents mean value which is statistically superior to Koza depth limiting baseline, $\downarrow$ represents mean value which is statistically inferior, and $\rightarrow$ represents no statistically significant difference.

From Table 5.5, we haven't observed any decrease in GP performance in terms of best raw fitness achieved when depth constraint crossover is used except for one case when $\epsilon = 0$ in symbolic regression problem, in which there is around 20% decrease in fitness. We think this is mainly due to to the fact that depth

constraint crossover with ϵ value 0 is very special. When the threshold value ϵ is set to 0, the subtrees swapped in and out must have exactly the same depth. As a result, the crossover cannot produce offsprings with different depths from their parents. In our experiment, the initial population is generated using Koza's ramp-half-and-half method with the maximum depth of 6. This means that, for all problems, when depth constraint crossover with $\epsilon = 0$ is used, the maximum depth of individuals cannot exceed 6. For symbolic regression problem, since most of functions' arity is 1, the initial population would be much smaller in terms of size compared to initial populations of other problem domains. This makes depth constraint crossover with $\epsilon = 0$ hard to explore the searching space to archive the same level of performance as other parameter settings.

In addition to Table 5.5, Figure 5.4 shows how the average size of run changes over generations. We can see that in artificial ant, multiplexer 11 and even-5 parity problems, depth constraint crossover with ϵ value 5 produces the amount of bloat quite similar to Koza-style depth limiting method. But, on symbolic regression problem however, depth constraint crossover with ϵ value 5 produces considerably more bloating (on average 85.49) compared to depth limiting method (on average 59.10). We think this is because in symbolic regression problem, the bloating force is much stronger. This can be partially supported by results in Table 5.1, in which we can see that the amount of removal bias is two times bigger in regression problem compared to values in the other three problems. A smaller ϵ threshold value is required for depth constraint crossover to properly control bloating in symbolic regression problem. Across all four benchmark problem domains, ϵ value 1 and 2 produce less bloating compared to depth-limiting method in a statistically significant manner. ϵ equals 3 and 4 produce less bloating in boolean domains but not in a statistically significant manner. As a special case, ϵ value 0 completely denies the ability of offsprings to have depths different from their parents. This represents the strictest control over increase in population's average depth. As a result, depth constraint crossover with ϵ value 0 produces minimal amount of bloating compared to all other threshold settings. But in regression problem, we see worse fitness, and in the other three domains, we see worse performance although not in a statistically significant manner. Overall, depth constraint crossover with ϵ value 1 gives the optimal control of bloating without scarifying fitness. With ϵ value 1, depth constraint crossover produces on average 32.11% less bloating compared to depth limiting method.

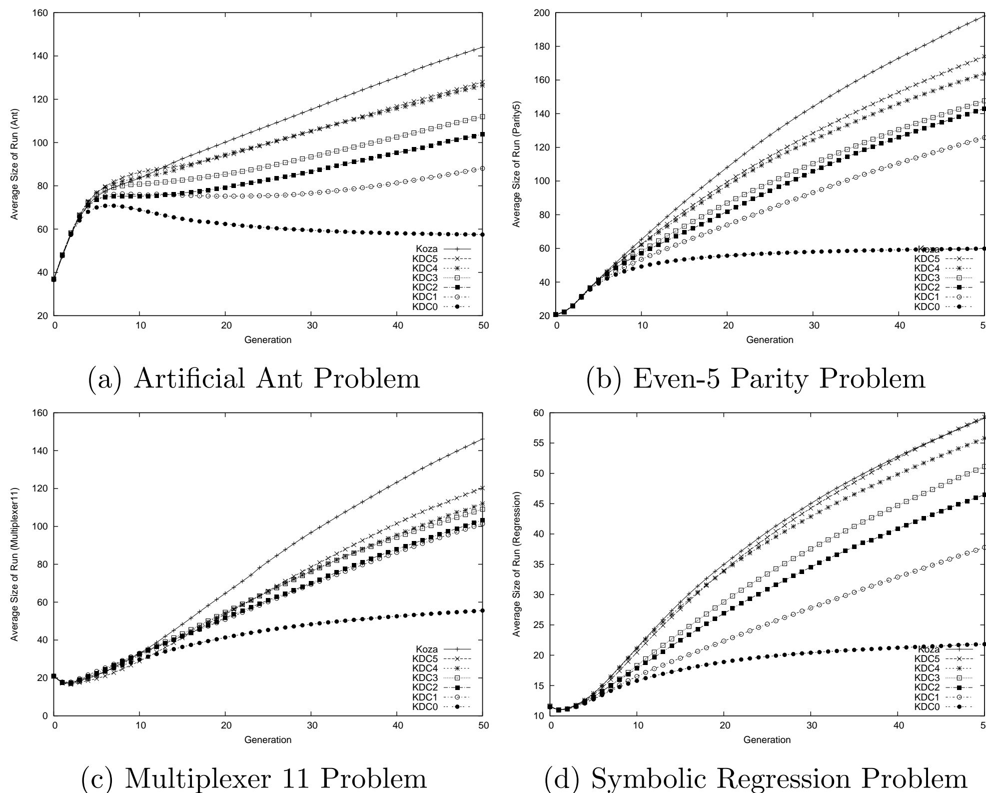
ϵ	Problem	Q_{bias}	$\sigma_{Q_{\text{bias}}}$	Parent Diff	$\sigma_{\text{Parent Diff}}$	Diff2	σ_{Diff2}
$\epsilon = 0$	Ant	0.0000	0.0000	0.0045	0.0668	0.0010	0.0669
	Regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	Multiplexer 11	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	Even-5 Parity	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
$\epsilon = 1$	Ant	0.6050	0.4889	1.4022	1.3147	0.0082	2.1025
	Regression	0.6231	0.4846	1.3197	1.5210	0.0167	2.1669
	Multiplexer 11	0.5885	0.4921	1.0183	1.0861	0.0513	1.7030
	Even-5 Parity	0.5875	0.4923	0.9714	0.9365	−0.0194	1.5752
$\epsilon = 2$	Ant	0.9322	0.7610	2.4339	2.2833	0.0304	3.5893
	Regression	1.0519	0.7633	2.2581	2.2977	0.0476	3.5091
	Multiplexer 11	0.9552	0.7641	1.8888	1.8476	0.0350	2.9711
	Even-5 Parity	0.9431	0.7547	1.5355	1.4499	0.0185	2.4754
$\epsilon = 3$	Ant	1.2224	1.0291	3.3482	3.0197	0.0525	4.8368
	Regression	1.4034	1.0403	3.1234	2.7446	−0.0458	4.5966
	Multiplexer 11	1.2513	1.0291	2.4395	2.4017	−0.0165	3.8461
	Even-5 Parity	1.2396	1.0266	2.1169	2.0974	−0.0203	3.4589
$\epsilon = 4$	Ant	1.4488	1.2831	4.0871	3.6243	0.0270	5.8586
	Regression	1.7693	1.3218	3.6898	3.3595	0.0246	5.5345
	Multiplexer 11	1.5126	1.3097	3.0394	3.0919	−0.0165	4.8461
	Even-5 Parity	1.4689	1.2825	2.7682	2.7683	−0.0269	4.4721
$\epsilon = 5$	Ant	1.6823	1.5711	4.8652	4.3124	0.0006	6.9865
	Regression	2.0486	1.5924	4.0663	3.7846	−0.0290	6.2411
	Multiplexer 11	1.7242	1.5652	3.4812	3.4591	0.0628	5.5637
	Even-5 Parity	1.6916	1.5360	3.3472	3.2382	0.0213	5.2655

Table 5.6: Removal Bias at Generation 49 with Depth Constraint Crossover

Table 5.6 summarises the amount of removal bias at generation 49 when depth constraint crossover is used in the experiment. Comparing data in Table 5.6 to data in Table 5.3, we can see that depth constraint crossover can effectively reduce the amount of removal bias and the amount of removal bias is positive related to the threshold parameter ϵ. The effectiveness of depth constraint crossover in reducing bloat shows that *reducing the amount of removal bias can be used as an effective way to control bloating.* This experiment result supports the *depth difference hypothesis*.

In [LP06], Luke shows that combining depth limiting with other bloating control methods always results in better control of bloat. Based on this observation, in the next experiment, we compare depth constraint crossover in combination with depth limiting against plain depth limiting. The experiment setup is the same as last experiment. The only difference is that in addition to depth constraint crossover, which adds constraints to selected crossover points, the depth of offsprings created also cannot exceeds 17. Experiment results are summarized in Table 5.7. Compared to experiment results in Table 5.5, we can find that depth

constraint crossover combined with depth limiting is able to significantly outperform plain depth constraint crossover when threshold parameter ϵ is relatively big (3, 4, 5 in the experiment) in control bloating. However, when threshold is smaller (2 or 1 in the experiment) the improvement is relatively small. This is due to the fact that Koza-style depth limiting only becomes effective when offsprings' depth reaches 17. When ϵ is relatively small, the population's average depth increases in a much slower manner. As a result, Koza-style depth limiting is less effective. An extreme case is when ϵ equals 0, in which the population's average depth cannot grow beyond 6. In this case, the Koza-style depth limiting is not at all effective, and therefore, the experiment gives the same result as previous experiment. Similar to Figure 5.4, Figure 5.5 shows how size of run changes over generations. Same as previous experiment, threshold ϵ value 1 gives optimal control of bloating without loss of fitness. With ϵ value 1, depth constraint crossover combined with depth limiting produces on average 35.51% less bloating compared to plain depth limiting method.



(a) Artificial Ant Problem

(b) Even-5 Parity Problem

(c) Multiplexer 11 Problem

(d) Symbolic Regression Problem

Figure 5.5: Size of Run Changes over 50 Generations for Depth Constraint Crossover combined with Depth Limiting and Koza-style Depth Limiting

Artificial Ant Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	$\sigma_{fitness}$
Koza	0	36.5072	2.1070	4.1998	0.0539	58.3000	7.0150
	50	144.0344	43.0130	12.1597	0.6649	29.1020	9.5110
$\epsilon = 5$	0	37.1081	2.2816	4.2111	0.0530	59.3200	7.9207
	50	127.9321 →	43.3762	12.0351	0.8574	28.6939 →	10.2941
$\epsilon = 4$	0	36.7764	1.7009	4.2013	0.0456	58.3000	8.8414
	50	126.4094 →	34.8137	11.8947	0.8183	29.1042 →	6.3022
$\epsilon = 3$	0	36.8772	1.7609	4.2164	0.0460	60.1400	5.7654
	50	111.9589 ↑	29.4106	11.0159	0.9195	27.8400 →	8.9986
$\epsilon = 2$	0	36.8946	1.9465	4.2053	0.0509	59.9000	8.3120
	50	103.8140 ↑	39.2420	10.2524	0.9907	28.4490 →	8.2563
$\epsilon = 1$	0	36.6100	1.9817	4.2049	0.0441	60.4200	6.5271
	50	88.0542 ↑	25.4204	8.5616	0.6602	27.5918 →	8.7387
$\epsilon = 0$	0	36.6874	1.8300	4.2010	0.0526	59.2600	7.3670
	50	57.4887 ↑	23.5606	5.3487	0.5224	31.6531 →	7.7947

Symbolic Regression Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	$\sigma_{fitness}$
Koza	0	11.5321	0.2897	4.5721	0.0350	1.0706	0.3470
	50	59.0995	15.5746	12.5412	0.5956	0.0880	0.0646
$\epsilon = 5$	0	11.5190	0.3760	4.5731	0.0428	1.0786	0.2997
	50	59.2834 →	14.5976	13.1756	0.6022	0.0638 →	0.0492
$\epsilon = 4$	0	11.4683	0.2849	4.5726	0.0391	1.0788	0.3582
	50	55.8001 →	11.9861	13.1431	0.6172	0.0761 →	0.0651
$\epsilon = 3$	0	11.6088	0.3364	4.5708	0.0473	1.1353	0.4123
	50	51.1228 ↑	11.2239	12.3284	0.8633	0.0777 →	0.0601
$\epsilon = 2$	0	11.5346	0.2796	4.5768	0.0450	1.2460	0.3070
	50	46.4787 ↑	12.4366	11.4310	0.8801	0.0763 →	0.0693
$\epsilon = 1$	0	11.4620	0.3341	4.5680	0.0421	1.0802	0.3140
	50	37.7814 ↑	10.2628	9.5780	1.0445	0.0935 →	0.0790
$\epsilon = 0$	0	11.5370	0.3612	4.5710	0.0457	1.2310	0.3567
	50	21.8227 ↑	7.7429	5.3728	0.5544	0.2689 ↓	0.1808

Multiplexer 11 Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	$\sigma_{fitness}$
Koza	0	20.8731	1.1156	3.7884	0.0549	766.3200	18.1035
	50	146.1525	35.4965	12.0101	0.6416	256.3878	85.8078
$\epsilon = 5$	0	20.8292	1.1571	3.8014	0.0613	770.0800	16.2232
	50	120.2280 →	29.4349	11.3901	0.6936	245.4286 →	91.7379
$\epsilon = 4$	0	20.9148	1.0941	3.7932	0.0550	764.8800	26.1631
	50	112.0201 ↑	27.1307	10.9528	0.7222	256.7347 →	90.1404
$\epsilon = 3$	0	20.9527	1.0022	3.7885	0.0570	765.3600	23.5370
	50	109.0671 ↑	31.1328	10.7898	0.9347	266.7200 →	87.9161
$\epsilon = 2$	0	20.8327	0.9657	3.7904	0.0564	767.2800	18.2768
	50	103.2645 ↑	31.2748	9.7734	0.8896	254.1200 →	89.4341
$\epsilon = 1$	0	21.0098	1.0512	3.7902	0.0605	772.2600	27.2755
	50	101.4085 ↑	30.5448	8.2726	0.6367	225.8000 →	86.4095
$\epsilon = 0$	0	20.8796	0.9276	3.7901	0.0575	770.6600	19.2433
	50	55.5466 ↑	24.6967	4.9741	0.5823	250.5600 →	97.7401

Even-5 Parity Problem							
Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	$\sigma_{fitness}$
Koza	0	20.6750	0.8614	3.9663	0.0502	13.9400	0.5444
	50	198.0517	34.8353	13.2706	0.4673	6.4600	1.7574
$\epsilon = 5$	0	20.8728	0.8094	3.9752	0.0524	14.0600	0.5064
	50	173.9904 ↑	32.4364	13.0044	0.5505	6.1000 →	1.6882
$\epsilon = 4$	0	20.7510	0.9450	3.9757	0.0569	14.0400	0.5276
	50	163.7528 ↑	30.0571	12.5871	0.5832	6.4200 →	1.7099
$\epsilon = 3$	0	20.7396	0.8875	3.9715	0.0462	14.1400	0.6003
	50	147.6448 ↑	23.9076	11.7555	0.6925	5.9800 →	1.8920
$\epsilon = 2$	0	20.4891	0.6922	3.9559	0.0442	14.0400	0.5276
	50	143.0514 ↑	24.2787	10.8385	0.6220	5.8000 →	1.5875
$\epsilon = 1$	0	20.5921	0.8109	3.9578	0.0463	14.0200	0.6161
	50	125.7805 ↑	25.8096	9.2304	0.4777	5.2200 ↑	1.8143
$\epsilon = 0$	0	20.5762	0.7667	3.9624	0.0426	14.1600	0.5783
	50	59.8392 ↑	0.8709	5.8739	0.0149	7.0400 →	1.3410

Table 5.7: Summary of Experiment Results Comparing Koza-Style Depth Limiting Method and Depth Constraint Crossover combined with Depth Limiting Method

From Figure 5.4 and Figure 5.5, we can clearly see that, if only size of run is considered, parameter ϵ value 0 produces the least amount of bloating compared to other ϵ values. If we compare the size of run at generation 50 between depth constraint crossover with ϵ value 0 and depth limiting method, depth constraint crossover with ϵ value 0 produces 63.73% less bloat. When compared to depth constraint crossover with ϵ value 1, ϵ value 0 produces 46.43% less bloat. However, as we discussed earlier, setting ϵ to 0 does result in worse best fitness achieved, especially in regression problem.

In order to leverage the superior performance advantage in controlling bloating for ϵ value 0, we modify the depth constraint crossover to allow parameter ϵ to be real number in the range between 0 and 1 ($\epsilon \in [0, 1]$). The implementation of depth constraint crossover is changed as follows to handle real numbered ϵ value. Similar as in Algorithm 6, after the crossover point is selected in the second parent and the amount of removal bias ($|d1 - d2|$) is calculated, if the amount of removal bias is bigger than 1, then the crossover point is rejected immediately. If the amount of removal bias equals 0, then the crossover point is accepted. If the amount of removal bias equals 1, then a random number between 0 and 1 is generated. If the random number is smaller than the real numbered threshold ϵ, then the crossover point is accepted, otherwise the crossover point is rejected and a new one is selected.

