THESIS

Sub-Symbolic Representation and Search Operators For Genetic Programming

by

Jonathan Page

Thesis submitted to the School of Computer Science
of the University of Birmingham for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
December 1999

Acknowledgments

Over the last decade or so, I have spoken to a lot of PhD students from a lot of disciplines, and I have heard a lot of horror stories. The usual suspects feature large in all these: aimlessness, lack of motivation, experiments that resolutely refuse to give the desired results and a general sense of impending disaster. Inevitably, mention is made of the supervisor. I have listened to bitter accounts of supervisors who are almost permanently unavailable, who will not listen to, or are overly dismissive of, their students' ideas, or who know little of their students' research areas. It has also become clear that the problems are generally not due to malice or negligence on the part of the supervisor. In the majority of cases, it is simply that the demands of providing good supervision come into conflict with those of being a successful researcher whilst at the same time managing a heavy teaching load. How Riccardo Poli manages all three remains a mystery to me. In the last four years, I have watched the explosive growth of Riccardo's publications list and seen his name appear on an increasing number of organising and program committees of major conferences. Yet in all this time, when I have asked to see him, I have never had to wait more than a day. These meetings have never been hurried, and whilst I have often walked into his office despondent, I have always emerged inspired and motivated. He displays a level of patience and understanding that one would normally only expect in a supervisor of considerable experience, and it is therefore all the more astonishing that I should have been amongst his first PhD supervisees. I remain indebted to Riccardo, as I hope will many future students.

I have received much useful advice from the members of my thesis group, Aaron Sloman and Manfred Kerber. I know from speaking to their supervisees that both possess much the same qualities as Riccardo, and our meetings have always left me encouraged and optimistic. Latterly I have also received a great deal of useful feedback from Bill Langdon.

A special mention must go to Indu Kaila whose appearence on the scene coincided with the beginning of the write-up. As a result, she has had to tolerate six months of moodiness and irritability, and she has done so with unstinting love and patience. Thanks, Indu.

My housemate of over two years, John Hodsoll, has also experienced the highs and lows of my research at the sharp end. His dry wit in all matters, and particularly those connected with both his research and my own, has had a considerable effect on sustaining my morale.

Whilst some research students become married to their work, others require a certain amount of extra-curricular activity to function normally. I fall into the latter category, and I would like to thank Erica Richardson, Bill Westwater, Paul Holtom and Daniel Hardy for reminding me, every friday night, that there is life outside evolutionary algorithms. From amongst my peers, I would especially like to thank Tim Kovacs, Ian Wright, Alessio Lomuscio, Sofia Guerra, Dorota Iskra, John Woodward, Stuart Reynolds and Axel Grossmann, all of whom have made me laugh when I have had no cause for laughing.

Outside Birmingham, Lars Williams, Kirsty Stevenson and Hugh Seton have remained close friends and provided me with a long-distance social life.

Finally, I would like to thank my family who have provided much support and encouragement for well over ten years. I hereby promise to get a proper job.

Abstract

Genetic Programming (GP) is one of the more recent additions to a diverse body of biologically-inspired search techniques known as evolutionary algorithms (EAs). GP differs from most other EAs in that candidate solutions are executable programs of arbitrary size which are evaluated according to how well they perform on a specified task. Whilst GP has been successfully applied to a number of problem domains, there remain many tasks on which performance is notably poor.

This thesis argues that poor performance is, in many cases, due to the nature of the search that GP's operators perform. GP's primary operator, crossover, has been shown to perform a predominantly local search of the program space, and this can make global optima difficult to locate if the fitness landscape is characterised by long stretches of very weak or neutral fitness gradients.

Recently, a GP search operator that performs global search has been proposed. This operator is known as *GP uniform crossover*, and in this thesis we test it extensively on a number of benchmark problems. We also propose a novel *sub-symbolic representation* for function nodes which can be used in conjunction with GP uniform crossover, and which affords GP the additional ability to perform a very local, directed search of the program space.

The experiments reported in this thesis raise a number of issues that are discussed in detail. The sub-symbolic representation, as used here, constrains the size of the function set and in many cases this is significantly larger than those used in most GP implementations. Despite of this, we report enhanced performance on a number of problems and this leads us to question the received wisdom which states that the recommends a minimalist approach to function set design. The use of different operators and representations transforms the search space and highlights a number of properties of GP program spaces that have so far been overlooked. We discuss these and, in the light of our findings, reevaluate some of the hypotheses that have been put forward concerning the program spaces of some well-known functions. The search operators used here also constrain the complexity of the programs that may be

evolved. We examine the advantages of this, particularly with respect to controlling program size and resultant run-time, and the disadvantages.

The issues arising from these studies are diverse and many have received little attention in the literature. We believe that the findings reported here pose numerous questions, and we suggest several directions for future research.

Contents

1 Introduction **2**

2 Evolutionary Computation and Genetic Programming **8**

2.1 Introduction 8

2.2 An Overview of Genetic Algorithms and Genetic Programming 9

2.2.1 Core Aspects of GAs and GP 11

2.3 Genetic Programming 14

2.3.1 Tree Representation in GP 15

2.3.2 Evaluation in GP 18

2.3.3 GP Search Operators 19

2.4 Conclusion 22

3 The Efficiency of GP Search **23**

3.1 Bloat 24

3.1.1 Introns and Bloat 25

3.1.2 Other Causes of Bloat 28

3.1.3 Counteracting Bloat 29

3.2 Efficiency of GP's Search Operators 30

3.2.1 Does GP Crossover Construct Building Blocks? 31

3.2.2 Global and Local Search in GP 35

3.3 Summary 37

4 Global And Local Search Operators and Boolean Representations 40

4.1 Introduction 40

4.2 Tree Representation and Search Operators 42

4.3 Genotype, Phenotype and Fitness in GP 47

4.3.1 GA Search Operators and Boolean Phenotype Spaces 48

4.4 Summary 53

5 Evolving Boolean Functions with Binary-Represented Genetic Algorithms 54

5.1 Introduction 54

5.2 Previous Work 55

5.3 Experimental Set-up 58

5.4 Experiments with the Arity-2 Function Set 60

5.4.1 Even-n-Parity Problem 60

5.4.2 6-Multiplexer 66

5.5 Experiments with the Arity-3 Function Set 67

5.5.1 6-Multiplexer 67

5.5.2 Even-n-Parity Problem 70

5.6 Summary of Results 74

5.7 Discussion 75

5.8 Conclusion 77

6 Global GP Operators and Boolean Functions **79**

6.1 Introduction 79

6.2 GP Uniform Crossover and Point Mutation 80

6.2.1 GP Smooth Uniform Crossover and Point Mutation 83

6.3 Experimental Studies 84

6.3.1 Experimental Set-up 84

6.4 Experiments with the Arity-2 Function Set 86

6.4.1 Even-n-Parity Problem 86

6.4.2 6-Multiplexer 91

6.4.3 Building Blocks and the Parity Functions 92

6.4.4 The Even-n- Parity Problems and Automatically Defined Functions 96

6.5 Experiments with the Arity-3 Function Set 101

6.5.1 6-Multiplexer 101

6.5.2 Even-n-Parity Problem 104

6.6 Summary of Results 109

6.7 Discussion 111

6.8 Conclusion 115

7 Symbolic Regression of Polynomial Functions **116**

7.1 Introduction 116

7.2 Interpolated Arithmetic Operators 118

7.3 Representing Interpolated Operators with a Binary Encoding 121

7.4 Experimental Studies 123

7.4.1 Order 3 and 5 Polynomials with Integer Coefficients 124

7.4.2 Polynomials with Real-Valued Coefficients 131

7.4.3 "Building Block" Functions 137

7.5 Partially-Interpolated Function Nodes 148

7.5.1 Experiments with the Partially-Interpolated Function Set . . . 149

7.6 Summary of Results 153

7.7 Discussion 156

7.8 Conclusion 161

8 Interpolating Arithmetic Functions with Real-Encoded Parameters. 163

8.1 Encoding Interpolated Arithmetic Operators with Real-Valued Parameters 164

8.2 Experiments with the Real-Encoded Representation 165

8.2.1 The Koza Polynomial, $2.718x^2 + 3.1416x$ 166

8.2.2 The Small 2-bit Building Block Function 168

8.2.3 The Large 2-bit Building Block Function 172

8.2.4 The 3-bit Building Block Function 174

8.3 Summary 178

8.4 Conclusion 178

9 Conclusions 181

9.1 Contributions Made by this Thesis 182

9.2 Further Work 184

9.2.1 Extending GP-SUX 184

9.2.2 Alternative Search Strategies 185

9.2.3 Function Sets 186

9.2.4 Bias Induction 186

9.2.5 Combining GP-SUX with other Techniques 187

9.3 Summary 188

A Applying the Sub-Symbolic Approach to Other Problems – An Example. 189

List of Figures

2.1 The core algorithm for GAs and GP. 10

2.2 GA Uniform Crossover illustrated on two parental bitstrings. Two offspring are generated in this example, although many implementations discard one. 11

2.3 An example GP syntax tree. The terminal (leaf) nodes code arguments to the function which are evaluated by the upper function nodes. 15

2.4 Standard GP crossover acting on two parental syntax trees. The sub-trees selected for crossover are highlighted. 20

2.5 Standard GP mutation acting on a parental syntax trees. The highlighted sub-tree is replaced by another generated at random. 21

3.1 Inviable and inoperative code in a GP syntax tree. 26

4.1 Encoding a two dimensional syntax tree on a one-dimensional string using prefix notation. 42

4.2 Uniform crossover applied to two one dimensional strings, and its effect on the offspring syntax trees. 43

4.3 Uniform crossover applied to a binary bitstring encoding a Boolean syntax tree. The values of the bits at each node specify the truth table of the function. 45

5.1 A Depth 3 solution to the even-6-parity problem 63

5.2 A Depth 5 solution to the even-6-parity problem 63

5.3 Simplified form of the tree shown in Figure 5.2. 64

5.4 A Depth 5 solution to the 6-multiplexer problem using arity-2 functions . 67

5.5 A Depth 2 solution to the 6-multiplexer problem using arity-3 functions . 69

5.6 Simplified version of the tree shown in Figure 5.5. 69

5.7 Depth 2 solution to the even-6-parity problem using arity-3 functions. . . 72

5.8 Simplified version of the tree shown in Figure 5.7. 72

5.9 Depth 3 solution to the even-6-parity problem using arity-3 functions. . . 73

5.10 Simplified version of the tree shown in Figure 5.9. 73

6.1 Two parental parse trees prior to GP-UX. 81

6.2 Offspring trees after GP-UX. 82

6.3 Uniform crossover on the sub-symbolic representation. 83

6.4 29 node solution to the even-6-parity problem found by GP-SUX on the maximum initial depth = 5 condition. 91

6.5 Simplified version of the solution shown in Figure 6.4 92

6.6 58 node arity-3 solution to the even-7-parity problem found by standard GP 105

6.7 Simplified version of the solution shown in Figure 6.6 106

6.8 40 node solution to the even-8-parity found by GP-SUX with maximum initial tree depth 4. 109

6.9 Simplified version of the solution shown in Figure 6.8. 110

7.1 Example solution to the quartic polynomial function $x^4 + x^3 + x^2 + x$. . . 117

7.2 A syntax tree equivalent to a single interpolated arithmetic node constructed from pure arithmetic operators. The terminal set includes the three interpolating parameters, and the symbols $a1$ and $a2$, which are the arguments to the interpolated node. 120

7.3 Example tree using 2-bit binary encoding. The three parameters encoded on this tree are $\alpha = \frac{1}{3}$, $\beta = \frac{2}{3}$ and $\gamma = \frac{2}{3}$. 122

7.4 Graph of the quintic polynomial function $x^5 - 2x^3 + x$ over the interval [0, 1]. 125

7.5 Graph of the triptic polynomial function $x^3 - x$ over the interval [0, 1]. . . 126

7.6 An 11 node solution to the triptic polynomial function $x^3 - x$ found by GP-UX with the 2-bit function set. This tree contains 2 interpolated arithmetic nodes 128

7.7 A 13 node solution to the triptic polynomial function $x^3 - x$ found by GP-SUX with the 3-bit function set. This tree contains 3 interpolated arithmetic nodes 128

7.8 Graph of the Koza function $2.718x^2 - 3.1416x$ over the interval [-1.0,+1.0]. 132

7.9 A 67 node solution to the Koza function $2.718x^2 - 3.1416x$ found by standard GP with the 1-bit function set. 134

7.10 Sub-optimal solution to the Koza polynomial found by GP-UX with the 2-bit function set. This tree evaluates $0.037x^3 + 2.741x^2 + 2.963x$. 135

7.11 Sub-optimal solution to the Koza polynomial found by GP-SUX with the 2-bit function set. This tree evaluates $-0.037x^3 + 2.741x^2 + 3.148x$. . . . 135

7.12 Solution to the Koza polynomial found by GP-SUX with the 3-bit function set. This tree evaluates $2.714x^2 + 3.143x$. 136

7.13 The target tree from which the smaller 2-bit building block function is derived. 138

7.14 Graph of the smaller 2-bit building block function over the interval [-1.0,+1.0]. 139

7.15 11 node solution to the smaller 2-bit building block function found by GP-UX. 141

7.16 Target tree from which the larger 2-bit building block function is derived. . 142

7.17 Graph of the larger 2-bit building block function over the interval [-1.0,+1.0]. 142

7.18 Target tree from which the 3-bit building block function is derived. 144

7.19 Plot of the 3-bit building block function over the interval [-1.0,+1.0]. . . . 145

7.20 Sub-optimal solution to the 3-bit building block function found by GP-UX with the 3-bit function set. 146

7.21 Sub-optimal solution to the 3-bit building block function found by GP-SUX with the 3-bit function set. 147

7.22 Plot of the function generated by the tree shown in Figure 7.21 over the interval [-1.0,+1.0]. 147

7.23 Solution to the Koza polynomial found by GP-SUX with the 3-bit partially interpolated function set. Both operator and parameter are given for the two interpolated functions in this tree; the remainder are uninterpolated. This tree evaluates to $\frac{19}{7}x^2 + \frac{22}{7}x$. 152

8.1 Example tree using real-valued parameter encoding. 164

8.2 Solution to the Koza fitness function found by GP-SUX using the fully-interpolated real-valued parameter encoding. The parameters given in all the figures presented in this chapter are rounded to 2 decimal places for ease of display. 167

8.3 Solution to the Koza fitness function found by GP-UX using the fully-interpolated real-valued parameter encoding. 168

8.4 Solution to the smaller 2-bit building block function found by GP-UX using the fully-interpolated real-valued parameter encoding. 171

8.5 Solution to the smaller 2-bit building block function found by GP-SUX using the fully-interpolated real-valued parameter encoding. 171

8.6 Sub-optimal solution to the larger 2-bit building block function found by GP-SUX using the fully-interpolated real-valued parameter encoding. . . . 173

8.7 Sub-optimal solution to the larger 2-bit building block function found by GP-UX using real-valued parameter encoding. 173

8.8 Sub-optimal solution to the 3-bit building block function found by GP-SUX using real-valued parameter encoding. 175

8.9 Sub-optimal solution to the 3-bit building block function found by GP-SUX using real-valued parameter encoding. 176

8.10 The target 3-bit tree with real-encoded parameters. 176

List of Tables

4.1 Impact on phenotype, in terms of average number of bits flipped in the concatenated output string, as a result of a single point mutation applied to a randomly selected node at depth d. The results given are for full trees of depth 6 with the arity 2 function set and symbolic representation. . . . 49

4.2 Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 2 function set and sub-symbolic representation. 49

4.3 Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 3 function set and symbolic representation. 51

4.4 Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 3 function set and sub-symbolic representation. 51

5.1 Minimum efforts (in thousands of fitness evaluations) required to solve the even-n-parity problem using various methods. 56

5.2 Parameters used in all the experiments reported in this chapter. 59

5.3 Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. 60

5.4 Minimum efforts required by the symbolic representation to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. 61

5.5 Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem, population size = 200. 61

5.6 Minimum efforts required by the symbolic representation to solve the even-n-parity problem, population size = 200. 62

5.7 Minimum efforts and number of successes (out of 50) for the even-8-parity problem with terminal mutation. Only trees of depth 3 were considered. . 65

5.8 Minimum efforts and number of successes (out of 50) for the 6-multiplexer problem with the full arity-3 function set, sub-symbolic representation and population size = 50. 68

5.9 Minimum efforts and number of successes for the 6-multiplexer problem with the sub-symbolic representation and population size = 200. 68

5.10 Minimum efforts and number of successes for the 6-multiplexer problem with the symbolic representation and population size = 200. 69

5.11 Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem with a full arity-3 function set and population size = 50. 70

5.12 Minimum efforts required by the symbolic representation to solve the even-n-parity problem with a full arity-3 function set and population size = 50. 70

5.13 Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem for with population size = 200. 71

5.14 Minimum efforts required by the symbolic representation to solve the even-n-parity problem with population size = 200. 71

6.1 Parameters used in all the experiments reported in this chapter. 84

6.2 Parameters used in the standard GP runs reported in this chapter. 85

6.3 Parameters used in the GP-UX and GP-SUX runs reported in this chapter. 85

6.4 Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using a full arity-2 function set. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given. 87

6.5 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 88

6.6 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 89

6.7 Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity 2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 90

6.8 Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 91

6.9 Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the reduced function set {`AND`, `OR`, `NAND`, `NOR`, `EQ`, `XOR`}. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given. 93

6.10 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using the reduced function set {AND, OR, NAND, NOR, EQ, XOR} and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 94

6.11 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using the reduced function set {AND, OR, NAND, NOR, EQ, XOR} and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 95

6.12 Minimum efforts required by GP-UX and GP-PM with a population size of 50 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with EQ and XOR omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 97

6.13 Minimum efforts required by GP-UX and GP-PM with a population size of 200 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with EQ and XOR omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 98

6.14 Minimum efforts required by GP-SUX and GP-SPM with a population size of 50 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with EQ and XOR omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 99

6.15 Minimum efforts required by GP-SUX and GP-SPM with a population size of 200 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with EQ and XOR omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 100

6.16 Minimum efforts required by Standard GP with 1 ADF branch to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the full and reduced arity-2 function sets. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given. 100

6.17 Minimum efforts required by Standard GP with 2 ADF branches to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the full and reduced arity-2 function sets. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given. 101

6.18 Minimum efforts required by standard GP to solve the 6-Multiplexer problem using a full arity-3 function set. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 102

6.19 Minimum efforts required by GP-UX and GP-PM to solve the 6-Multiplexer problem using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 102

6.20 Minimum efforts required by GP-SUX and GP-SPM to solve the 6-Multiplexer problem using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 103

6.21 Minimum efforts required by GP-SUX and GP-SPM to solve the 6-Multiplexer problem using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 104

6.22 Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using a full arity-3 function set. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given. 105

6.23 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 107

6.24 Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 107

6.25 Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 108

6.26 Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given. 109

6.27 Number of generations and evaluations required to solve high-order versions of the even-n-parity problem. The number of fitness cases per individual is also shown to emphasise how rapidly the actual computational effort increases with n. 114

7.1 The effects of varying α, β and γ on which operators are interpolated to produce the return value. The value `non-int` indicates that the parameter is a non-integer value in the range $0 < p < 1$. The value `any` indicates that the parameter can take *any* value without affecting the value returned by the function 121

7.2 Parameters used in all the experiments reported in this chapter. 123

7.3 Minimum effort required by standard GP, GP-UX and GP-SUX to solve the triptic polynomial. 126

7.4 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated over all 50 runs of the triptic polynomial. With the exception of the total nodes evaluated, these data are based on runs where GP failed to find a solution, and the number of runs (out of 50) is also given. 127

7.5 Minimum effort required by standard GP, GP-UX and GP-SUX to solve the quintic polynomial. 129

7.6 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over all 50 runs of the quintic polynomial. The number of runs on which the fitness and complexity data are based is also given for each condition. 130

7.7 Minimum effort required by standard GP, GP-UX and GP-SUX to solve the Koza function. 133

7.8 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and total number of nodes evaluated over 50 runs of the Koza polynomial. The number of runs on which the fitness and complexity data are based is also given for each condition. 133

7.9 Minimum effort required by standard GP, GP-UX and GP-SUX to solve the smaller 2-bit building block function. 139

7.10 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated complete 50 runs of the smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition. 140

7.11 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated over 50 runs of the larger 2-bit building block function. 143

7.12 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the 3-bit building block function. 145

7.13 Minimum efforts required by standard GP, GP-UX and GP-SUX to solve the Koza function using the partially interpolated function set. 150

7.14 Minimum efforts required by standard GP, GP-UX and GP-SUX to solve the Small 2-bit building block function using the partially interpolated function set. 153

7.15 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition. 154

7.16 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Large 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition. 155

7.17 Minimum effort required by standard GP, GP-UX and GP-SUX to solve the 3-bit building block function using the partially interpolated function set. . 156

7.18 Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition. 157

8.1 Minimum effort required by standard GP, GP-UX and GP-SUX running the fully-interpolated real-encoded representation to solve the Koza function. 166

8.2 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Koza function with the fully-interpolated real-valued representation. 166

8.3 Minimum effort required by standard GP, GP-UX and GP-SUX running the partially interpolated real-encoded representation to solve the Koza function. 169

8.4 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Koza function with the partially-interpolated real-valued representation. 169

8.5 Minimum effort required by standard GP, GP-UX and GP-SUX running the fully-interpolated real-encoded representation to solve the smaller 2-bit building block function. 169

8.6 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and total number of nodes evaluated over 50 runs of the smaller 2-bit building block function with the fully-interpolated real-encoded representation. 170

8.7 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the small 2-bit function with the partially-interpolated real-valued representation. 172

8.8 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the larger 2-bit building block function with the fully-interpolated real-encoded representation. 172

8.9 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the large 2-bit function with the partially-interpolated real-valued representation. 174

8.10 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the 3-bit building block function with the fully-interpolated real-valued encoding. 175

8.11 Minimum effort required by standard GP, GP-UX and GP-SUX running the partially interpolated real-encoded representation to solve the 3-bit function. 177

8.12 Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the 3-bit function with the partially-interpolated real-valued representation. 177

Chapter 1

Introduction

The idea of developing machine learning techniques based on abstractions of natural adaptive processes is not new. The work of McCulloch and Pitts [75] on the perceptron and Friedberg's use of evolutionary principles in an optimisation algorithm in the 1940s and 1950s are each examples of what may be collectively termed natural computation [10]. Moreover, these two techniques were to prove the forerunners of two distinct classes of artificial intelligence technique. The first, which simulate the neural processes in the brains of higher animals, are today known as *artificial neural networks* [40]. The second simulates the progressive adaptation of organisms to their environments via the mechanisms of evolution and natural selection. A number of variants on this theme exist, and they are collectively known as *evolutionary algorithms* (EAs) [8].

Of these two, the latter is of interest to this thesis. Specifically, the studies reported in this thesis are concerned with a recent newcomer to the field of EAs – *Genetic Programming* (GP) [58]. The aim of GP is ambitious. It represents the latest step towards automatic programming, wherein computers are able to program themselves to solve a given problem. In this respect it differs from many other EAs, which are principally concerned with the optimisation of problem-specific parameters. For this reason it has already attracted a great deal of attention from researchers in the field of machine learning.

GP evolves computer programs. A population of candidate programs is created randomly and their ability to perform a user-specified task is evaluated. Although we would be very surprised if any of these programs represent perfect solutions to the problem at hand (unless, of course, it is very trivial), we would nevertheless expect some programs to perform better than others. We might further reason that these above-average programs are superior because they possess some quality, however small, that is useful in solving the problem. Different programs will also possess different qualities, and combining these may yield further improvements. In GP, this reasoning underpins the processes of *fitness-based selection* and *recombination*. The result is a new population, fitter on average than the initial random population. We also hope to find at least one improvement over the best program observed in the previous generation.

Selection introduces competition for survival and reproduction amongst the individuals in the population. To ensure a reasonable chance of survival, a program must perform at least as well as, and ideally better than, its peers. If we repeat the process of evaluation, selection and recombination, this becomes harder and harder to achieve. There is a constant pressure to evolve, analogous to the pressure that drives natural organisms towards the development of adaptive innovations.

The use of evolutionary adaptation as a machine learning technique was already well-established by the time GP was developed. However, there had been great difficulty applying the principles to automatic programming. Computer programs are brittle, by which we mean that even very small changes to a functioning program generally results in a completely non-functional program (as opposed to a program that performs some function, regardless of whether it is better or worse than its antecedent). If we generated a population of random programs by randomly concatenating operators, functions and variables, it is very unlikely that *any* of them would compile, far less run. The work of Cramer [19], Fujiki and Dickinson [31] and Koza [58] demonstrated a constrained representation whereby programs can be generated at random and still be assured of performing some meaningful function. Koza in particular also proposed a number of search operators which when applied to selected parent programs always generate computationally valid offspring.

The simplicity of GP's representation and operators belies its behavioural complexity. Providing they satisfy certain limited constraints, any function or operator can be incorporated into a program representation. These may be familiar constructs, such as arithmetic and logic operators, or highly esoteric, user-defined functions, such as navigation operations that control a robot's movement around its environment. The size of a GP program can also vary, such that tasks of arbitrary complexity may potentially be tackled. GP is thus a general algorithm, and has been successfully applied to many diverse problems in science and engineering.

Nevertheless, GP is in its infancy. We still lack a complete theoretical account of how GP populations search program spaces, and there exists no general method for predicting whether a problem will be easy or difficult for GP to solve. It is possible that we will never acquire mathematical tools that are sufficiently powerful to allow us to achieve these goals, in which case we will be forced to fall back on informal explanations based on empirical data.

Such empirical observations have already highlighted a number of properties that appear to be more or less inherent in GP. From the point of view of this thesis, the most important of these are that GP programs have a tendency to increase rapidly in size as the population evolves, and that search often stagnates. The concept of stagnation requires some qualification in this context. Although GP continues to generate programs that are structurally and syntactically different from their parents, their behaviour frequently shows a marked tendency to be increasingly similar. The distinction is emphasised throughout this thesis: for evolution to work, the system must perform an efficient search of the *phenotype* space. We are not interested in creating a myriad different programs that all do the same thing.