In the next experiment, we test the performance of depth constraint crossover with real numbered ϵ. We use the same setting as previous two experiments. In this experiment, we test the following ϵ values: 0.5, 0.1, 0.01, 0.005 and 0.001. The experiment result is summarized in Table 5.8. In the table, in order to establish means for statistical significance, we perform T-test with a 99% confidence interval on size of run and raw fitness. In the table, $\uparrow$ represents mean value which is statistically superior to depth constraint crossover combined with depth limiting with threshold 1, $\downarrow$ represents mean value which is statistically inferior, and $\rightarrow$ represents no statistically significant difference.

From Table 5.8, we can find that when ϵ equals 0.5, there is very little performance difference compared to depth constraint crossover combined with depth limiting with threshold 1. Performance improvements in terms of average size of run can be observed when ϵ equals 0.1 in symbolic regression and parity problem, however not in a statistical significant manner in artificial ant and multiplexer 11 problem. ϵ value 0.001 gives very similar performance compared to when ϵ

Artificial Ant Problem

Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
$\epsilon = 1$	0	36.6100	1.9817	4.2049	0.0441	60.4200	6.5271
	50	88.0542	25.4204	8.5616	0.6602	27.5918	8.7387
$\epsilon = 0.5$	0	37.3096	1.9525	4.2177	0.0557	59.6400	6.5750
	50	81.4541 →	29.4061	8.0549	0.7041	24.3125 →	11.0475
$\epsilon = 0.1$	0	36.7370	1.4787	4.2032	0.04163	59.0600	5.3307
	50	80.4786 →	32.4146	6.9326	0.5944	28.6531 →	8.6769
$\epsilon = 0.01$	0	37.0866	1.9283	4.2144	0.0494	60.6600	4.7691
	50	67.5618 →	23.0569	5.7645	0.4947	31.2708 →	7.6151
$\epsilon = 0.005$	0	36.5176	1.9974	4.1939	0.0474	59.4000	6.8264
	50	64.1212 ↑	26.2522	5.6577	0.4206	29.2857 →	8.5905
$\epsilon = 0.001$	0	36.7023	1.8154	4.1985	0.05598	59.3800	6.5173
	50	61.8453 ↑	27.2933	5.4212	0.5685	30.5600 →	6.5212
$\epsilon = 0$	0	36.6874	1.8300	4.2010	0.0526	59.2600	7.3670
	50	57.4887 ↑	23.5606	5.3487	0.5224	31.6531 →	7.7947

Symbolic Regression Problem

Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
$\epsilon = 1$	0	11.4620	0.3341	4.5680	0.04207	1.0802	0.3140
	50	37.7814	10.2628	9.5780	1.0445	0.0935	0.0790
$\epsilon = 0.5$	0	11.5874	0.3307	4.5803	0.0439	1.1227	0.3468
	50	32.3185 →	11.2485	8.2206	0.9553	0.1142 →	0.1112
$\epsilon = 0.1$	0	11.5194	0.3229	4.5741	0.0409	1.0695	0.3870
	50	23.6785 ↑	7.2165	6.8247	1.1137	0.1872 →	0.1650
$\epsilon = 0.01$	0	11.5431	0.3068	4.5724	0.0386	1.1039	0.3581
	50	21.2308 ↑	7.2065	5.8125	0.5882	0.2357 ↓	0.1384
$\epsilon = 0.005$	0	11.5732	0.3030	4.5774	0.0440	1.1778	0.3494
	50	21.1401 ↑	4.8813	5.9088	0.5581	0.2696 ↓	0.1854
$\epsilon = 0.001$	0	11.5080	0.3016	4.5721	0.0445	1.2226	0.2910
	50	23.3780 ↑	6.7325	5.6184	0.5115	0.2182 ↓	0.1309
$\epsilon = 0$	0	11.5370	0.3612	4.5710	0.0457	1.2310	0.3567
	50	21.8227 ↑	7.7429	5.3728	0.5544	0.2689 ↓	0.1808

Multiplexer 11 Problem

Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
$\epsilon = 1$	0	21.0098	1.0512	3.7902	0.0605	772.2600	27.2755
	50	101.4085	30.5448	8.2726	0.6367	225.8000	86.4095
$\epsilon = 0.5$	0	20.7296	0.9292	3.7707	0.0559	766.2800	14.7730
	50	82.9456 →	23.8601	7.8956	0.5120	231.9200 →	75.8071
$\epsilon = 0.1$	0	21.1228	1.0134	3.8033	0.0559	770.4200	16.1667
	50	82.9599 →	25.6596	6.6344	0.5139	223.4792 →	85.8790
$\epsilon = 0.01$	0	21.0441	0.8632	3.7994	0.0501	768.4000	14.6697
	50	69.4039 ↑	26.1322	5.6265	0.5296	215.1600 →	98.7620
$\epsilon = 0.005$	0	20.7317	1.1003	3.7914	0.0631	770.0000	26.7791
	50	63.9289 ↑	24.1838	5.4746	0.4537	241.2200 →	88.0312
$\epsilon = 0.001$	0	20.8502	0.9651	3.7783	0.0537	769.0000	15.5653
	50	56.5472 ↑	23.4380	5.0931	0.5314	257.9592 →	95.4649
$\epsilon = 0$	0	20.8796	0.9276	3.7901	0.0575	770.6600	19.2433
	50	55.5466 ↑	24.6967	4.9741	0.5823	250.5600 →	97.7401

Even-5 Parity Problem

Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
$\epsilon = 1$	0	20.5921	0.8109	3.9578	0.0463	14.0200	0.6161
	50	125.7805	25.8096	9.2304	0.4777	5.2200	1.8143
$\epsilon = 0.5$	0	20.4971	0.9301	3.9610	0.0493	14.1400	0.4477
	50	120.6688 →	25.5293	8.7823	0.4863	5.5833 →	2.1779
$\epsilon = 0.1$	0	20.6328	0.8822	3.9616	0.0497	14.1000	0.5000
	50	102.8551 ↑	20.3228	7.7146	0.4112	5.4082 →	1.7487
$\epsilon = 0.01$	0	20.6890	0.9316	3.9628	0.0510	14.1400	0.4903
	50	74.1753 ↑	11.5057	6.4449	0.3135	6.0200 →	2.1118
$\epsilon = 0.005$	0	20.6026	0.9208	3.9654	0.0501	14.1200	0.4308
	50	71.4626 ↑	11.0315	6.2973	0.3337	6.0400 →	1.7996
$\epsilon = 0.001$	0	20.5880	0.7558	3.9658	0.0364	14.1400	0.5295
	50	61.7714 ↑	3.8767	5.9509	0.1318	7.0600 ↓	1.4200
$\epsilon = 0$	0	20.5762	0.7667	3.9624	0.0426	14.1600	0.5783
	50	59.8392 ↑	0.8709	5.8739	0.0149	7.0400 ↓	1.3410

Table 5.8: Summary of Experiment Results of Depth Constraint Crossover with Real Number Threshold Parameter ϵ

equals to 0. In regression problem, ϵ values 0.1, 0.01, 0.005 and 0.001 give better performance in terms of size of run, but we have observed worse best raw fitness achieved. ϵ value 0.5 produces less bloating, but gives worse best raw fitness, however, both of which are not in statistical significant manners. In the other three boolean problem domains, ϵ value 0.005 achieves the best control of bloating without performance loss in terms of best raw fitness achieved. In these three domains, ϵ value 0.005 produces 35.77% less bloating compared to ϵ value 1. It produces 58.55% less bloating compared to Koza style depth limiting method.

5.5 Depth Difference Hypothesis

In the previous section, we developed depth constraint crossover as a new bloating control method. The effectiveness of depth constraint crossover in controlling bloating shows that by reducing the amount of removal bias, the amount of bloating can be reduced. This result, in combination to the observation in Section 5.3 that there is a strong correlation between the amount of removal bias and the average depth of the population, support depth difference hypothesis, *the depth difference between the two subtrees swapped causes the bloating.* In this section, we give a more in-depth discussion about depth difference hypothesis.

Before we dig into the depth difference hypothesis, we would like to firstly briefly review the relationship between individual program tree depth and size. Tree structure is a widely used data structure in computer science. In general tree structure, there is a relative weak positive correlation between depth and size of the tree. This is due to the fact that trees can have different shapes. However, when the tree shape is defined, the tree depth defines the upper bound for tree size. For example, for binary tree, in which each node has at most two child nodes, the maximum number of nodes (size) allowed for a tree with depth d is:

$$s = 2^d - 1.$$

On the other hand, given the size of the tree, it is very hard to work out the depth of the tree even for binary trees. As a result, when controlling bloating, controlling the depth increase is more appropriate compared to controlling the size increase directly, since the former gives a guaranteed upper limit on size.

Now, we know that tree depth places an upper limit on tree size and increasing

in tree depth results in increasing in this threshold, which allows the tree size to grow. Then, how the depth of the tree changes in GP? At the individual program tree level, the only way for an individual to increase or decrease its own depth is via crossover or mutation, which performs structural changes on the individual. Here, we only consider crossover, since mutation is rarely used in GP. In crossover, it is possible for the individual to increase its depth by having subtree of smaller depth removed and subtree with bigger depth inserted. However, only this effect cannot increase the population's average depth. This is because, at the same time, there is another individual's depth decreases by having subtree of bigger depth swapped with subtree with smaller depth.

At the population level, population's average depth is increased over generation by selection favoring deeper trees. Deeper and potentially bigger trees are favored for several reasons. Firstly, they are more resilient to the destructiveness of crossover, based on defensive against crossover theory. Secondly, they are more likely fitter than smaller individuals based on fitness causes bloat theory [LP97b]. Finally, in crossover, the growing individual created by removing smaller subtree and attaching bigger subtree is more likely to be fitter than the shrinking individual. This can be explained using the experiment result of crossover effect in the last chapter. The growing individual, which has deeper crossover point, is more likely to be created by constructive crossover compared to the shrinking individual. So, this combined effect of crossover creating deeper individuals at individual level and the selection favoring deeper individuals at population level represent the fundamental cause of increase in population's average depth over generations. Although both effects are necessary to the formation of bloating, we think the former is relatively more essential since it is the most fundamental mechanism in GP for individual's depth to increase. Based on depth difference hypothesis, in depth constraint crossover, where the depth difference between subtrees swapped in crossover is controlled by a pre-defined threshold, deeper individuals are much less likely to be generated via crossover, and thus effectively controls bloating.

Depth constraint crossover with ϵ value 0 represents the most extreme scenario which gives the strictest control of bloating. When ϵ equals 0, subtrees swapped in crossover must have the same depth. This requirement ensures that offsprings created by crossover inherit the same depth as their parents and hence completely denies crossover's ability to increase individual's depth. This means that, if the initial population is generated using ramp-half-and-half methods with

maximum depth 6, the maximum population depth cannot exceed 6. However, if we review the experiment result in Table 5.5, surprisingly, even with this rather strict restriction, we only observed statistically significant worse performance in regression problem. We think the reason for this is that in regression problem, the function set have relatively small arity. This makes the initial population much smaller compared to other domains, which we believe is also too small to evolve any competitive candidate solutions. The average tree size is around 11 at generation 0 in regression problem, while in multiplexer 11 and parity 5 problems, the average size is around 20 and in ant problem, the average size is 36, which is three times as big as the average in regression problem. Apart from regression problem, there is no decrease in the best raw fitness achieved in the other three boolean problems. This interesting observation suggests that the relationship between increasing in the population's depth/size and finding fitter candidate solutions may be much weaker than previously thought.

If the depth difference hypothesis holds and it is the depth difference between subtree swapped in crossover which drives the bloating, what would be the cause of the depth difference between subtrees swapped in the first place? Intuitively, we think the depth difference between subtrees swapped in crossover is driven by the depth difference between two parents. Assuming that parents selected in crossover have similar shapes, i.e. number of nodes at a given depth are approximately the same, the depth of subtree selected from the parent then depends on the distribution of nodes over depth for the parent tree. If the depth of the two parents are the same, then it is more likely that the depth of subtrees selected are similar. Experiment evidence to support this proposition can be found in Table 5.1, 5.3 and 5.6, in which we record Diff2, which is the difference between parent depth difference and subtrees swapped depth difference. If we review Diff2 across above three experiments, we can see that the average value is very close to 0. This result supports our view that it is the depth difference between parents in crossover which results in the depth difference between subtree swapped. If the depth difference between subtree swapped is driven by the depth difference between parents selected in crossover, instead of controlling the former as in depth constraint crossover, will controlling the later directly feasible? We think even through the former is driven by the later, however, this relationship is not strong enough to give an accurate enough control of bloating. This can be verified by the value of σ_{Diff2} in Table 5.1, 5.3 and 5.6, which are considerably big.

Param	Gen	Size of Run	σ_{size}	Depth of Run	σ_{depth}	Raw Fitness	σ_{fitness}
Artificial Ant Problem							
$\epsilon = 0.005$	0	36.5176	1.9974	4.1939	0.0474	59.4000	6.8264
	50	64.1212	26.2522	5.6577	0.4206	29.2857	8.5905
Double	0	37.0169	1.6276	4.1995	0.0452	60.0800	7.3887
	50	66.7920	19.2286	9.9729	1.1211	22.8958	10.0088
Lexicographic	0	36.6843	1.7532	4.2013	0.0440	61.1200	5.6979
	50	66.8094	31.3975	8.6428	1.8406	25.7708	7.7278
Proportional	0	37.3178	1.8752	4.2168	0.0454	60.1000	7.2422
	50	56.9323	15.8125	8.9785	1.0414	22.8511	8.4501
RatioBucket	0	36.9909	1.4740	4.2107	0.0405	60.7400	6.2284
	50	**50.7024**	20.6484	8.1215	1.6454	**21.4884**	9.1661
Tarpeian	0	36.8238	1.9061	4.2114	0.0461	60.1600	6.0212
	50	79.1673	20.8880	10.3141	0.8706	26.0638	9.0352
Symbolic Regression Problem							
$\epsilon = 1$	0	11.4620	0.3341	4.5680	0.04207	1.0802	0.3140
	50	37.7814	10.2628	9.5780	1.0445	**0.0935**	0.0790
Double	0	11.5423	0.2909	4.5658	0.0458	1.1681	0.3622
	50	33.3359	14.3639	10.2822	2.4552	0.1879	0.2785
Lexicographic	0	11.4922	0.3016	4.5741	0.0406	1.2327	0.3099
	50	57.0514	22.1228	12.0107	2.0388	0.1516	0.2592
Proportional	0	11.5124	0.3594	4.5722	0.0426	1.2201	0.3277
	50	**26.7825**	16.1713	7.7715	3.0394	0.2952	0.3673
RatioBucket	0	11.5324	0.2959	4.5685	0.0435	1.1073	0.3581
	50	44.6117	16.9906	11.1136	2.1283	0.1336	0.2381
Tarpeian	0	11.5136	0.3162	4.5709	0.0410	1.2422	0.3332
	50	33.3742	13.2137	9.7025	2.5560	0.2155	0.2338
Multiplexer 11 Problem							
$\epsilon = 0.005$	0	20.7317	1.1003	3.7914	0.0631	770.0000	26.7791
	50	63.9289	24.1838	5.4746	0.4537	**241.2200**	88.0312
Double	0	20.9084	1.0136	3.7968	0.0555	768.0400	20.5621
	50	73.2230	22.7514	9.8726	1.0185	294.0851	119.2123
Lexicographic	0	20.8380	1.0467	3.7848	0.0613	768.1600	12.1051
	50	97.9815	56.8519	9.6649	2.0350	258.6400	94.3860
Proportional	0	20.9548	1.3185	3.7887	0.0593	765.4800	17.2363
	50	**51.2006**	26.2164	7.7052	1.5114	367.9000	119.0899
RatioBucket	0	20.6647	1.0971	3.7816	0.0555	767.9600	25.6421
	50	84.7025	50.0376	9.1237	2.6173	279.6735	111.1528
Tarpeian	0	20.8227	1.0269	3.7850	0.0574	769.2800	13.1317
	50	73.9358	28.9790	9.7769	1.1089	326.6600	104.2588
Even-5 Parity Problem							
$\epsilon = 0.005$	0	20.6026	0.9208	3.9654	0.0501	14.1200	0.4308
	50	**71.4626**	11.0315	6.2973	0.3337	6.0400	1.7996
Double	0	20.6922	0.8236	3.9691	0.0461	14.0400	0.4454
	50	128.6236	30.1495	11.8982	0.6911	6.2600	1.7528
Lexicographic	0	20.7195	0.9078	3.9760	0.0506	14.2000	0.4899
	50	109.6214	28.5508	10.8912	1.0532	**5.8400**	2.0626
Proportional	0	20.8817	0.7230	3.9847	0.0447	14.0800	0.4833
	50	105.8656	29.3556	10.9360	0.6253	6.9400	1.6782
RatioBucket	0	20.6222	0.8520	3.9652	0.0530	13.9600	0.2800
	50	101.1207	25.1416	10.4136	1.0227	5.9000	1.7916
Tarpeian	0	20.6286	0.9577	3.9665	0.0554	14.1400	0.6328
	50	126.0209	27.4786	11.8730	0.5083	7.1000	1.5133

Table 5.9: Summary of Experiment Results Comparing Depth Constraint Crossover with Other Bloating Control Methods

In the final experiment in this section, we compare depth constraint crossover with a number of existing bloating control methods exist in literature including double tournament [LP02a], proportional tournament [LP02a], lexicographic tournament [LP02b], ratio bucketed tournament [LP06] and Tarpeian selection [Pol03]. These techniques have been compared in detail previously in [LP06] by Luke and Panait. As a result, here for each bloating control method, we use the optimal parameter setting found in [LP06] to compare against depth constraint crossover. For double tournament selection, the first tournament is based on size and the second is based on fitness. The tournament size for the first tournament is 1.4 and the tournament size for the second tournament is 7. For proportional tournament, the tournament size is 7 and 20% time, the proportional tournament selection is based on size and the other 80% is based on fitness. For lexicographic tournament, the tournament size is 7. For ratio bucket tournament selection, the size of tournament is 7 and the number of buckets is 2. For Tarpeian selection, 30% of "big" individuals are killed in every generation. For depth constraint crossover, tournament size is 5. Threshold ϵ is set to 0.005 for boolean problems and 1 for symbolic regression problem. The rest of experiment parameters are the same as previous experiments.

The experiment result can be found in Table 5.9. In artificial ant problem, ratio bucket selection produces least amount of bloating (50.7024). Depth constraint crossover gives similar performance compared to double tournament and lexicographic selection. In symbolic regression problem, proportional selection produces least amount of bloating (26.7825), but with the cost of fitness. Depth constraint crossover produces average amount of bloating (37.7814) but best raw fitness (0.0935). Similarly, in multiplexer 11 problem, depth constraint crossover produces average amount of bloating (63.9289) and best raw fitness (241.2200). In even-5 parity problem, depth constraint crossover produces least amount of bloating (71.4626) and the average raw fitness (6.0400). Overall, the performance of depth constraint crossover is reliable and very competitive across all problem domains.

5.6 Conclusion

In this chapter, we extend the removal bias bloating theory to develop depth difference hypothesis. Depth difference hypothesis blames the depth difference

between subtree swapped in crossover as the root cause of bloating and this depth difference is driven by depth difference of parents selected. In order to support depth difference hypothesis, we quantitatively define the depth difference of subtree swapped as amount of removal bias. Experiment results show that, there is a strong correlation between average depth of the population and average amount of removal bias, and more importantly, controlling the amount of removal bias reduces the average depth of the population and vice visa. This empirical evidence supports the depth difference hypothesis theory. Motivated by depth difference hypothesis theory, depth constraint crossover, which controls the amount of removal bias directly, has been developed as a new bloating control method. A number of experiments conducted show that depth constraint crossover, despite the simplicity, is capable of effectively controlling bloating by controlling the population average depth without affecting GP performance in terms of best fitness achieved. And with the optimal control parameter we found ($\epsilon = 0.005$ which gives best control of bloating while maintaining the performance in terms of best raw fitness archived) via experiments, depth constraint crossover is very competitive compared to other existing bloating control methods.

Chapter 6

Norm-Referenced Fitness Evaluation

6.1 Introduction

Optimization of genetic programming has always been a hot topic in genetic programming research. The goal of the optimization is to improve the efficiency of GP in order to solve more complex problems. A lion share of these optimizations are bloating control methods which we have discussed in detail in previous chapters. Bloating control methods concentrate on bloating. By controlling bloating, genetic programming algorithm can be executed much faster and runs for much more generations, which improves the capacity of GP for more complex problems. Different from bloating control, other optimization methods focus on improving GP's ability to find fitter solutions faster, which directly improve the efficiency of GP. Some of these methods improve GP performance by exploring alternative representations of individuals. Examples of these methods include automated defined functions [Koz94], Linear GP, Graph-based GP and so on. Other methods improve GP performance by developing more efficient genetic operators. One example of these methods is greedy over-selection introduced by Koza in [Koz92]. In greedy over-selection, the population is divided into two groups, Group I and Group II, based on individual's fitness value. Group I's individuals are fitter than Group II's. When an individual is selected, individual is selected from Group I with 80% probability. In this way, greedy over-selection greatly improves the selection intensity, and hence improves the convergence speed of GP.

In this chapter, we explore an alternative of the original fitness function, norm-referenced fitness function, which is able to improve GP performance. Norm-referenced fitness function is motivated by the concept of norm-referenced test. It evaluates an individual's performance not only based on how well the individual performs, but also taking into account other individuals' performance within the same population. Experiments and analysis show that, norm-referenced fitness function is able to significantly improve GP performance. This work presented in this chapter greatly extends our previous published work in [LZ11]. The rest of this chapter is organized as follows. In the next section, we discuss the limitation of the original fitness function and the motivation of norm-referenced fitness function. After that, we introduce two internal fitness measures and use them to build the norm-referenced fitness function. Then, we study empirically the performance of norm-referenced fitness function firstly in even 5 parity problem in detail and then extend to other GP problem domains and OneMax problem in GA as well. We also use the experiment results to develop the *implicit bias theory*, which forms the theoretical foundation of norm-referenced fitness function. Finally, we conclude this chapter with partial norm-referenced fitness function, which addresses one potential runtime performance limitation of norm-referenced fitness function when used in conjunction with tournament selections.

6.2 Motivation

In educational assessment, there are mainly two different types of tests widely used: criterion-referenced test and norm-referenced test [Bon96]. In criterion-referenced test, as the name implies, exam takers are measured against a number of predefined objective criteria [Bon96]. A typical example of criterion-referenced test is driving test. In norm-referenced test, candidates are measured against a predefined group of exam takers, to give an estimation of their relative positions in that population [Bon96]. Examples of norm-referenced test include IQ test, many entrance exams such as Graduate Record Examinations (GRE) [Wik13]. The main difference between norm- and criterion-referenced test is that, the former tries to show the relative ranking of test subjects while the later assesses the mastery of certain material [Bon96]. As a result, entrance exams are generally norm-referenced exams since institutions are more interested in exam takers'

relative ranking, while most of the end of term diagnostic exams are criterion-referenced tests because lecturers are more interested in whether students have mastered the course material or not.

One limitation of criterion-referenced test is the possibility that exam takers are judged by exam questions which are not appropriate to their level. One extreme example would be testing students in high school Mathematics using Calculus. Every student would probably fail in this test, but this test does not assess those students' mathematical knowledge at all. To avoid this problem, question setters must make sure their expectation matches exam takers' actual level. On the other hand, since norm-referenced test does not seek to enforce any expectation over what exam takers should be able to do, it does not have this limitation.

In GP, the role of fitness function is very similar to an exam. Individuals within population are "exam takers". The fitness value obtained is the exam result. GP system then uses the exam result to guide the selection of parents in breeding phase. For example, in fitness proportionate selection, the probability that an individual is selected is proportional to the fitness of that individual. The exact form of fitness function may vary from problem to problem. Typically, a fitness function consists of a number of test cases. Each test case consists of a number of inputs and a desired output. The test case acts as a "question" in exam. An individual's fitness value for a test case is usually the distance from individual's actual output to the desired output. More formally, let x be an individual, we use $f_i(x)$ to represent the fitness value of individual x for test case i. Then, for a fitness function which consists of n test cases, denoted as $F(x)$, we have:

$$F(x) = \sum_{i=1}^{n} f_i(x). \tag{6.1}$$

The fitness function is a criterion-referenced test. This is because an individual's "absolute" performance is used to judge the quality of that individual. As a result, the fitness function in GP does have the expectation mis-match problem we discussed previously. For example, evaluating the newly generated population using fitness function is pretty similar to test high school students Mathematics using Calculus, as the fitness function is too "hard" to randomly created individuals. The fundamental problem of this miss-match between the expectation and the actual performance is that, it reduces the ability to differentiate between

better exam takers and worse exam takers. In another word, when using criterion-referenced test result to establish the relative ranking of test takers, one needs to ensure that test questions match the exam takers' average level, such that the results are able to adequately differentiate those exam takers' level. On the other hand, a norm-referenced test would be a better fit in this scenario because, firstly norm-referenced test does not enforce any expectation of test subjects' level, and secondly the primary purpose of the norm-referenced test is to give ranking information telling which exam taker performs at an average level, which exam taker does better, and which exam taker does worse.

6.3 Internal Fitness Measure

As we discussed in the previous section, the fundamental problem of the criterion-referenced fitness function used in GP is that, it only subjectively judges the quality of candidate solutions without taking into account other individuals' performance within the same population. To overcome this problem, in this section, we build a more comprehensive fitness function which not only considers individual's subjective raw fitness, but also takes into account the current population's performance. We call the former "external" fitness measure because it is defined by the user and is provided as input into the GP system. We call the later "internal" fitness measure because it is derived from the population and it is independent to the problem GP system is solving.

In this section, we consider two ways to build the "internal" fitness measure. Fitness function is usually a summation of errors from every single test case. Usually, we consider all test cases as a single fitness value. The objective of GP system is to minimize $F(x)$. Now, if we consider each test case independently, for a problem with n test cases, given a population P which consists of m individuals, for each test case, we can calculate the population P's performance:

$$e_i = \frac{\sum_{j=1}^{m} S(f_i(x_j))}{m} \tag{6.2}$$

where $S(.)$ is a scaling function and $S(x) \in [0, 1]$ and $f(.)$ is the same as in (6.1). Let $\boldsymbol{e} = (e_1, e_2, ..., e_n)$, then $\boldsymbol{e}$ represents the population's performance for all test

cases. Normalizing this error vector $\boldsymbol{e}$, we get a weight vector $\boldsymbol{w}^-$, in which:

$$w_i^- = \frac{e_i}{\sum_{i=1}^{n} e_i} \tag{6.3}$$

This weight vector $\boldsymbol{w}^-$ represents the relative importance of test cases using error as the criterion.

Similar to the calculation above, if we use accuracy rather than error as the criterion, we can calculate the population P's performance, $\boldsymbol{a} = (a_1, a_2, ..., a_n)$, where:

$$a_i = 1 - e_i \tag{6.4}$$

Normalizing $\boldsymbol{a}$, we get the weight vector $\boldsymbol{w}^+$, where:

$$w_i^+ = \frac{a_i}{\sum_{i=1}^{n} a_i} \tag{6.5}$$

This weight vector $\boldsymbol{w}^+$ represents the relative importance of test cases using accuracy as the criterion.

We can use $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$ as internal fitness measures to create adjustments of original fitness function. Let $\boldsymbol{f} = (f_1(x), f_2(x), ..., f_n(x))^T$, if only $\boldsymbol{w}^-$ is used, we get:

$$F_{adj}^-(x) = \boldsymbol{w}^- \cdot \boldsymbol{f}$$

If only $\boldsymbol{w}^+$ is used, we get:

$$F_{adj}^+(x) = \boldsymbol{w}^+ \cdot \boldsymbol{f}.$$

In the next, we give a brief example calculating $\boldsymbol{w}^-$, $\boldsymbol{w}^+$, $F_{adj}^-(x)$ and $F_{adj}^+(x)$, using even parity 3 problem domain which has 8 test cases as an example. Considering a population of 4 individuals: $P = \{p_1, p_2, p_3, p_4\}$. The fitness of each individual is listed in table 6.1. For this population P, using (6.2), we get:

Individual	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	F
p_1	0	0	0	0	0	1	1	0	2
p_2	1	0	1	0	1	1	1	0	5
p_3	1	0	1	0	1	0	1	0	4
p_4	0	0	0	0	1	1	1	0	3

Table 6.1: Fitness Values of 4 Individuals in P

$$\begin{aligned} \boldsymbol{e} &= (\frac{2}{8}, \frac{0}{8}, \frac{2}{8}, \frac{0}{8}, \frac{3}{8}, \frac{3}{8}, \frac{4}{8}, \frac{0}{8}) \\ &= (0.25, 0, 0.25, 0, 0.375, 0.375, 0.5, 0) \end{aligned}$$

Where $S(x) = x$ is used as the scaling function. Normalizing $\boldsymbol{e}$ using (6.3), we get:

$$\begin{aligned} \boldsymbol{w}^- &= (\frac{0.25}{1.75}, \frac{0}{1.75}, \frac{0.25}{1.75}, \frac{0}{1.75}, \frac{0.375}{1.75}, \frac{0.375}{1.75}, \frac{0.5}{1.75}, \frac{0}{1.75}) \\ &= (0.143, 0, 0.143, 0, 0.214, 0.214, 0.286, 0) \end{aligned}$$

Similarly, we can also calculate $\boldsymbol{a}$ using (6.4):

$$\begin{aligned} \boldsymbol{a} &= (1 - \frac{2}{8}, 1 - \frac{0}{8}, 1 - \frac{2}{8}, 1 - \frac{0}{8}, 1 - \frac{3}{8}, 1 - \frac{3}{8}, 1 - \frac{4}{8}, 1 - \frac{0}{8}) \\ &= (0.75, 1, 0.75, 1, 0.625, 0.625, 0.5, 1) \end{aligned}$$

Normalizing $\boldsymbol{a}$ using (6.5), we get:

$$\begin{aligned} \boldsymbol{w}^+ &= (\frac{0.75}{6.25}, \frac{1}{6.25}, \frac{0.75}{6.25}, \frac{1}{6.25}, \frac{0.625}{6.25}, \frac{0.625}{6.25}, \frac{0.5}{6.25}, \frac{1}{6.25}) \\ &= (0.12, 0.16, 0.12, 0.16, 0.1, 0.1, 0.08, 0.16) \end{aligned}$$

Using $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$, we can calculate F^-_{adj} and F^+_{adj}. For example, for p_1:

$$\begin{aligned} F^-_{adj}(p_1) &= (0.143, 0, 0.143, 0, 0.214, 0.214, 0.286, 0) \cdot \\ &\quad (0, 0, 0, 0, 0, 1, 1, 0)^T \\ &= 0.5 \\ F^+_{adj}(p_1) &= (0.12, 0.16, 0.12, 0.16, 0.1, 0.1, 0.08, 0.16) \cdot \\ &\quad (0, 0, 0, 0, 0, 1, 1, 0)^T \\ &= 1.08. \end{aligned}$$

F^-_{adj} and F^+_{adj} consider the present population's performance from two different perspectives of view. If we use the examination analogy, in which all test cases are exam questions, using the original fitness function, all "questions" are worth the same mark. In F^-_{adj}, in which the original fitness function is adjusted using $\boldsymbol{w}^-$, "hard" questions which most of students cannot answer are worth more marks. This encourages students to try to solve those hard questions. In F^+_{adj}, where the original fitness function is adjusted using $\boldsymbol{w}^+$, "easy" questions which most of students answer correctly are worth more marks. This promotes students to concentrate on easy questions while ignoring hard ones. The accumulated effect

of F^-_{adj} drifts the population towards unexplored region of searching space. This is because a "hard" test case attracts the population by having a bigger weight. But once the population performs better on that "hard" test case, its weight becomes smaller. Then the population moves its "interest" to other "hard" test cases. In another word, F^-_{adj} destabilizes the evolution by keeping changing the searching direction. On the other hand, the accumulated effect of F^+_{adj} drives the population to converge to the current state of evolution. This is because "easy" test cases, which have bigger weights, have even bigger weights as the population evolves. In another word, F^+_{adj} stabilizes the evolution by enforcing the search direction.

In fact, F^-_{adj} and F^+_{adj} represent two different but equally important aspects of the evolution system: the exploration and exploitation [BT95]. On one hand, F^-_{adj} constantly changes the searching direction to explore the whole searching space. On the other hand, F^+_{adj} drifts the population into convergence, maintaining the stability of the system. The balance between exploration and exploitation is critical to the behavior of GP system [BT95]. As a result, we can combine F^-_{adj} and F^+_{adj} to build a more comprehensive fitness function based on population's performance:

$$\begin{aligned} F_{adj}(x) &= \lambda \cdot F^-_{adj}(x) + (1-\lambda) \cdot F^+_{adj}(x) \\ &= (\lambda \boldsymbol{w}^- + (1-\lambda)\boldsymbol{w}^+) \cdot \boldsymbol{f} \\ &= \boldsymbol{w} \cdot \boldsymbol{f} \end{aligned} \tag{6.6}$$

where λ is a parameter and $\lambda \in [0, 1]$. We call this new fitness function (6.6) the norm-referenced fitness function because it not only considers the raw fitness value $\boldsymbol{f}$, but also takes into account population's performance. Similar to the norm-referenced test, which gives relative rank of test taker, F_{adj} gives an individual's fitness value which is relative to the present population's performance.