This thesis examines these issues by considering the nature of the search that various GP operators perform. More specifically, we are interested in the degree to which a given operator performs global or local search. In a recent paper, Poli [101] demonstrated that standard GP crossover is primarily a local search operator, and in subsequent work with Langdon [107] introduced a pair of GA-inspired search operators that perform an initially global then increasingly local search. These operators – GP uniform crossover (GP-UX) and GP point mutation (GP-PM) – and the benefits they offer are given a detailed empirical examination in this thesis. In addition, we expand on Poli and Langdon's work, introducing a novel *sub-symbolic* representation that can be used in conjunction with GP-UX and GP-PM. We call the resultant operator-representation pairings *GP smooth uniform crossover* (GP-SUX) and *GP smooth point mutation* (GP-SPM). These operators retain the property of an initial global search, but are capable of performing a very local search of the program space in the later stages of the run.

Whilst investigating global and local search strategies through these operators was the primary aim of this work, a number of other issues are also addressed. In particular, we consider the issues of operator homology, code bloat and the size and content of the primitive function set. The work centres around two families of benchmark regression functions: the induction of Boolean functions (specifically the even-n-parity and 6-multiplexer problems) and symbolic regression problems. The main findings of this thesis are:

- GP-UX and GP-SUX significantly outperform standard GP on the benchmark Boolean even-n-parity problems for $5 \leq n \leq 8$. Furthermore GP-SUX outperforms GP-UX on the same problems. We conclude that an initially global search followed by a very local search of small regions of the program space yields very good results on this family of functions.

- GP-UX, however, performs rather poorly on the Boolean 6-multiplexer problem relative to GP-SUX and standard GP. This problem seems to require a more local search than GP-UX is capable of performing.

- Using the sub-symbolic representation without constraints generally requires the use of function sets that may at first glance be considered abnormally, and even prohibitively, large. In spite of this, performance on the Boolean problems mentioned above is not impaired, and in many cases is actually improved. We show that this is *not* simply because the larger function sets include building blocks that are absent from smaller, more commonly-used Boolean primitive sets such as {`AND, OR, NAND, NOR`}.

- The strict homology observed by GP-UX and GP-SUX means that the depth of syntax trees cannot increase beyond the current population maximum. Whilst this provides a very effective means of controlling bloat, it does render both operators susceptible to deceptive attractors in the search space. These attractors can lead the population to converge on trees too small to solve the problem. We report examples of such attractors in the 6-multiplexer function and a number of symbolic regression problems. However, they are not sufficiently powerful to counter the pressure to bloat in standard GP populations, and this explains why they have not hitherto been observed in standard GP experiments.

- GP-UX and GP-SUX with a variety of adapted function representations are able to discover parsimonious solutions to relatively simple, low-order polynomial functions. However, performance deteriorates rapidly with increased problem complexity. Without constraints, populations running under both operators rapidly move into the space of very small trees from which they are subsequently unable to escape and in which solutions are generally either very difficult or impossible to find.

- In most of the experiments reported in this thesis we use a binary encoding for the sub-symbolic representation. This is not necessary, however, and we demonstrate its use in conjunction with a real-valued encoding and modified search operators on a number of sub-symbolic regression problems previously attempted with a binary representation. In some cases, this gives significant performance enhancements over the binary case.

The remainder of this thesis is structured as follows.

Chapter 2 gives an overview of standard GP. Since Koza's initial development, GP has come to be used as an umbrella term for a number of systems in which variable length programs are evolved. The studies reported here concentrate on the most common variant in which programs are represented as syntax trees. Accordingly, Chapter 2 focuses attention on the representation, evaluation and search operators used in this approach.

Chapter 3 reviews previous studies that have addressed common difficulties experienced by GP. We highlight the problems presented by excessive code growth and discuss attempts to explain and curtail it. We also discuss a number of criticisms of the search operators and proposed improvements. Finally we define the concepts of global and local search and discuss their application to GP.

Chapter 4 provides a full description the *sub-symbolic representation.* The defining feature of this is that the atom of genetic information is shifted to a lower level than the function node. The representation is well-suited to use in conjunction with the GP uniform crossover and point mutation [107] operators used later in the thesis. It can also be used with a hierarchical GA, acting on fixed-size tree representations. Since this constitutes a simpler special case of the GP operators used subsequently, we first describe a pure GA implementation. The results of a number of experiments run with this system on two Boolean regression problems are reported in Chapter 5.

Chapter 6 applies the GP uniform crossover and point mutation operators to the same problems, and combines them with the sub-symbolic representation to create two new search operators for GP – *GP smooth uniform crossover* and *GP smooth point mutation* . Performance is compared with both standard GP, and the GA implementation described in the previous two chapters. A number of interesting results pertaining to the operators, representation and function sets used are discussed.

In Chapter 7 we demonstrate that the sub-symbolic representation can also be adapted for use on other classes of problems. In this case, we encode a set of functions that *interpolate* between addition, subtraction, multiplication and division and apply them to a number of symbolic regression problems. This is achieved with 3 interpolating parameters, encoded in binary on each function node. Our results lead us to question the use of such a representation, however, and we also examine a simpler form of interpolation in which a single parameter interpolates either addition and subtraction or multiplication and division.

Chapter 8 illustrates the use of a real-valued encoding in the sub-symbolic representation. We replicate many of the experiments reported in Chapter 7 but encode the interpolating parameters as floating point, rather than binary, values. In the case of GP-SUX, this requires some modifications to the search operators to allow them to work on the new representation. We discuss the advantages and disadvantages of this encoding.

Finally, Chapter 9 summarises the results obtained in Chapters 5, 6, 7 and 8. We discuss the contributions that the work reported in this thesis has made to the field of GP and conclude with a number of areas which, we feel, deserve further investigation.

Chapter 2

Evolutionary Computation and Genetic Programming

2.1 Introduction

Prior to the publication of Darwin's *Origin of Species* [15], the prevailing dogma had held that species were created to be perfectly adapted to the niches they inhabited. Whilst opposition to this creationist account of life had been growing, Darwin was the first to propose a coherent and tenable alternative theory. Darwin argued that the existence of diverse species was testimony to the fact that organisms adapt over time in response to the pressures of their environment. Specifically, his theory of evolution argues that:

- Organisms reproduce, and given infinite resources will increase in number.
- Resources, however, are not infinite and competition for them is inevitable. Individuals that compete effectively will survive and reproduce whilst those that do not will perish.
- A reproducing organism passes many of its characteristics onto its offspring.
- These characteristics are not always perfectly copied. As a result, variations are observed in the population.

- These variations may lead to further differences in ability to compete for resources.

Evolutionary adaptation will occur in *any* system exhibiting these properties [74]. For example, Dawkins' memes hypothesis [22] argues persuasively for the evolution of cultural entities in human societies. We should also note that the quality of a variation is relative and depends on the demands of the local environment. We would not therefore expect to observe the same dominant biological variations in the arctic tundra as we would in a tropical rain forest.

The idea of applying these principles to artificial search and optimisation algorithms is not new. In 1958, Friedberg [29, 30] developed the first example of what is now known as an evolutionary algorithm (EA) [8, 9, 125]. Subsequent variations included evolution strategies (ES) [111, 120], evolutionary programming [27, 26] and classifier systems [41, 132]. These algorithms all incorporate the following abstractions of Darwinian evolution:

- An initial random population of one or more candidate solutions.
- Reproduction with variation to produce new solutions.
- Evaluation of individuals against a cost function (termed the *fitness function*).
- Probabilistic selection for survival and future reproduction based on fitness evaluation.

Two further evolutionary algorithms of particular interest are the genetic algorithm (GA) [42, 35] and genetic programming (GP) [57, 58, 59]. These will be discussed in detail in the next section.

2.2 An Overview of Genetic Algorithms and Genetic Programming

GAs and GP are probably the two best-known and most widely-studied classes of evolutionary algorithms. Both utilise a population of candidate solutions (generally of fixed size), probabilistic selection on the basis of fitness, and recombination and mutation search operators. The skeleton algorithm, common to both, is shown in Figure 2.1

```
Randomly initialise population.
Evaluate fitness of population.
t = 0.
while (t < max_generations and solution not found)
  Select individuals for reproduction.
  Reproduction.
  Evaluate fitness of new population.
  t = t + 1.
endwhile.
```

Figure 2.1: The core algorithm for GAs and GP.

GAs and GP differ from many of the previously mentioned EAs in their use of populations of more than one individual and their heavy reliance on the recombination operator. The primary difference is that GP systems evolve programs of arbitrary complexity comprising predefined functional primitives whereas most GAs seek to optimise parameters for a given problem. This leads to a difference in the manner in which solutions are represented: GA chromosomes are generally of fixed length (although see Goldberg's work on messy GAs [37, 36] for an exception to this) whereas the size of GP genomes is allowed to vary[1]. Furthermore, whilst both GAs and GP employ recombination and mutation search operators, their implementation must be tailored to the representation upon which they act to ensure valid offspring.

Many of the steps outlined in Figure 2.1 are now so well-known in the EA community that further discussion will be limited to those points that directly impinge on the studies reported in this thesis. For more extensive reviews, the reader is directed to [35, 82, 79, 81].

[1]Some readers may find this distinction contentious, pointing out that techniques such as messy GAs that employ variable-length representations are, nevertheless, still GAs. For this reason we emphasise that throughout this thesis the term GA should be taken to mean "fixed-length GA" unless otherwise stated.

2.2.1 Core Aspects of GAs and GP

In this subsection, we briefly discuss some of the points in the algorithm shown in Figure 2.1.

Search Operators

Given the difference in representation between GAs and GP, it is no surprise that their search operators differ radically in implementation. In this section we will briefly consider the points shared by GA and GP search operators, and discuss one pair of GA search operators used in Chapter 5 – uniform crossover[127] and point mutation acting on binary strings. A more detailed consideration of GP search operators is given in the next section.

Most GA and GP implementations employ a single sexual reproduction operator (recombination) and a single asexual operator (mutation). The former requires two or more parents (although two are generally preferred) whilst the latter requires one.

An example of a recombination operator in GAs is uniform crossover. Given two parental bitstrings, crossover points are selected along each string with some probability (typically 0.5) and offspring generated by exchanging the bits at this point. The process is illustrated in Figure 2.2.

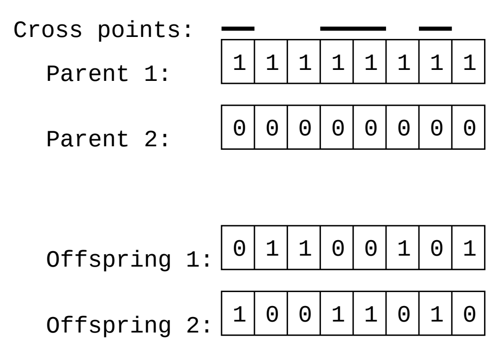

Figure 2.2: GA Uniform Crossover illustrated on two parental bitstrings. Two offspring are generated in this example, although many implementations discard one.

A single application of the point mutation operator selects a random bit on the parental bitstring and flips it.

Other crossover and mutation operators acting on binary and non-binary representations have been proposed but will not be discussed here. The interested reader is again referred to [78, 35, 81].

Fitness Evaluation

We noted in Section 2.1 that a necessary element in the evolutionary process is that individuals must exhibit differences in their ability to compete for some finite resource, and that the degree to which an individual possesses this ability is proportional to its capacity to reproduce. The measure of this ability is known as the fitness, and in biological systems it is implicit: the fitness of a gene or organism is reflected in its capacity to reproduce itself. Seen in this light, the Darwinian concept of survival of the fittest becomes tautologous – fitness is a measure of an individual's capacity to reproduce, and an individual's reproductive rate is directly proportional to its fitness.

In contrast to biological systems, most EAs select for reproduction explicitly, and to do this we need some way of evaluating one individual against another. The method used almost universally is the fitness function, which gives a quantitative measure of how well the individual performs on a pre-defined task. Put another way, the fitness function performs a mapping between points in the space of possible solutions (the genotype space) and points in the fitness space.

Transmission Functions and Fitness Landscapes

Given a genotype g in a set of possible genotypes G, the set of offspring accessible from a single application of the search operators forms a subset of G, which we will refer to as O. The elements contained within O can therefore be thought of as being located in the neighbourhood of g. It is, however, important to note that the size and content of O will be determined by the representation of the genotype and the nature of the operators acting upon it.

We should also note that as the fitness of g increases, the proportion of possible offspring that are fitter than g will generally decrease. As a run progresses, fitter variants will therefore become progressively more difficult to find. Searching the genotype space is thus analogous to traversing an undulating landscape, looking for the highest point – as we climb higher, we find an ever-decreasing number of points that are higher still. The image is enduring and has given rise to the concept of the *fitness landscape* [134]. It must be emphasised that the routes through this landscape are dependent on the representation and operators used. The point is important because, depending on the problem, we may eventually obtain a sub-optimal genotype for which no fitter variants can be reached with a single application of the search operators (i.e. all the elements of O exhibit fitness equal or inferior to g). The genotype is then unable to evolve further. If the genotype is able to compete with the other individuals in the population, it will continue to reproduce, and may eventually exclude all other solutions. Under these circumstances, the population is said to have converged on a sub-optimal point from which it cannot escape.

Altenberg [3] develops these ideas further in his concept of the *transmission function*, which describes the probabilistic mapping of parent points onto offspring points. Seen in this context, operators and representation are relevant only to the extent to which they affect the offspring that can be generated from any given parents, and the probability of generating a specific offspring. An equivalent change in the transmission function brought about by, for example, an alteration of the representation could also be brought about by modifying the operators. The crucial point is that it is the relationship between the transmission function (as opposed to just the operators or representation) and the fitness function that dictates whether a problem will be difficult for the system. The frequently observed problems of epistasis, multimodality and deception are therefore not simply inherent to the fitness function, but arise from the fitness landscape generated by the transmission function and the test problem. We may note that many apparently simple functions can be rendered very difficult to solve by transforming the transmission function. However, whilst the converse is also true for many difficult problems, improvements to the transmission functions will not carry over to *all* functions. Furthermore, finding an optimal transmission function for a highly complex problem may require knowledge of the search space that is, in practice, infeasible.

Two things are immediately apparent. Firstly, an operator-representation pairing that results in many local optima will make locating the global optimum a difficult task for an EA. Secondly, the ruggedness of the fitness landscape for a given problem can be tuned by modifying the operators, the representation, or both. This is a central theme to this thesis.

Selection

A number of selection algorithms have been proposed, each equally applicable to either GAs or GP. Currently one of the most popular, and the one used throughout this thesis, is tournament selection. In tournament selection, n individuals are selected from the population and their fitnesses compared. The fittest individual is copied to the new population, where it may survive unchanged or transformed with the search operators. The principal advantages of tournament selection over other approaches are its simplicity, the relatively low computational overhead and its tuneability. With reference to the last of these, we may see that increasing the value of the tournament size, n, decreases the probability that a relatively unfit solution will be selected. The tournament size therefore constitutes a simple means of regulating selection pressure.

2.3 Genetic Programming

Genetic programming originated as an extension of genetic algorithm research. Cramer [19] developed what is now credited as the first EA to use variable-sized tree structures as genomes. However, it was the publication of Koza's *Genetic Programming* [58] that demonstrated the power and generality of GP. Since then, a number of different representations have been proposed, including linear machine code instructions (e.g. [87, 44]) and graph-based implementations (e.g. PADO [131], PDGP [100, 99]). GP has also been successfully applied to a number of diverse areas such as the autoparallelisation of serial programs [119], DNA sequence classification [38], financial prediction [96, 112], image processing [98, 97], natural language processing [24], robot control [118] and music generation [126]. However, since only tree-based genomes are considered in this thesis, further discussion will be restricted to these representations. Furthermore, we shall use the term standard GP to refer to the canonical tree-based implementation without automatically defined functions (ADFs), indexed memory etc.

As we have seen, GP differs from GAs principally in its operators and representation. In addition, it evolves executable programs, rather than vectors of parameters. The discussion in this section will therefore focus on these areas.

2.3.1 Tree Representation in GP

Candidate solutions are represented in GP as syntax trees consisting of function and terminal nodes. Functions and terminals are differentiated by their number of arguments, or arity. Function nodes accept at least one argument whilst terminals accept none. An important constraint in many implementations, however, is that instances of both must all return a value.

The functions and terminals from which a GP syntax tree is constructed are drawn from user-defined function and terminal sets. An example tree, constructed using the Boolean function set {`AND, OR, NAND, NOR`} and a terminal set comprising the Boolean input variables {`x1, x2, x3`} is shown in Figure 2.3.

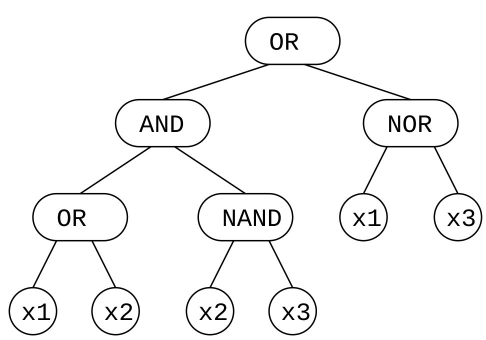

Figure 2.3: An example GP syntax tree. The terminal (leaf) nodes code arguments to the function which are evaluated by the upper function nodes.

Throughout this thesis we shall illustrate GP trees using diagrams such as that shown in Figure 2.3. We will also make references to sub-trees and code fragments in the text, and to do this we will use a prefix notation. Thus, the tree in Figure 2.3 can alternatively be defined as {`OR` {`AND` { `OR x1 x2`} { `NAND x2 x3`} } {`NOR x1 x3`}}.

The use of variable inputs means that the tree may be evaluated over a number of fitness cases. Since the tree in Figure 2.3 accepts Boolean inputs, the number of possible fitness cases is finite ($2^3 = 8$ with this function set). However other representations (e.g. those encoding a continuous domain) may potentially be evaluated over an infinite number of fitness cases.

Choice of Function and Terminal Set

As may be seen from Figure 2.3, function nodes occupy the internal regions of the tree whereas terminals are located at the leaves. Another way of looking at this is to think of terminals as functions of arity 0. Both functions and terminals must return a value, and this value must provide a valid argument to any of the functions in the function set (for example, the Boolean functions in this example would generally need some modification to be able to handle floating point arguments). This constraint – known as closure [58] – ensures that any tree constructed from the available functions and terminals will exhibit valid behaviour (although, see the work of Montana [86] on strongly-typed GP for a variation on this rule). More importantly, we must ensure that all possible values that may be returned by the node are defined. For example, the arithmetic division function returns an undefined value when its second argument is 0, and provision against this contingency must be built into its definition.

Aside of these constraints, the choice of function and terminal sets is left to the user, and selecting an appropriate combination for the problem is something of an art. The importance of this is clear – the choice of function set directly affects the transmission function – yet it is an area that has been almost wholly ignored in the literature, and is one of the themes of this thesis. Koza [58] examined the effect of adding more Boolean functions to the function set on a single Boolean induction problem (the 6-multiplexer). He found that performance progressively deteriorated when the size of the set went beyond 6 functions, and observed the best performance with the (logically complete) function set {`NAND`}. Rosca [116] observed that increasing the size of the function set from 4 to 8 increased the fitness diversity of randomly generated trees on the even-5-parity problem, but that this effect is slightly reduced when the size is further increased to 16 functions. Two rules of thumb emerged from these studies, which can be summarised as:

- The function set should be sufficiently powerful to permit the generation of the desired behaviour. For example, if the task is to evolve a Boolean function, a minimal requirement should be that the function set is logically complete (i.e. any Boolean function can in principle be constructed from the given primitives).

- The function set should, however, be kept as small as possible since every additional function increases the size of the search space and this may impede search [11, 56][2].

Whilst the first rule is almost self-evident, the unquestioned acceptance of the second has recently been challenged [69]. Moreover, we note that Rosca's study only gives an indication of the relative densities of certain solutions within the program space of specific function sets – it says nothing of how easy other solutions will be to find with a search algorithm operating on these representations. This leaves us with a single study on a single problem from which the second conclusion is largely drawn. This conclusion is questioned in all the studies reported in this thesis.

Initialisation

Given a pre-specified maximum depth, GP trees are usually initialised in one of two ways:

- **Full Initialisation.** If the current depth at which a node is to be added is less than the maximum, a randomly selected function is always inserted. The number of nodes that must then be inserted at the next level is $\sum_{i=1}^{n} a_i$ where n is the number of nodes at the current depth, and a_i is the arity of node i. If the current depth is equal to the maximum specified, a randomly selected

[2]In view of recent work by Koza and his co-workers on the Genetic Programming Problem Solver (GPPS) [60], we emphasise that both these rules apply to "standard" approaches to solving a given problem with GP, that is, where the function and terminal sets are user-defined and reflect some knowledge of the problem domain. GPPS, of which more will be said in Chapter 5, deliberately utilises a large, generic set of functions and in this sense is an exception to the rules. However, since the aim of this technique is primarily to enhance generality – in that the task of selecting appropriate functions for the problem is shifted from the user to GP – rather than performance, we do not feel it detracts from the main thrust of the argument.

terminal is inserted. Trees initialised in this way therefore exhibit the property that every terminal is located at the maximum depth.

- **Grow**. If the current depth is equal to the maximum, a terminal is always inserted. However, if it is less than the maximum, a terminal may be inserted with some probability. Otherwise a function is inserted and initialisation continues. Trees initialised with this method may therefore vary widely in structure and depth.

In many implementations, including all the studies reported in this thesis, these two methods are combined in what is known as ramped half-and-half initialisation [58]. With this method, the population is initialised in equal proportions with maximum depths in the range $[D_{min}, D_{max}]$[3]. Furthermore, half the trees are initialised with the full and half with the grow method. This method ensures initial tree diversity in terms of both size and structure.

2.3.2 Evaluation in GP

Fitness evaluation in GP is more complicated than in GAs. GA chromosomes code a *feature vector* [5] and fitness in GAs can be viewed as a measure of how well these features compare with a pre-defined optimum. In contrast, GP chromosomes are executable programs and fitness is a measure of how well these programs accomplish some desired task. Evaluation therefore implies execution, frequently over a number of distinct fitness cases.

Execution of a program is in general more computationally intensive than simple evaluation of a feature vector. Furthermore, when each tree is executed over multiple fitness cases, the evaluation stage rapidly becomes the most expensive part of the algorithm. For this reason, Koza [58] argued that a reasonable measure assessing the efficiency of a particular algorithm on a given problem is the number of individuals that must be evaluated to find a solution. At first sight, this might appear to be simply a factor of the population size and the number of generations that have elapsed when a solution is found. The problem is that EAs feature a stochastic element such that solutions may be found more quickly on some runs than others. Koza's measure

[3] In our studies, we set D_{min} to 0, i.e. trees of one terminal were possible

takes account of this by calculating the number of independent runs that must be performed to yield a solution by generation g with some probability p.[4] The number of independent runs, R, that must be performed with a population of size M to yield a solution with probability p by generation g is given by:

$$R = \frac{log(1-p)}{log(1-P(M,g))}$$

where $P(M,g)$ is the cumulative probability that a solution is found by generation g (for details of how this expression is derived, see [58][p.193-194]). The accuracy of this measure depends on the reliability of $P(M,g)$, which should be made on the basis of a number of independent runs. In this thesis, all our estimates were based on the results of 50 runs.

Having obtained an estimate of the number of independent runs required to find a solution, the number of individuals that must be evaluated is $R \times g \times M$. The minimum over g of this value is now generally referred to in the literature as the *computational effort* required.

Note that this value assumes a single fitness evaluation to be an atomic operation – it does not take account of the number of fitness cases over which each program is executed to perform an evaluation. Moreover, its validity as a reliable means of comparison hinges on the assumption that all chromosomes will require the same amount of resources to execute. When variable-length representations are used this frequently does not hold, since the time taken to execute a program increases with its size. We will return to this point in Chapter 7.

2.3.3 GP Search Operators

The GP algorithm was derived from GAs, and it is therefore no surprise to find that it uses analogous search operators. Specifically, its principle operators are crossover and mutation. In this subsection, we describe each of these in turn.

[4] The value of p used in most studies, and the one used throughout this thesis, is 0.99.

Crossover

In its standard form, crossover generates offspring by exchanging genetic material between two parent trees. A single crossover point is selected randomly in each parent, and the sub-trees at these locations are swapped. Crossover points may be selected entirely at random, or with a bias towards non-terminal nodes (the former is used in the standard GP studies reported in this thesis). Crossover acting on two parent trees, and the resultant offspring, is illustrated in Figure 2.4.

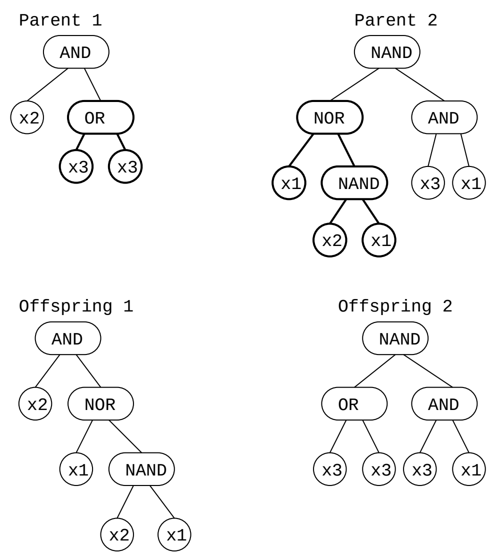

Figure 2.4: Standard GP crossover acting on two parental syntax trees. The sub-trees selected for crossover are highlighted.

We note that, providing the principle of closure is observed with respect to the functions and terminals used, no further constraints are necessary to ensure that valid offspring are generated.

Mutation

Standard GP mutation, like its GA counterpart, accepts a single parent. A mutation point is selected at random in the parent tree and the sub-tree replaced with another, generated at random from the given functions and terminals. The operation is illustrated in Figure 2.5.

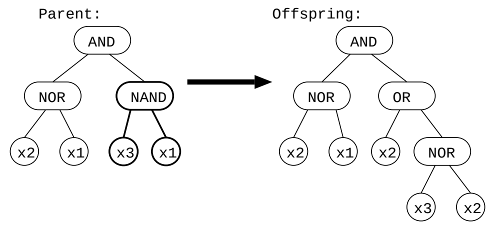

Figure 2.5: Standard GP mutation acting on a parental syntax trees. The highlighted sub-tree is replaced by another generated at random.