One inspiration of norm-referenced fitness function is boosting algorithm developed in machine learning [Sch90]. Boosting is a popular machine learning technique in supervised learning. The idea behind boosting is to construct a "strong" classifier by combining multiple "simple" and "weak" classifiers. Despite the variety in weighting scheme of the "weak" classifiers and Mathematical hypothesis made, most of boosting algorithms iteratively learn a number of "weak" classifiers and then adding them together based on some weighting mechanism to form the final "strong" classifier. During this iteration process, after a "weak" classifier is added, the training data set is usually re-weighted, either based on

the performance of the "weak" classifier learnt in this iteration, or based on the performance of the combined "strong" classifier so far. Training data which are misclassified gain weight and data points which are classified correctly lose weight. In this way, in the future iterations, the weak classifier generated would be more focus on the part of training set previously misclassified. This rationale is shared in norm-referenced fitness function when the weights for test cases are adjusted in each generation.

Similarly, in the context of genetic algorithm, Eiben et al [EVDHVH98] introduce Stepwise Adaptation of Weights (SAW) method to adjust weights of test cases in graph coloring problem. In SAW, every T_p number of fitness evaluations, the weight of test cases which have been colored wrongly by the best individual in the population is creased by Δw. In [EVDHVH98], it is also shown that the exact value of the two newly introduced control parameter T_p and Δw "do not have a significant effect on the performance", as long as T_p is sufficiently small, and SAW is able to drastically improve the performance of both the convergence to the optimal solution and convergence speed in graph coloring problem studied. Comparing to SAW, norm-referenced fitness function adjusts the weights of the test cases based on the whole population instead of just the best so far individual. Using the population's average performance instead of the best so far individual's performance gives us the ability to perform a more detailed theoretical analysis of the algorithm, which we will discuss in Section 6.5. Furthermore, instead of applying a delta (Δw) to the existing weight of the test case in SAW, norm-referenced fitness function recalculate the weight for each test case every generation explicitly.

6.4 Norm-referenced Fitness

The norm-referenced fitness function introduced in the previous section can be implemented by porting into existing GP implementations using adjusted fitness. The concept of adjusted fitness is originally developed by Koza to "exaggerate the importance of small differences in the value of the standardized fitness as the standardized fitness approaches 0" [Koz92]. It was mainly used for fitness proportionate selection. Later on, because of the development of various bloating control methods, the usage of adjusted fitness has been extended. Nowadays, most of GP implementations use adjusted fitness calculation as a custom point

where users can alter raw fitness value, applying linear parametric parsimony pressure for example, and underlying selection methods are based on adjusted fitness rather than raw fitness values [SA05].

In the case of norm-referenced fitness function, the adjustment of fitness can be implemented in two steps. After every individual's fitness in the current population is calculated, we firstly calculate $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$ using Algorithm 7.

Algorithm 7 Algorithm to Build $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$

1: **function** build-metric(population p) : $\boldsymbol{w}^-$, $\boldsymbol{w}^+$
2: $\boldsymbol{e} \leftarrow (e_1 = 0, \dots, e_n = 0), i = 1 \dots n$
3: $\boldsymbol{a} \leftarrow (a_i = 0, \dots, a_n = 0), i = 1 \dots n$
4: **for** testcase $i = 1 \dots n$ **do**
5: **for all** individual $x \in p$ **do**
6: $e_i = e_i + S(f_i(x))$
7: $a_i = a_i + (1 - S(f_i(x)))$
8: **end for**
9: **end for**
10: $\boldsymbol{w}^- \leftarrow (w_1^- = \frac{e_1}{\sum_{i=1}^n e_i}, \dots, w_n^- = \frac{e_n}{\sum_{i=1}^n e_i}), i = 1 \dots n$
11: $\boldsymbol{w}^+ \leftarrow (w_1^+ = \frac{a_1}{\sum_{i=1}^n a_i}, \dots, w_n^+ = \frac{a_n}{\sum_{i=1}^n a_i}), i = 1 \dots n$
12: **return** $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$
13: **end function**

Then, with $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$, we can calculate fitness adjustments for every individual in the population using Algorithm 8.

Algorithm 8 Algorithm to Calculate Fitness Adjustment

1: **function** calc-adjusted-fitness(individual x, $\boldsymbol{w}^-$, $\boldsymbol{w}^+$, λ) : f_{adj}
2: $f_{adj} \leftarrow 0$
3: **for** testcase $i = 1 \dots n$ **do**
4: $f_{adj} = f_{adj} + (\lambda \cdot w_i^- + (1 - \lambda) \cdot w_i^+) \cdot f_i(x)$
5: **end for**
6: **return** f_{adj}
7: **end function**

Unlike common fitness adjustment calculations such as linear parametric parsimony pressure, in which the adjusted fitness is calculated immediately after the calculation of raw fitness for an individual, norm-referenced fitness adjustment can only be calculated after all individuals' raw fitness within the population are

evaluated. This is because the calculation of $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$ require every individual's raw fitness information. In addition, to avoid repeated evaluation, every individual's fitness needs to be stored as a vector indexed by the test case number rather than a single aggregated value. This is because individual test case fitness information i.e. $f(x)$, is required in both Algorithm 7 and Algorithm 8. Without re-evaluation, the runtime overheads to implement the adjustment can be neglected.

6.4.1 Initial Experiments

In the first experiment, we test the performance of norm-referenced fitness function using even parity 5 problem domain. We firstly randomly generate 50 initial populations. Then, for each population generated, we perform GP runs firstly using the original fitness function, and then using the norm-referenced fitness function developed. We test the parameter λ ranging from 0 to 1 in the step of 0.1 for norm-referenced fitness function. So, for each population initialization, we have 11 GP runs, one using the original fitness function, and the other 10 using norm-referenced fitness function with different λ values. Since the output of GP system is usually the best individual in the last generation, to compare the performance, we use *best individual in the current generation* as the criterion. The rest of experiment parameters are as follows. The population size is 500. Tournament selection is used and tournament size is set to 5. In breeding process, only crossover and reproduction are used with probability 90% and 10% respectively. GP runs for 50 generations. We use $S(x) = x$ as the scaling function in (6.2). We use GPLab [SA05] as the testing platform.

Table 6.2 summaries the experiment result we get. The best fitness column is the average best fitness achieved at generation 50 over 50 initializations. In order to illustrate the statistical significance, we perform T-Test with 95% confidence between the original statistics (in the first row) and each λ value. The test results are given in column T-Test in Table 6.2, where $\uparrow$ represents mean value which is statistically superior to original fitness function, $\downarrow$ represents mean value which is statistically inferior, and $\rightarrow$ represents no statistically significant difference. From Table 6.2, we find that norm-referenced fitness function developed outperforms the original fitness function when parameter λ is around 0.5. λ equals to 0.5 gives best performance improvement (16.08%). But, when λ is small, the performance is worse compared to the original fitness function. Because we are using the same

Fitness Function	λ	Best Fitness	σ	T-Test
Original		7.46	1.459	
Norm-referenced	1.0	7.70	1.170	$\rightarrow$
	0.9	7.60	1.095	$\rightarrow$
	0.8	7.50	1.063	$\rightarrow$
	0.7	6.92	1.197	$\uparrow$
	0.6	6.32	1.618	$\uparrow$
	0.5	**6.26**	**1.180**	$\uparrow$
	0.4	6.84	2.043	$\rightarrow$
	0.3	9.92	2.252	$\downarrow$
	0.2	11.46	1.846	$\downarrow$
	0.1	11.52	1.526	$\downarrow$
	0.0	11.44	1.627	$\downarrow$

Table 6.2: Best Fitness at Gen 50 in Even Parity 5 Problem

50 initializations across experiments of different parameters, in addition to the average performance comparison above, it is also possible to compare performance in a one-to-one basis, as in Table 6.3. From Table 6.3, we can find that, when

λ	Num of Better	Num of Equal	Num of Worse
1.0	15	15	20
0.9	17	7	26
0.8	18	11	21
0.7	24	8	18
0.6	26	13	11
0.5	**29**	**13**	**8**
0.4	25	10	15
0.3	6	6	38
0.2	2	2	46
0.1	0	2	48
0.0	0	3	47

Table 6.3: Number of GP Runs when Norm-referenced Fitness Function Performs Better, Equal, or Worse Compared to the According Original Fitness function with Same Initialization

$\lambda = 0.4, 0.5, 0.6, 0.7$, norm-referenced fitness function performs better in around 50% of initializations.

6.4.2 Analysis of Selection Intensity

Norm-referenced fitness function adjusts the original fitness function using the population's performance. This adjustment changes individuals' fitness ranks and ultimately affects the selection of parents in the breeding phase. Thus, it is possible to analyze the effect of norm-referenced fitness function by studying how it affects the selection of parents. Here, we use selection intensity to analyze the effect of the norm-referenced fitness function. Selection intensity is developed by Blickle and Thiele in [BT96]. The selection intensity I of a selection method is:

$$I = \frac{\bar{M}^* - \bar{M}}{\bar{\sigma}}$$

where $\bar{M}^*$ is the expected mean fitness after selection, $\bar{M}$ is the expected mean fitness before selection, and $\bar{\sigma}$ is the mean fitness variance before selection. The selection intensity subjects to the distribution of fitness before selection. As a result, in [BT96], Blickle and Thiele restricted the fitness distribution to normalized Gaussian distribution in order to derive mathematical formulas, such that different selection methods can be compared. In our case, however, the goal is to analyze how norm-referenced fitness function affects the selection method throughout the evolution process. Thus, we need to consider different distributions of fitness.

The simulation of selection intensity is designed as follows. We select a single GP run from previous experiment in which the original fitness function is used. For each generation, we simulate tournament selections with tournament size 5 firstly using original fitness function, then using norm-referenced fitness function with λ ranging from 0 to 1 in step of 0.1. We then calculate the selection intensity for each case. In order to reduce the randomness in tournament selection, we use the same random number generator for each simulation, i.e. the same 5 random individuals are selected, only the selection of the best may be altered based on different fitness functions and parameter λ settings. The simulation result is in Figure 6.1.

From this simulation, we get two direct observations. Firstly, the selection intensity of tournament selection with the original fitness function increases in later generations. The average selection intensity for the original fitness function from generation 0 to 20 is 0.192, while this average increases to 0.810 for generation 21 to 50. Secondly, for tournament selection with norm-referenced fitness function,

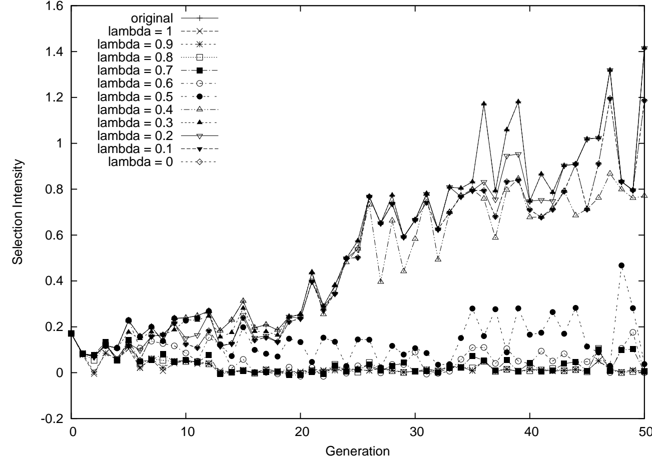


Figure 6.1: The Selection Intensity of Tournament Selection for the Original Fitness Function and Norm-referenced Fitness Functions with Different λ Settings

the selection intensity behaviors can be divided into three categories. For λ values from 0.6 to 1, where $\boldsymbol{w}^-$ dominates F_{adj}, the selection intensity fluctuates around 0. For λ values 0 to 0.4, where $\boldsymbol{w}^+$ dominates F_{adj}, the selection intensity follows the same trend as the original fitness function. For λ value 0.5, the selection intensity seems to be independent to generation. The mean selection intensity is 0.149, which is very close to the initial selection intensity in generation 0, which is 0.171. This observation confirms our discussion in Section 6.3. $\boldsymbol{w}^-$ promotes exploration by reducing the selection intensity, while $\boldsymbol{w}^+$ promotes exploitation by increasing the selection intensity. λ value 0.5 gives the best balance between exploration and exploitation. As a result, it gives the best performance in the initial experiment.

6.5 Implicit Bias Theory

To further study the selection intensity, let's review F_{adj}. From (6.6), we have:

$$\begin{aligned} F_{adj}(x) &= (\lambda \boldsymbol{w}^- + (1-\lambda)\boldsymbol{w}^+) \cdot \boldsymbol{f} \\ &= \textstyle\sum_{i=1}^n (\lambda w_i^- + (1-\lambda) w_i^+) \cdot f_i(x) \end{aligned}$$

Substitute w_i^- and w_i^+ using (6.3) and (6.5), we get:

$$F_{adj}(x) = \textstyle\sum_{i=1}^n (\lambda \frac{e_i}{\sum_{i=1}^n e_i} + (1-\lambda) \frac{a_i}{\sum_{i=1}^n a_i}) \cdot f_i(x)$$

Substitute a_i using (6.4), we have:

$$\begin{aligned} F_{adj}(x) &= \textstyle\sum_{i=1}^{n}(\lambda \frac{e_i}{\sum_{i=1}^{n} e_i} + (1-\lambda)\frac{1-e_i}{\sum_{i=1}^{n}(1-e_i)}) \cdot f_i(x) \\ &= \textstyle\sum_{i=1}^{n} \frac{e_i(\lambda n - \sum_{i=1}^{n} e_i) + (1-\lambda)\sum_{i=1}^{n} e_i}{(n - \sum_{i=1}^{n} e_i)\sum_{i=1}^{n} e_i} \cdot f_i(x) \end{aligned}$$

Let $\lambda = \frac{\sum_{i=1}^{n} e_i}{n}$, then:

$$\begin{aligned} F_{adj}(x) &= \textstyle\sum_{i=1}^{n} \frac{n \cdot e_i \cdot 0 + (n - \sum_{i=1}^{n} e_i)\sum_{i=1}^{n} e_i}{n \cdot (n - \sum_{i=1}^{n} e_i)\sum_{i=1}^{n} e_i} \cdot f_i(x) \\ &= \frac{1}{n} \cdot \textstyle\sum_{i=1}^{n} f_i(x) \\ &= \frac{F(x)}{n} \end{aligned}$$

which is independent of e_i. On one hand, we could say that norm-referenced fitness function is useless when parameter $\lambda = \frac{\sum_{i=1}^{n} e_i}{n}$. On the other hand, we can also say that the original fitness function is a "special" kind of norm-referenced fitness function, in which the parameter λ is dynamically adjusted to $\frac{\sum_{i=1}^{n} e_i}{n}$ for each generation. We call this value $\lambda_{original}$, i.e.

$$\lambda_{original} = \frac{\sum_{i=1}^{n} e_i}{n}$$

Since e_i represents the population's average error for a single test case i, $\frac{\sum_{i=1}^{n} e_i}{n}$ then is the population's average error for the problem currently solving. Within the evolution process, the population's fitness generally improves. So, this value $\frac{\sum_{i=1}^{n} e_i}{n}$ becomes smaller as GP system evolves. In another word, in the original fitness function, the $\lambda_{original}$ value decreases through generations. Since smaller λ means stronger effect of $\boldsymbol{w}^{+}$, thus, there is an *implicit bias* within the original fitness function increasing the selection intensity as GP system evolves, "pushing" the system to converge.