Crossover, Mutation and Building Blocks

Like most GA researchers, Koza considered the crossover operator the more important of the two. In [58] he argued informally that, under selection, highly fit sub-trees would emerge in the population and that crossover would exchange them between parents creating ever-fitter offspring. Koza emphasised the similarities between his argument and the GA building block hypothesis [35]. GP, he suggested, works by combining small, fit code fragments into larger *building blocks*. If these building blocks afford individuals an evolutionary advantage, we should expect them to be propagated under the influence of selection and reproduction in a manner similar to that predicted by the GA Schema theorem [42]. Koza recognised that, as in GAs, crossover may act as a disruptive force, substituting inferior code into highly fit building blocks, but reasoned that as they increase in number, it becomes more likely that they will be transferred intact to other regions in offspring trees. Thus, for example, the arity-2 `EQ` function is a useful building block for solving the even-n-parity problems. If a sub-tree equivalent to this function should be evolved, we should within a few generations expect to see numerous instances of it in fit candidate solutions. It may also be used in the construction of larger building blocks, such as the even-3 and -4 parity functions.

These arguments led Koza to argue for crossover's primacy as a search operator. Mutation, by virtue of inserting random, untested code into offspring programs, should be less efficient when it comes to constructing building blocks of this kind. Instead, its purpose was seen as reintroducing lost genetic material. However, Koza also argued that the complete elimination from the population of a vital function or terminal was very unlikely with GP crossover and that the use of the mutation operator as a means of reintroducing genetic material was in many cases unnecessary. These arguments have since come under attack from a number of different angles and, at the time of writing, remain a controversial area of debate in the GP community [11]. We address some of these issues in the next chapter.

2.4 Conclusion

The discussion so far has provided a brief historical perspective of GP, and given a high-level description of the most widely used tree-based form. We have seen that GP's representation and search operators allow it to search program spaces where the programs may vary in length. That such a comparatively simple and unconstrained algorithm is able to do so is remarkable, and its diversity of application and success on many of these applications is therefore all the more impressive. However its poor performance in a number of areas, together with certain undesirable properties, have fuelled a large body of research to which this thesis contributes further. In the next chapter we set the scene for the studies reported in later chapters by reviewing the key problems associated with GP and current attempts to address them.

Chapter 3

The Efficiency of GP Search

Genetic Programming is computationally expensive. Typical populations comprise several hundred individuals, evaluated over many generations. In addition, a normal GP experiment normally consists of multiple runs. Whilst this is also true of GA experiments, GP suffers from two further considerations. Firstly, the use of variable inputs at the terminals means that a GP tree can exhibit different behaviour in different situations. A useful fitness measure should take account of this, and for this reason GP trees are almost always evaluated over multiple fitness cases. As we have seen, whilst the number of chromosome evaluations performed in a canonical GA run is simply the product of the number of generations and the population size, this value must additionally be multiplied by the number of fitness cases in GP. A number of studies have attempted to address this issue directly, evaluating sub-sections of the population over limited fitness cases [34, 130], or, in the case of Sub-Machine Code GP, evaluating multiple fitness cases in parallel [109, 102].

The second consideration is that GP trees can vary in size. If trees become very large, this can seriously compromise performance, in terms of both memory use and processing time. Koza's statistic for algorithm performance, given in the previous chapter, rightly assumes that evaluation is the most computationally expensive stage of a GP run, but also assumes that the memory and processing demands made by a *single* tree remain constant. In fact, this almost never happens. Trees show a marked tendency to increase in size as a run progresses – a phenomenon known as bloat – and this severly compromises performance.

These two factors clearly work to compound GP's problems. Bloat increases the length of time taken to evaluate a single individual once, and each individual must be evaluated over numerous fitness cases. Attempts to reduce the number of program evaluations aside, it is usually inevitable that a tree will have to be evaluated more than once. Consequently, most studies have concentrated on identifying the causes of bloat and seeking remedies to it. We review these studies in detail in the next subsection.

3.1 Bloat

Bloat is the observed tendency of candidate programs to increase in size as a GP run continues. Bloat appears to be an inherent feature of search algorithms using variable-length representations [71, 68], and has been observed in many problems in both tree-based (e.g. [58, 45, 137, 12, 115, 122, 43, 129, 4, 113]) and linear GP (e.g. [88]).

Large candidate solutions are undesirable for a number of reasons. We have already seen that maintaining populations of large trees is computationally very expensive. We should also note that large solutions can be difficult to understand and that there is evidence that, for certain problem domains, they may be less general than smaller programs when tested on data outwith the training set [55, 128]. Conversely, it has been argued that much of the code in bloated solutions is redundant and that the proportion of the nodes that actually contribute to the evaluated behaviour is quite small [88, 122]. Under these circumstances, search may stagnate as many, if not most, of the offspring generated by crossover will exhibit behaviour identical to their parents.

Analogies have been drawn between non-coding fragments in GP solutions and biological introns [58, 4], and the terms are now often used interchangeably in the literature. Introns were originally thought to be beneficial since they reduce the likelihood that crossover will destroy acquired building blocks [6, 89, 90] (similar arguments have been made for the explicit use of introns in GAs – see [72, 135]). Whilst this may be true, we argue that this suggests the need for alternative search operators for certain problems, rather than that introns are beneficial per se. Furthermore, a number of recent studies have implicated introns as a contributary factor in bloat. We will discuss these studies below.

3.1.1 Introns and Bloat

Introns are found in the DNA of the overwhelming majority of higher organisms. Intron sequences are removed before the translation of RNA to protein molecules, and therefore play no direct part in determining the organism's phenotype. In many organisms, introns make up the majority of the genome and there is no correlation between organism complexity and length of DNA (cf. the "C value paradox" [14]).

The complex molecular processes involved in biological ontogeny and the fact that the role introns play in it is poorly understood mean that the analogy drawn between biological introns and introns in GP is at best weak [20]. This, of course, need not be a problem providing we are aware of the fact and that we do not try to overstretch the metaphor. Of more significance, however, is the fact that the term intron has been applied to variant forms of redundant code fragments in GP, and that conclusions drawn from consideration of one are assumed to apply to the others.

This imprecision has lead to a distinction between *inoperative* and *inviable* code [71, 123, 28]. Inoperative and inviable sub-trees are defined in [71] thus:

> A node n in a program's syntax tree is operative if the removal of the subtree rooted at that node (and replacing it with a null operation) will change the program's output. Conversely a node is inoperative if it is not operative. [71][p. 165]
>
> A node n in a program's syntax tree is viable if there exists a sub-tree such that replacing the sub-tree rooted at node n with the new sub-tree will change the program's output on some input. Conversely a node is inviable if it is not viable. [71][p. 165]

This distinction can be clarified by considering the example tree in Figure 3.1.

In this tree, the sub-tree `{- x x}` is inoperative since it will evaluate to 0 for any input. We could therefore replace the sub-tree `{- x {- x x}}` with `{- x 0}` (or,

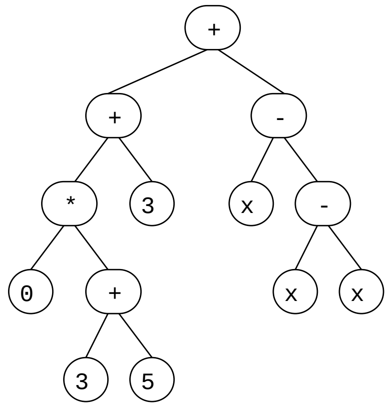

Figure 3.1: Inviable and inoperative code in a GP syntax tree.

more simply, `x`) without affecting the value returned by the tree for any input. Note, however, that other changes to the sub-tree {`- x x`} affecting the value it returned would in turn affect the behaviour exhibited by the tree.

The sub-tree {`* 0` {`+ 3 5`}} multiplies its second argument by 0 and will return 0 irrespective of what the second argument happens to be. We could substitute the sub-tree {`+ 3 5`} with *any* fragment without affecting the overall behaviour of the tree; the sub-tree {`+ 3 5`} is therefore inviable. We may also note that, as a result of this, the sub-tree {`* 0` {`+ 3 5`}} is inoperative.

If we define an intron as any code fragment that could be removed without affecting the tree's behaviour, then it becomes clear that both inviable and inoperative sub-trees are introns. However, it would be a mistake then to conclude that a search operator acting on any form of intron will result in an offspring that is semantically identical to its parent – this only holds for inviable code.

The distinction is important because introns have been cited as a primary cause of bloat in GP [13, 77, 88, 115, 124]. The theory proposed is roughly as follows. As a run progresses, the average fitness of the population increases and new improvements become more difficult to find. The effect of crossover on relative offspring fitness, if any, is then generally deleterious [116]. When this happens, a sub-optimal, but nevertheless viable, strategy to ensure offspring survival is to ensure offspring exhibit the same behaviour (and hence have the same fitness) as their parents. McPhee and

Miller [77] argue that selection will favour individuals that are probabilistically more able to do this over those that are not, i.e. there is a replication accuracy force (RAF). This may in practice be quite difficult to achieve in GP since even when crossover is applied to two syntactically identical parents, the offspring are usually syntactically different. However, if the code modified is inviable, the offspring will be semantically equivalent to their parents and therefore, from the point of view of selection, accurate replications.

A number of predictions follow from this. In a stagnant population, selection will favour those trees where the ratio of operative code (i.e. code contributing to fitness) to inviable code is low, since the chances of crossover acting (presumably destructively) on operative code are minimised. Other conditions remaining equal, selection should therefore favour trees whose quantity of operative code is small, that is, trees which when stripped of their inoperative branches encode parsimonious solutions [11, 88, 122]. Of course, this also applies to sub-optimal solutions, and such pressure may force the population to converge prematurely on small coding trees. Stagnation is then virtually inevitable as the inviable code prevents crossover from generating competitive operative code. On the other hand, selection will also favour trees containing ever larger amounts of inviable code.

A number of experimental studies confirm these predictions. For example, Soule and Foster [122] examined the effects of destructive and non-destructive crossover operators on code growth in a maze navigation problem. There are a number of variant forms of non-destructive crossover (e.g. [3, 93]), which will be discussed more fully below. The core principle is that offspring are created as normal and if their fitness exceeds that of their parents they replace them in the population. Crossover can therefore only act to improve fitness since all destructive applications are rejected. Under these circumstances the pressure towards large programs as a means of protecting against the destructive effects of crossover is removed, and we should not expect to observe code growth on such a pronounced scale. This is precisely what Soule and Foster found. With destructive crossover, code bloated rapidly and almost all of it was inoperative. In addition, they noted that while code growth still occurred with the non-destructive operator, it was proportional to improvements in fitness. Other studies substantially confirm these observations (e.g. [93, 43, 71]).

3.1.2 Other Causes of Bloat

While introns undoubtedly contribute to bloat, they are not the only cause. Tackett [129] argued that bloat arose as a result of the transmission of inferior or neutral code fragments along with fit building blocks during crossover – a process analogous to the hypothesised "hitchhiking effect" in GAs [83, 84, 81]. More recently, Soule and Foster [123] have shown that code growth is an emergent feature of crossover, and Langdon and others [71, 65, 66] have argued that code growth is to be expected in any search algorithm that uses variable-length representations. We review these arguments in this subsection.

Soule and Foster [123] argue that a *removal bias* arises as a result of inviable code. Briefly put, their reasoning runs as follows. The probability of a crossover affecting viable code is inversely proportional to depth of the crossover point, since inviable code tends to be located towards the leaves of the tree. Offspring trees are therefore more likely to replicate the behaviour of their parents if crossover is performed deep in the tree of the parent donating the root node. However, if this point occurs within an inviable code fragment, the size of the replacement sub-tree is irrelevant since it will be rendered inviable and will not affect offspring behaviour. Selection acting on robustness (i.e. the preservation of equal fitness in the absence of fitness improvements) will favour trees where small program fragments are replaced with sub-trees of average size. This will lead to a gradual increase in average tree size at each generation. Whilst the growth in size as a *proportion* of the current average is small, exponential increases in average tree size are often observed over many generations.

Langdon *et al.* [71] go further, suggesting a more general cause of bloat. A number of earlier studies of a range of GP problems showed that, beyond a certain threshold size, the density of programs with a given fitness is approximately constant for any given program complexity [66, 68, 69]. The population is therefore forced into the space of larger and larger trees simply because these regions contain many more instances of programs that exhibit the same behaviour. They also argue that the fitness landscape may be smoother in the regions of larger programs, particularly on continuous domain problems, since tiny improvements in fitness can be made more easily.

3.1.3 Counteracting Bloat

Bloat, then, is an insidious process that would seem to be endemic to all search algorithms that employ a variable-length representation. As an understanding of the causes of bloat has been developed, a number of possible remedies have also been proposed. We examine some of these in this subsection.

Koza [58] provided a simple means of curtailing bloat by imposing a maximum depth on the size of tree that may be generated by the search operators. This is perhaps more of a necessity than anything else, given finite computing resources. There is also evidence that when trees are close to this maximum size, performance can be adversely affected [32, 64].

Subsequent early studies, as we have seen, were directed at counteracting the destructive effects of crossover on fit code. The earliest example was Tackett's brood recombination algorithm [129] (although see also [2]). In this approach, crossover is applied more than once on the same parents to produce a pool, or brood, of offspring. These offspring are then evaluated, and the fittest inserted into the population. Tackett avoids the increased overhead of evaluating additional offspring by estimating fitness on the basis of limited fitness trials. He reports performance superior to GP on a number of test problems. Other approaches involving non-destructive crossover exist (e.g. [93, 94, 39]) but these raise issues pertaining to the nature of GP search, and discussion is therefore deferred to Section 3.2.

Another means of combatting bloat is to impose a penalty on large programs as part of the fitness function [12, 45, 137, 121]. There are, however, a number of problems with this approach. Firstly, it alters the structure of the fitness landscape and may impede search for a solution [58]. Conversely, it may also prove ineffective, if the pressure towards larger solutions is sufficiently strong. Finally, as a general consequence of the previous points, it is not immediately obvious how heavy the penalty should be [115].

Ryan [117] has suggested an interesting means of controlling program size that side-steps these issues. Programs are initially added to a reproduction pool purely on the basis of their performance on the fitness function. However, programs that are not sufficiently fit to be included are given a second chance if they are both reasonably fit and parsimonious. Such programs are placed in a second pool. Ryan terms programs of the first type "civil servants", since they are efficient, and those of the second "pygmies", since they are small. In Ryan's algorithm, pygmies can only reproduce with civil servants. Ryan found that this approach enhanced both performance and solution parsimony on a sorting task. He also emphasised that this approach need not be restricted to controlling program size, but could be applied to any problem which required the optimisation of more than one feature or value.

Langdon [71, 62] has recently proposed a number of crossover and mutation operators in which the size of the replacement tree is controlled so that it is, on average, the same as that of the tree it replaces. Whilst he found that these operators offered little, if any, improvements in search efficiency over the standard operators, bloat was significantly reduced.

This section has given a detailed overview of a pervasive problem afflicting GP – the bloating of candidate solutions. The conclusion that seems to follow from the studies reported is that complex, yet parsimonious, solutions to many problems are difficult to construct and that as the run continues, the population instead concentrates on preserving the gains that it has accrued rather than improving on them. This is a worrying conclusion as it implies that evolvability (that is, the ability to find offspring that are fitter than their parents [3]) is not maintained throughout the run. This calls into question the efficiency of the search that GP is performing, and we discuss this issue in the next section.

3.2 Efficiency of GP's Search Operators

The preceding discussion of bloat highlighted the fact that, as a run progresses, there is an increasing likelihood that standard GP crossover applied to operative code will have a detrimental effect on offspring fitness. We noted that introns were initially viewed in a positive light as a defence against this destructive influence [6, 89, 90]. However, this can be interpreted as an argument against the use of crossover on many problems, rather than an argument in favour of introns (and therefore bloat). Unfortunately, standard GP crossover's problems are not limited to bloat. Other studies, which we will now review, have argued against Koza's hypothesis that crossover works by assembling highly fit sub-tree building blocks, have attacked Koza's assertion that crossover is a radically different and far more important search operator than mutation, and questioned the nature of the search that crossover performs.

3.2.1 Does GP Crossover Construct Building Blocks?

In [58], Koza argued that GP assembles solutions of increasing fitness by exchanging highly fit sub-trees between parents. His theory hinges on the ideas that trees possessing such trees will, on average, be fitter than those that do not, that such trees (and hence such sub-trees) will tend to proliferate under selection, and that the likelihood of the fit sub-trees being transmitted to other trees intact thereby increases. His theory is thus an informal analogy of the schema theorem for GAs [42]. Moreover, it implies that crossover is of primary significance as a search operator – since mutation introduces random, untested genetic material into the population it should, as in GAs, primarily be used to maintain diversity or, as total loss of genetic information is not generally a problem in GP populations, not at all.

The first prediction, then, is that GP mutation used in place of crossover should result in a deterioration of performance. Lang [61] compared standard crossover with "headless chicken crossover" (HCC). This operator is derived from a GA operator of the same name developed by Jones [51] and involves performing crossover on two parents – one drawn from the population and one generated randomly. Whilst the GA operator behaves in a similar manner to crossover in the early stages of a GA run, the effects of GP-HCC are syntactically equivalent to GP mutation. Lang observed superior performance of this operator on a single problem – the 3-multiplexer function – and, somewhat unwisely, concluded that it was therefore better than crossover *in general*.

Whilst Lang's conclusions have subsequently been shown to be premature, questions continue to be asked about crossover's supposed superiority over other operators such as mutation. Angeline [7] found that both standard GP mutation and variant forms of HCC performed at least on a par with standard GP crossover on a diverse range of test problems. Similar results with different mutation operators have also been reported [93, 16, 17, 73].

The conclusion that appears to follow from these studies is that standard GP crossover acts as a form of macro-mutation operator. There is little evidence to support Koza's argument that crossover progressively constructs large, fit sub-trees from smaller evolved building blocks. In fact, the reverse seems to be true – beyond a certain size, crossover's effect on building blocks is predominantly destructive.

This is in marked contrast to what is found with biological recombination. Crossover of DNA segments almost always results in offspring that exhibit fitness comparable to that of their parents. Sexual reproduction takes place, by definition, between members of the same species and base pair exchange occurs at the same locations on both parental chromosomes. Where genetic material is identical in both parents (and the vast majority of genetic information is shared by every member of a species), the offspring inherit their parents' traits. The relatively few differences may lead to observable differences in the offspring, although this only occurs within tightly controlled boundaries.

Nature has clearly gone to considerable lengths to constrain the amount of variation that is possible as a result of sexual reproduction, and in highly complex organisms it is easy to see why; failure to do so would almost certainly prove catastrophic for the resultant offspring. Banzhaf *et al.* [11] offer an interesting analogy from computer science:

> Indeed, crossing over two programs is a little like taking two highly fit word processing programs, Word for Windows and WordPerfect, cutting the executables in half and swapping the two cut segments. Would anyone expect this to work? Of course not. [11][p. 158]

This *homology*, whilst a feature of many EAs such as GAs, is ignored in standard GP. Crossover takes a sub-tree from one parent and inserts it at a *random* location in the other. No account is taken of the behaviour exhibited by either tree, or the role that each sub-tree performs in determining this behaviour. Moreover, standard GP crossover ignores the fact that the fitness of a building block is highly dependent on the context in which it finds itself [92]. A function is only useful if it is called under appropriate circumstances.

A number of operators that enforce either partial or total homology have been proposed. Ito *et al.* [47, 48] describe a modified crossover operator in which crossover depths are probabilistically selected with a bias towards depths close to the root and then crossover points chosen from the nodes at the selected depth. In a subsequent study, the depth parameter was allowed to co-evolve with the trees [49], and Ito *et al.* reported that the favoured depth moved closer to the root as the run progressed, i.e. crossover exchanged larger building blocks.

Langdon [62] extends his size-fair crossover operator with a homologous operator. This works by selecting a sub-tree in the second tree that matches as closely as possible the size and position of that first (as opposed to size fair crossover, which ignores the position of the second sub-tree). Unfortunately, as with size-fair crossover, he did not observe improvements in performance relative to standard crossover.

One problem with exchanging single sub-trees at the same depth is that most crossover points will tend to be selected close to the leaves [32, 114] and this means that nodes close to the root may converge prematurely. Harries and Smith [39] therefore additionally compared a number of operators in which depth constraints are only partially enforced. They found that while strict depth-dependent crossover (where crossover locations are selected at the same depth in both trees) performed comparatively poorly on two Boolean regression tasks, an operator that performed depth-dependent crossover half the time and standard crossover otherwise, consistently outperformed pure standard crossover on a variety of problems.

The operators described so far have all exhibited what may be termed weak homology. By this we mean that whilst some attention is paid to node position in the tree, it is not strictly enforced. Thus, for example, whilst a depth-dependent crossover operator exchanges sub-trees at the same depth, and will therefore be likely to swap roughly the same quantities of information, it may nevertheless insert each sub-tree at a different position in the recipient tree. Context in all these studies is, at best, weakly preserved.

d'Haeseleer's strong context-preserving crossover (SCPC) [23] permits crossover only between sub-trees at the same depths and positions in the tree. Thus, for two parental nodes to be considered for crossover, all nodes above them must share the same arities in both trees. d'Haeseleer noted that this approach is similar to GA crossover, and that both lead to rapid, and frequently premature, lexical convergence in the population. Whilst the problem is overcome in GAs by the inclusion of a mutation operator, d'Haeseleer was concerned that imposing strict constraints on depth and position in crossover constituted overkill, in that the occasional migration of building blocks to other parts of the tree where they could be usefully reused would be prevented. For this reason he used SCPC in combination with standard crossover in most of his runs. He found that such combinations were superior to standard crossover alone on the Boolean 11-multiplexer function and a robot navigation task.

Poli and Langdon's derivation of a schema theorem for GP [106, 108] led them to develop an operator called GP One Point Crossover [105]. A set of possible crossover locations is generated by performing a traversal of each parental tree, stopping each expansion as soon as a mis-match in node arities is found. A single crossover point is selected at random from this set and the sub-trees at this location in each tree are swapped. It is thus very similar to the SCPC operator proposed by d'Haeseleer. However, Poli and Langdon do not rely on the partial incorporation of standard crossover to maintain genetic diversity. Instead they use a point mutation operator, which selects a single node at random and replaces it with another of the same arity, drawn at random from the function (or terminal) set (see also [16]). This elegantly sidesteps the problem associated with premature convergence of upper nodes in the tree since every node has an equal probability of being mutated.

GP one-point crossover was derived from the GA crossover operator of the same name. Poli and Langdon developed another operator GA-inspired crossover, which forms the basis of much of the work reported in this thesis, called GP uniform crossover (GP-UX) [107]. GP-UX generates crossover points in much the same way as GP one point crossover, but then performs crossover on *all* the selected nodes with some probability. The nature of the crossover performed depends on whether or not the arities match in both trees – if they do, only the nodes are exchanged, otherwise the nodes and their sub-trees are swapped. GP-UX is described more fully in Chapter 6.

GP-UX has the additional property that exchange of nodes close to the root remains equally likely throughout the run. This is not the case for GP one point crossover because structural convergence means that crossover points at deeper locations become progressively more likely. However, we should expect lexical convergence in both, with the result that exchange of material will have no effect on offspring behaviour. Again, to prevent *premature* convergence, some level of point mutation should be used.

Although GP-UX and GP one-point crossover are homologous operators, this was not the main motivation behind their development. Instead, Poli and Langdon were seeking to address an issue that has become central to the theme of this thesis: the locality of search performed by standard GP. We explore this matter in the next section.

3.2.2 Global and Local Search in GP

One area of research that has received very little attention is the nature of the search that GP operators such as crossover perform. Most GA crossover operators generate offspring by combining genetic material from each parent in roughly equal quantities. At the beginning of the run, when diversity is high, this frequently results in offspring that are both structurally and semantically very different from their parents. As the population begins to converge, many of the bits become fixed in the population. Exchanging these bits during crossover has no effect, and offspring increasingly come to resemble their parents. Mutation, unless used with a very high probability, typically flips relatively few bits in the genome and also produces an offspring that is semantically and structurally similar to its parent. This allows the population to sample in parallel many hyperplanes early in the run, and to focus on, and search more thoroughly, a limited number of likely regions later. Another way of looking at this is to say that the population performs an initial global search and a subsequent increasingly local search of the entire space. The ability to move to distant points in the search space, at least initially, affords GAs advantages in some problems over approaches such as high-climbing since they are less prone to becoming trapped at local optima. It should also be noted that, if the operators are well-designed and applied to a suitable problem, movements in the genotype space usually correlate with movements in the fitness space, i.e. small changes to the genome correspond to small changes in fitness.

Attempting to construct an analogous argument for GP search operators is complicated by the fact that they act upon variable length representations. What does it mean, for example, to perform a global sampling of a GP program space? Is such an enterprise feasible, given the size of GP search spaces? In two recent papers, Poli [101, 107] showed that even in the face of these considerations, standard GP crossover is a good deal less explorative than appears to have been assumed. Poli's work centres on the amount of genetic information exchanged between parents as a proportion of overall size, and shows that crossover produces offspring that generally inherit most of their code from one parent. The conclusion he draws is that, on average, GP crossover performs only a very local search of the region occupied by each parent. GP populations may therefore behave in a manner similar to a population of parallel hill-climbers.

Poli also showed that GP uniform crossover initially performs a comparatively global search of the program space, in that offspring inherit genetic information from each of their parents in roughly equal quantities. The strictly homologous nature of the operator means that the nodes exchanged during crossover will eventually converge with the result that search will become increasingly local as the run continues.

With the exception of d'Haeselheer's context-preserving crossover, all the crossover operators reviewed in this chapter exchange single sub-trees between parents. The amount of material donated by each parent will therefore almost always be biased in favour of one or the other. Seen in the light of our definition of global search, these operators are all, to a greater or lesser degree, local search operators.

Poli's studies define globality and locality entirely in terms of the space of programs, and make few explicit predictions concerning the effects on behaviour[1]. It is clear from the literature reviewed in this chapter that GP populations frequently stagnate before a solution is found, in spite of the fact that new programs continue to be sampled. Can we expect GP uniform crossover to behave differently?

A number of studies indicate that we can. Igel and Chellapilla [46] used a distance metric similar to that described in [91] to measure the impact that various search operators had on offspring trees and found some support for their hypothesis that perturbations of the genotype correlate with changes in fitness. There was stronger evidence for a second hypothesis, originally proposed by Rosca and Ballard [114], that the impact an operation has on behaviour is inversely proportional to the distance of application from the root node. This is likely to be of particular significance because the selection of standard GP crossover points becomes increasingly biased towards

[1] Although we may note that, in the absence of a mutation operator, GP uniform crossover will, like its GA counterparts, eventually converge on a point from which further evolution is impossible.

lower nodes throughout the run [76, 33]. GP uniform crossover and point mutation are able to circumvent the consequent stagnation because they are applied to each node in the tree with equal probability. We provide additional evidence consistent with these findings in the next chapter.

It could be argued that the concept of global search proposed here is only one of many that may be applied to GP. For example, O'Reilly and Oppacher [94] describe a number of hybrid search algorithms in which standard GP crossover is used with simulated annealing and hill-climbing. They argue that their algorithm combines global, explorative search via the crossover operator, with local search being provided by the hill-climbing and (low-temperature) simulated-annealing components. With the hypothesis proposed here, we might argue the opposite – that standard crossover is performing local search around the space occupied by the two parents and that the global component of the algorithm arises when two unsuccessful parents are finally rejected and two new parents are generated at random. This feature may be termed global in the sense that it periodically allows the algorithm to sample distant points in the program space and thus escape from local optima.

It is important to note that we are not criticising local search per se, or suggesting that the strategy performed by GP uniform crossover – an initially global search followed by a progressively more local search – is necessarily optimal for all problems. The implications of the No Free Lunch theorems [25] are that there will be some problems for which a purely local search as performed by standard GP, or the essentially local search with an occasional random leap exhibited by O'Reilly and Oppacher's hybrid algorithms, represent a better approach. We instead suggest that GP can only profit from the development of operators that utilise different, well-understood search strategies, and that such operators are at present conspicuous by their absence in the literature. The work reported in the following chapters aims to address this.

3.3 Summary

We conclude this chapter with a summary of lessons that may be learned from the literature reviewed. These are:

- In most problems, GP programs show a marked tendency to increase rapidly in size throughout the run. This has been observed in both linear and tree-based genomes, and there is evidence to suggest that code growth is an endemic feature of search algorithms that use a variable length representation.

- Code growth in standard GP is exacerbated by two factors: inviable code and a removal bias that favours the replacement of small, inviable fragments with sub-trees of arbitrary size.

- Standard GP's blind approach to context means that large building blocks are unlikely to be constructed from small, fit sub-trees. Instead, standard GP crossover becomes increasingly destructive as partial solutions are evolved[2].

- Standard GP crossover can be viewed as a local search operator in that offspring inherit the majority of their genetic material from one parent.

- The bias towards nodes close to the terminals in the selection of crossover points means that nodes close to the root may converge prematurely.

- Even when modifications affect operative code, the bias towards lower nodes means that the impact on behaviour is small.

With the proposal of GP uniform crossover and point mutation [107], we have a pair of operators with the following properties:

- The ability to perform global and local search of the program space.

- The (as yet hypothesised) ability to perform global and local search of the fitness space.

- The operators are applied to each node in the tree with equal probability.

- Strict homology in node and sub-tree exchange.

[2] Again, we emphasise that this is not necessarily undesirable in all cases. If the acquired building blocks are leading search into sub-optimal regions of the space (that is, they are proving deceptive), we will actually want them to be disrupted. Alternatively, crossover acting in this way may serve simply to maintain genetic diversity.

- A fixed, and generally relatively small, upper bound on program size that is defined at initialisation.
- An implicit parsimony pressure exerted by point mutation (cf. [46]).

The remainder of this thesis examines the extent to which these properties enhance or impede search and their effect on the quality of solutions found. We compare performance of standard GP with that of GP uniform crossover as defined in [107]. We also introduce an extension of the GP uniform crossover and point mutation operators in conjunction with a novel representation for function nodes. We call these operators *GP smooth uniform crossover* and *GP smooth point mutation*, and refer to the representation as the *sub-symbolic node representation*. We test these operators and representation on a number of Boolean and symbolic regression problems. Full details are given in the following chapters.

Chapter 4

Global And Local Search Operators and Boolean Representations

4.1 Introduction

In their canonical forms, GAs and GP share many common features. Both utilise a randomly initialised population of individuals, a fitness function for evaluating the population, a fitness-based selection procedure, and unary and binary search operators for the generation of new individuals. The main differences lie in the genome representations and the operators that act upon them. Most GA crossover operators exchange information between corresponding locii on the string, and since the chromosomes are of fixed length it is unnecessary to impose constraints on the operator to ensure valid offspring. This is not the case in GP, however, because the parent trees may not both have nodes at the selected location, or the nodes may be of different arities. The exchange of information between parents in GP therefore needs to be constrained for all generated offspring to be valid.

In this chapter we describe an alternative to the standard GP approach – constraining the tree representation so that GA operators may work upon it. Specifically, each individual comprises a full syntax tree consisting of nodes of the same arity, flattened to form a linear chromosome. Assuming the nodes and terminals to be atomic units, GA crossover operators may be used unconstrained and will always yield valid offspring. GA point mutation can also be used with minor modifications. Whilst intended as a foundation study for work reported later in the thesis, these GA-based studies are interesting for a number of reasons:

- The GA search operators used here are essentially simplified versions of the GP uniform crossover and point mutation operators described in [107] and used in Chapters 5, 6 and 7.
- Both the GA and GP uniform crossover operators perform global search in the sense that offspring inherit genetic information from their parents in roughly equal quantities. We wish to investigate whether this property offers benefit for algorithms searching spaces of syntax trees.
- The constraints on the size and structure of the GA representation mean that the behaviour of the system will be easier to analyse than those in which the structure is allowed to vary. Nevertheless, many of the inferences based on the analysis given in this chapter will also apply to the GP search operators used later.
- We want to see whether, and under what circumstances, fixed structure representations facilitate or impede search relative to when variable length chromosomes are used.

The remainder of this chapter is structured as follows. In the next section we describe how full syntax trees may be flattened onto linear genomes so that GA operators can act upon them with little or no constraint. The experiments reported in the following two chapters address Boolean regression problems, and in this chapter we discuss two ways of representing Boolean function nodes. We then draw a distinction between phenotype and genotype in GP, discuss why this is important, and analyse the impact the search operators have on each under the two representations.

4.2 Tree Representation and Search Operators

Coding GP syntax trees as GA strings requires fixing both the size and structure of every individual in the population. We do this by considering only full trees of a pre-defined depth, constructed from functions of the same arity. Trees are flattened and represented on the chromosome using a prefix notation similar to that described in [54], an illustration of which can be seen in Figure 4.1.

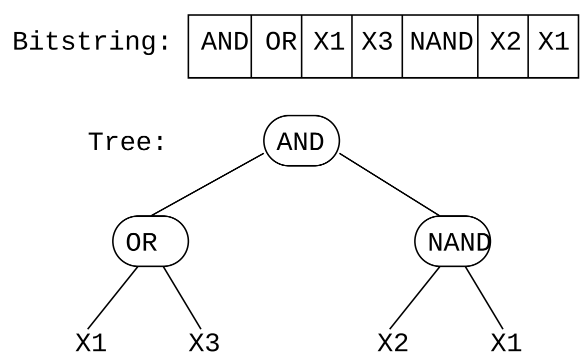

Figure 4.1: Encoding a two dimensional syntax tree on a one-dimensional string using prefix notation.

This representation allows the use of virtually unmodified GA search operators. In our investigations of variable-length chromosomes in chapters 6 and 7, we use GP uniform crossover and GP point mutation [107], and we therefore use GA uniform crossover [127] and point mutation here. With this representation, slight modifications are made to each operator. Uniform crossover generates offspring by swapping each of the parental function nodes with 50% probability. Terminal nodes, however, are not considered, for reasons that will be discussed below. A single application of the point mutation operator selects a random function node, and replaces it with another drawn at random from the function set. Again, terminal nodes are not considered. The action of GA uniform crossover applied to trees represented in this way is illustrated in Figure 4.2.

There has been very little work that addresses the issue of representation in GP. In most implementations, representation considerations do not go beyond including primitive function nodes that it is assumed will prove useful to the population in

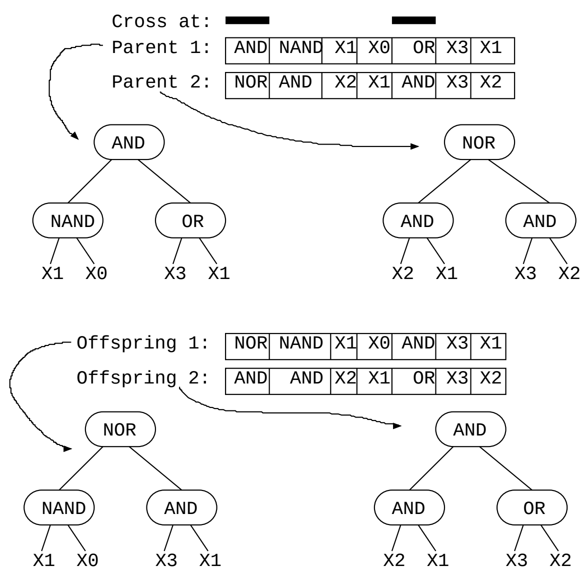

Figure 4.2: Uniform crossover applied to two one dimensional strings, and its effect on the offspring syntax trees.

evolving a solution. This seems to ignore the importance placed on selecting a suitable representation in other EAs, and for this reason a major part of this thesis will be devoted to investigating the question.

Many attempts to solve Boolean regression problems with GP use a function set consisting of the four arity 2 functions `AND, OR, NAND, NOR`. Whilst an expression equivalent to any Boolean function can be constructed from this function set, it may nevertheless prove limiting for two reasons:

- It is biased. 12 of the 16 Boolean functions of arity 2 are omitted. If the function to be evolved does not require any of the omitted functions, this may be seen as building knowledge into the system. Conversely, if one or more of the omitted functions are necessary in solving the problem, they will have to be constructed from the primitives, possibly at multiple locations throughout the tree.
- The smallest syntactical change possible with our operators – point mutation – may still have a sizeable impact on the values returned by the tree. For example, if the root node of the tree {`NAND X1 X0`} is mutated to `AND`, the output of the tree is flipped in all four fitness cases.

The first point can be addressed by simply including all 16 Boolean functions in the set. Interestingly, this function set subsumes the arity 1 function set, including two functions that return one or other of their arguments unchanged, and two that return their negations.

In addressing the second point we begin by noting that a Boolean function of arity n can be defined by its truth table, which may in turn be converted to a bitstring of 2^n bits. Thus the arity-2 function `AND` can be defined by the bitstring `0001`, where the least significant bit refers to the case where both inputs evaluate to `true`. Flipping a single bit of a function represented in this way alters the value returned by the node in just one of the four fitness cases.

We now have two levels at which we may view an atom of genetic information: the function and the bit. We call the former the *symbolic representation* and the latter the *sub-symbolic representation*. Adopting the sub-symbolic representation for functions gives us a linear binary chromosome on which we may use *unconstrained binary GA search operators*, at least on non-terminal nodes. We cannot, however, use such unconstrained operators on the terminals (unless, of course, the terminal set happens to consist of 2^n inputs) since this may result in invalid symbols. For this reason, we encoded the terminals separately and, in most of our experiments, did not apply the search operators to them at all. An illustration of GA uniform crossover applied to two trees under the sub-symbolic representation is given in Figure 4.3.

The discussion so far has focussed on functions of arity 2. Encoding functions of higher arity is not difficult, but it does have serious implications for the size of the search space if we wish to include the full set. In the studies reported here, we also considered functions of arity 3. The 3 Boolean arguments give 256 possible truth table outputs, and the size of a full tree of depth D will also be significantly larger since each function has three branches. We may note, however, that a function set represented in this way necessarily subsumes all functions of arity 1 and 2, these being the functions where the other arguments play no part in influencing the output (e.g. {`00010001 x1 x2 x3`} `=` {`AND x2 x3`}). Of the remainder, 24 perform some variation of `IF-THEN-ELSE` with arity-1 arguments (e.g {`10100011 x1 x2 x3`} `= IF x1 THEN x2 ELSE NOT x3`)). Unfortunately, unlike `XOR` and `EQ`, these functions are sensitive to the ordering of their arguments, so they can't be used with the same freedom.

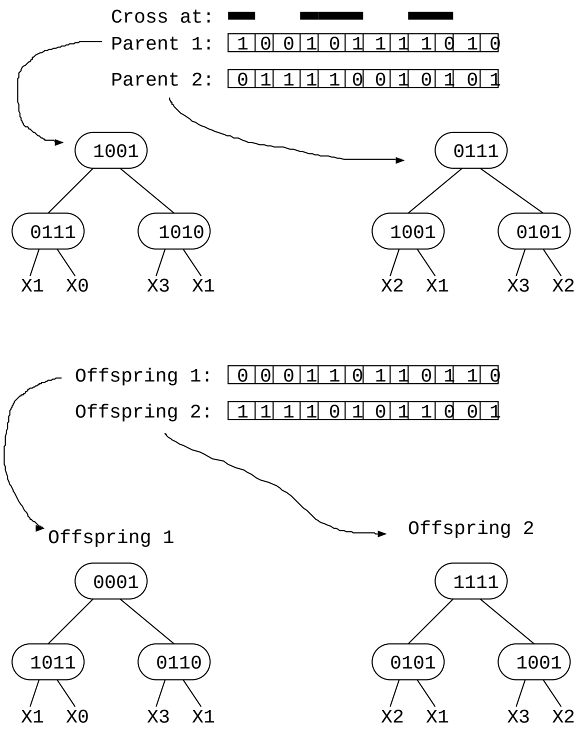

Figure 4.3: Uniform crossover applied to a binary bitstring encoding a Boolean syntax tree. The values of the bits at each node specify the truth table of the function.

We have described two ways of representing Boolean syntax trees as linear GA chromosomes. In both cases, the chromosomes encode full trees consisting of functions of the same arity (2 or 3). The functions are represented as truth tables specifying the value returned for any of the possible inputs, and any function that can be represented in this way is valid. To a human observer, trees encoded under either representation will be indistinguishable, appearing simply as linear bitstrings. The difference is only "seen" by the search operators. To the symbolic search operators, the atom of genetic information is the function – crossover and mutation can only exchange or replace entire nodes. The sub-symbolic search operators, however, act at the bit level. As far as they are concerned, the fact that the chromosome encodes a hierarchical syntax tree is therefore irrelevant.

The difference in genetic atomicity between the two representations has subtle implications for the manner in which search is performed under the two representations. Symbolic uniform crossover swaps, on average, half the function nodes in their entirety, and leaves the remainder unaffected. Sub-symbolic uniform crossover, however, swaps half the *bits* on average. *Both* operators therefore exchange about 50% of the genetic material between parents – it is the distribution of the exchanged information that varies. In the case of mutation, the number of operations applied to an individual is the number of bits multiplied by the mutation probability, p_m. Depending on the representation, bits or functions are then selected at random and mutated. Since a single symbolic point mutation generally flips more bits than a sub-symbolic point mutation, it will perturb the genotype to a greater degree. Furthermore, the distributions of bitwise mutations will again vary – symbolic mutations will tend to occur in clumps, whilst sub-symbolic mutations will be spread more evenly.

These differences have further implications for the locality of search each operator is able to perform on the genotype space. Symbolic point mutation will maintain quite a high genetic diversity in the population, particularly when the larger arity-3 function set is used, and this will prevent a truly local search. Sub-symbolic mutation, however, perturbs the genotype to a far smaller degree and will permit a very local search of relatively small areas of the search space once the population has converged.

The preceding discussion has dealt with search of the genotype space. In GP, chromosome evaluation is not performed directly on the syntax tree, but is instead a function of its behaviour over one or more test scenarios. Put another way, fitness evaluation in GP, as in natural evolution, is a function of the individual's *phenotype*. In the next section, we discuss the meaning of the term when applied to GP individuals and then go on to analyse the degree to which movement around the genotype space, as effected by the search operators, correlates with movement around the genotype space.

4.3 Genotype, Phenotype and Fitness in GP

In evolutionary biology, distinctions are frequently drawn between an organism's genotype, phenotype and fitness. At the most basic level, the genotype is simply the specific sequence of nucleic acids in the DNA. Although the mapping process is extremely complex, variations in this sequence ultimately lead to differences in the physical and behavioural characteristics of an organism. Those characteristics under genetic control are collectively termed the phenotype. The fitness of a gene is defined as the ability of the gene to reproduce itself, and it is important to note that whilst it is *genes* that are selected for reproduction, the selection process operates at the fitness level, which is a function of the phenotype. We also note that as a result of introns and the redundancy of the genetic code, it is possible for two different genotypes to produce identical phenotypes and hence to possess the same fitness.

In EA research, the terms fitness and genotype are also widely used. References to an individual's phenotype are made less frequently, although in systems such as GP, where fitness is generally evaluated over a number of fitness cases, the term is both valid and useful. In GP, an individual's phenotype may be defined as the set of its outputs over all the assigned fitness cases. As in biological genetics, it is possible for two different genotypes to produce the same phenotype, especially if the coding is redundant or, as is more often the case, intron branches may occur. Furthermore, it is possible for two individuals exhibiting different phenotypes to possess the same fitness.

We can illustrate this by considering the example of Boolean regression tasks for GP. A Boolean problem of n arguments is typically evaluated over all its 2^n possible inputs giving 2^n binary return values, and the concatenated bitstring of these outputs may be termed the chromosome's phenotype. The mapping from the genotype to the phenotype space is clearly surjective, since a given output can be obtained from many different trees. Depending on the fitness function, the mapping from phenotype to fitness space may also be surjective so that, for example, a string of 64 1s and a string of 64 0s will both score 32 hits on the even-6-parity problem.

Why is the phenotype significant? Principally because GP fitness functions operate on the chromosome's phenotype, as opposed to directly on the genotype. Efficient search operators should therefore not only modify genetic information, but do so in such a way that differences at the phenotypic level are frequently observed. If the fitness landscape is rugged, it may also be desirable for some of these differences to be quite large to enable the population to escape from local minima. For Boolean problems, we can obtain some idea of how well any search operator satisfies these criteria by taking the Hamming distance between parent and offspring phenotype strings.

4.3.1 GA Search Operators and Boolean Phenotype Spaces

It seems likely that the effect that a search operator applied to a single node has on the phenotype will depend on a number of factors: the arity of the function set, the representation used, the number of input variables, the depth of the tree and the depth of the node upon which the operator is applied within the tree.

We examined the effect empirically for point mutation as follows. We generated 10,000 trees of depth D (i.e. $D-1$ function levels and one level of terminals) and mutated a single, randomly-selected node at depth d, where $0 \leq d < D$, the root node being located at level 0. The experiments were repeated for both the symbolic and sub-symbolic representations. Given k input terminals, we concatenated the values returned by the tree over the 2^k fitness cases before and after application of the mutation operator into two binary output vectors. A measure of the impact on phenotype produced by a single application of the mutation operator at depth d is therefore the Hamming distance between these two strings.

Tables 4.1 and 4.2 shows the average number of bits flipped in the output strings using full arity-2 trees of depth 6 under both representations, with $k = 4, 5$ and 6 and $0 \leq d < 6$. Other values of D were tried, and these produced substantially the same results. From these results, it is clear that a single application of the point mutation operator acting on the sub-symbolic representation flips the output returned in approximately $\frac{1}{2^{d+2}}$ fitness cases. When the symbolic representation is used, this value is doubled, i.e. the value is altered in $\frac{1}{2^{d+1}}$ fitness cases.

Table 4.1: Impact on phenotype, in terms of average number of bits flipped in the concatenated output string, as a result of a single point mutation applied to a randomly selected node at depth d. The results given are for full trees of depth 6 with the arity 2 function set and symbolic representation.

	Inputs (fitness cases)		
d	4 (16)	5 (32)	6 (64)
0	7.93	16.01	32.08
1	4.003	8.01	15.86
2	1.97	4.09	8.01
3	0.99	2.05	3.95
4	0.53	0.996	2.04
5	0.27	0.49	0.97

Table 4.2: Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 2 function set and sub-symbolic representation.

	Inputs (fitness cases)		
d	4 (16)	5 (32)	6 (64)
0	3.94	8.09	16.07
1	1.97	3.95	7.88
2	1.04	2.08	4.07
3	0.49	1.002	1.97
4	0.23	0.5	1.002
5	0.118	0.2572	0.4872

These effects are essentially what we would predict. Point mutation acting on the sub-symbolic representation flips one of the four bits of an arity 2 truth table. If applied to the root node this will usually flip the value returned by the tree in $\frac{1}{4}$ of the fitness cases. Point mutation acting on the symbolic representation replaces the function with another drawn at random from the function set. With a function set of all 16 arity-2 functions, the replacement is, on average, 2 Hamming points distant from the node it replaces, as opposed to 1 when the sub-symbolic representation is used. This represents a doubling of the number of return values changed over all the possible fitness cases.

It is clear from Tables 4.1 and 4.2, however, that the relationship is only approximate – sub-symbolic point mutation, for example, does not flip the output in exactly $\frac{1}{2^{d+2}}$ of the fitness cases every time it is applied. This is because the exact value depends on the arguments provided to the function node. Consider the case when one of the subtrees evaluates to a constant expression, e.g. {`0110 X0 1`} `= NOT X0`. Flipping the first or third bits of this node has no effect on the value returned. Flipping the second or fourth gives the functions `always-OFF` and `always-ON` respectively. A sub-symbolic point mutation to this node therefore either has no effect on the node's behaviour or it changes the value returned in half the fitness cases. Similar effects are observed when the arguments are duplicated (e.g. {`1011 X0 X0`}).

We repeated the experiments using the arity-3 function set. The results are shown in Tables 4.3 and 4.4. Again, the results are in line with what we might predict. A symbolic point mutation replaces the mutated function with another node drawn at random from the set of 256 functions. As with the arity 2 function set, this represents, on average, a flipping of half the bits in the node. However, a single sub-symbolic point mutation flips one of eight bits in the node, i.e. it changes the output returned in half as many fitness cases as it does when applied to an arity-2 function. We can therefore see that a single symbolic point mutation acting on an arity 3 node at depth d, changes the output returned by the tree in $\frac{1}{2^{d+1}}$ of the fitness cases, whilst a sub-symbolic mutation does so in $\frac{1}{2^{d+3}}$ of the fitness cases.

These results demonstrate that both mutation operators are capable of perturbing the phenotype to quite a large degree if they are applied to nodes close to the root of the tree. However, sub-symbolic point mutation makes smaller changes on average, producing offspring more closely related to their parents in terms of both phenotype and genotype. In other words, sub-symbolic point mutation performs a more local

Table 4.3: Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 3 function set and symbolic representation.

	Inputs (fitness cases)		
d	4 (16)	5 (32)	6 (64)
0	7.99	16.07	32.13
1	3.98	7.9	15.91
2	2.002	4.005	7.88
3	0.999	2.02	3.9
4	0.49	0.99	1.997
5	0.26	0.5	1.01

Table 4.4: Impact on phenotype of a single point mutation applied to a node at depth d. The results given are for full trees of depth 6 with the arity 3 function set and sub-symbolic representation.

	Inputs (fitness cases)		
d	4 (16)	5 (32)	6 (64)
0	1.98	3.99	7.98
1	1.01	2.01	3.95
2	0.49	1.003	1.98
3	0.25	0.51	1.03
4	0.135	0.24	0.499
5	0.067	0.133	0.246

search of the phenotype as well as the genotype space than symbolic point mutation.

The behaviours of GA crossover operators are more difficult to analyse since the effect on the offspring depends on how closely related the parents are. Two genotypically distant parents often give rise to offspring that bear little phenotypic resemblance to either. Closely related parents tend to produce similar offspring in terms of both phenotype and genotype, since much of the information exchanged is the same in both parents. The latter situation becomes more likely as a run progresses and the population converges, with the result that an increasingly local area is searched. Clearly, genotypically identical parents will produce genotypically and phenotypically identical offspring, and to prevent irreversible premature convergence in the population, a certain amount of mutation should always be used.

We have seen that the effect of a single point mutation on the phenotype is inversely proportional to the depth of the mutated node. In considering the effects of crossover on a function node, it is useful to think of uniform crossover as a macro-mutation operator where the number of bits flipped in a single application is a function of the Hamming distance, h, between the two parent nodes. As the population converges, what may be termed the "macro-mutation probability" decreases, and may reach 0 in the absence of a point mutation operator. Since uniform crossover acting on the symbolic representation exchanges nodes in their entirety, the number of bits flipped in both offspring is simply h. Uniform crossover acting on the sub-symbolic representation swaps, on average, half the bits so that $\frac{h}{2}$ are flipped. It therefore follows that uniform crossover acting on two 4-bit parent nodes at depth d will alter the output returned by the offspring trees in approximately $\frac{h}{2^{d+2}}$ fitness cases for with standard representation, and $\frac{h}{2 \times 2^{d+2}} = \frac{h}{2^{d+3}}$ fitness cases with the sub-symbolic representation.

It must be emphasised that we have been discussing the effect that results from the application of uniform crossover to a single node. In practice, symbolic uniform crossover will be applied to half the nodes in the tree, regardless of their arity. In contrast, sub-symbolic uniform crossover is applied to half the *bits* in the tree, such that the probability of a node escaping crossover altogether is $\frac{1}{2^n}$, where n is the arity of the function. Sub-symbolic uniform crossover will therefore be applied to some degree to the majority of the nodes in the tree, particularly when the arity 3 function set is used. This is offset, however, by the fact that the sub-symbolic mutation operator will almost certainly maintain a much lower genetic diversity, with the result that many of the exchanged bits will be the same in both parents. With this in mind, it seems likely that sub-symbolic uniform crossover, whilst performing an equally global search of the phenotype space early in the run, will be able to perform a far more local search than symbolic uniform crossover once the population has converged.

4.4 Summary

In this chapter we have presented two pairings of search operators and representation which are capable of performing an initial global search of both genotype and phenotype spaces. For simplicity, we have fixed the structure of the representation so that we are essentially dealing with a GA. We have argued that whilst both representations allow the operators to perform an initially global search, the sub-symbolic representation also affords a converged population the ability to make a very local search of a highly fit region of the search space. The next chapter investigates what advantages, if any, this will afford the sub-symbolic representation as we compare it against the symbolic representation on two well-known benchmark Boolean regression problems – the even-n-parity function and the 6-multiplexer.