Figure 6.2 gives an example of how $\lambda_{original}$ changes when the original fitness function is used. In the initial population at generation 0, the $\lambda_{original}$ is 0.49975, which is very close to 0.5. This is not an accident. Parity problem is a binary problem. The output can only be either 1 or 0. Because the initial population is randomly generated, as a result, the expected value of $\lambda_{original}$ in

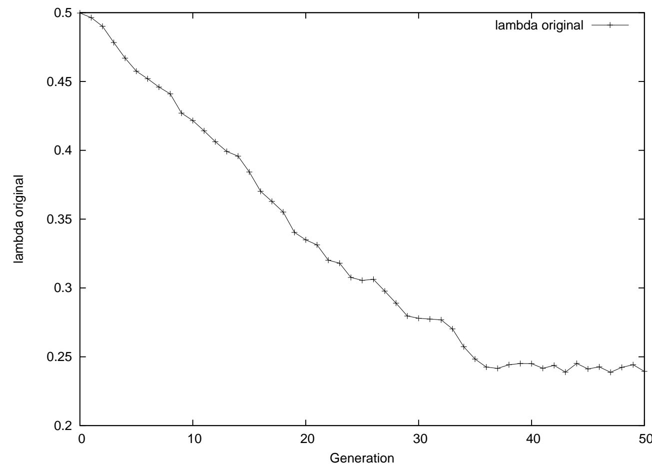


Figure 6.2: $\lambda_{original}$ for a GP Run in Even Parity 5 Problem using Original Fitness Function

generation 0 would be 0.5. With $\lambda_{original}$ equals to 0.5 initially, the original fitness function achieves a good balance between exploration and exploitation. But, as the population's fitness gets better, the $\lambda_{original}$ gets smaller, the original fitness function then puts more weight on exploitation, breaking the balance between exploration and exploitation. As a result, after around 35 generations, the GP system's evolution speed has been greatly reduced. So, in conclusion, the main problem of original fitness function is the implicit bias towards exploitation in later generations. On the other hand, because of the fixed λ parameter setting, norm-referenced fitness function does not have this implicit bias.

Using this theory of implicit bias, we can explain why different λ values achieve different performances in the initial experiment. In the initial experiment, λ values ranging from 0 to 0.3 failed to perform well. This is because those λ values are too small such that the GP system does not have enough ability to explore the searching space. As a result, in the selection of parameter λ, we generally should avoid small numbers. This is because small λ values not only cannot reduce the implicit bias towards exploitation in the original fitness function, but also enforce the bias. So, smaller λ values may lead to early convergence of the system, which is not desired.

On the other hand, when λ is too big, the selection intensity would be very small. This does remove the implicit bias in later generations, but it put too much emphasis to exploration in early generations, such that the evolution will be slowed down. This is why λ values 1, 0.9 and 0.8 haven't achieved better

performance compared to the original fitness function in the initial experiment. In even parity 5 problem domain, norm-referenced fitness function with λ value 0.5 gives the best balance between the exploration and exploitation. In early generations, it mimics the behavior of the original fitness function to give a good balance between the exploration and exploitation. In later generations, when the implicit bias starts to affect the original fitness function breaking the balance, the norm-referenced fitness function with λ equals to 0.5 still maintains the balance.

The theory of the implicit bias towards exploitation in original fitness function not only can be used to explain why norm-referenced fitness function works as we discussed above, the significance of this theory is much more profound. In fact, this theory offers a new perspective while tuning selection pressure in GP. Selection pressure is the key feature of selection methods [PLM08] and it plays a critical role in GP system [XZA07]. Furthermore, selection pressure plays an important role in bloating [GEBK04]. But, current researches about selection pressure control are mainly concentrated on modifications of selection methods. For example, in [XZA06a], a modification of standard tournament selection using population clustering is developed. The theory of implicit bias shows that, in addition to selection methods, *fitness function also plays an important role in the formation of selection pressure*. Adjusting parameter λ in norm-referenced fitness function provides a complete new and effective approach to tune the selection pressure. For example, from Figure 6.1, we can see that λ value 1 is able to significantly reduces the selection intensity of tournament selection to around 0 for all generations. λ value 0.5 is able to maintain the selection intensity of tournament selection at a constant level, rather than increasing over generations.

In addition, the implicit bias theory also gives an unique insight into the convergence of GP. Previous researches regarding the convergence of GP are generally based on the loss of diversity symptom. It is widely believed that it is the crossover which results in population lossing diversity over generations and the pre-mature convergence of GP. For example, in [Lan96a], Langdon regards the common phenomenon in GP that improvements in the best fitness value in the population occurs rarely after 20 to 30 generations as the "death of crossover". As a result, a number of "clever" GP operators are developed such as in [CM02] and [SD06] which are able to preserve the diversity of the population. However, the implicit bias theory reveals another and arguably more fundamental cause of convergence: the fitness function of GP system itself "pushes" the population

towards convergence when the population's average performance gets better and better. In fact, we think that the lose of diversity could merely be a symptom of the curse of evolution. This is because selection methods which have been thought as the root cause of the loss of diversity is controlled and driven by fitness function.

In the early stage of evaluation in which the population's average performance is not very good ($\lambda_{original}$ is big), the original fitness function is able to balance the exploration and exploitation, driving the population to evolve quickly. However, as the population's average performance getting better and $\lambda_{original}$ getting smaller, according to our discussion in Section 6.3, the fitness function is more "interested" in individuals copying their parents' behavior, and failed to capture "better" individuals which perform well on "hard" test cases, because the "hard" test cases have relatively smaller weights. So, the same fitness function, which promotes the evolution in the early stage of evolution, becomes the obsolete to the evolution in later generations. We call this phenomenon the *curse of evolution*. The curse of evolution shows that, in the later generations when the evolution slows down, it not only because it is increasingly hard to find fitter solutions, but also and more likely due to the fitness function is no longer able to effectively evolve the population. In addition, the curse of evolution shows that "the selection favoring representations which have the same fitness as those from which they were created" [LP97b] may not due to either the fact that crossover cannot produce "fitter" individuals or any selection scheme. In fact, in the later generations, offsprings having the same phenotype behavior as their parents are favored because the fitness function favors individuals copying the population's behavior.

Finally, the curse of evolution represents a first evidence showing that the population plays a much more important role in the selection of "fitter" individuals. The original fitness function, which we thought is quite subjective, can actually be affected by the performance of population. On the other hand, the curse of evolution also implies that, there is actually a conflict between finding the optimal solution, which is what we expect GP to perform, and what GP actually does, finding fitter populations, especially in later stage of evolution. In another word, unfit individuals within the population is quite useful to balance the population's performance such that the the population is able to positively affects the fitness function.

6.5.1 Further Experiment in Even Parity 5 Domain

In the simulation in previous section, we find that bigger λ values result in smaller selection intensity and conclude that this slows down the convergence speed of the algorithm. In the next experiment, we study whether smaller selection intensity leads to better convergence of algorithm in longer term despite the speed. In this experiment, we randomly choose 10 initializations created in the first experiment and run them for 300 generations instead of 50. Given enough time for GP algorithm to evolve, we then check if the norm-referenced fitness function leads to better convergence for big λ values. The rest of parameters are the same as in initial experiment. Here, we test norm-referenced fitness with λ values ranging from 0.5 to 1 in step of 0.1.

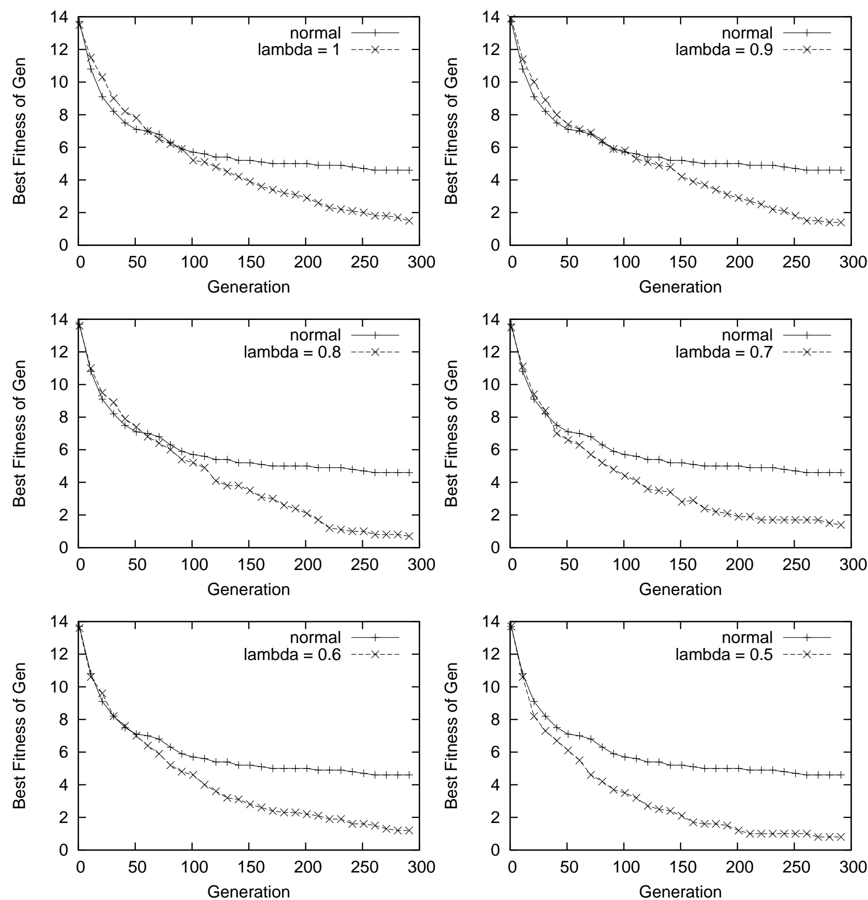


Figure 6.3: Comparison of Best Fitness of Generation Changes over Generations Between Original Fitness Function and Norm-referenced Fitness Functions with Different λ Settings

The experiment result is summarized in Table 6.4 and Figure 6.3. In Table

Initialization	Original	Norm-referenced						Best Param
		$\lambda = 1.0$	$\lambda = 0.9$	$\lambda = 0.8$	$\lambda = 0.7$	$\lambda = 0.6$	$\lambda = 0.5$	
1	3 (300)	2 (300)	**0 (110)**	1 (300)	**0 (140)**	**0 (232)**	**0 (193)**	$\lambda = 0.9$
2	3 (300)	**0 (240)**	1 (300)	**0 (213)**	**0 (151)**	**0 (192)**	**0 (100)**	$\lambda = 0.5$
3	5 (300)	**0 (219)**	3 (300)	1 (300)	1 (300)	2 (300)	1 (300)	$\lambda = 1.0$
4	5 (300)	3 (300)	1 (300)	1 (300)	1 (300)	3 (300)	**0 (156)**	$\lambda = 0.5$
5	8 (300)	**0 (290)**	1 (300)	**0 (204)**	2 (300)	3 (300)	**0 (190)**	$\lambda = 0.5$
6	5 (300)	4 (300)	**0 (244)**	**0 (213)**	3 (300)	1 (300)	**0 (209)**	$\lambda = 0.8$
7	3 (300)	**0 (215)**	**0 (252)**	**0 (188)**	1 (300)	**0 (215)**	3 (300)	$\lambda = 1.0, 0.6$
8	4 (300)	2 (300)	5 (300)	1 (300)	**0 (281)**	1 (300)	**0 (271)**	$\lambda = 1.0$
9	4 (300)	3 (300)	3 (300)	**0 (227)**	1 (300)	1 (300)	1 (300)	$\lambda = 0.8$
10	6 (300)	**0 (198)**	**0 (240)**	3 (300)	5 (300)	**0 (207)**	3 (300)	$\lambda = 1.0$
Avg. Fitness	4.6	1.4	1.4	0.7	1.4	1.1	0.8	

Table 6.4: Experiment Results for 10 Initializations Running 300 Generations

6.4, each cell gives the best raw fitness achieved. The number in brackets is the generation when the best raw fitness is achieved. GP runs in which the optimal solution is found (raw fitness 0) are highlighted. From Figure 6.3, we can find that, original fitness function performs very well in the first 100 generations. The best fitness of generations has been improved from 13.7 at generation 0 to 5.7 at generation 100. But after that, it failed to continuously improve the best fitness of generation. It takes 200 generations to improve the best fitness of generation to 4.6 at generation 300. On the other hand, norm-referenced fitness function is able to continuously improve the best fitness of generation. Bigger λ value results in slower improvements in early generations. When λ equals to 1, it takes 62 generations for norm-referenced fitness function to outperform original fitness function. It takes 72 generations when λ equals to 0.9. This number is reduced to 61 when λ equals to 0.8, 45 when λ equals to 0.7, 50 when λ equals to 0.6, and 10 when λ equals to 0.5. But, given 300 generations to evolve, norm-referenced fitness function outperforms original fitness function in almost all cases for all λ values experimented. In addition, there is no clear difference in performance for different λ values. This suggests that even though bigger λ value results in smaller selection intensity and it slows down the evolution, given enough time to evolve, big λ value can always achieve the same level of performance as the optimal λ (0.5 in our case), and greatly outperforms the original fitness function. Thus, when selecting λ value, we can increase the number of generations GP runs to reduce the sensitivity of the parameter λ.

6.5.2 Experiment in Other Problem Domains

In previous sections, we study the performance of norm-referenced fitness function in detail using one problem domain even parity 5. In this section, we extend our

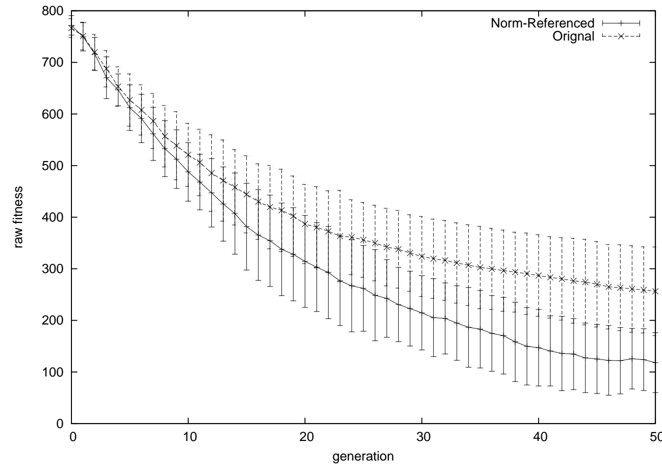


Figure 6.4: Best Fitness by Generation for Norm-referenced and Original Fitness Function in Multiplexer 11 Problem

experiments to a number of other interesting problem domains including two other boolean problems, multiplexer 11 and artificial ant, a real numbered problem, the sextic regression problem and finally OneMax problem which is widely used in GA.

The first problem domain we are interested in is multiplexer 11 problem. Unlike even parity 5 problem which only contains 32 test cases, the interesting aspect of multiplexer 11 problem is that, it contains 2048 independent test cases. The experiment of norm-referenced fitness function in multiplexer 11 problem is designed as follows. Norm-referenced fitness function with λ value 0.5 is compared with original fitness function using the best raw fitness achieved in each generation as criterion. The population size is set to 500. Tournament selection is used with tournament size 5. Crossover and reproduction are used with 90% and 10% probability respectively. The experiment result is summarized in Figure 6.4. From the result, we can find that norm-referenced fitness function significantly outperforms the original fitness function especially in later generations. This behavior is quite similar to what is observed in even parity 5 problem. This suggests that norm-referenced fitness function is able to scale up to handle problem domains which contain a lot of test cases.

In the next, we study the behavior of norm-referenced fitness function in artificial ant problem. The nature of artificial ant problem is quite different compared to multiplexer and parity problems. In the later two problems, the individuals are functions without side effects and test cases are independent to each other.

However, this is not the case for artificial ant problem. In artificial ant problem, the individuals are not functions without side effects and the performance of the ant not only depends on the correctness of the program but also the location of the ant. Moreover, each food on the ground are not independent to each other in artificial ant problem. A successful move for the ant to eat one food could result in a number of subsequent foods to be eaten by the ant easily. Since the test cases are correlated to each other, the population is more likely to converge since exploring alternative better solutions are much harder compared to when the test cases are not correlated. As a result, we believe that a bigger λ value for norm-referenced fitness function, which prompts exploration over exploitation, is required to improve the overall performance. In the next experiment, we compare the performance of norm-referenced fitness function and original fitness function in ant problem. For norm-referenced fitness function, we test λ equals 0.5, 0.6, 0.7, 0.8, 0.9 and 1.0. The rest of experiment parameters are the same as previous experiment in multiplexer 11 problem. The experiment result is summarized in Table 6.5. To establish the statistical significance, we use T-Test with 99% confidence to compare original fitness function and norm-referenced fitness function. From Table 6.5, we can see that norm-referenced fitness function with λ value

Fitness Function	λ	Best Fitness	σ	T-Test
Original		29.102	9.511	
Norm-referenced	1.0	24.163	8.683	$\rightarrow$
	0.9	**22.286**	**8.051**	$\uparrow$
	0.8	24.673	8.175	$\rightarrow$
	0.7	25.021	7.157	$\rightarrow$
	0.6	22.531	8.325	$\uparrow$
	0.5	26.532	6.869	$\rightarrow$

Table 6.5: Best Fitness at Gen 50 in Artificial Ant Problem

0.5, 0.7, 0.8 and 1.0 perform better than original fitness function, but not in a statistical significant manner. λ value 0.6 and 0.9 perform better than original fitness function in a statistical significant manner. λ value 0.9 performs best and it outperforms the original fitness function by 23.42%. This confirms the expectation that in ant problem where test cases are correlated to each other, bigger λ value performs better compared to λ value 0.5.