Chapter 5

Evolving Boolean Functions with Binary-Represented Genetic Algorithms

5.1 Introduction

In the previous chapter, we described in detail a GA-based encoding of syntax trees. Within the framework of Boolean functions, we also described two ways of representing function nodes within the trees. In this chapter, we examine the performance of these two representations on two well-known Boolean test problems: the even-n-parity problem and the 6-multiplexer.

In the even-n-parity problem, the task is to return 1 (`true`) if an even number of the n inputs evaluate to 1, 0 (`false`) otherwise. This problem has been studied extensively in the GP literature and, as we shall see in Section 5.2, has been shown to be a very difficult problem for standard GP to solve. Whilst the reasons for this will be discussed in full in this chapter, it is immediately apparent that a contributing factor is that flipping the value of one of the inputs requires flipping the value returned by the tree.

The n-multiplexer consists of p address bits and 2^p data bits, so that $n = p + 2^p$. For each fitness case, the task is to return the value of the data bit at the location specified by the address bits. In this chapter, we consider a version of the problem with 2 address and 4 data bits – the 6-multiplexer. This problem has hitherto been considered unimodal, and is in fact a much easier problem for standard GP to solve than the equivalent (in terms of inputs and fitness cases) even-6-parity problem. Koza [58] estimates the minimum effort required for standard GP to solve the 6-multiplexer to be 245,000.

In this chapter, we compare performance of the symbolic and sub-symbolic representations on these two problems using fixed-size full syntax trees encoded as GA chromosomes. Full details of the experimental set-up are given in Section 5.3. We investigated the behaviour of the two representations on the two problems using two different function sets: a full arity-2 Boolean function set of 16 functions, and a full arity-3 Boolean function set of 256 functions. The results with the arity-2 function set are presented in Section 5.4 and those with the arity-3 function set are given in Section 5.5. The results are summarised in Section 5.6 and the chapter concludes with a discussion. We begin, however, with a review of previous work on the even-n-parity problem.

5.2 Previous Work

Koza provided a detailed treatment of the even-n-parity problem in his extensive discussions of the standard GP paradigm [58] and Automatically Defined Functions (ADFs) [59]. In both cases he restricted himself to the four arity-2 Boolean functions {`AND`, `OR`, `NAND`, `NOR`} and whatever input terminals were necessary for the problem. The omission of the `XOR` and `EQ` primitives from the set creates additional problems for standard GP, since it must independently evolve semantically equivalent blocks of code at numerous locations throughout the program tree. Unsurprisingly, solving the parity problems using standard GP without ADFs is computationally expensive and Koza was unable to obtain a result for values of $n > 5$. Of course, the discovery and reuse of building blocks is the idea behind ADFs, and Koza reported greater success when they were used. The solutions he reported for $n = 3, 4, 5, 6, 7, 8, 11$ all evolved, and made extensive use of, code equivalent to either the `XOR` or `EQ` primitives on their function-defining branches. The estimated computational effort for these and, where available, the other studies discussed here are

	N				
Approach	5	6	7	8	9
Standard GP [59]	6,528	70,176[a]	n/a	n/a	n/a
GP + ADFs [59]	464	1,344	1,440	solved[b]	
GPPS 1.0 [60]	9,087	31,250	n/a	n/a	n/a
CGP [80]	49	n/a	n/a	n/a	n/a
EP [17]	2,100	n/a	n/a	n/a	n/a
EP + ADFs [17]	126	121	169	321	586
GP + ADFs [1]	359	627	n/a	n/a	n/a

[a]Estimated.

[b]Koza solved parity problems for n up to and including 11. However, the heavy computation load required prevented him from performing sufficient runs to calculate the effort for $n > 5$.

Table 5.1: Minimum efforts (in thousands of fitness evaluations) required to solve the even-n-parity problem using various methods.

given in Table 5.1.

Koza *et al.* [60], also tackle the even-n-parity functions with the *Genetic Programming Problem Solver* (GPPS). This is a sophisticated extension to the GP paradigm which allows GP to utilise a wide range of arithmetic and Boolean functions, ADFs, loops, and indexed memory. All these features are made available regardless of the problem, and it is then up to GPPS to decide whether and in what manner it uses each of them. It can be seen from Table 5.1 that GPPS is outperformed by standard GP on the even-5-parity problem (although it significantly improves on Koza's estimate for $n = 6$), and markedly inferior to GP with ADFs. Such comparisons are simplistic, however, as they do not take into account the motivation behind GPPS. The primary goal of GPPS, as its name suggests, is generality, and Koza *et al.* have quite justifiably opted to trade performance for ease of application. Nevertheless, it does improve on Koza's standard GP estimate for $n = 6$, most probably because ADFs are available.

Chellapilla [17] essentially replicates these studies using Evolutionary Programming (EP), omitting the crossover operator and using instead a variety of mutation operators. His results compare favourably with those of Koza and he uses them to argue that the significance of crossover has been overstated.

In [80], Miller uses an approach known as Cartesian Genetic Programming (CGP) to solve a number of Boolean logic problems including the even-n-parity functions for $3 \leq n \leq 5$. CGP uses linear chromosomes that both encode a 2-dimensional array of functions, and specify their connections. He gives a number of results for various array and population sizes, of which the best is an effort of 49,204 obtained with a 30x30 array and a population of 4 individuals.

Aler [1] presents a modification of Koza's ADF technique in which function and result-producing branches are evolved in separate populations. The main branch-evolving population uses the ADF of the best individual from the ADF population of the previous generation and vice versa. His results on the even-5 and -6-parity functions also compare well with those of Koza.

Gathercole and Ross [34] use GP *without* ADFs and a fitness function in which evaluation of the individual ceases once a given number of fitness cases have been misclassified. If this threshold is reached, the remaining, untested, cases are also judged as misclassifications. The result is that low-fitness programs can be identified without being evaluated on the full training set, with considerable savings in CPU time. The order of presentation of the fitness cases is modified at run-time so that the evolving population is exposed to cases of increasing difficulty. Gathercole and Ross's results are very encouraging although they do not estimate the effort.

The studies discussed so far have all treated the even-n-parity problem for different values of n as distinct tasks and evolved specific solutions for each. Wong and Leung [133] describe an alternative approach in which logic programs are evolved. Programs operate recursively on a list of Boolean values (each list representing a single fitness case), rather than a set of distinct input terminals. Whilst the solutions reported by Wong and Leung are general solutions to the even-n-parity problem, their method is necessarily heavily constrained to avoid infinite recursions and contains a great deal of problem-specific information. Less biased is the approach of Yu [136] who evolves recursive general solutions to the even-n-parity problem using lambda abstractions.

A brief examination of Table 5.1 shows that, as n increases, the problem rapidly becomes very difficult for GP systems which use the traditional function set without any favourable inbuilt bias. For $n \geq 6$, the effort is at least of the order of 10^5, and for many methods the problem becomes intractable. In the next section we describe a GP algorithm that finds parsimonious solutions to the parity problems with a high level of efficiency.

In the next section, the parameters used in both problems are given and the reasons for their choice are explained. The two sections following provide and analyse the results of 50 repeated runs on each problem.

5.3 Experimental Set-up

We compared performance of fixed-length full trees using two representations. These were:

- **Symbolic**, where the function is the atomic unit. Uniform crossover acting at a given point exchanges the nodes in their entirety between the two parents. Point mutation replaces the node with another drawn at random from the function set.
- **Sub-symbolic**, where the bit of the truth table is the atomic unit. Uniform crossover exchanges bits between each parent with 50% probability. A single point mutation flips the value of a bit at a randomly selected location.

We examined two function sets: the set of all 16 Boolean functions of arity 2, and the set of all 256 Boolean functions of arity 3. Both are larger than usual in GP studies, the latter significantly so. We may observe, however, that they are unbiased in that no assumptions concerning useful building blocks for the test problem are made.

We also varied tree depth. The size of the search space increases exponentially with each increment in depth, and increases more rapidly for arity-3 functions. A full tree of depth D (i.e. $D-1$ function levels and 1 level of terminals) comprises $2^{D-1}-1$ functions and 2^{D-1} terminals for an arity-2 function set, and $\sum_{i=0}^{D-2} 3^i$ functions and 3^{D-1} terminals for arity-3. In the sub-symbolic version, this corresponds to $4 \times 2^{D-1} - 1$ bits for the representation of the function levels of a full tree of arity-2 nodes, whilst for arity-3, $8 \times \sum_{i=0}^{D-1} 3^i$ are required. The size of the search space thus increases exponentially for each additional level, and we are forced to trade search space dimensionality against the minimum complexity for a solution to exist. More will be said on this matter below, although we may observe that a lower limit on tree depth should be one that allows each of the n terminals to occur at least once in the tree. Terminals were encoded on a separate string and were not subjected to the action of the search operators, i.e. their values did not change after initialisation.

The experimental parameters are summarised in Table 5.2. We used two population sizes for each condition, namely 50 and 200 individuals. The crossover probability is the proportion of the population that underwent crossover at each generation. The probability of exchanging atomic genetic information – bits or functions depending on the representation – during crossover was 0.5. The mutation probability is a function of the number of bits, B, required to encode either representation. The total number of mutations performed at each generation is thus $B \times p_m \times Popsize$. In each case, a single mutation was applied to a randomly-selected node or bit (depending on the representation) of a randomly-selected individual.

Table 5.2: Parameters used in all the experiments reported in this chapter.

Number of runs	50
Population sizes	50, 200
Maximum generations	100
Tournament size	7
Tree Depth	3 - 7
Crossover prob.	0.7
Mutation prob.	0.1

5.4 Experiments with the Arity-2 Function Set

5.4.1 Even-n-Parity Problem

The minimum efforts required to solve the even-n-parity problem for $5 \leq n \leq 8$ for the sub-symbolic and symbolic representations, with populations of 50 and 200 individuals, are given in Tables 5.3 to 5.6.

Table 5.3: Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses.

	n			
Tree Depth	5	6	7	8
3	2,100 (48)	5,700 (30)	21,600 (13)	159,600 (1)
4	950 (50)	1,900 (50)	3,800 (48)	7,350 (42)
5	1,450 (50)	1,950 (50)	2,450 (50)	2,550 (50)
6	2,800 (50)	2,450 (50)	5,600 (48)	4,450 (50)
7	3,750 (50)	5,900 (49)	6,750 (48)	14,250 (35)

Table 5.4: Minimum efforts required by the symbolic representation to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses.

	n			
Tree Depth	5	6	7	8
3	4,000 (48)	9,750 (33)	59,400 (3)	216,600 (1)
4	3,550 (50)	4,800 (49)	5,800 (48)	13,800 (38)
5	4,050 (49)	5,900 (47)	9,000 (48)	12,600 (43)
6	8,100 (48)	9,600 (45)	18,300 (33)	33,600 (25)
7	15,600 (39)	28,200 (28)	39,200 (22)	77,900 (11)

Table 5.5: Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem, population size = 200.

	n			
Tree Depth	5	6	7	8
3	2,600 (50)	6,400 (48)	35,000 (27)	347,200 (4)
4	4,600 (50)	4,000 (50)	5,200 (50)	10,000 (50)
5	5,200 (50)	5,400 (50)	7,400 (50)	8,800 (50)
6	6,000 (50)	7,800 (50)	10,600 (50)	20,400 (49)
7	17,200 (50)	17,600 (49)	33,000 (49)	47,400 (43)

Clearly the even-n-parity function – a difficult problem for standard GP to solve – is trivially easy for the GA encoded approach. The performance of the sub-symbolic representation on the even-6-parity function is some 4 orders of magnitude better than standard GP. Furthermore the effort required to solve the function with each increment in n increases linearly for both representations, whereas it increases by an order of magnitude for standard GP. This is due to at least three factors:

- The globality of the search operators.

Table 5.6: Minimum efforts required by the symbolic representation to solve the even-n-parity problem, population size = 200.

	n			
Tree Depth	5	6	7	8
3	4,000 (50)	10,400 (47)	38,400 (26)	450,000 (4)
4	3,600 (50)	6,200 (50)	7,600 (50)	17,600 (48)
5	8,000 (50)	9,800 (50)	18,000 (50)	26,800 (49)
6	12,000 (50)	30,400 (49)	44,800 (44)	84,000 (33)
7	44,400 (42)	97,000 (31)	201,600 (18)	502,200 (8)

- The fixed size of the search space.
- The inclusion of the `EQ` and `XOR` building blocks in the function set.

The locality of GP's crossover operator has already been discussed in previous chapters. Clearly a more global search operator is well-suited to solving the parity functions. The constraint on the size of the search space is also likely to be significant, since the bloating of candidate solutions is prevented.

The inclusion of the `XOR` and `EQ` building blocks in the function set proved very useful. All the correct solutions observed constained numerous instances of both. An example depth 3 solution is shown in Figure 5.1. We see that all the nodes are either `XOR` or `EQ` except one – `0101` – which returns the second argument unmodified. Interestingly, the intron branch thus created – {`0110 X4 X3`} – also contains the `XOR` function. Presumably this subtree was useful in antecedent partial solutions.

Larger trees also contained numerous instances of the building blocks, although they accounted for a smaller proportion of the nodes used. A solution of depth 5 is shown in Figure 5.2.

Analysing this tree we see many instances of the functions `0011` and `1100`, which return the first argument and its negation respectively, and `0101` and `1010` which do the same with the second argument. The behaviour of the function `0100` that supplies the root node with its first argument is modified by the fact that its first

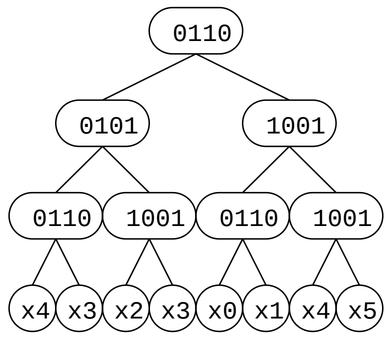

Figure 5.1: A Depth 3 solution to the even-6-parity problem

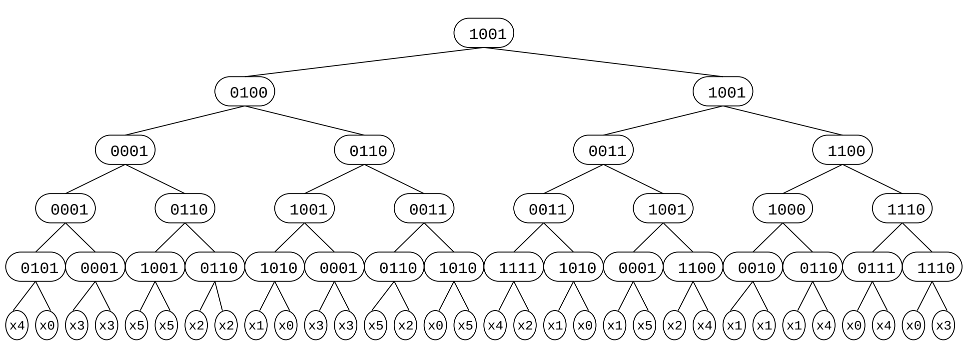

Figure 5.2: A Depth 5 solution to the even-6-parity problem

argument evaluates to `false` so that it returns its second argument unchanged (i.e. it behaves as an arity-1 function). Arity-1 functions were a prevalent feature of the larger solutions and occurred in a number of ways:

- As pure arity-1 equivalent functions that either returned one of the arguments unchanged (e.g. `0011`, `0101`) or its negation (e.g. `1100`, `1010`).
- By supplying a constant sub-tree as one of the arguments, as described above.
- By duplicating the arguments. Examples in the tree shown in Figure 5.2 are {`1001 X5 X5`}, {`0110 X2 X2`} and {`0001 X3 X3`} which return `true`, `false` and `X3` respectively.

Large quantities of inviable code also occurred frequently in larger trees, arising through functions that ignored one or both of their arguments.

In *every* solution analysed, the removal of intron branches and arity-1 functions resulted in a tree of striking simplicity containing only instances of the `EQ` and `XOR` building blocks. The simplified version of the tree shown in Figure 5.2 is shown in Figure 5.3.

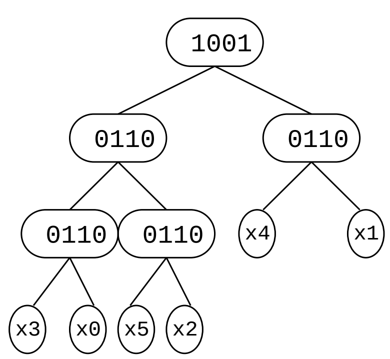

Figure 5.3: Simplified form of the tree shown in Figure 5.2.

Deeper trees clearly afford the system increased flexibility in solving the parity problems via the mechanisms described above. This accounts for the superior performance of depth 4 and 5 trees over those of depth 3, despite the larger search space. There is a trade-off between flexibility and the size of the search space, however, and for trees of depth 6 and above, performance begins to deteriorate.

There is a noticeable deterioration in performance of depth 3 trees as n is increased from 7 to 8. Depth 3 trees consist of 7 function nodes and 8 terminals, which offer no redundancy at all for the even-8-parity function. A solution must therefore include exactly one instance of each of the 8 terminals, and exclude all functions but the `XOR` or `EQ` building blocks. This gives very little gradient information, with the vast majority of trees scoring exactly half marks, and since the terminal symbols were not mutated it was common for the population to lose one or more of them.

A simple way of overcoming the problem of loss of terminal symbols was to adapt the search operators so that terminals were also considered. Crossover was modified so that terminal symbols were also exchanged with 50% probability. The mutation operator replaced a randomly selected terminal symbol with another drawn at random from the set with a 10% probability, mutating function nodes as normal in the remainder of cases. This applied to both the symbolic and subsymbolic representations. The results are shown in Table 5.7.

Table 5.7: Minimum efforts and number of successes (out of 50) for the even-8-parity problem with terminal mutation. Only trees of depth 3 were considered.

Representation	Pop. size	Effort
Sub-symbolic	50	89900 (10)
Sub-symbolic	200	110400 (26)
Symbolic	50	59850 (12)
Symbolic	200	123200 (27)

These results, whilst a significant improvement over the case when the search operators ignored the terminal symbols, are still considerably worse than we might predict given the increase in effort required with increasing n for higher depths. Moreover, when we tested the adapted search operators on the other conditions (i.e. other values of n and deeper trees) we observed no change in performance. This is a surprising result when one considers that even for $n = 5$ a sizeable proportion of an initial population of depth 3 trees will not include the full complement of terminal symbols necessary to solve the problem. Furthermore, with no mutation operator to replace lost terminals, we might expect the population to converge frequently on trees in which one or more terminal is missing. In practice this almost never happened. The reason is that any tree lacking one or more of the terminal symbols *can only score half marks*. Even with $n = 7$ and depth 3, individuals scoring above half marks were quickly identified and these quickly came to dominate the population, thus excluding solutions in which one or more terminals were absent.

The discussion up to now has not considered performance differences between the symbolic and sub-symbolic representations. Returning to Tables 5.3 to 5.6, we see that the sub-symbolic representation outperformed the symbolic representation on almost every condition (the exception being depth = 4, population size = 200, $n = 4$, where the symbolic representation was clearly fortunate). In general, the sub-symbolic representation outperformed the symbolic representation by just less than a factor of 2 for depth 3 trees, and by a factor of 3 to 5 for the depth 7 trees. It is clear that the sub-symbolic representation, by virtue of its ability to perform both global and very local search, is able to negotiate the search space far more effectively than the symbolic representation, and this effect increases with the size of the space.

5.4.2 6-Multiplexer

We tested performance on the 6-Multiplexer using the same conditions as for the even-n-parity problem. The symbolic representation failed to find a solution in any of the runs, whilst the sub-symbolic representation solved the problem once each for trees of depth 5 and 6. The single solution of depth 5 found by the sub-symbolic representation is shown in Figure 5.4. This solution is difficult to analyse – there are far fewer introns than in a typical parity solution of the same size and the coding branches interact in a complex manner.

Given the ease with which the parity problems were solved, the poor performance on the multiplexer was initially surprising. It is, after all, easier for standard GP to solve than the parity problems. A likely reason is that the multiplexer is an extension of the arity-3 `IF` function, in much the same way that the even parity functions are extensions of `EQ`, and this building block was of course absent from the arity-2 function set. In fact, most GP implementations that solve the multiplexer include `IF` in the function set. With only an arity-2 function set, the system is forced into trying to construct sub-trees that perform tasks equivalent to several `IF` functions – a formidable task given the number of functions necessary. Including the `IF` building

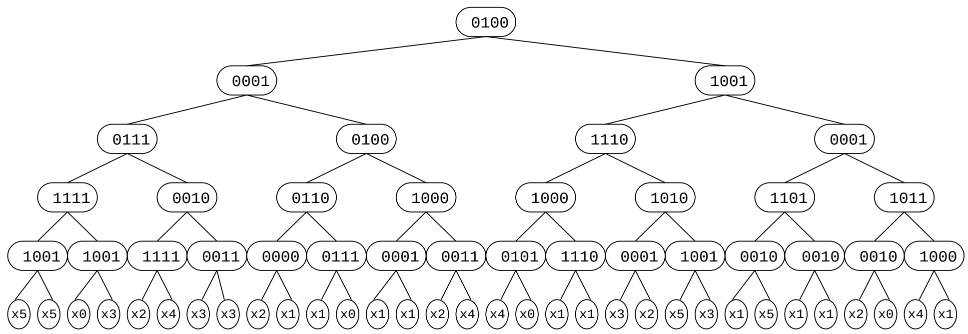

Figure 5.4: A Depth 5 solution to the 6-multiplexer problem using arity-2 functions

block in the function set means considering the set of arity-3 functions, and this is the subject of the next section.

5.5 Experiments with the Arity-3 Function Set

5.5.1 6-Multiplexer

We tested the sub-symbolic and symbolic representations on the 6-multiplexer using the arity-3 function set and $2 \leq D \leq 6$. The results are given in Tables 5.8 to 5.10. The symbolic representation solved the problem on only 2 occasions when the smaller population size was used, and the efforts are not given.

Despite the larger search space, both representations showed considerable improvements in performance relative to when the arity-2 function set was used. The symbolic representation did solve the problem with the smaller population size, but generally seems to require larger populations to do so regularly. In contrast, the sub-symbolic representation regularly found solutions at a variety of depths with both population sizes. Solutions were found more often with the larger population size although it required more effort on average. The effort required by the sub-symbolic representation also compares well with Koza's estimate of 243,000 for standard GP with the function set {`IF, NOT, OR, AND`}, outperforming it by about an order of magnitude on the depth = 5, population size = 50 condition.

Table 5.8: Minimum efforts and number of successes (out of 50) for the 6-multiplexer problem with the full arity-3 function set, sub-symbolic representation and population size = 50.

Depth	Effort
2	216,600 (2)
3	46,150 (16)
4	29,050 (27)
5	72,000 (14)
6	—

Table 5.9: Minimum efforts and number of successes for the 6-multiplexer problem with the sub-symbolic representation and population size = 200.

Depth	Effort
2	684,000 (1)
3	72,000 (31)
4	56,400 (43)
5	39,200 (46)
6	62,400 (39)

The arity-3 functions and the explosive tree size made analysing even moderately deep trees difficult. However, an elegant depth 2 solution to the 6-multiplexer is shown in Figure 5.5.

The root node is an `IF` function, returning the negation of the third argument if its first argument is true and the negation of the second otherwise. The leftmost function on the second level is an arity-2 function that ignores its second argument, and in fact evaluates to {`EQ X0 X1`}. It therefore returns `true` if both the address bits are identical, `false` if they are different. The middle function on the second level is equivalent to `IF X1 THEN NOT X5 ELSE NOT X2`, i.e. it returns `NOT X5` when the address bits are both 1 and `NOT X2` when they are both 0. Similarly, the rightmost

Table 5.10: Minimum efforts and number of successes for the 6-multiplexer problem with the symbolic representation and population size = 200.

Depth	Effort
2	1,732,800 (1)
3	411,600 (10)
4	464,400 (10)
5	—
6	—

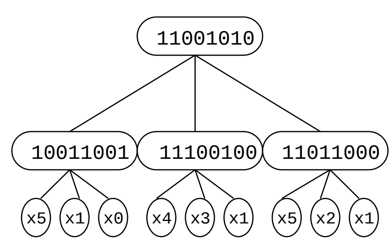

Figure 5.5: A Depth 2 solution to the 6-multiplexer problem using arity-3 functions

function returns `NOT X4` when the first address bit is 1 and the second 0, and `NOT X3` when the first is 0 and the second 1. The negations are all inverted by the root node, giving one solution to the 6 Multiplexer. A simplified version of the tree can be seen in Figure 5.6.

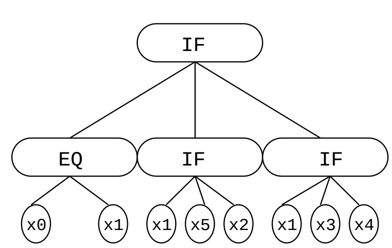

Figure 5.6: Simplified version of the tree shown in Figure 5.5.

The performance of the sub-symbolic representation, whilst an improvement on standard GP, remains significantly inferior to that on the parity problems. The most obvious reason for this is that the increase in the size of the search space is not matched by an increase in the number of solutions, i.e. the ratio of solutions to total number of programs decreases on this problem with this function set. We discuss this issue in Section 5.7

5.5.2 Even-n-Parity Problem

We repeated the parity experiments reported in Section 5.4 using the arity-3 function set for tree depths of 2 to 4. The results are shown in Tables 5.11 to 5.14.

Table 5.11: Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem with a full arity-3 function set and population size = 50.

	n			
Tree Depth	5	6	7	8
2	2,100 (50)	3,000 (49)	17,600 (28)	64,400 (7)
3	12,600 (38)	35,000 (23)	73,150 (14)	189,200 (5)
4	649,800 (1)	542,400 (2)	—	—

Table 5.12: Minimum efforts required by the symbolic representation to solve the even-n-parity problem with a full arity-3 function set and population size = 50.

	n			
Tree Depth	5	6	7	8
2	74,250 (14)	271,200 (3)	— (0)	— (0)
3	581,400 (1)	— (0)	— (0)	— (0)
4	— (0)	— (0)	— (0)	— (0)

It is clear that performance is still better on the parity problems than it is on the multiplexer when the larger function set is used. It is generally inferior to that of similar-sized trees when arity-2 functions are used, but this is not surprising given

Table 5.13: Minimum efforts required by the sub-symbolic representation to solve the even-n-parity problem for with population size = 200.

	n			
Tree Depth	5	6	7	8
2	2,800 (50)	2,600 (50)	5,200 (49)	15,600 (29)
3	5,000 (50)	9,200 (50)	8,400 (50)	7,800 (50)
4	11,000 (50)	20,400 (48)	16,800 (49)	22,400 (49)

Table 5.14: Minimum efforts required by the symbolic representation to solve the even-n-parity problem with population size = 200.

	n			
Tree Depth	5	6	7	8
2	14,800 (50)	30,000 (48)	31,800 (47)	77,000 (29)
3	28,000 (47)	44,000 (32)	58,800 (40)	67,200 (39)
4	52,800 (43)	68,800 (39)	111,600 (28)	237,600 (17)

the large number of functions in the set. Performance in both conditions is markedly superior when larger population sizes are used, presumably for the same reason.

As predicted, the sub-symbolic representation outperforms the symbolic to a much greater extent than it did on the arity-2 function set – typically by a factor of about 5, although greater differences were observed. The explanation proposed – that the sub-symbolic representation permits a significantly more local search to be performed in the later stages of the run – will be discussed in Section 5.7.

A depth 2 solution to the even-6-parity problem is shown in Figure 5.7.

This tree contains two instances of the node `10010110 (even-3-parity)` and one instance of `01101001 (odd-3-parity)`. The arity of the remaining node is reduced to 2 by the duplicate argument and it evaluates to {`EQ X4 X5`}. The simplified tree is shown in Figure 5.8.

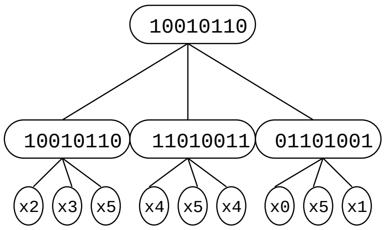

Figure 5.7: Depth 2 solution to the even-6-parity problem using arity-3 functions.

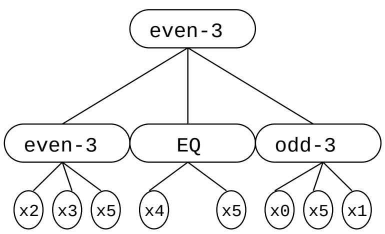

Figure 5.8: Simplified version of the tree shown in Figure 5.7.

Larger trees, as with arity-2 functions, make wider use of the functions in the set. A depth 3 solution is shown in 5.9. The root node of this tree is equivalent to {`EQ ARG1 ARG2`}, i.e. the third sub-tree is ignored. Considering the first argument, we see that its second sub-tree is the `always-OFF` function, which effectively reduces its arity by 2 such that it evaluates to `NOT ARG1`. This input evaluates to {`XOR X2 X1`} which, when negated, means that the root node's first argument is simply {`EQ X2 X1`}. The root node's second argument is the `even-3-parity` function. Its first argument evaluates, via argument duplication, to {`XOR X4 X3`}, its second is simply `X0`, and its third evaluates, again by argument duplication, to `X5`. The simplified tree is given in Figure 5.10

We can see then that the enlarged function set contains sufficient building blocks to enable the system to solve the parity problem. In addition to the even-3 and odd-3 parity functions there are six functions that ignore one of their arguments and evaluate to either `XOR` or `EQ`. Numerous arity-3 functions can also be adapted to serve the same purpose if one of the expressions returns the same value for all fitness cases, or one of the arguments is duplicated. Thus, whilst we observe a greater diversity of function nodes in arity-3 solutions than is the case with arity-2, the functionality of

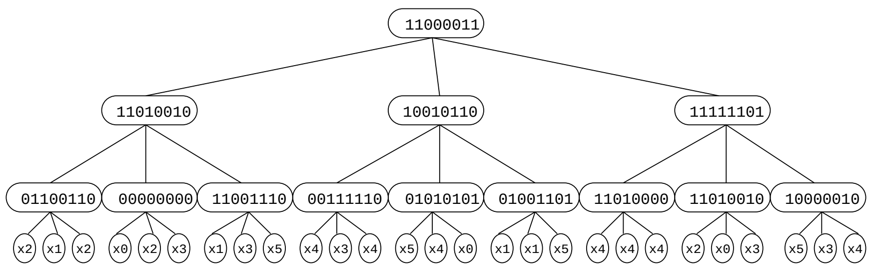

Figure 5.9: Depth 3 solution to the even-6-parity problem using arity-3 functions.

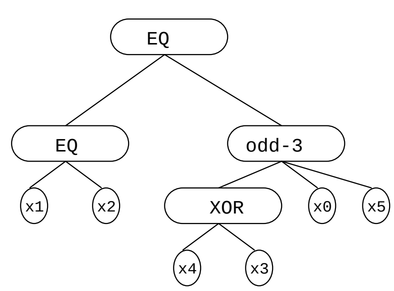

Figure 5.10: Simplified version of the tree shown in Figure 5.9.

the nodes is strikingly similar.

5.6 Summary of Results

The main findings of the experiments reported in this chapter can be summarised as follows:

- Our algorithm, which uses fixed-structure syntax trees and GA search operators, significantly outperformed standard GP on the even-n-parity problem for $5 \leq n \leq 8$. This was true of both the symbolic and sub-symbolic representation, and was observed with both the arity-2 and arity-3 function sets. The sub-symbolic representation with the arity-2 function set did particularly well, typically outperforming standard GP by between 3 and 6 orders of magnitude. In addition, the sub-symbolic representation with the arity-3 function set outperformed standard GP on the 6-multiplexer by about an order of magnitude.

- Performance with the sub-symbolic representation was almost always superior to that with the symbolic representation on both problems and with both function sets. The amount by which the sub-symbolic representation outperformed the symbolic was more pronounced with the arity-3 function set than with the arity-2.

- Both representations performed better on the even-n-parity problems than on the 6-multiplexer, in spite of observations in the literature indicating that the former is a more difficult problem for GP to solve. Performance on the 6-multiplexer was very poor with the arity-2 function set, which lacks the building blocks necessary to solve the problem.

- On the even-n-parity functions, the effort required to solve the problem increased linearly with n. This is in contrast to the estimates given by Koza [58], which suggest that the effort required by standard GP increases exponentially with n.

- Performance was sensitive to the depth of the trees. It is apparent that the ratio of the number of solutions to the number of possible trees is not constant for all tree depths. In addition, we found that on the even-8-parity problem, solutions could be very difficult to find with very small trees.

Many of these points have already been discussed in the previous sections. One point that has received very little attention, however, is the fact that performance was significantly better on the even-n-parity problems than on the 6-multiplexer in spite of the fact that the multiplexer is considered an easier problem to solve. We address this issue in the next section.

5.7 Discussion

Standard GP populations search a potentially huge program space, yet attempt to do so with local search operators which, in effect, make them behave as parallel hill-climbers [101, 107]. Of course, a great many problems can be solved with exactly this approach and standard GP has been applied to a diverse range of scenarios with good effect. Unfortunately, there are also many problems where some other strategy seems to be required and when it is, standard GP is unable to perform the sort of search necessary. The even-n-parity function is an example of such a problem, and becomes trivially easy when global search operators are used. The 6-multiplexer, on the other hand, seems to benefit from a more local search.

It is important to note that whilst we did not vary the per-node (or per-bit, in the case of the sub-symbolic representation) probability of crossover in these studies, it is nevertheless a tuneable parameter. Moreover, the selection of a value other than that used here (0.5) would result in offspring that inherited most of their material from one parent. Uniform crossover acting on both representations can therefore be easily adjusted to perform local search if the problem seems to require it.

One question that arises from these results is why performance should be so much better on the parity problems than on the multiplexer when the opposite has generally been observed for standard GP. The explanation previously proposed for the difference in standard GP's performance on the two tasks is that the fitness landscape of the multiplexer is unimodal [11]. Juels and Wattenberg give a demonstration of this in [52]. However, their argument assumes the existence of a mutation operator that can flip a single incorrect bit in the output vector. In general, this will require the insertion of a specific sub-tree at a specific location – an unlikely occurrence with standard mutation and a potentially impossible one with standard crossover, since the necessary sub-tree may not exist in the population.

In fact, from the point of view of these GA operators, the multiplexer's landscape is not unimodal and contains attractors from which it can be impossible to escape. We can see this by considering the case of very small trees which do not contain an instance of every terminal necessary to solve the problem. It is possible early in the run for such trees to outperform trees which do include all the terminals, with the result that the population converges on the former and one or more terminals are lost. Since terminal mutution is not used, these terminals cannot be reintroduced and no solution can be found. That this proved to be a significant problem for both representations operating on small trees is reflected in the poor performance. Standard GP almost never gets trapped in these attractors, however, because the pressure towards larger trees ensures that vital terminals are not lost from the population. It is therefore able to move into the region of the space where a local search will frequently yield a solution.

Larger trees mean a much larger search space, particularly when the arity-3 function set is used (as was shown to be necessary for the multiplexer). The use of the arity-3 function set with the symbolic representation also has a serious impact on the ability of the search operators to perform local search. If we assume from standard GP's good performance on the problem that solving the multiplexer requires a predominantly local search, we might predict that performance with the symbolic representation will be comparatively poor. This is exactly what we found – the symbolic representation generally required a larger population to find a solution at all, and even then was only able to do so within a fairly narrow margin of tree depths.

In contrast, the sub-symbolic representation consistently solved the multiplexer with both population sizes and with a wide variety of tree depths. Providing the tree depth is sufficiently large to enable the population to avoid the deceptive attractors, a very local search of the fit region of the search space can be performed and solutions frequently found.

Deceptive attractors of the kind seen in the multiplexer do not exist in the parity functions. This is because any tree that does not consider each of the n input variables must score exactly half marks. Populations could therefore only lose terminals by genetic drift. In practice this happened rarely, and only under specific conditions (e.g. for small trees when $n = 8$). Generally, an above-average individual was found within a couple of generations and the population rapidly converged around individuals in which every terminal was present. The operators were then free to modify the functions so that a solution could be found.

The results indicate that solutions to the parity functions are clustered within relatively tiny areas of the total search space. Furthermore, at a fine resolution (i.e. that "seen" by local search operators), the fitness information guiding a population towards these regions is weak and is characterised by extensive plateaus of zero gradient. An obvious example of such a plateau is the sub-space occupied by programs in which one or more of the input variables is ignored. The gradient can only be followed if global operators are used, and we can see from the results that these operators can follow the gradient very well. Once the fit region of the space is located, a local search of the vicinity generally yields a solution very quickly. The sub-symbolic operators are able to perform local search regardless of the arity of the function set, and performed well on both arities. The inability of the symbolic representation operators to perform a local search of the fit regions when the arity-3 function set was used explains the impairment of their performance.

5.8 Conclusion

The results of this preliminary investigation give us good grounds for believing that GP will profit from the addition of a global search operator. The search spaces of the parity function in particular appear to demand such an operator. In these experiments, however, the representation has been considerably constrained, since only full trees of a fixed size were considered. We have seen that performance can be very sensitive to tree depth and that the optimum is dependent on a combination of the problem, the arity of the function set and the representation used. It is conceivable that performance will also be affected by tree structure, although an extensive examination within the current framework is clearly infeasible. In order to do this, we need operators that can work on variable structure trees. Such operators are desirable for two reasons: firstly they remove the task of finding the optimal depth from the user and secondly because they give GP the freedom to discover optimal tree sizes and structures for itself. We address this issue in the next chapter.

Chapter 6

Global GP Operators and Boolean Functions

6.1 Introduction

In the previous chapter we examined the hypothesis that one reason for GP's poor performance on problems such as even-n-parity is that its operators perform local search. To do this we adapted the representation of a standard GA so that the bit-string encoded a fixed-length full parse tree of arbitrary depth. One of the conclusions drawn was that performance is sensitive to the depth of the tree – deeper trees allow the system more flexibility in constructing the primitives it requires to solve the problem, but at the expense of a larger search space. It is also possible that tree structure (i.e. whether trees tend to be sparse or bushy) may affect performance, although this was not investigated in the previous chapter. At least for tree size with fixed structure, we have seen that the optimum setting depends on the problem, the arity of the functions used and the representation, and that identifying this for a given problem is time-consuming. In this chapter, the task of identifying good regions of the size and structure spaces in which to concentrate search is passed to the GP system, and to do this we must use different search operators and adapt the representations so that they may work upon them.

How should we do this? Much of the trouble with standard GP crossover stems from the fact that it is local in the sense that relatively small amounts of information are exchanged between parents, and it is biased in that crossover locations tend to be selected towards the leaves. Our search operators should be directed at overcoming these problems. These considerations recently led Poli and Langdon [107] to develop a number of GA-inspired operators for GP, including one-point crossover and GP uniform crossover (GP-UX). We used GA uniform crossover in the previous chapter and its GP counterpart will form the basis of the studies described here.

The remainder of this chapter is structured as follows. The next section describes GP uniform crossover and a GA-inspired mutation operator called GP point mutation (GP-PM). Adapting the sub-symbolic representation of the GA system for these operators is quite straightforward and the development of a novel pairing of operator and representation called *GP smooth uniform crossover* (GP-SUX) is described in Section 6.2.1. In Section 6.3 we describe our experimental set-up, and in Sections 6.4 and 6.5 we test GP-UX and GP-SUX on the parity and multiplexer problems, comparing performance with both the GA system of the last chapter and standard GP using the same function sets. Our experiments produce a number of interesting results and possible causes of these are explored by a number of experiments reported in Sections 6.4.3 and 6.4.4. We conclude with a summary and a discussion of the results.

6.2 GP Uniform Crossover and Point Mutation

GP uniform crossover [107] is a search operator inspired by the GA operator of the same name. We saw in the last chapter that GA uniform crossover constructs offspring on a bitwise basis, exchanging each allele between parents with a 50% probability. The information at each gene location is thus equally likely to have come from either parent and each parent donates half of its genetic material on average. The whole operation relies on the fact that both parents are of the same structure and the same length. No such assumptions can be made in GP since the parent trees will almost always contain unequal numbers of nodes and be structurally dissimilar.

GP uniform crossover begins with the observation that many parse trees are at least partially similar structurally, particularly when the function set consists of functions of the same arity. This means that if we start at the root node and work our way down, we can frequently go some way before finding nodes of differing arity at the same location. Furthermore we can swap every node up to this point with its counterpart without altering the structure of either tree. Working down from the root node, we can define two regions of a pair of trees as follows. Any node in one tree having a corresponding node at the same location in the other is said to be located within the *common region*. Those pairs of nodes within the common region that have the same arity are referred to as *interior*. We should note that the common region necessarily subsumes the interior. The interior and common region of two example trees is shown in Figure 6.1.

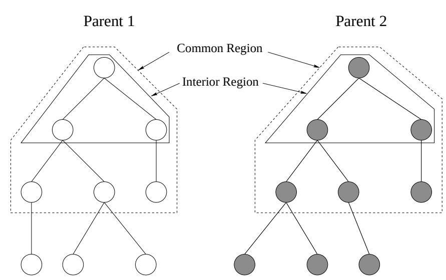

Figure 6.1: Two parental parse trees prior to GP-UX.

The uniform crossover algorithm is then as follows. Once the interior nodes have been identified, the parent trees are both copied. Interior nodes are selected for crossover with some probability p_c. Crossover involves exchanging the selected nodes between the trees, with those nodes not selected remaining unaffected. Non-interior nodes within the common region can also be crossed, but in this case the nodes and their sub-trees are swapped in their entirety. As in GA uniform crossover, the value of p_c is generally set to 0.5, resulting in an exchange of 50% of the interior nodes. The result of uniform crossover applied to the trees in Figure 6.1 is shown in Figure 6.2.

With this in mind, we can make a number of observations about the properties of GP-UX:

- In contrast to standard GP crossover, program depth cannot increase beyond that of the deepest at initialisation. We do not therefore need to impose an upper limit on the size of tree that can be generated by the crossover operator.

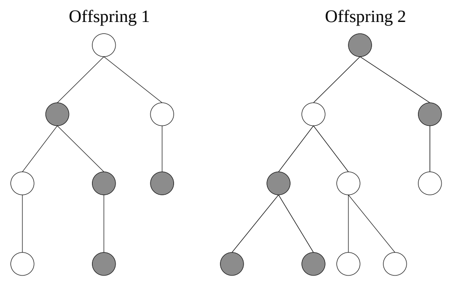

Figure 6.2: Offspring trees after GP-UX.

- Since, under selection, the population will begin to converge on trees of a certain structure, the proportion of nodes within two randomly-selected trees that fall within the interior region will increase as the run progresses.
- Uniform crossover does not introduce new information into the population, and there is no way of obtaining it from other parts of the tree. This can lead to premature lexical convergence, whereby a sub-optimal gene becomes fixed at a given location.

The problem of premature lexical convergence is frequently observed in GAs, and is generally overcome by including a mutation operator to maintain diversity in the population. The operator used by Poli and Langdon, and the one we use here – GP point mutation (GP-PM) – is also inspired by its GA counterpart. GP-PM substitutes a single function node with a randomly-selected replacement of the same arity. As in GA mutation, the number of mutations performed per individual is a function of the program size and a user-defined mutation rate. Since in GP program lengths vary, this means that larger programs will, on average, be perturbed to a greater degree than smaller ones. Since such perturbations are generally detrimental to the fitness of a highly-evolved program, this will generate an emergent parsimony pressure [105].

GP-UX and GP-PM are clearly quite similar to the GA operators used in the last chapter with the standard representation. In fact, the only difference lies in the exchange of sub-trees outside the interior region which allows structurally dissimilar trees to be considered. Adapting the operators to work with the sub-symbolic representation is, not surprisingly, easy and the mechanism is described in the next section.

6.2.1 GP Smooth Uniform Crossover and Point Mutation

We saw in the previous chapter that affording the operators the ability to exchange information at the bit level, rather than the function level, allows GP to perform very local as well as global search, and that this almost always led to improvements in performance. This section describes the incorporation of the sub-symbolic representation into the variable-length tree system to produce two new search operators which we call *GP smooth uniform crossover* (GP-SUX) and *GP smooth point mutation* (GP-SPM).

For Boolean function nodes, as before, the first step is to represent the function nodes as truth tables of 2^n bits, where n is the arity of the function. GP-SUX then proceeds as for GP-UX until a pair of interior nodes is selected for exchange. Now however each bit of the node is exchanged with 50% probability, as opposed to the whole node. The operation is illustrated in Figure 6.3. In the case of non-interior nodes falling within the common region, the nodes will either be functions of differing arity or one of the nodes will be a terminal. In both cases, it is meaningless to try to exchange bits between them and the sub-trees are simply swapped as with GP-UX.

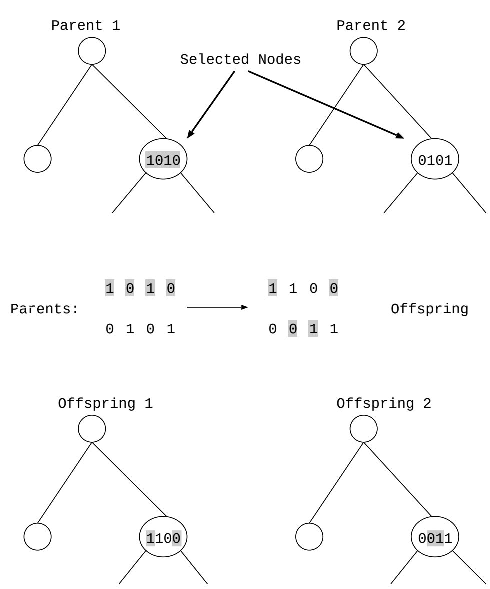

Figure 6.3: Uniform crossover on the sub-symbolic representation.

Point mutation operating on Boolean function nodes is also very simple. When a function is selected for mutation, a single randomly-selected bit is flipped. Any function node in the tree is eligible for mutation. Terminals are not mutated.

6.3 Experimental Studies

A number of motivations guided the work described in this and the following chapter. Firstly, we wished to perform an in-depth empirical study of the GP uniform crossover and point mutation operators [107]. With the sub-symbolic representation, we are able to perform global search in GP whilst retaining the capacity to perform very local search, and we wanted to see what benefits, if any, this conferred upon the system. There was also the desire to demonstrate that these operators allow GP to search the space of smaller programs efficiently and thus find parsimonious solutions, rather than driving search into the realms of ever-larger programs that exhibit similar behaviour to their smaller ancestors. Whilst we noted earlier that the nature of GP-UX and GP-PM are such that the depth of the largest tree is constrained at initialisation, it is hoped that the solutions found will be considerably smaller than this upper limit.

6.3.1 Experimental Set-up

We compared GP using the standard crossover and point mutation operators, GP with GP-UX and GP-PM and GP using GP-SUX and GP-SPM on the even-n-parity and 6-multiplexer problems. The parameters common to all the experiments are given in Table 6.1.

Table 6.1: Parameters used in all the experiments reported in this chapter.

Number of runs	50
Population sizes	50, 200
Maximum generations	100
Tournament size	7
Initialisation method	ramped

We used two population sizes, both quite small by GP standards. Pilot studies indicated that, as with the GA studies reported in the previous chapter, GP with uniform crossover was quite capable of finding solutions with a population of just 50 individuals, but we also used a larger population to give standard GP a fighting chance. The remaining parameters are summarised in Table 6.2 for the standard operators, and Table 6.3 for the uniform crossover and point mutation operators.

Table 6.2: Parameters used in the standard GP runs reported in this chapter.

Max. initial tree depth	6
Max. tree depth	15
Crossover probability	0.9
Mutation probability	0.01

Table 6.3: Parameters used in the GP-UX and GP-SUX runs reported in this chapter.

Max. initial tree depth	3-10
Crossover probability	0.3
Mutation probability	0.1

The meaning of many of these parameters differs subtly depending on the condition and some explanation is necessary. The maximum initial tree depth is given by the depth of the deepest terminal node, with the root node located at level 0. A depth 4 full tree consisting of arity-2 functions will therefore contain 16 terminal and 15 function nodes. The results reported in the previous chapter give us reason to believe that performance may be sensitive to the maximum possible tree depth and for this reason this parameter was also varied in these studies. The crossover probability parameter gives the proportion of the new population generated by crossover, the remainder being produced by cloning. The values for both operators differ significantly from those normally used in standard GP studies. We selected them on the basis of pilot studies which indicated that parameters of about these values were necessary to obtain reasonable performance from the system. The operator parameters in the standard crossover condition are simply those suggested by Koza [58]. In the standard condition, the mutation rate is the proportion of the population that underwent a single standard GP mutation at each generation. In the case of point mutation, however, *every* individual underwent mutation (with an elitist strategy) and the parameter gives the proportion of nodes mutated.

Two function sets were considered, these being the set of all 16 arity-2 functions and the set of all 256 arity-3 functions. The sets were not combined, partly because this would have had a significant detrimental effect on the number of nodes within the interior region in the uniform crossover and partly because the arity-2 functions are included in the arity-3 function set anyway.

6.4 Experiments with the Arity-2 Function Set

6.4.1 Even-n-Parity Problem

We tested GP-UX with GP-PM, GP-SUX with GP-SPM and standard GP on the even-n-parity problem with $5 \leq n \leq 8$. We used a function set comprising all 16 arity-2 Boolean functions and 2 population sizes of 50 and 200 individuals. In the GP-UX and GP-SUX conditions we also varied the maximum depth of trees at initialisation. The lower limit on this, as with the GA studies reported in the previous chapter, was 3 (i.e. 7 function nodes and 8 terminals in a full tree of depth 3 – the minimum required to solve the problem with $n = 8$). The upper limit of 10 was slightly higher than in the GA studies since it seemed likely that the population would converge on smaller trees.

The results are for standard GP are given in Table 6.4. The results for GP-UX with GP-PM and GP-SUX with GP-SPM are given in Tables 6.5 to 6.8.

Standard GP with a population of 50 individuals performed predictably poorly on the parity problems, solving the even-5-parity function just once and failing to find a solution to any of the higher-order versions. However, there was a marked improvement when the larger population was used. The estimate of 112,000 is over an order of magnitude better than Koza's value of 6,528,000 [59]. This was presumably due to the inclusion of the `EQ` and `XOR` building blocks in the function set [70], although the small population size also contributed by minimising the effort expended evaluating

Table 6.4: Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using a full arity-2 function set. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given.

	n							
Pop. Size	5		6		7		8	
	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
50	296,400 (1)	275	no solution	—	no solution	—	no solution	—
200	112,000 (10)	126.6	547,200 (1)	73	no solution	—	no solution	—

individuals on failed runs[1]. Standard GP also solved the even-6-parity problem once with a population of 200 individuals. This gave an effort of 547,200 – an improvement of 2 orders of magnitude on Koza's estimate. No solutions were found to any of the higer-order problems.

GP-UX solved all versions of the problem with both population sizes. In both cases, there was a slight increase in optimal maximum tree size at initialisation with increasing n. With smaller trees, performance was slightly better with the larger population whereas the reverse held for tree depths of 8 and above. Again, this appears to be due to the minimisation of ineffective evaluations in the smaller population as solutions became more difficult to find. In the larger population there was generally a steady increase in the average complexity of correct solutions, at least up to initial tree depths of 8. For depths of 9 and 10, however, the effect was much weaker. The trend towards larger solutions with increasing n was not particularly strong for any of the depths in the smaller population condition. Smaller populations with large initial tree depths mean that the space of small trees is relatively poorly sampled. The population may therefore converge prematurely on large trees where solutions were more difficult to locate, particularly for higher values of n. For initial depths of 10 in both conditions we find both that the complexity of solutions and the number of successful runs were lower for $n = 8$ than $n = 6$ implying that GP-UX was more likely to solve the even-8-parity function on the (comparatively few) occasions when

[1]Pilot studies using larger populations confirmed this. Whilst standard GP did solve the parity problems more frequently, the minimum effort required nevertheless increased.