All previous experiments deal with discrete domain, in which $f(x)$ is binary. This makes the selection of scaling function S in (6.2) quite simple. In regression

problems, however, $f(x)$ usually is not bounded, i.e. $f(x) \in [0, \infty)$. In this case, for certain unfit individuals, $f(x)$ could be extremely large. These outliers affect the calculation of e_i and ultimately affect the calculations of both $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$. For example, using GP to solve quartic symbolic regression problem ($x^4 + x^3 + x^2 + x$ with 20 points generated from [-1, 1] as test cases), individual $2x^4 + x^3 + 3x^2$'s error for test case $x = 1$ is $6 - 4 = 2$. Another individual $e^{e^{e^x}}$'s error for the same test case is 3814275.1. If we add those two errors together directly in (6.2), the effect of the first error value will be neglected. Moreover, from pure implementation point of view, adding very big numbers together in (6.2) may cause floating point number overflow. Then e_i will be ∞ and the whole calculation of $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$ will be wrong.

To solve this problem, we need an effective scaling function in (6.2) to control those outliers. In this thesis, we develop a simple linear scaling function as follows. Given a population $P = \{x_i | 1 \le i \le m\}$, for a problem which contains n test cases, let

$$E = \{f_i(x_j) | 1 \le i \le m, 1 \le j \le n\}$$

Let $a = min(E)$ i.e. the minimal value in E, and $b = trimmean(E)$, i.e. the trimmed mean value of E, then the scaling function S is:

$$S(x) = \begin{cases} 0, & x = a \\ \frac{1}{2(b-a)}x - \frac{a}{2(b-a)}, & a < x \le 2b - a \\ 1, & x > 2b - a \end{cases} \tag{6.7}$$

This scaling function uses a linear function to map the minimal error value (a) to 0, and the trimmed mean value (b) to 0.5. All values bigger than $2(b - a)$ are mapped to 1. For every generation, we rebuild E and recalculate the value of a and b before calculating e_i and a_i. One thing to note is that, $S(f(x))$ here is an adjustment of $f(x)$ relative to present population's performance. Comparing the value of $S(f(x))$ from different generations would be meaningless.

We test the performance of norm-referenced fitness function with the linear scaling function using sextic regression problem introduced in [Koz94]. In sextic problem, the equation to be regressed is $y = x^6 - 2x^4 + x^2$ where $x \in [-1, 1]$. The experiment is designed as follows. Similar to the initial experiment, we firstly generate 50 initializations. Then for each initialization, we firstly run GP with

original fitness function, and then run with norm-referenced fitness function. In the later, the parameter λ ranges from 0.5 to 1 in step of 0.1. We also use best individual in the last generation as comparison criteria. The population size is 500. Tournament selection is used and the tournament size is 5. crossover and reproduction are used for selection of parent in breading with probability 90% and 10% respectively. GP runs for 50 generations. In the scaling function (6.7), when calculating trimmed mean, we discard 10% values at both high and low ends.

The experiment results are summarized in Table 6.6. In sextic problem, we find that λ equals to 1.0 gives the best performance improvement (40.35%). The

Fitness Function	λ	Best Fitness	σ	T-Test
Original		0.347	0.318	
Norm-referenced	**1.0**	**0.229**	**0.207**	↑
	0.9	0.288	0.278	↑
	0.8	0.313	0.339	→
	0.7	0.306	0.319	→
	0.6	0.364	0.342	→
	0.5	0.329	0.301	→

Table 6.6: Best Fitness at Gen 50 in Sextic Problem

optimal λ value in sextic problem is much bigger compared to parity problem. We think this is mainly because of the usage of scaling function (6.7), rather than the change of problem domain. In previous discussion, we find that the original fitness function is a "special" kind of norm-referenced fitness function, in which the parameter $\lambda_{original}$ equals to $\frac{\sum_{i=1}^{n} e_i}{n}$ for each generation. In parity problem, this value goes down as GP evolves. But in sextic problem, because of the linear scaling function (6.7), in which the trimmed mean is always mapped to 0.5, the value of $\lambda_{original}$ does not change through generations and

$$\lambda_{original} \approx 0.5$$

for all generations. In early generations, when individuals are generally unfit, this value is reduced by the scaling function. In later generations, when individuals are generally fit, the scaling function amplifies this value. As we discussed previously, big λ value prevents GP system from converging in later generations and this leads to better performance. Since the λ has been amplified by the scaling function in

later generations, as a result, we need even bigger λ values to effectively control the convergence speed. In our case, the scaling function (6.7) amplifies λ in sextic problem. So, bigger value ($\lambda = 1$) performs best. In conclusion, the selection of scaling function has much bigger implication to the norm-referenced fitness function. It may affect the value of optimal parameter λ. For the linear scaling function (6.7) developed, we can generally choose big λ values like 0.9 or 1.

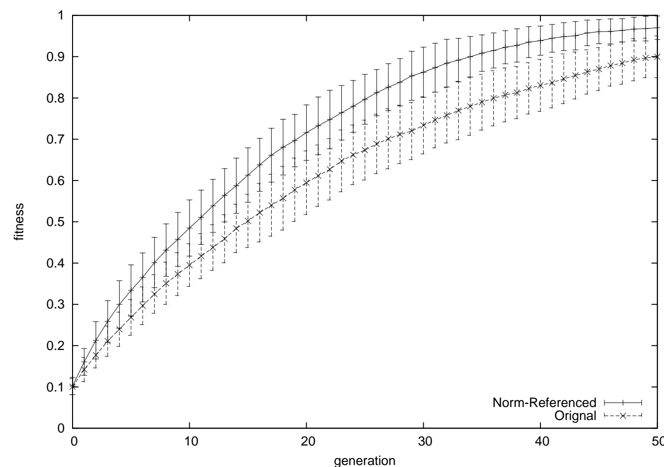


Figure 6.5: Average Population Fitness Change over Generations for Norm-referenced and Original Fitness Function in OneMax Problem

Even though the norm-referenced fitness function is developed in the context of genetic programming, the development of norm-referenced fitness function however does not use any GP specific domain information. As a result, the norm-referenced fitness function can be easily applied to other evolutionary computation methods. In the last experiment in this section, we apply norm-referenced fitness function to genetic algorithm using OneMax problem. OneMax problem [SE91] is arguably one the most widely known and used benchmark problems in genetic algorithm. The goal of OneMax problem is to maximize the number of ones of a bitstring. Formally, the goal of OneMax problem is to find a bit string $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, with $x_i \in \{0, 1\}$, which maximizes:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i$$

In our experiment, we optimize 20 bits bitstrings, i.e. $N = 20$. The rest of control parameters are defined as follows. Population size is set to 10. Only crossover operator is used for breeding. Tournament selection with tournament size 2 is

used for selection of parents in crossover. GA runs for 50 generations. In the initial genome generation, each bit has 10% chance to be 1 and 90% to be 0. For norm-referenced fitness function, we set the λ equals 0.5. We use average fitness of population at each generation to compare the performance of norm-referenced fitness function and original fitness function. The experiment is performed using ECJ [Luk09]. The experiment result can be viewed in Figure 6.5. Unlike previous experiments in which smaller raw fitness values are better. In OneMax problem, bigger fitness values are better and the optimal raw fitness value is 1. From the figure, we can see that norm-referenced fitness function is able to outperform original fitness function in a statistical significant manner. At generation 50, norm-referenced fitness function outperforms original fitness function by 7.84%. Finally, we experiment the performance of norm-referenced fitness function as N increases from 20 to 50, 100, 150 and 200. The experiment result is summarized in Table 6.7. In Table 6.7, the value before $\pm$ is the average best fitness achieved and the value after is the standard deviation. From Table 6.7, we can see that,

N	Norm-referenced	Original
20	0.970 ± 0.028	0.900 ± 0.051
50	0.850 ± 0.036	0.755 ± 0.043
100	0.722 ± 0.030	0.649 ± 0.028
150	0.647 ± 0.023	0.599 ± 0.022
200	0.604 ± 0.024	0.568 ± 0.021

Table 6.7: OneMax Problem Performance for N in 20, 50, 100, 150 and 200

as N increases and onemax problem becomes harder, both norm-referenced and original fitness function performs worse as expected. But, norm-referenced fitness function always able to outperforms the original fitness function for a given N value.

6.6 Partial Norm-referenced Fitness

At the beginning of this chapter, when we gave the description of the norm-referenced fitness function algorithm, we explained that norm-referenced fitness adjustment can only be calculated after all individuals within the population are evaluated because the calculation of $\boldsymbol{w}^-$ and $\boldsymbol{w}^+$ require every individual's raw fitness information. In general, this limitation would not cause any runtime performance issue. But, when using tournament selection with norm-referenced

fitness function, the above limitation may result in two problems which affects runtime performance of GP due to two unique features of tournament selection. The first feature of tournament selection is that it suffers from "not-sampled issue" [XZ12], i.e. not all individuals are guaranteed to be selected at least once as part of tournament. These not sampled individuals which originally not required to be evaluated now need to be evaluated because their fitness value are required as part of the calculation for $\boldsymbol{w}^{-}$ and $\boldsymbol{w}^{+}$. The second feature is that, tournament selection supports parallel processing of multiple tournaments. However, when it is used with norm-referenced fitness function, no tournament selection can be processed until the evaluation of all individuals within the population is completed. In another word, when using norm-referenced fitness function, tournament selection no longer can be used in a multi-threaded manner. This problem, although may not be a big issue in single threaded implementation of GP, may result in inferior runtime performance in multi-threaded implementation of GP. In this section, we address this limitation of norm-referenced fitness function by implementing partial norm-referenced fitness function.

Partial norm-referenced fitness function implements the same concept as norm-referenced fitness function but only at tournament level. Given a tournament T in tournament selection which contains t individuals, and a problem with n test cases, for each test case i, we can calculate the tournament T's performance:

$$e_i^p = \frac{\sum_{x_j \in T} S(f_i(x_j))}{t} \tag{6.8}$$

where $S(.)$ is a scaling function and $S(x) \in [0, 1]$ and $f(.)$ is the same as in (6.1). Let $\boldsymbol{e^p} = (e_1^p, e_2^p, ..., e_n^p)$, then $\boldsymbol{e^p}$ represents the tournament T's performance for all test cases. Similar to (6.3), normalizing $\boldsymbol{e^p}$, we get $\boldsymbol{w^{p-}}$, where:

$$w_i^{p-} = \frac{e_i^p}{\sum_{i=1}^{n} e_i^p} \tag{6.9}$$

Similar to (6.4), we can calculate the tournament T's performance, $\boldsymbol{a^p} = (a_1^p, a_2^p, ..., a_n^p)$, where:

$$a_i^p = 1 - e_i^p \tag{6.10}$$

Normalizing $\boldsymbol{a^p}$, we get the weight vector $\boldsymbol{w^{p+}}$, where:

$$w_i^{p+} = \frac{a_i^p}{\sum_{i=1}^{n} a_i^p} \tag{6.11}$$

With $\boldsymbol{w^{p-}}$ and $\boldsymbol{w^{p+}}$, we define the partial norm-referenced fitness function F^p_{adj} for tournament T as:

$$\begin{aligned} F^p_{adj}(x) &= \lambda^p \cdot F^{p-}_{adj}(x) + (1-\lambda^p) \cdot F^{p+}_{adj}(x) \\ &= (\lambda^p \boldsymbol{w^{p-}} + (1-\lambda^p)\boldsymbol{w^{p+}}) \cdot \boldsymbol{f} \\ &= \boldsymbol{w^p} \cdot \boldsymbol{f} \end{aligned} \tag{6.12}$$

where $x \in T$.

In essence, F^p_{adj} represents an approximation of F_{adj}. The quality of this estimation depends on the size of the tournament and how "well" the tournament of individuals are selected from the population. Norm-referenced fitness function F_{adj} is a global weight optimization for a given population, while the partial norm-referenced fitness function is a local weight optimization for a given tournament in tournament selection. As a result, the performance of partial norm-referenced fitness may not be as good as norm-referenced fitness function. Norm-referenced fitness function can be implemented by modifying tournament selection. After the tournament is randomly selected, we calculate $\boldsymbol{w^p}$ and then adjust each individual in the tournament's raw fitness, and then use the adjusted fitness to determine the winner of the tournament. A very important thing to note is that although in all norm-referenced fitness function experiments above, we only use tournament selection, this is only because the performance of tournament selection is good compared to other selection methods such as fitness proportionate selection. The norm-referenced fitness function is a generic modification of fitness function and can be used in conjunction with all other selection methods. However, the partial norm-referenced fitness function is only a modification of tournament selection and hence can only be used in conjunction with tournament selection to address the potential runtime performance limitation of norm-referenced fitness function.

In the next, we compare partial norm-referenced fitness function, norm-referenced fitness function and original fitness function using artificial ant, multiplexer 11 and even parity 5 problem. For partial norm-referenced fitness function and norm-referenced fitness function, we experiment parameter λ values 0.5 to 1.0, in the step of 0.1. Tournament selection of tournament size 5 is used. GP runs for 300 generations. Crossover and reproduction are used with 90% and 10% probability respectively. The experiment is performed using ECJ [Luk09]. For each parameter setting, we perform 50 GP runs to calculate the average performance. Table 6.8 summarizes the experiment result of partial norm-referenced,

Artificial Ant Problem					
Fitness Function	λ	Gen 50	Gen 100	Gen 200	Gen 300
Original		28.46 ± 11.18	26.14 ± 11.65	22.76 ± 12.42	21.54 ± 12.21
Norm-referenced	1.0	27.24 ± 6.78	23.64 ± 6.56	19.52 ± 6.66	17.18 ± 6.49
	0.9	23.16 ± 7.86	19.26 ± 7.39	14.80 ± 7.34	**12.52 ± 7.38**
	0.8	25.22 ± 7.41	19.46 ± 7.76	17.16 ± 7.76	14.84 ± 6.83
	0.7	22.96 ± 10.45	18.22 ± 9.99	14.78 ± 9.60	13.40 ± 8.55
	0.6	22.60 ± 8.27	18.92 ± 8.68	15.68 ± 8.34	14.46 ± 7.95
	0.5	24.62 ± 7.57	20.34 ± 8.54	17.28 ± 8.02	15.34 ± 7.75
Partial Norm-referenced	1.0	26.14 ± 9.25	23.60 ± 10.57	19.34 ± 10.28	**17.14 ± 10.45**
	0.9	27.90 ± 8.76	23.84 ± 8.92	20.32 ± 8.94	17.98 ± 8.18
	0.8	27.10 ± 10.24	23.56 ± 10.06	21.60 ± 9.48	19.88 ± 9.16
	0.7	26.64 ± 9.41	23.36 ± 9.69	19.96 ± 8.73	18.06 ± 8.74
	0.6	25.14 ± 9.65	23.22 ± 9.71	20.96 ± 10.13	19.76 ± 9.41
	0.5	25.90 ± 8.96	23.10 ± 8.91	20.18 ± 9.62	18.94 ± 9.69
Even Parity 5 Problem					
Fitness Function	λ	Gen 50	Gen 100	Gen 200	Gen 300
Original		6.48 ± 1.88	5.48 ± 2.13	4.20 ± 2.14	3.62 ± 2.14
Norm-referenced	1.0	6.92 ± 1.29	3.96 ± 1.59	1.04 ± 1.15	0.24 ± 0.55
	0.9	6.42 ± 1.46	3.68 ± 1.67	1.1 ± 1.20	0.48 ± 0.78
	0.8	6.20 ± 1.37	3.10 ± 1.63	0.96 ± 1.20	0.44 ± 0.83
	0.7	5.82 ± 1.65	3.04 ± 1.71	0.84 ± 1.12	0.28 ± 0.53
	0.6	5.58 ± 1.27	2.64 ± 1.42	0.64 ± 0.87	0.22 ± 0.46
	0.5	5.70 ± 1.36	2.52 ± 1.50	0.52 ± 0.78	**0.20 ± 0.44**
Partial Norm-referenced	1.0	10.56 ± 1.20	7.74 ± 2.30	5.06 ± 2.36	3.58 ± 2.44
	0.9	9.82 ± 1.24	7.24 ± 2.03	4.94 ± 2.93	3.74 ± 2.73
	0.8	10.02 ± 1.46	7.20 ± 1.93	4.92 ± 2.76	3.56 ± 2.91
	0.7	8.96 ± 1.41	7.06 ± 1.79	4.84 ± 2.63	3.64 ± 2.76
	0.6	8.36 ± 1.31	5.84 ± 1.94	3.22 ± 2.26	2.18 ± 2.35
	0.5	7.44 ± 1.91	4.84 ± 2.27	2.52 ± 2.24	**1.82 ± 2.02**
Multiplexer 11 Problem					
Fitness Function	λ	Gen 50	Gen 100	Gen 200	Gen 300
Original		268.72 ± 73.22	187.68 ± 76.25	130.40 ± 92.32	112.16 ± 97.86
Norm-referenced	1.0	83.48 ± 92.67	6.64 ± 22.22	0.00 ± 0.00	0.00 ± 0.00
	0.9	95.96 ± 98.55	13.28 ± 29.22	0.00 ± 0.00	0.00 ± 0.00
	0.8	81.72 ± 90.86	14.64 ± 35.63	1.28 ± 8.96	0.00 ± 0.00
	0.7	104.28 ± 80.20	11.1 ± 27.93	2.56 ± 12.54	0.00 ± 0.00
	0.6	91.48 ± 82.28	7.36 ± 24.57	4.16 ± 19.94	3.84 ± 19.87
	0.5	86.16 ± 81.40	**10.4 ± 27.07**	0.00 ± 0.00	0.00 ± 0.00
Partial Norm-referenced	1.0	210.8 ± 115.10	82.56 ± 85.64	18.24 ± 45.71	7.68 ± 27.57
	0.9	230.92 ± 95.16	73.80 ± 71.50	**4.00 ± 15.20**	0.00 ± 0.00
	0.8	227.36 ± 114.58	92.66 ± 98.87	13.48 ± 38.91	2.56 ± 17.92
	0.7	194.64 ± 85.96	68.64 ± 72.77	20.80 ± 42.72	11.84 ± 32.13
	0.6	189.92 ± 99.96	69.38 ± 96.99	25.28 ± 80.07	18.24 ± 76.80
	0.5	143.24 ± 87.61	51.46 ± 58.79	18.24 ± 42.21	10.88 ± 34.83

Table 6.8: Best Raw Fitness at Generation 50, 100, 200, 300 using Partial Norm-referenced, Norm-referenced and Original Fitness Function

norm-referenced and original fitness function. The value before ± is the average best raw fitness achieved and the value after is the standard deviation. In artificial ant problem, all λ values give better performance compared to original fitness function. As we discussed in previous section, the optimal λ parameter is bigger. For norm-referenced fitness function, the optimal λ value is 0.9 and at generation 300, it is 41.88% better than the original fitness function. For partial norm-referenced fitness function, the optimal λ value is 1.0 and at generation 300, it is 20.43% better. In even parity 5 problem, the optimal λ parameter for norm-referenced fitness function is 0.5, with which, it is able to outperform the original fitness function by 94.48% at generation 300. Partial norm-referenced

results in worse performance in the first 50 generations, but it is able to catch up and eventually outperforms original fitness function. The optimal λ parameter value is also 0.5, in which it outperforms the original fitness function by 49.72% at generation 300. Norm-referenced and partial norm-referenced fitness function performs extremely well in multiplexer 11 problem. For norm-referenced fitness function, with the optimal λ value 0.5, all 50 GP runs are able to find optimal solution within no more than 133 generations. For partial norm-referenced fitness function, considering only the first 50 generation, bigger λ values result in worse performance. But, given enough generation to evolve, all λ values are able to significantly outperform the original fitness function. λ value 0.9 gives optimal performance. All 50 GP runs are able to find optimal solution within 284 generations.

This experiment also expands the experiment in Section 6.5.1, in which only even parity 5 problem is studied with 10 initialization. Experiment result for norm-referenced fitness function confirms the findings in Section 6.5.1. On the other hand, for partial norm-referenced fitness function, overall, the runtime performance gain is not without cost. The partial fitness function is able to outperform the original fitness function, but its performance in terms of best fitness achieved is not as good as norm-referenced fitness function. The choice between partial norm-referenced fitness function and norm-referenced fitness function depends on the requirement from the problem GP is solving. For problem domain which is time sensitive, it is more appropriate to use the partial norm-referenced fitness function in order to leverage the parallel processing ability of tournament selection.

6.7 Conclusion

In this chapter, we develop an dynamic fitness function to improve GP performance. This is motivated by the analysis in Section 6.3 showing that the original fitness function in GP may not be appropriate for the population. Then, we develop the norm-referenced fitness function by taking into account not only the raw fitness achieved by the individual, but also how well the rest of the individuals perform within the same population. With the proper controlling parameter λ, the norm-referenced fitness function is able to "guide" the population towards unexplored area of searching space and at the same time maintain the stability

of the evolution. Further analysis of norm-referenced fitness function in Section 6.5 reveals that the original fitness function widely used can be thought as a special case of norm-referenced fitness function with a changing λ value $\lambda_{original}$. This finding then reveals that the original fitness function suffers from a implicit bias towards exploitation. We name this implicit bias as the curse of evolution and it acts as the theoretical foundation of norm-referenced function. Empirical evidences from not only GP problems but also GA problems have also been given in this chapter to support the performance norm-referenced fitness function. In these experiments, it is shown that norm-referenced fitness function has very promising performance over the original fitness function. We also find in Section 6.5.1 that, given enough generation to evolve, the sensitivity of the choice of parameter λ can be greatly reduced. Finally, to address the potential runtime performance limitation of norm-referenced fitness function when used in conjunction with tournament selection, we develop a modification of tournament selection called partial norm-referenced fitness function which applies the same concept of norm-referenced fitness function but only at the local tournament level. Experiments of partial norm-referenced fitness function show that it is able to outperform the original fitness function but is not as good as norm-referenced fitness function. This leaves the user of the algorithm to balance the choice between better performance or better runtime performance.

The promising performance of norm-referenced fitness function is not accidental. In fact, it simulates one important aspect of evolution [Rob91], the feedback loop of evolution from the environment, i.e. the evolution of species changes the environment and this change of environment then affects the evolution of the species. With the original fitness function, the objective of the evolution is fixed and not affected by the evolution of the population. This kind of no feedback evolution certainly happens in real world. For example, polar bears may adapt to colder climates by growing thicker fur, but the climate is not affected by the increased fur at all [Rob91]. However, other evolution scenarios, the environment does response to the evolution of species and thus forms a feedback loop (either positive or negative). A example of the feedback loop can be found in the evolution of energy technology, in which the adaption of fossil fuels has caused the global warming which drives the exploration of alternative energy source. In the norm-referenced fitness function, the evolution objective is specially tailored according to the present population's performance. As a result, the fitness function

evolves by itself along with the population. This work represents an attempt to establish a feedback loop in genetic programming. The competitive performance of norm-referenced fitness function shows that adding the feedback loop dynamic into GP is a promising further research direction.

Chapter 7

Conclusion and Further Work

One advantage of genetic programming is the ability to perform structural optimization at the same time of performing the parameter optimization. This unique feature of GP makes it a very promising technique especially when solving complex problems in which the structure of the solution is not known a priori. However, scaling up GP is not at all a trivial task. One unique challenge faced would be bloating, which presents a serious obsolete to GP's scalability. This suggests that scaling GP not only requires tactical changes of the algorithm, but also systematic advancements of the understanding of GP dynamics.

This thesis seeks to improve the scalability of GP from two different perspectives and by answering three questions raised in the objectives section in Chapter 1. The first question "How bad is the bloating?" and the second question "How can bloating be controlled?" concentrate on fighting bloating as bloating is so far the biggest obsolete when scaling up GP. The third question "Apart from bloating control, how can we improve the GP performance in general?" addresses the GP scalability issue from a different perspective: if bloating cannot be eliminated completely, can we speed up GP to find the optimal solution faster? By answering these three questions, the works described in this thesis develop a better systematic understanding of GP from a number of different perspectives and hence improve the overall performance of GP in order to solve more complex problems.

7.1 Contributions

To answer the first question “How bad is the bloating?”, we developed activation rate in Chapter 4. Activation rate quantitatively models the fact that not all tree nodes in an individual contribute to the fitness of the individual. We further linked this fact to the characteristic of function set of the problem domain as descent rate. Using descent rate, we were able to accurately estimate the activation rate of a tree node of a certain depth. This gave us the ability to theoretically study how the computation effort of an individual program tree increases as the depth of the tree increases. We experimented with two theoretical tree size growth models deducted based on previous researches of tree growth, and found that the computation effort required to evaluate these trees increases as the depth of the tree increases, but in a slower manner compared to the increase in tree size. In addition to the above result, we studied an alternative tree evaluation algorithm, the bottom-up tree evaluation, using activation rate. Bottom-up tree evaluation is developed to address a scenario in tree evaluation where top-down evaluation is costly and less effective. Using activation rate as a theoretical tool, we found that bottom-up tree evaluation algorithm outperforms standard top-down tree evaluation when the program tree depth is relatively small (no bigger than 8). Also, we use activation rate to perform experiments to study crossover effects. Using activation rate, we were able to analyze the effect of crossover point selection and the effect of subtree swapping separately. We found that the effect of subtree swapping is generally stable, however, constructive crossover is more likely to happen when deeper crossover point is selected. Finally, we use activation rate to develop semi-intron crossover. Experiment of semi-intron crossover with different parameter setting gave an interesting anti-modification point depth theory empirical evidence.

To answer the second question “How bloating can be controlled?”, we reviewed one famous bloating theory, removal bias. We extended the theory by quantitatively define the removal bias as the depth difference between subtree swapped in crossover. Experiments of this quantitative measure of removal bias revealed that there is a strong positive correlation between the depth difference between subtree swapped in crossover and the average depth increase of the population. We called this observation the depth difference hypothesis. We showed that the depth difference between subtree swapped, which is driven by the depth difference between parents selected for crossover, represents the most fundamental

mechanism for GP to produce offsprings with different depths. Also, this genotype depth difference between parents is completely irrelevant to their phenotype performance. As a result, motivated by this depth difference hypothesis, we developed the depth constraint crossover which adds an additional constraint in crossover point selection limiting the depth difference between subtree swapped. Experiments of depth constraint crossover showed that it is very competitive compared to a number of existing bloating control methods.

In addition to studying bloating and proposing a new bloating control method, we seek alternative methods to improve GP performance, in order to answer the third question "Apart from bloating control, how can we improve the GP performance in general?". This is motivated by the fact that if the ultimate purpose of the bloating control is to allow GP to be applied to more complex problems, in theory, the same effect could also be achieved by improving GP performance by itself rather than fighting bloating. We developed a dynamic fitness function called norm-referenced fitness function. Unlike the original fitness function used in standard GP which is fixed, norm-referenced fitness function dynamically adjusts the weight of each test case based on the population's performance on all test cases. The population's performance is measured by balancing two conflicting metric: the exploration and the exploitation. Experiments of the norm-referenced fitness function showed that it is able to significantly improve the GP performance compared to the original fitness function.

Our analysis of the original fitness function showed that, it can be considered as a special case of the norm-referenced fitness function in which the balance between exploration and exploitation can only be maintained in early generations. Once the original fitness function brings the population's average performance to a certain level, the balance of exploration and exploitation breaks and there is an implicit bias towards exploitation. So, original fitness function which drives evolution in early stage ends up becoming obsolete of evolution later on. We called this curse of evolution, which we consider as the theoretical evidence explaining why the original fitness function fails.

We also developed a modification of standard tournament selection called partial norm-referenced fitness function which applies the same concept of norm-referenced fitness function but only at tournament level. The partial norm-referenced fitness function addresses one potential runtime performance limitation of norm-referenced function in the cost of phenotype performance. But it is

still able to significantly outperforms the original fitness function especially given enough generations to evolve.

7.2 Further works

This thesis explored a variety of areas in genetic programming including bloating, fitness evaluation, crossover effects, crossover point selection, and fitness function. As a result, there are many choices for future directions of research. In this section, I list four of them.

Developing More Effective Crossover. The analysis of crossover effect using activation rate reveals that deeper crossover point results in higher chance of constructive crossover. An intuitive application of this observation is to place a minimum depth in the selection of crossover point. However, the choice of this minimum is hard to deduct. This is because on one hand, bigger minimum depth would results in higher chance of constructive crossover, but on the other hand, deeper crossover point results in smaller change on the individual's fitness as well. Moreover, based on modification point depth theory, deeper crossover point produces more bloating. Is there any connection between constructive crossover and bloating? Finding the optimal setting for this minimum depth requires balancing between fitness and bloating.

Exploring GP with Fixed Depth Population. When experimenting depth constraint crossover, we found that when we set the parameter ϵ's value to 0 to completely deny the ability for individual to grow its depth, it produces least amount of bloating with surprisingly reasonable fitness performance. In fact, apart from symbolic regression problem, in the other three domains, there is no statistically significant loss in fitness. This suggests that the relationship between tree size growth at genotype level and the fitness improvement at phenotype level may be much weaker than we previously thought. Equipped with a more sophisticated initial population generation algorithm, depth constraint crossover with ϵ equals 0 may able to be sufficient in finding optimal solution.

Extending the Curse of Evolution. During the analysis of norm-referenced fitness function, we discover one limitation of the original fitness function and named it as the curse of evolution. In addition to support norm-referenced fitness function, curse of evolution can also be used to explain the convergence of the evolution. The curse of evolution reveals one fundamental limitation of standard

GP setup. However, we believe norm-referenced fitness function only represents one possible approach to solve the curse of evolution. Alternative methods to resolve curse of evolution may lie in "cleverer" context-aware selection method which compensates the limitation of the original fitness function, similar to partial norm-referenced fitness function.

Exploring the Efficiency of Combining Depth Constraint Crossover and Norm-referenced Fitness Function. The depth constraint crossover developed in this thesis has shown its effectiveness in controlling bloating. On the other hand, the norm-referenced fitness function shows great potential in improving the GP performance. One next step is to explore how effectively we can tune GP when combining these two methods in more complex real world problems.

Bibliography

[ACEAS+08] Eva Alfaro-Cid, Anna Esparcia-Alcázar, Ken Sharman, Francisco Fernández de Vega, and J. J. Merelo. Prune and plant: A new bloat control method for genetic programming. In *HIS '08: Proceedings of the 2008 Eighth International Conference on Hybrid Intelligent Systems*, pages 31–35, Washington, DC, USA, 2008. IEEE Computer Society.

[Alt94a] L. Altenberg. The evolution of evolvability in genetic programming. In K. E. Kinnear, editor, *Advances in Genetic Programming*, pages 47–74. MIT Press, Cambridge, MA, 1994.

[Alt94b] 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 1994. World Scientific Publishing.

[Ang94] Peter J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in genetic programming*, pages 75–97. MIT Press, Cambridge, MA, USA, 1994.

[Ang96] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 21–29, Cambridge, MA, USA, 1996. MIT Press.

[Ang97] 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.

[Ang98] Peter J. Angeline. Subtree crossover causes bloat. 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 745–752, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[BA02] Terry Van Belle and David H. Ackley. Uniform subtree mutation. In *Proceedings of the 5th European Conference on Genetic Programming*, EuroGP '02, pages 152–161, London, UK, UK, 2002. Springer-Verlag.

[BB03] Markus Brameier and Wolfgang Banzhaf. Neutral variations cause bloat in linear GP. 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 286–296, Essex, 14-16 April 2003. Springer-Verlag.

[BBSW03] Wolfgang Banzhaf, Markus Brameier, Marc Stautner, and Klaus Weinert. Genetic programming and its application in machining technology. In Hans-Paul Schwefel, Ingo Wegener, and Klaus Weinert, editors, *Advances in Computational Intelligence: Theory and Practice*, Natural Computing Series, chapter 7, pages 194–242. Springer, 2003.

[BBTZ01] Stefan Bleuler, Martin Brack, Lothar Thiele, and Eckart Zitzler. Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 536–543, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[BBZ] Stefan Bleuler, Johannes Bader, and Eckart Zitzler. In Joshua Knowles, David Corne, and Kalyanmoy Deb, editors, *Multiobjective Problem Solving from Nature: from concepts to applications*, chapter 9, pages 177–200.

[BGS96] 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.

[BJ08] Lawrence Beadle and Colin G Johnson. Semantically driven crossover in genetic programming. In *IEEE World Congress on Computational Intelligence*, pages 111–116. IEEE, January 2008.

[BJ09] Lawrence Beadle and Colin G. John'son. Semantically driven mutation in genetic programming. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 1336–1342, Piscataway, NJ, USA, 2009. IEEE Press.

[BL02] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, 2002.

[Bli96] Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996.

[Bon96] Linda A. Bond. Norm- and criterion-referenced testing. *Practical Assessment, Research & Evaluation*, 5(2), Retrieved January 27, 2011, from `http://http://pareonline.net/getvn.asp?v=5&n=2,1996`.

[Bra04] Markus Brameier. *On Linear Genetic Programming*. PhD thesis, Fachbereich Informatik, Universität Dortmund, Germany, February 2004.