Table 6.5: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	18,400 (26)	12.8	136,800 (4)	13.5	501,600 (1)	15.0	— (0)	—
4	6,900 (46)	18.6	16,200 (31)	20.3	47,500 (15)	25.3	157,500 (5)	26.2
5	6,000 (48)	38.9	10,800 (40)	37.7	21,600 (30)	38.7	64,800 (13)	37.6
6	6,150 (48)	63.4	12,200 (44)	74.05	21,250 (34)	52.8	61,750 (15)	54.6
7	7,600 (47)	96.4	15,200 (37)	75.1	42,750 (19)	78.2	63,650 (15)	89.4
8	7,000 (48)	137.8	14,200 (38)	121.4	32,400 (24)	110.5	55,100 (14)	90.1
9	9,750 (44)	197.8	19,800 (28)	170.9	55,200 (17)	212.6	75,650 (12)	137.7
10	11,600 (37)	257.1	37,200 (23)	333.1	63,750 (14)	189.3	173,800 (5)	174.6

it converged on a smaller tree size.

GP-SUX outperformed GP-UX on almost every condition, and generally did so by a factor of between 2 and 4. The largest differences in performance were observed with deeper trees, suggesting that GP-SUX is far more efficient at searching large spaces than GP-UX. As with GP-UX, the optimal maximum tree depth at initialisation increased slightly with n, although the effect was much less pronounced and larger trees seem occasionally to have been preferred. In fact, the average complexity of correct solutions found by GP-SUX was generally higher than that of GP-UX. Whether this is because the sub-symbolic representation allows a more efficient search of larger spaces, or because the GP-SPM operator perturbs solutions less and therefore exerts less of a counter-pressure against the attraction towards larger solutions, or that it is a combination of the two is unclear.

Comparing GP-UX and GP-SUX with their GA-based equivalents we see that the latter outperform the former with smaller tree sizes, and that the former outperform the latter with larger trees. The ability of the GP-based populations to converge

Table 6.6: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	11,400 (49)	13.1	81,600 (26)	12.9	675,000 (4)	14.0	1,094,400 (1)	15.0
4	4,200 (50)	19.4	15,600 (48)	23.5	56,800 (37)	23.2	136,800 (19)	25.1
5	5,200 (50)	30.1	7,200 (50)	40.0	18,400 (49)	41.5	52,000 (40)	42.8
6	5,600 (50)	36.0	9,400 (50)	59.6	14,000 (50)	64.0	30,600 (47)	62.5
7	5,600 (50)	44.7	13,800 (50)	77	38,400 (42)	87.6	52,200 (42)	95
8	9,600 (50)	76.6	21,600 (48)	114.9	52,800 (37)	135.5	144,000 (21)	155.2
9	11,600 (50)	109.8	25,800 (46)	141.3	70,400 (36)	185.1	136,000 (22)	133.2
10	13,800 (48)	90.54	32,400 (42)	225.2	202,400 (13)	171.3	183,600 (17)	215.5

on smaller trees than the maximum possible given the initial depth is undoubtedly responsible for the superiority of GP over GA with large trees. When tree size was close to or below the optimum size, however, the added task of locating good tree sizes and structures seems to have led to a deterioration in GP's performance relative to that of the GA system. One illustration can be seen for $n \geq 7$ with tree depth $= 3$. Whilst these certainly proved difficult for the GA system to tackle, GP found solutions rarely or not at all. Populations in these conditions frequently converged on trees too small to solve the problem. As with the GA studies, this could only have happened by genetic drift since any tree scoring above half marks would necessarily have had to have included all the terminals and therefore sufficient function nodes to solve the problem. That such individuals were rarely found in the first few generations seems to confirm that these conditions offer very little gradient information for a population to follow. Whilst the optimal depth for the GP system was higher than that of the GA, the average complexities of correct solutions in these GP conditions is generally in close agreement with those predicted by the GA results.

Table 6.7: Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity 2 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	9,200(27)	12.38	45,200 (6)	14.33	— (0)	—	— (0)	—
4	3,150 (47)	19.93	15,400 (22)	23.2	24,650 (25)	15.0	86,800 (5)	23.4
5	2,600 (47)	38.96	6,800 (42)	39.8	12,800 (31)	40.2	22,750 (20)	42.8
6	3,900 (45)	59.13	4,200 (47)	58.5	9,750 (40)	65.3	16,200 (35)	70.8
7	1,800 (50)	107.64	4,950 (49)	110.7	7,800 (46)	102.7	11,400 (40)	111.9
8	3,600(49)	185.12	6,250 (48)	206.7	11,000 (44)	191.6	21,600 (33)	211.1
9	5,400 (47)	291.1	6,200 (42)	200.2	12,000 (39)	308	18,500 (36)	292.6
10	4,500 (48)	323.63	9,300 (38)	322.7	16,800 (31)	279.3	34,800 (24)	415.7

Both GP-UX and GP-SUX considerably outperformed standard GP. On the even-6-parity function, both operators outperformed standard GP with the standard function set by 4 orders of magnitude at best, and standard GP with the full arity-2 function set by about 2 orders of magnitude. Clearly, the even-n-parity functions are not difficult problems for GP to solve if global search operators are used.

We conclude this section with an illustration of sample solutions. A 29 node solution to the even-6-parity function found by GP-SUX is shown in Figure 6.4. There are 3 instances of the `EQ` function and 2 of `XOR` in this tree. A sizeable proportion of the nodes in this tree do little or nothing, however (e.g. the `0010` sub-tree evaluates to `NOT X4`) and the main coding branch is dominated by the building blocks. With the introns removed, the tree in fact contains only `EQ` and `XOR` functions as can be seen in Figure 6.5.

Table 6.8: Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	10,800 (46)	13.5	34,000 (24)	13.8	195,000 (7)	14.43	866,400 (1)	15.0
4	5,600 (50)	21.8	9,600 (50)	24.6	30,000 (39)	22.85	57,600 (26)	26.2
5	3,000 (50)	33.6	6,800 (50)	43.9	7,400 (50)	41.8	21,600 (46)	43.9
6	3,600 (50)	41.6	4,400 (50)	42.2	6,600 (50)	76.5	11,000 (50)	72.6
7	5,600 (50)	65.4	5,000 (50)	78.3	13,200 (50)	125.2	17,000 (49)	103
8	3,800 (50)	71.8	5,600 (50)	112.9	17,200 (49)	146.9	24,000 (48)	224.2
9	4,400 (50)	91.6	5,800 (50)	129.9	16,800 (49)	164.9	33,600 (46)	279.3
10	5,000 (50)	153.62	7,800 (50)	195.4	25,600 (47)	274.1	36,400 (47)	311.2

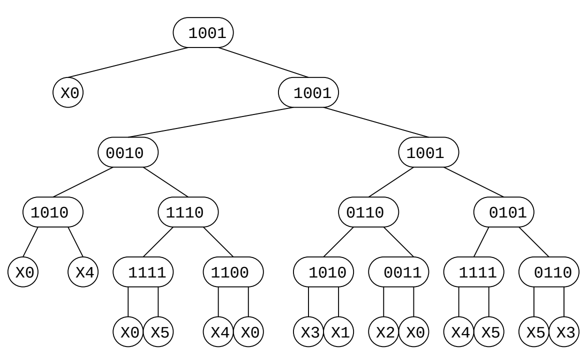

Figure 6.4: 29 node solution to the even-6-parity problem found by GP-SUX on the maximum initial depth = 5 condition.

6.4.2 6-Multiplexer

We tested standard GP, GP-UX with GP-PM and GP-SUX with GP-SPM on the 6-multiplexer using the same parameters and function set as those for the even-n-

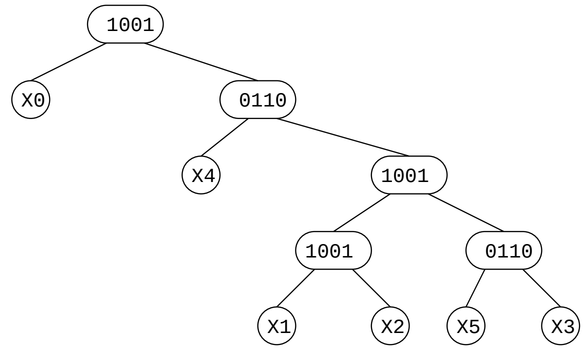

Figure 6.5: Simplified version of the solution shown in Figure 6.4

parity problems. No solutions were found in any of the conditions. The result was not altogether surprising given the poor performance of the GA system on the same function set. The arity-2 function set omits the building blocks required to construct solutions to the multiplexer and GP is therefore forced to construct these at multiple locations throughout the tree. More will be said on this below, but for now it is sufficient to note that the additional task of searching the spaces of program size and structure rendered the problem insoluble even for GP-SUX.

6.4.3 Building Blocks and the Parity Functions

One question that arises from the results of this and the previous chapter is the degree to which the building block primitives influence performance. More specifically, we might ask whether standard GP, GP-UX and GP-SUX are performing well in spite of, rather than because of, the larger function set and whether performance would not be further enhanced with a smaller function set that retained the functions `EQ` and `XOR`. It would also be interesting to see what would happen if these two primitives were omitted from the initial function set, but could be generated during the run by mutation. We could then investigate the extent to which each operator is able to identify and distribute the building blocks throughout the population.

We performed two additional sets of experiments to examine these questions. In the first, the function set was restricted to {`AND, OR, NAND, NOR, EQ, XOR`}. This is a small function set which includes the building block primitves `EQ` and `XOR`. If the previous results were simply due to the inclusion of the building blocks, we should expect further performance enhancements here, since many apparently extraneous functions have been removed. Of course, the nature of the sub-symbolic representation meant that we could not run such a function set without severely constraining the operators, and since this would almost certainly bias the results we did not run these experiments with GP-SUX. The second set of experiments used the full arity-2 function set, but omitted the `EQ` and `XOR` primitives at initialisation. These functions could only be introduced into the population by the mutation operator, although once generated they could be propagated by crossover. We performed these experiments with standard GP, GP-UX and GP-SUX. The remaining parameters were the same as those listed in Section 6.3.1.

The results for standard GP running the reduced function set {`AND, OR, NAND, NOR, EQ, XOR`} are given in Table 6.9.

Table 6.9: Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the reduced function set {`AND, OR, NAND, NOR, EQ, XOR`}. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given.

	n							
Pop. Size	5		6		7		8	
	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
50	15,800 (20)	94.8	55,400 (6)	132.3	145,600 (3)	156.3	441,800 (1)	109.0
200	19,600 (38)	90.5	70,000 (17)	121.6	113,600 (14)	207.0	189,000 (12)	182.3

Comparing these results to those given in Table 6.4, we can see that standard GP does indeed profit from a smaller function set that includes the necessary building block primitives. Performance was markedly superior to that with the full arity-2 function set across all conditions, and the problem was solved at least once in all cases. At this point, the tentative conclusion is that, at least for standard GP, the inclusion of extraneous functions in the function set impairs performance. However, this issue will receive further consideration in Section 6.5.2.

The results for GP-UX running the reduced function set {`AND, OR, NAND, NOR, EQ, XOR`} are given in Tables 6.10 and 6.11.

Table 6.10: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using the reduced function set {`AND, OR, NAND, NOR, EQ, XOR`} and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	9,250 (17)	13.0	13,050 (17)	13.2	64,150 (3)	13.0	58,500 (2)	15.0
4	11,450 (22)	19.4	13,900 (16)	23.6	72,000 (7)	25.0	95,750 (4)	29.5
5	14,600 (24)	31.5	23,850 (23)	37.4	58,500 (9)	39.9	37,150 (10)	37.4
6	13,400 (25)	39.0	29,750 (19)	47.0	53,800 (6)	44.3	154,550 (5)	47.0
7	13,200 (32)	32.8	29,300 (16)	66.1	160,200 (3)	61.7	47,600 (7)	80.7
8	11,100 (24)	38.0	35,000 (11)	62.8	92,000 (6)	49.3	52,400 (6)	56.0
9	13,400 (24)	46.6	30,800 (13)	63.0	76,800 (8)	56.0	58,500 (2)	45.0
10	11,350 (19)	46.1	58,500 (9)	165.7	104,600 (4)	82.5	161,600 (1)	27.0

Comparing these results with those in Tables 6.5 and 6.6, the following points are apparent:

- GP-UX generally performed better with the reduced function set for maximum depths less than 5. However, particularly for intermediate depths of between 5 and 8, GP-UX with the full arity-2 function set generally outperforms the reduced set by about a factor of 2.
- Significantly fewer solutions were found in almost every case, compared with the arity-2 function set.
- The average complexities of correct solutions increased much more slowly with maximum initialisation depth than when the full arity-2 function set was used.

Table 6.11: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using the reduced function set {`AND, OR, NAND, NOR, EQ, XOR`} and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	11,000 (36)	12.9	16,000 (31)	14.4	46,200 (21)	13.7	98,800 (6)	15.0
4	13,200 (36)	19.4	14,000 (35)	20.9	24,400 (29)	24.5	69,200 (14)	23.4
5	10,000 (39)	20.2	19,600 (33)	26.2	64,400 (18)	36.9	88,400 (14)	36.6
6	15,200 (36)	23.8	27,600 (26)	25.9	42,800 (21)	36.8	73,400 (18)	40.0
7	12,400 (39)	21.9	25,800 (31)	36.2	93,200 (17)	61.1	59,800 (21)	60.3
8	14,000 (40)	25.3	26,000 (32)	41.4	46,200 (22)	55.9	122,000 (12)	77.2
9	14,000 (41)	38.5	25,800 (34)	49.2	85,600 (15)	79.5	212,000 (5)	46.2
10	14,000 (36)	24.4	23,000 (32)	44.5	37,200 (22)	40.4	634,800 (2)	110.0

The key result of these experiments is that for depths between 5 and 8, the reduced function set significantly impairs search, even though the building blocks are present. The hypothesis we propose to account for these results is that the reduced function set provides a weaker gradient for global operators such as GP-UX to follow, and may generate more local optima upon which the population can converge. Correct solutions were generally found early in the run or not at all, as can be seen by the relatively low effort compared to the number of solutions found. Furthermore, large solutions were rarely found. The impression is that GP-UX with the reduced function set would "chance" upon a solution, rather than move steadily towards it via incremental improvements. Such a strategy was facilitated by the fact that the smaller function set made it more likely for an incorrect node to be replaced with the correct one during mutation, and that the higher proportion of building blocks in the initial population meant that many trees were already closer to the optimum than they would have been had the full set been used. In those conditions where there was very little gradient information (i.e. small maximum depths), GP-UX was more likely to find a solution with the reduced function set simply because many of the trees it started with contained a large proportion of correct nodes. In contrast, at greater depths the full function set yielded superior performance because the advantages it offered in terms of navigation generally outweighed those arising as a result of the smaller search space.

Standard GP running the full arity-2 function set with `EQ` and `XOR` suppressed at initialisation failed to find a solution on any of the runs. This is not an entirely surprising result since the only way these primitives could be introduced into the population would be via sub-tree mutation which was used with a low probability and which would, in any case, have generated comparatively few instances.

The results for GP-UX with GP-PM running the full arity-2 function set with `EQ` and `XOR` suppressed at initialisation are given in Tables 6.12 and 6.13, and those for GP-SUX with GP-SPM are given in Tables 6.14 and 6.15. It is clear that the omission of the building blocks from the initial population has no discernible impact on performance for either GP-UX or GP-SUX. The mutation operators in both cases generate many instances of the `EQ` and `XOR` primitives, and these are then exchanged by the crossover operators.

6.4.4 The Even-n- Parity Problems and Automatically Defined Functions

In [59], Koza demonstrated significant performance enhancements on the even-n-parity functions with automatically defined functions (see Table 5.1). In these studies he used the function set {`AND, OR, NAND, NOR`}, and the main role of the ADF branches was to evolve building block functions such as `EQ`, `XOR` and `even-3-parity` which could then be used at multiple points throughout the value-returning branch.

ADFs may offer another benefit, however, in that they allow GP to alter the behaviour of function nodes close to the root, without perturbing the sub-trees beneath. This is, of course, a property of the point mutation operators we have been using in conjunction with GP-SUX and GP-UX, and given their marked superiority over standard GP on the even-n-parity problems, it is worthwhile to ask whether such a property will prove beneficial for standard GP.

Table 6.12: Minimum efforts required by GP-UX and GP-PM with a population size of 50 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with `EQ` and `XOR` omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	21,000 (22)	13.2	112,200 (5)	14.4	212,000 (1)	15.0	2,412,000 (1)	15.0
4	6,900 (47)	17.3	21,600 (28)	23.3	41,050 (19)	27.1	231,500 (5)	31.4
5	5,600 (47)	38.9	10,000 (41)	39.6	23,150 (29)	40.7	60,200 (15)	39.4
6	8,000 (45)	56.2	11,200 (42)	76.1	21,000 (37)	57.3	63,450 (19)	49.4
7	7,400 (44)	98.9	14,300 (39)	80.3	25,100 (31)	90.1	57,900 (15)	82.3
8	9,200 (45)	152.1	16,100 (36)	93.4	30,800 (26)	123.8	68,400 (14)	104.4
9	10,100 (40)	211.4	17,200 (33)	150.3	75,100 (10)	189.6	84,200 (12)	139.7
10	12,400 (32)	209.8	38,900 (22)	298.3	59,250 (16)	322.1	112,100 (7)	142.9

Illustrating the possible benefits is best done with an example. Consider a syntax tree comprising a single arity-2 ADF – `ADF0` – and a value returning branch of {`ADF0` {`EQ X0 X1`} `X2`}, where the target is the even-3-parity function. If `ADF0` evaluates to `AND`, the tree will score 6 out of 8 hits. If, however, the ADF node is altered so that it evaluates to `XOR` and the remainder of the branch is left unchanged, the tree evaluates to the target function. This is a very simple example, but the point is hopefully clear. In large trees, making small modifications to nodes close to the root whilst preserving the semantics of lower nodes is virtually impossible with standard crossover or mutation. ADFs, however, allow such changes to be made.

We wanted to investigate whether this property would produce any further performance enhancements over those reported so far in this chapter, so we tested standard GP running ADFs with the full and reduced (i.e. {`AND, OR, NAND, NOR, EQ, XOR`}) arity-2 function sets on the parity functions. Since the `EQ` and `XOR` building block primitives are present in both sets, any gains would presumably be due to the point mutation-like behaviour described above.

Table 6.13: Minimum efforts required by GP-UX and GP-PM with a population size of 200 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with `EQ` and `XOR` omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	9,200 (49)	13.3	92,400 (22)	13.1	412,000 (2)	14.0	1,312,000 (1)	15.0
4	4,000 (50)	17.4	13,200 (49)	21.2	58,600 (38)	24.9	128,600 (20)	27.2
5	5,400 (50)	32.1	7,600 (50)	41.3	16,600 (48)	47.5	38,200 (44)	51.1
6	5,800 (50)	38.8	10,400 (50)	56.4	17,200 (49)	61.8	36,400 (45)	73.5
7	5,600 (50)	49.1	19,200 (49)	80.1	36,000 (44)	82.1	81,200 (30)	88.1
8	10,400 (50)	88.1	23,400 (49)	109.3	60,200 (36)	123.4	108,200 (21)	113.2
9	12,200 (50)	113.3	21,000 (48)	121.7	65,200 (37)	142.8	159,600 (19)	181.1
10	15,800 (46)	115.7	35,200 (40)	217.2	16,800 (25)	202.1	198,200 (14)	227.1

We tested standard GP with one and two arity-2 ADF branches on the even-n-parity problems for $5 \leq n \leq 8$. The remaining parameters are the same as those given in Tables 6.1 and 6.2. The results of these experiments are shown in Tables 6.16 and 6.17.

Comparing the results in Table 6.16 with those in Tables 6.4 and 6.9, we can see that the addition of a single ADF branch does not yield any significant performance enhancements. In fact, the only significant differences – with a population size of 200 for $n = 5$ with the full set, and $n = 8$ with the reduced set – seem to show an impairment with ADFs. The differences are more pronounced with 2 ADF branches, where GP is clearly impaired in all cases.

Examining the correct solutions, we noted that only in a tiny minority of cases did the ADF branches evaluate to `EQ` or `XOR`. Furthermore, when ADFs were used in the value-returning branch, they tended to occur towards the leaves where there impact was minimal. In many cases they were not used at all.

Table 6.14: Minimum efforts required by GP-SUX and GP-SPM with a population size of 50 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with `EQ` and `XOR` omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	10,600(25)	12.4	43,750 (8)	14.1	— (0)	—	— (0)	—
4	3,000 (47)	21.8	17,200 (22)	25.1	24,000 (27)	23.0	62,100 (9)	28.3
5	3,300 (46)	37.2	7,550 (41)	41.9	10,100 (32)	39.9	29,350 (18)	46.1
6	2,750 (49)	55.1	4,000 (49)	60.1	8,250 (42)	70.3	21,450 (31)	62.1
7	3,100 (49)	113.7	4,800 (49)	102.8	7,600 (48)	99.1	13,100 (39)	131.2
8	4,600 (48)	169.1	5,400 (46)	131.7	9,800 (46)	187.6	19,650 (38)	213.2
9	5,000 (47)	275.0	7,150 (40)	211.8	11,800 (40)	279.1	28,300 (36)	313.9
10	8,800 (46)	303.8	12,100 (32)	281.4	21,300 (29)	259.1	33,300 (26)	395.5

The conclusion, at least as far as the parity functions go, is that ADFs are essentially extraneous when the primitive function set already contains the necessary building blocks required to solve the problem. With only one ADF branch, the impact is not significant, but when two (or more, presumably) are used, performance is impaired. The most likely explanation for this is that the presence of the `EQ` and `XOR` building blocks in the function set obviates the need to evolve them as ADFs. With the reduced funtion set, the result is that the the size of the function set is increased, and since the ADFs rarely evaluate to the desired functions, this has an inhibitory effect. With the full set, the ADFs can only evaluate to functions already in the primitive set, so a bias is introduced. Again, since the ADFs are unlikely to exhibit the desired behaviour, the bias leads away from `EQ` and `XOR`. It is therefore clear that the success of GP-SUX and GP-UX on the parity functions is not purely a result of their ability to make small changes to nodes close to the root of the tree without disturbing those below. We shall not consider ADFs further in the remainder of this thesis.

Table 6.15: Minimum efforts required by GP-SUX and GP-SPM with a population size of 200 to solve the even-n-parity problem for various values of n using the full arity-2 function set, but with `EQ` and `XOR` omitted at initialisation. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
3	9,600 (49)	13.1	47,000 (22)	14.0	172,200 (8)	14.5	466,200 (1)	15.0
4	4,800 (50)	23.1	10,200 (50)	27.6	19,200 (42)	29.0	60,200 (25)	22.1
5	4,200 (50)	38.1	5,800 (50)	41.3	8,200 (50)	41.8	19,200 (47)	45.9
6	3,000 (50)	37.3	5,200 (50)	44.5	7,600 (50)	56.3	15,400 (50)	71.9
7	3,400 (50)	60.2	5,200 (50)	68.3	14,000 (50)	98.7	20,600 (48)	112.1
8	4,200 (50)	73.8	6,200 (50)	123.1	15,000 (49)	166.1	37,600 (46)	210.4
9	4,400 (50)	109.1	6,800 (50)	139.2	21,800 (48)	190.3	34,200 (46)	235.4
10	5,400 (50)	123.9	7,000 (50)	162.5	38,200 (46)	188.9	39,200 (44)	240.0

Table 6.16: Minimum efforts required by Standard GP with 1 ADF branch to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the full and reduced arity-2 function sets. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given.

		n							
		5		6		7		8	
Pop	Fn Set	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
50	Full	171,000 (1)	22.0	1,114,000 (1)	172.0	no solution	—	no solution	—
200	Full	113,000 (0)	150.0	705,000 (3)	173.3	361,600 (2)	85.0	no solution	—
50	Red'd	40,500 (11)	155.09	45,200 (12)	229.3	193,800 (3)	581.3	228,000 (2)	187.0
200	Red'd	26,000 (36)	132.5	52,000 (23)	134.7	246,600 (10)	254.4	700,600 (3)	206.7

Table 6.17: Minimum efforts required by Standard GP with 2 ADF branches to solve the even-n-parity problem with population sizes of 50 and 200 individuals using the full and reduced arity-2 function sets. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given.

		n							
		5		6		7		8	
Pop	Fn Set	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
50	Full	467,400 (1)	143.0	no solution	—	no solution	—	no solution	—
200	Full	203,400 (4)	170.0	957,600 (2)	393.0	no solution	—	no solution	—
50	Red'd	57,000 (10)	231.0	86,500 (8)	228.3	395,500 (2)	151.0	no solution	—
200	Red'd	33,800 (29)	168.7	105,400 (17)	208.1	197,400 (11)	225.5	415,600 (2)	288.3

6.5 Experiments with the Arity-3 Function Set

6.5.1 6-Multiplexer

We repeated the experiments described in Section 6.4.2 using the function set of all 256 arity-3 function sets. In [58], Koza reports a minimum effort for standard GP on the 6-Multiplexer of 245,000 based on a population of 500 individuals, using the strongly-biased function set of {`IF, NOT, OR, AND`} and to facilitate comparison we also ran standard GP with this population size. The results for standard GP are given in Table 6.18.

The maximum initial tree depth was varied in the GP-UX and GP-SUX conditions, but with a different range to that for the arity-2 function set. The lower limit was set at 2 (i.e. 9 terminals and 4 function nodes) and the upper limit at 7 (i.e. 729 terminals and 364 function nodes). The remaining parameters were the same as those given above. GP-UX failed to find a solution in any of the smaller population size conditions. The remaining results for GP-UX with GP-PM and GP-SUX with GP-SPM are given in Tables 6.19 to 6.21.

We can see that standard GP with a population of 200 individuals outperforms Koza's

Table 6.18: Minimum efforts required by standard GP to solve the 6-Multiplexer problem using a full arity-3 function set. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

Pop. Size	Effort	Complexity
50	1,117,200 (1)	760
200	100,800 (19)	197.9
500	216,000 (24)	204.6

estimate by a factor of 2. The result with 500 individuals is closer – but still superior – to Koza's estimate. It is conceivable that, in spite of its size, the large function set used here offered GP many additional building blocks not present in the standard set. Moreover, it is clear that standard GP has little difficulty solving the 6-multiplexer with populations smaller than those suggested by Koza, and that the effort expended is considerably reduced as a result.