[BS73] Fischer Black and Myron S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[BT94] 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).

[BT95] Tobias Blickle and Lothar Thiele. A mathematical analysis of tournament selection. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 9–16, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[BT96] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evol. Comput.*, 4:361–394, December 1996.

[CC99] Michael J. Cavaretta and Kumar Chellapilla. Data mining using genetic programming: The implications of parsimony on generalization error. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1330–1337, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[Che97] Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.

[CM02] Vic Ciesielski and Dylan Mawhinney. Prevention of early convergence in genetic programming by replacement of similar programs. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 67–72. IEEE Press, 12-17 May 2002.

[da 08] Sara Guilherme Oliveira da Silva. *Controlling Bloat: Individual and Population Based Approaches in Genetic Programming.* PhD thesis, Coimbra University, Portugal, April 2008.

[DP07] 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 *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1588–1595, New York, NY, USA, 2007. ACM.

[DP08] 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*, volume 4971 of *LNCS*, pages 158–169, Naples, 26-28 March 2008. Springer.

[dVGPG04] Francisco Fernandez de Vega, German Galeano Gil, Juan Antonio Gomez Pulido, and Jose Luis Guisado. Control of bloat in genetic programming by means of the island model. 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 263–271, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[dWP01] Edwin D. de Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. 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 11–18, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[EN01] Anikó Ekárt and S. Z. Németh. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, 2001.

[EVDHVH98] A. E. Eiben, J. K. Van Der Hauw, and J. I. Van Hemert. Graph

coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, June 1998.

[FO82] Philippe Flajolet and Andrew Odlyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25(2):171 – 213, 1982.

[GAMRRP07] M. Garcia-Arnau, D. Manrique, J. Rios, and A. Rodriguez-Paton. Initialization method for grammar-guided genetic programming. *Knowledge-Based Systems*, 20(2):127–133, March 2007. AI 2006, The 26th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence.

[GB89] John J. Greffenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 20–27, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[GD91] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. San Francisco, CA: Morgan Kaufmann, 1991.

[GEBK04] Steven Gustafson, Anikó Ekárt, Edmund Burke, and Graham Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 5:271–290, September 2004.

[GGP11] Marc-André Gardner, Christian Gagné, and Marc Parizeau. Bloat control in genetic programming with a histogram-based accept-reject method. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 187–188, New York, NY, USA, 2011. ACM.

[GR96] 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.

[GTV02] Mario Giacobini, Marco Tomassini, and Leonardo Vanneschi. Limiting the number of fitness cases in genetic programming using statistics. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 371–380, London, UK, 2002. Springer-Verlag.

[Har12] Robin Harper. Spatial co-evolution: quicker, fitter and less bloated. In Terry Soule, Anne Auger, Jason Moore, David Pelta, Christine Solnon, Mike Preuss, Alan Dorin, Yew-Soon Ong, Christian Blum, Dario Landa Silva, Frank Neumann, Tina Yu, Aniko Ekart, Will Browne, Tim Kovacs, Man-Leung Wong, Clara Pizzuti, Jon Rowe, Tobias Friedrich, Giovanni Squillero, Nicolas Bredeche, Stephen Smith, Alison Motsinger-Reif, Jose Lozano, Martin Pelikan, Silja Meyer-Nienberg, Christian Igel, Greg Hornby, Rene Doursat, Steve Gustafson, Gustavo Olague, Shin Yoo, John Clark, Gabriela Ochoa, Gisele Pappa, Fernando Lobo, Daniel Tauritz, Jurgen Branke, and Kalyanmoy Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 759–766, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[Hay98] Thomas Haynes. Collective adaptation: The exchange of coding segments. *Evol. Comput.*, 6(4):311–338, 1998.

[HH08] Kassel Hingee and Marcus Hutter. Equivalence of probabilistic tournament and polynomial ranking selection. In *IEEE Congress on Evolutionary Computation*, pages 564–571. IEEE, 2008.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[HS97] Kim Harries and Peter Smith. Exploring alternative operators and search strategies 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 147–155, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Iba96] Hitoshi Iba. Random tree generation for genetic programming. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberger, and Hans-Paul Schwefel, editors, *PPSN*, volume 1141 of *Lecture Notes in Computer Science*, pages 144–153. Springer, 1996.

[IC99] Christian Igel and Kumar Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. 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 1061–1068, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[IdGS94] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Advances in genetic programming. chapter Genetic programming using a minimum description length principle, pages 265–284. MIT Press, Cambridge, MA, USA, 1994.

[IIS99] Takuya Ito, Hitoshi Iba, and Satoshi Sato. A self-tuning mechanism for depth-dependent crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 16, pages 377–399. MIT Press, Cambridge, MA, USA, June 1999.

[Jac05] David Jackson. Fitness evaluation avoidance in boolean GP problems. 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 2530–2536, Edinburgh, UK, 2-5 September 2005. IEEE Press.

[JF97] Terence Soule James and James A. Foster. Code size and depth flows in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320. Morgan Kaufmann, 1997.

[KABK99] John R. Koza, David Andre, Forrest H. Bennett, and Martin A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[KCS06] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[Kei04] Maarten Keijzer. Alternatives in subtree caching for genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *EuroGP*, volume 3003 of *Lecture Notes in Computer Science*, pages 328–337. Springer, 2004.

[KEK93] Jr. Kenneth E. Kinnear. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 287–294, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[Kin93] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 287–294, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[KM94] Mike J. Keith and Martin C. Martin. Genetic programming in c++: implementation issues. In *Advances in genetic programming*, pages 285–310. MIT Press, Cambridge, MA, USA, 1994.

[KM99] Tatiana Kalganova and Julian Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, EH '99, pages 54–63, Washington, DC, USA, 1999. IEEE Computer Society.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, December 1992.

[Koz94] John R. Koza. *Genetic programming II: automatic discovery of reusable programs.* MIT Press, Cambridge, MA, USA, 1994.

[Lan96a] W. B. Langdon. *Data Structures and Genetic Programming.* PhD thesis, University College, London, 27 September 1996.

[Lan96b] William B. Langdon. Data structures and genetic programming. In *Advances in genetic programming: volume 2*, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[Lan98] 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.

[Lan00a] W. B. Langdon. Quadratic bloat in genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[Lan00b] W. B. Langdon. Size fair and homologous tree crossovers for tree genetic programming. *Genetic Programming and Evolvable Machines*, 1(1-2):95–119, 2000.

[Lan09] W. B. Langdon. A CUDA SIMT interpreter for genetic programming. Technical Report TR-09-05, Department of Computer Science, King's College London, Strand, WC2R 2LS, UK, 18 June 2009. Revised.

[LBP03] Sean Luke, Gabriel Catalin Balan, and Liviu Panait. Population implosion in genetic programming. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 1729–1739, Berlin, Heidelberg, 2003. Springer-Verlag.

[Lev91] James R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127. Morgan Kaufmann, 1991.

[LP97a] W. B. Langdon and R. Poli. An analysis of the MAX problem 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 222–230, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[LP97b] W. B. Langdon and R. Poli. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*, pages 23–27. Springer-Verlag, 1997.

[LP98a] 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.

[LP98b] William B. Langdon and Riccardo Poli. Fitness causes bloat: Mutation. In *EuroGP '98: Proceedings of the First European Workshop on Genetic Programming*, pages 37–48, London, UK, 1998. Springer-Verlag.

[LP02a] Sean Luke and Liviu Panait. Fighting bloat with nonparametric parsimony pressure. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 411–421, London, UK, 2002. Springer-Verlag.

[LP02b] Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[LP06] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evol. Comput.*, 14(3):309–344, 2006.

[LS97] 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.

[LS98] 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.

[LSPF99] William B. Langdon, Tery Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In *Advances in genetic programming: volume 3*, pages 163–190. MIT Press, Cambridge, MA, USA, 1999.

[Luk00a] 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 2000.

[Luk00b] 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, 2000.

[Luk00c] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000.

[Luk03] Sean Luke. Modification point depth and genome growth in genetic programming. *Evol. Comput.*, 11(1):67–106, 2003.

[Luk09] Sean Luke. ECJ 19: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2009.

[LZ10a] Geng Li and Xiao-Jun Zeng. Bottom-up tree evaluation in tree-based genetic programming. In Ying Tan, Yuhui Shi, and Kay Chen Tan, editors, *The first International Conference on Swarm Intelligence*, volume 6145 of *Lecture Notes in Computer Science*, pages 513–522. Springer, 2010.

[LZ10b] Geng Li and Xiao-Jun Zeng. Controlling bloating using depth constraint crossover. In *Computational Intelligence (UKCI), 2010 UK Workshop on Computation Intelligence*, Sept. 2010.

[LZ11] Geng Li and Xiao-Jun Zeng. Genetic programming with a norm-referenced fitness function. In Natalio Krasnogor and Pier Luca Lanzi, editors, *13th Annual Genetic and Evolutionary Computation Conference*, GECCO '11, pages 1323–1330, 2011.

[MFH92] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1992.

[MH08] Julian Francis Miller and Simon L. Harding. Cartesian genetic programming. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, GECCO '08, pages 2701–2726, New York, NY, USA, 2008. ACM.

[MM95] Nicholas F. Mcphee and Justin D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference*, ICGA95, pages 303–309, Pittsburgh, PA, USA, JanuaryMay-JanuarySeptember 1995. Morgan Kaufmann.

[NB95] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 310–317, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[NFB96] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In *Advances in genetic programming: volume 2*, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[OCPT97] Mouloud Oussaidène, Bastien Chopard, Olivier V. Pictet, and Marco Tomassini. Parallel genetic programming and its application to trading model induction. *Parallel Computing*, 23(8):1183–1198, August 1997.

[PL97a] Riccardo Poli and W. B. Langdon. Genetic programming with one-point crossover. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer-Verlag London, 23-27 June 1997.

[PL97b] 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.

[PL98] 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.

[PL04] Liviu Panait and Sean Luke. Alternative bloat control methods. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 630–641, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[PLD07] 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.

[PLM08] 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. (With contributions by J. R. Koza).

[PM08] Riccardo Poli and Nicholas Freitag McPhee. Parsimony pressure made easy. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1267–1274, New York, NY, USA, 2008. ACM.

[PMV08] Riccardo Poli, Nicholas Freitag McPhee, and Leonardo Vanneschi. The impact of population size on code growth in gp: analysis and

empirical validation. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1275–1282, New York, NY, USA, 2008. ACM.

[Pol97] Riccardo Poli. Parallel distributed genetic programming applied to the evolution of natural language recognisers. In David Corne and Jonathan L. Shapiro, editors, *Evolutionary Computing, AISB Workshop*, volume 1305 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 1997.

[Pol03] 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.

[PS06] Alan Piszcz and Terence Soule. A survey of mutation techniques in genetic programming. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 951–952, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[Rob91] D S Robertson. Feedback theory and darwinian evolution. *Journal of Theoretical Biology*, 152(4):469–484, 1991.

[Ros95] Justinian P. Rosca. Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85. AAAI, 1995.

[RV10] Cristóbal Romero and Sebastián Ventura. Educational data mining: A review of the state of the art. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 40(6):601–618, 2010.

[SA05] Sara Silva and Jonas Almeida. Gplab-a genetic programming toolbox for matlab. In *In Proc. of the Nordic MATLAB Conference (NMC-2003*, pages 273–278, 2005.

[SC04] Sara Silva and Ernesto Costa. Dynamic limits for bloat control: Variations on size and depth. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 666–677, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[SC05a] 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.

[SC05b] Sara Silva and Ernesto Costa. Resource-limited genetic programming: the dynamic approach. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1673–1680, New York, NY, USA, 2005. ACM.

[Sch90] Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2):197–227, July 1990.

[SCZ09] Andy Song, Dunhai Chen, and Mengjie Zhang. Bloat control in genetic programming by evaluating contribution of nodes. In

GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pages 1893–1894, New York, NY, USA, 2009. ACM.

[SD06] Lothar M. Schmitt and Stefan Droste. Convergence to global optima for genetic programming systems with dynamically scaled operators. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 879–886, New York, NY, USA, 2006. ACM.

[SDV12] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. Operator equalisation for bloat free genetic programming and a survey of bloat control methods. *Genetic Programming and Evolvable Machines*, 13(2):197–238, June 2012.

[SE91] J.D. Schaffer and L.J. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R.K. Belew and L.B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[SF98a] Terence Soule and James A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evol. Comput.*, 6(4):293–309, December 1998.

[SF98b] 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.

[SFD96] Terence Soule, James A. Foster, and John Dickinson. Code growth 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 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[SH98] Peter W. H. Smith and Kim Harries. Code growth, explicitly defined introns, and alternative selection schemes. *Evol. Comput.*, 6(4):339–360, 1998.

[SH02] Terence Soule and Robert B. Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283–309, 2002.

[Sil11] Sara Silva. Reassembling operator equalisation: a secret revealed. In Natalio Krasnogor, Pier Luca Lanzi, Andries Engelbrecht, David Pelta, Carlos Gershenson, Giovanni Squillero, Alex Freitas, Marylyn Ritchie, Mike Preuss, Christian Gagne, Yew Soon Ong, Guenther Raidl, Marcus Gallager, Jose Lozano, Carlos Coello-Coello, Dario Landa Silva, Nikolaus Hansen, Silja Meyer-Nieberg, Jim Smith, Gus Eiben, Ester Bernado-Mansilla, Will Browne, Lee Spector, Tina Yu, Jeff Clune, Greg Hornby, Man-Leung Wong, Pierre Collet, Steve Gustafson, Jean-Paul Watson, Moshe Sipper, Simon Poulding, Gabriela Ochoa, Marc Schoenauer, Carsten Witt, and Anne Auger, editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1395–1402, Dublin, Ireland, 12-16 July 2011. ACM.

[Sou98] Terence Soule. *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.

[Spe96] Lee Spector. Simultaneous evolution of programs and their control structures. In *Advances in genetic programming: volume 2*, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[Str03] 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.

[SV09] Sara Silva and Leonardo Vanneschi. Operator equalisation, bloat and overfitting: a study on human oral bioavailability prediction. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1115–1122, New York, NY, USA, 2009. ACM.

[Tac94] 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.

[Tet96] Andrea G. B. Tettamanzi. Genetic programming without fitness. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 193–195, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[TNM13] Leonardo Trujillo, Enrique Naredo, and Yuliana Martinez. Preliminary study of bloat in genetic programming with behavior-based search. In Michael Emmerich, Andre Deutz, Oliver Schuetze, Thomas Baeck, Emilia Tantar, Alexandru-Adrian, Pierre Del Moral, Pierrick Legrand, Pascal Bouvry, and Carlos A. Coello, editors, *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV*, volume 227 of *Advances in Intelligent Systems and Computing*, pages 293–305, Leiden, Holland, July 10-13 2013. Springer.

[VTCC03] Leonardo Vanneschi, Marco Tomassini, Philippe Collard, and Manuel Clergue. Fitness distance correlation in structural mutation genetic programming. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 455–464, Berlin, Heidelberg, 2003. Springer-Verlag.

[Whi96] P. A. Whigham. Search bias, language bias and genetic programming. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 230–237, Cambridge, MA, USA, 1996. MIT Press.

[WHM+97] Mark Willis, Hugo Hiden, Peter Marenbach, Ben McKay, and Gary A. Montague. Genetic programming: An introduction and survey of applications. In Ali Zalzala, editor, *Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, University of Strathclyde, Glasgow, UK, 1-4 September 1997. Institution of Electrical Engineers.

[Wik13] Wikipedia. Graduate record examinations, 2013. [Online; accessed 24-Augest-2013].

[WL96] Annie S. Wu and Robert K. Lindsay. A survey of intron research in genetics. In *PPSN IV: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 101–110, London, UK, 1996. Springer-Verlag.

[XZ12] Huayang Xie and Mengjie Zhang. Impacts of sampling strategies in tournament selection for genetic programming. *Soft Comput.*, 16(4):615–633, April 2012.

[XZA06a] Huayang Xie, Mengjie Zhang, and Peter Andreae. Automatic selection pressure control in genetic programming. In *Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications - Volume 01*, ISDA '06, pages 435–440, Washington, DC, USA, 2006. IEEE Computer Society.

[XZA06b] Huayang Xie, Mengjie Zhang, and Peter Andreae. Population clustering in genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *EuroGP*, volume 3905 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 2006.

[XZA07] Huayang Xie, Mengjie Zhang, and Peter Andreae. An analysis of constructive crossover and selection pressure in genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1739–1748, New York, NY, USA, 2007. ACM.

[Zha97] Byoung-Tak Zhang. A taxonomy of control schemes for genetic code growth. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.

[ZM95] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evol. Comput.*, 3(1):17–38, 1995.