Table 6.19: Minimum efforts required by GP-UX and GP-PM to solve the 6-Multiplexer problem using a full arity-2 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

Depth	Effort	Complexity
2	no solution found	–
3	no solution found	–
4	2,034,000 (2)	38.5
5	1,401,200 (2)	35.5
6	1,687,200 (2)	49.0
7	2,192,200 (1)	73.2

GP-UX with GP-PM performed poorly on the 6-multiplexer. Solutions were only found with the larger population size, and even then only occasionally. In the previous chapter we saw that the standard representation was only able to find solutions with any regularity when larger populations were used and the trees contained 40 or 121 nodes. This result, and the average complexities of solutions given here, imply that GP-UX performance is very sensitive to tree size and can only find solutions with certainty when the population converges on such moderately-sized trees.

Table 6.20: Minimum efforts required by GP-SUX and GP-SPM to solve the 6-Multiplexer problem using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

Depth	Effort	Complexity
2	no solution	–
3	101,700 (6)	29.0
4	67,200 (12)	54.8
5	94,550 (9)	95.0
6	66,300 (13)	113.8
7	69,600 (14)	93.1

GP-SUX consistently found solutions with both population sizes for all depths above 2. It outperformed standard GP with both population sizes, although not as markedly as it did on the parity problems.

The deterioration in performance exhibited by GP-SUX and GP-UX on the 6-multiplexer relative to that on the parity functions is in line with the findings of the previous chapter. The deceptive nature of the landscape exhibited by the multiplexer in combination with these operators meant that terminals were often lost from the population. Unfortunately, when the size of candidate solutions is allowed to vary, there is an additional pressure towards solutions too small to solve the problem. For example, any tree consisting of a single data terminal (`X2, X3, X4, X5`) will score 40 hits. This presented a significant problem when the initial tree depth was small, since a relatively large proportion of the initial population contained too few nodes to solve the task but were nevertheless able to compete with those that did. In fact, neither

Table 6.21: Minimum efforts required by GP-SUX and GP-SPM to solve the 6-Multiplexer problem using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

Depth	Effort	Complexity
2	no solution	–
3	140,400 (13)	39.1
4	189,000 (12)	58.3
5	88,800 (27)	73.0
6	72,000 (28)	97.1
7	120,000 (24)	69

GP-SUX nor GP-UX solved the problem with the smallest depth of 2 levels. Furthermore, whilst performance increased with initial depth, up to 5 or 6 levels, the average complexity of solutions was always significantly less than the maximum possible. It would appear that these larger-than-optimal initial depths are necessary to allow GP-UX and GP-SUX to overcome the pressure towards small trees early in the run. This issue will be discussed in greater detail below.

6.5.2 Even-n-Parity Problem

We also repeated the experiments described in Section 6.4 using the function set of all 256 arity-3 functions. The results for standard GP are given in Table 6.22. The lower and upper limits in the GP-UX and GP-SUX conditions were set to 2 and 7 respectively, as in section 6.5.1. The results for GP-UX with GP-PM and GP-SUX with GP-SPM are given in Tables 6.23 to 6.26.

The results for standard GP are striking. With a function set of 256 functions and a population size of 200 individuals, standard GP found 5 solutions to the even-6-parity and 2 solutions to the even-7-parity problems. It also succeeded in solving the even-6-

Table 6.22: Minimum efforts required by Standard GP to solve the even-n-parity problem with population sizes of 50 and 200 individuals using a full arity-3 function set. The number of successful runs (out of 50) is given in parentheses. The average complexity of correct solutions is also given.

	n							
Pop. Size	5		6		7		8	
	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
50	168,750 (4)	737.5	228,000 (1)	73	no solution	–	no solution	–
200	78,000 (28)	586.4	319,000 (5)	1004.2	858,800 (2)	253	no solution	–

parity function with a population of 50 individuals. The results for the even-6-parity function are well over 2 orders of magnitude better than the estimate given by Koza (see Table 5.1). If we assume that the parity problem requires an increase in effort of an order of magnitude for each increment of n, the result for the even-7-parity function represents an improvement of 3 orders of magnitude. The issue of function sets has been largely unaddressed in the literature, and it is clear that more work is required in the area.

```
{ 00100110 { 10010011 { 00000000 X5 X5 { 00001011 X5 X4 X4  }  }
{ 00001101 { 10010001 X4 X3 X5  } X2 { 10010001 { 00001011 X5 X4 X3
} X3 X5  }  }{ 10010001 X4 X3 { 00001011 X5 X4 X4  }  }  }
{ 10010110 { 00100110 { 10010001 X3 X3 X3  } X4 X2  }
{ 00001011 X5 X4 X4  } X1  } { 11011000 { 01101010 { 10000011 X3 X3
X0  } X4 X6  } { 01001101 X3 X3 X6  } { 10000011 X0 X3 X0  }  }  }
```

Figure 6.6: 58 node arity-3 solution to the even-7-parity problem found by standard GP

It is possible that standard GP is profitting from the presence of certain universal logic modules (ULMs) in the function set. A universal logic element is defined as:

> A circuit configuration which is capable of realizing all possible 2^{2^n} functions of n independent binary input variables $x_1, ..., x_n, n \geq 2$, de-

pending upon the pattern of its input connections taken from the set $\{0, 1, x_1, \bar{x}_1, ..., x_n, \bar{x}_n\}$. [18]

Chen and Hurst [18] performed an analysis of six basic 3-argument ULMs with a view to identifying an optimal gate for the construction of a wide range of logic functions. Their preferred candidate module was in fact the simple 3-multiplexer function, and we might therefore predict that forms of this function would occur frequently in standard GP solutions.

One of the solutions to the even-7-parity problem found by standard GP is shown in Figure 6.6 (it is, unfortunately, too large to show in tree form). It can be seen that there are no 3-multiplexer nodes in this tree. Instead we see numerous instances of functions such as `10010001` and `10000011`, whose use is at first sight not clear. Analysing the tree shows that the leftmost argument of the root node evaluates to `true`, with the result that the root node returns the `XOR` of its second and third arguments. A simpler sub-tree which does the same thing is the first argument of the even-3-parity node – {`00100110` {`10010001 x3 x3 x3`} `x4 x2`} – which evaluates to `XOR x4 x2`. The rightmost argument of the root node evaluates to `odd-3-parity x3 x0 x6`. The simplified solution is shown in Figure 6.7.

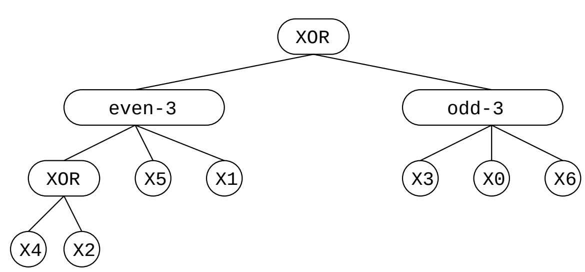

Figure 6.7: Simplified version of the solution shown in Figure 6.6

In this case at least, GP has succeeded in constructing useful building blocks from the (numerous) primitives at its disposal[2], although it does not seem to have used ULMs to do so. Whether this reflects a fundamental difference in the way that standard GP and GP-UX or GP-SUX solve problems such as the parity functions, and whether

[2]This is consistent with Koza's results with ADFs on the parity problems [59], where lower-order parity building blocks were consistently evolved.

ULMs facilitate search, remains unclear. Further work is required to investigate these questions.

Table 6.23: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
2	18,400 (36)	12.6	38,400 (22)	12.2	121,500 (8)	11.8	638,400 (1)	13
3	27,300 (30)	24.9	49,500 (19)	33.2	79,650 (12)	37.3	318,750 (3)	31
4	23,600 (28)	52.4	100,800 (10)	71.8	144,150 (7)	88	172,050 (6)	65
5	45,600 (17)	68.3	48,000 (19)	78.4	410,400 (2)	38.5	— (0)	—
6	37,500 (21)	53.3	175,150 (6)	48.0	285,000 (3)	77	729,600 (1)	121
7	42,000 (21)	92.8	91,350 (10)	142.9	763,000 (1)	175	542,000 (2)	73

Table 6.24: Minimum efforts required by GP-UX and GP-PM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
Tree Depth	5		6		7		8	
	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
2	25,200 (48)	12.4	22,400 (49)	12.7	31,600 (48)	12.5	132,600 (19)	13
3	22,800 (47)	23.5	49,200 (44)	32.2	59,400 (40)	31.5	81,600 (34)	34.1
4	19,800 (47)	32.6	48,000 (41)	42.6	75,200 (36)	51.7	94,000 (32)	61.1
5	32,000 (48)	48.3	49,200 (41)	50.4	90,000 (35)	82.3	118,800 (27)	110.8
6	34,400 (45)	40	58,000 (39)	64.4	107,800 (29)	136.2	174,200 (18)	142.8
7	32,500 (46)	54.1	65,000 (37)	106.2	152,000 (23)	118.7	218,400 (17)	131.6

GP-UX, whilst still able to solve the parity problem with the arity-3 function set, was clearly impaired relative to when arity-2 functions are used. We saw that with the GA system, the search space rapidly became too large for the operators to navigate efficiently, and much the same seems to hold for GP-UX with GP-PM. To reiterate, a single application of the GP-PM operator flips, on average, half the bits in the function node and therefore maintains a high diversity of function nodes in the population, especially when arity-3 functions are used. GP-UX is therefore unable to act as a truly local search operator, even in the latter stages of the run, and in large trees this can result in failure to find a solution. However, GP-UX's performance was generally better than its GA counterpart, particularly on large trees. This is presumably because GP-UX is not forced to search the very large spaces specified by the maximum initial depth. This is confirmed by the fact that the average complexity of correct solutions grew very slowly relative to the maximum possible size.

Table 6.25: Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 50 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
2	1,100 (50)	12.4	5,400 (41)	12.1	11,500 (27)	12.6	39,600 (7)	13
3	3,300 (50)	31.1	3550 (50)	33.6	4,300 (50)	34.2	5,600 (49)	36.1
4	4,500 (49)	60.1	7,350 (45)	75.3	12,000 (38)	79.2	10,800 (44)	95.6
5	8,100 (42)	119.4	9,750 (36)	111.9	18,500 (33)	142.9	23,750 (31)	178.9
6	9,200 (36)	138.7	10,200 (38)	115.2	34,850 (23)	260.7	33,600 (25)	265.8
7	9,450 (38)	125.4	14,950 (33)	197.0	23,500 (30)	201.9	49,350 (16)	437.7

GP-SUX, unsurprisingly, outperformed GP-UX on every condition and by a significantly greater margin than it did with the arity-2 function set. In fact, the performance of GP-SUX was in many cases comparable its performance with the arity-2 function set for trees of similar sizes. A 40 node solution to the even-8-parity problem found by GP-SUX with maximum initial depth = 4 is shown in Figure 6.8. In

Table 6.26: Minimum efforts required by GP-SUX and GP-SPM to solve the even-n-parity problem for various values of n using a full arity-3 function set and a population of 200 individuals. The number of successful runs (out of 50) is given in parentheses. The average complexities of correct solutions are also given.

	n							
	5		6		7		8	
Tree Depth	Effort	Comp.	Effort	Comp.	Effort	Comp.	Effort	Comp.
2	2,600 (50)	12.4	2,800 (50)	12.3	12,000 (48)	12.8	63,000 (19)	13
3	6,000 (50)	23.5	6,800 (50)	31.2	5,400 (50)	33.7	11,200 (50)	35.3
4	6,400 (50)	43.1	6,000 (50)	54.5	17,600 (49)	73.8	17,200 (49)	86.6
5	6,200 (50)	62.6	8,000 (50)	92.3	17,600 (48)	127.6	25,200 (49)	135.9
6	6,400 (50)	71.8	12,400 (49)	105.6	16,600 (50)	192.8	26,400 (48)	193.5
7	8,400 (50)	71.3	14,800 (48)	92.1	21,200 (48)	144.4	35,200 (46)	301.7

contrast to the standard GP solution, this tree makes use of many useful building blocks and is easy to analyse. Both the even-3 and odd-3 parity functions are used, and there are a number of instances of arity-2 and arity-1 equivalent functions. The simplified solution is shown in Figure 6.9.

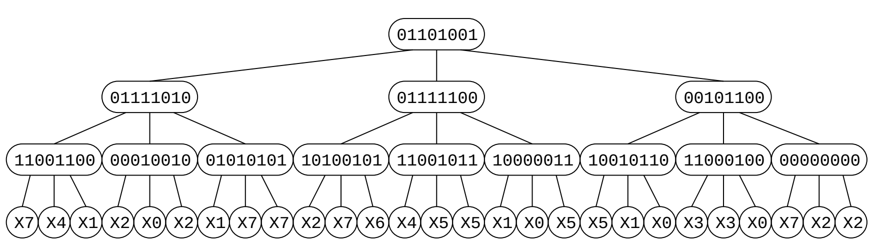

Figure 6.8: 40 node solution to the even-8-parity found by GP-SUX with maximum initial tree depth 4.

6.6 Summary of Results

The main results of the experiments reported in this chapter are summarised below.

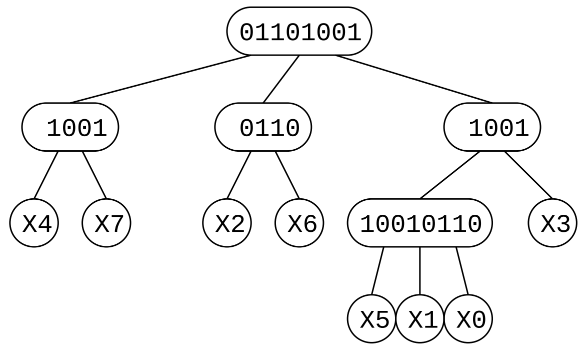

Figure 6.9: Simplified version of the solution shown in Figure 6.8.

- GP-UX and GP-SUX significantly outperformed standard GP on the even-n-parity problem.

- The performance of GP-UX and GP-SUX on the parity functions was, as in the GA studies of the previous chapter, sensitive to depth. In general, however, the operators exhibited superior performance to their GA counterparts with larger trees, and slightly inferior performance with smaller ones. In the former case, this is because the populations were able to move into the solution-rich space of smaller trees. Conversely, in the latter case they occasionally converged on trees that lacked the terminals necessary to solve the problem.

- Standard GP with the full arity-2 and arity-3 function sets also significantly outperformed standard GP with the standard function set of {`AND`, `OR`, `NAND`, `NOR`}. Surprisingly, the effect was most pronounced with the arity-3 function set, in spite of the fact that it contained 256 functions. Standard GP succeeded in finding 2 solutions to the even-7-parity problem, and 5 to the even-6-parity problem, with a population of just 200 individuals. The resultant effort in these cases was, respectively, 3 and 2 orders of magnitude better than the estimates given by Koza [58].

- Standard GP exhibited its best performance on the parity problems with a function set consisting of the primitives {`AND`, `OR`, `NAND`, `NOR`, `EQ`, `XOR`}. However, the performance of GP-UX with the same function set was impaired relative to when the full arity-2 set was used. GP-SUX was not tested as the representation did not permit the use of such a function set without the imposition of severe constraints on the operators.

- When the {EQ} and {XOR} primitives were supressed at intitialisation, standard GP's mutation operator was unable to generate them in sufficient quantities to solve the parity problems. The performance of GP-UX and GP-SUX was, however, unaffected.

- The addition of a single ADF branch had no significant effect on standard GP's performance on the even-n-parity problems when the EQ and XOR primitives were already present in the function set. With 2 ADF branches, performance was impaired relative to when no ADFs were used.

- Standard GP's performance on the 6-multiplexer seems to have been largely unaffected by the use of the arity-3 function set, relative to that used by Koza [58]. However, it was still quite capable of solving the problem with a smaller population than that used by Koza, and since less effort was expended evaluating individuals to no gain, this was reflected in an improvement in performance.

- GP-UX, like its GA counterpart, performed poorly on the 6-multiplexer, confirming our theory that a purely global search represents an inferior strategy on this problem. GP-SUX, however, outperformed standard GP on the same problem.

- On the 6-multiplexer, deceptive attractors presented a significant problem to both GP-UX and GP-SUX, and frequently resulted in the population converging on sub-optimally small trees. This compounded the problems faced by the fixed-length chromosomes described in the previous chapter and as a result the performances of GP-UX and GP-SUX on the 6-multiplexer were markedly inferior to those of their GA counterparts.

6.7 Discussion

The previous two chapters have addressed the issue of global and local search in GP, considering Boolean induction problems as a test case. It is known that standard GP's principal search operator, crossover, performs essentially local search [101][107] and the motivating hypothesis was that this is responsible for GP's poor performance on many problems. Recently, a pair of global search operators called GP uniform crossover (GP-UX) and GP point mutation (GP-PM) have been proposed [107], but they have received little further investigation until now. Our experiments with these operators confirm our findings of the previous chapter, namely that performing a global search of syntax tree spaces enhances performance by several orders of magnitude on the even-n-parity problem, and that combining global and local search operators improves performance still further on this function and also yields good results on a problem that seems to require local search.

Whilst GP-SUX has consistently outperformed GP-UX and standard GP over a range of conditions on 2 Boolean problems, we should not take this as evidence that the search strategy it offers is better per se. The implications of the No Free Lunch theorem [25] are that there will be problems on which either GP-UX or standard GP are superior, and further problems on which still other strategies are optimal. It must be emphasised in this respect that whilst GP-SUX with GP-SPM is able to perform both global and local search, it tends to do so in a set way, performing global search early in the run and local search later, once the population has converged. It is easy to conceive of problems where, for example, a very local search is all that is needed, in which the initial global search would be wasteful. However, it is also important to note that GP-SUX and GP-UX, like their GA equivalents, may be tuned to perform more local search by altering the per-node crossover probability. This could be done at run-time, with alterations of the parameter being made in later runs on the basis of the results of earlier ones. We would thus predict a gradual increase in performance over successive runs as an optimal strategy was found. This approach is redolent of the parameter-tuning performed by meta-GAs, although there is none of the overhead associated with maintaining populations of populations. Further work is needed in this area, and more will be said on the matter in Chapter 9.

These experiments have produced a number of interesting results. Firstly, we saw that whilst the 6-multiplexer is significantly easier for standard GP to solve than the even-n-parity functions, the reverse held for GP-UX and GP-SUX. As we may have predicted from the GA studies, GP-UX and GP-SUX performed better on the parity problems than on the multiplexer, whereas the reverse was observed for standard GP. In the previous chapter we saw that the multiplexer contains a number of deceptive attractors, where the population can frequently converge around small trees which score relatively highly despite lacking one or more of the terminal symbols necessary to solve the problem. In the absence of a terminal mutation operator, the population can thus irrevocably lose these terminal inputs. The situtation is, if anything, slightly worse for GP-UX and GP-SUX since the population may lose not only necessary terminals, but also necessary functions if it converges on very small trees. The problem is particularly noticeable when the maximum initialisation depth is small, and in these circumstances we frequently observed convergence on a single data terminal (each of which scores 40 out of 64 hits on the 6-multiplexer).

These observations, and the remarks made in Section 5.7 concerning the structure of the parity landscapes, led to the speculation that whilst the parity landscape may be multimodal, the solutions occupy a relatively small area of the search space. This area is easy to locate for global operators such as GP-UX and GP-SUX, whereas standard GP's local search operators have relatively little chance of finding it. GP-SUX's ability to perform a local search of this area also accounts for its superiority over the more global GP-UX.

We also speculate that should populations manage to escape the attractors present in the multiplexer, solutions can generally be found by local search, given the correct function set. In the experiments reported here, this function set consisted of 256 functions and the action of its point mutation operator severely impeded GP-UX population convergence. GP-UX was therefore unable to perform anything like the local search required and rarely succeeded in solving the problem whilst GP-SUX and standard GP, with their ability to perform local search, solved the problem without difficulty.

It may be argued that performance on these problems is very sensitive to the function set used and that the results we obtained on the parity functions in particular are not surprising given that the full arity-2 and arity-3 function sets contain a number of useful building block primitives. Our experiments with standard GP using the reduced arity-2 function set {`AND, OR, NAND, NOR, XOR, EQ`} are certainly consistent with this argument – standard GP performed far better with this set than when the full sets were used. However, we should then predict that standard GP would perform better with the full arity-2 function set than with the arity-3, but we in fact observed the opposite. The (admittedly limited) examination of solution trees found with the

Table 6.27: Number of generations and evaluations required to solve high-order versions of the even-n-parity problem. The number of fitness cases per individual is also shown to emphasise how rapidly the actual computational effort increases with n.

n	No. Generations	Individuals evaluated	Fitness cases
13	285	28,500	8,192
15	542	54,200	32,768
17	490	98,000	131,072
20	1188	356,400	1,048,576
22	2093	418,600	4,194,304

arity-3 set revealed that standard GP made little use of the numerous building blocks at its disposal and instead constructed its own from unlikely-looking primitives. It is possible that this function set includes other primitives whose facilitating role is unclear. A full investigation of this is unfortunately outwith the scope of this thesis.

Our experiments with GP-UX on the reduced function set demonstrated that the inclusion of {`EQ`} and {`XOR`} is not the only factor determining GP-UX's superior performance on the parity problems. Performance with this function set was in many cases impaired relative to that with the larger full arity-2 set and we argue that the inclusion of the other, apparently extraneous, primitives generates a smoother landscape for global search operators to follow.

Finally, the ease with which GP-SUX solved the even-n-parity functions with $n \leq 8$ led us to believe that it might be capable of tackling much higher-order versions of the problem. GP-SUX, in combination with sub-machine code GP[109][103] and demes has since found solutions to the even-n-parity functions with $n \leq 22$[3]. The results of these experiments are summarised in Table 6.27. Full details can be found in [110, 104].

[3]Sub-machine code (SMC-GP) is a technique whereby multiple fitness cases are evolved in parallel with considerable savings in execution time. Whilst the exponential increase in fitness cases associated with an increase in order of Boolean induction problems made it a necessity for the even-22-parity function, it was also used to great effect in the experiments reported in this and the previous chapter.

6.8 Conclusion

This chapter has provided a detailed description of a pair of GP search operators that are together capable of performing both global and local search. These operators have significantly outperformed all known GP algorithms on the even-n-parity and 6-multiplexer problems. Nevertheless, this represents success on only a single class of problem, and the representation in any case lends itself particularly well to problems of this kind. The next chapter describes one way in which the representation may be adapted for arithmetic operators and gives results for a number of symbolic regression problems.

Chapter 7

Symbolic Regression of Polynomial Functions

7.1 Introduction

Symbolic regression problems of one variable involve generating functions which approximate a target function, f, given a set of data pairs, (x_i, y_i), such that $y_i = f(x_i)$. In real-world applications, the target function is of course unknown, and the data pairs are obtained from empirical observation. A certain amount of measurement error will inevitably be associated with these observations, and we are therefore interested in inducing functions that provide an approximate fit of the data. For the purposes of testing algorithms on these tasks it is common to derive fitness pairs from a known function and to evaluate each candidate solution according to how well it approximates the target. Since it is possible to derive many semantically equivalent functions to the target, the fitness function evaluates the error observed between the candidate and target values over a number of fitness points. There are a number of ways of doing this, for example summing the absolute differences between the observed and target values, taking the mean squared error, or taking the root of the sum of the squared differences. As stated above, we should not expect a candidate solution to fit the target data perfectly so we should also specify a small tolerance margin of error, within which a fitness case is said to have been correctly classified. The success criterion is therefore not a total error of zero, but an acceptable classification of all the fitness points.

Since the aim of symbolic regression is the induction of a symbolic function, most implementations include mathematical operators such as addition and multiplication in the function set and may additionally include trigonometric functions, root functions, exponential functions etc., if it is felt that the search would profit from them. The terminal set should, of course, include instances of each input variable. As an example, consider the target function $x^4 + x^3 + x^2 + x$. The function set for this function might be the four arithmetic operators $\{+, -, *, pdiv^1\}$ whilst the terminal may consist of the single input variable $\{x\}$. A minimal solution to this problem is shown in Figure 7.1.

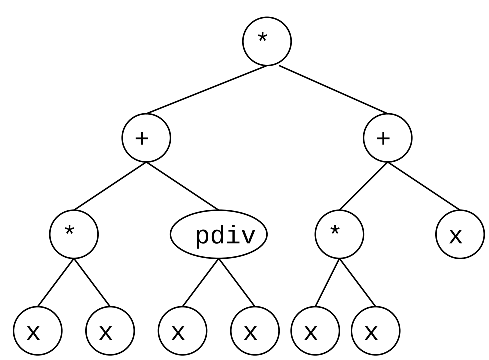

Figure 7.1: Example solution to the quartic polynomial function $x^4 + x^3 + x^2 + x$.

This tree evaluates $(x^2 + 1)(x^2 + x) = x^4 + x^3 + x^2 + x$. Significant in this solution is the sub-tree {`pdiv X X`}, which evaluates to the constant 1. Whilst GP is also capable of generating non-integer constants (e.g. {`pdiv X` {`+ X X`}} `= 0.5`), it is often forced to construct large sub-trees to do so, and this has a detrimental effect both on performance and solution parsimony. In this example, this is not really an issue since the constant actually simplifies the solution and other simple solutions that do not use constants at all can easily be constructed. However, many problems involve finding expressions that include rational coefficients that may not easily be generated using this method. The most widely-used solution to this problem is the inclusion of floating-point constants in the terminal set [58][page 242], although Patrick and York's

[1]The inclusion of the division operator in the function set can lead to invalid trees through division by 0. In the experiments reported in this chapter, we use a protected division function which in the case of division by 0 returns the numerator unchanged.

method [95] of co-evolving constants using small GA populations is an interesting alternative.

This chapter draws on the work of the previous chapters to present an alternative approach: interpolating arithmetic operators. The idea behind this is that included in the function set along with the pure arithmetic functions of addition, subtraction, multiplication and (protected) division are a number of interpolated operators that combine the values returned by two or more of the pure operators and return a value somewhere in between. A single node thus performs two functions:

- It allows small movements to be made around the phenotype space.
- In some cases, it can generate rational coefficients for free.

In this chapter, we present a general method for interpolating between a set of four related operators. The method involves defining the function node as a vector of independent parameters that control the contribution each of the four operators makes to the final return value. Full details of this are given in Section 7.2. The following sections describe the testing of various interpolated function sets on a number of problems. In Section 7.3, we encode the parameters as binary bit-strings, with the result that the GP-UX and GP-SUX search operators can be used without modification. We then compare the performance of standard GP, GP-UX and GP-SUX on a number of polynomial fitness functions using three function sets derived from this representation. Details of these experiments are given in Section 7.4. The results of these experiments are for the most part disappointing, and in Section 7.5 we modify the representation so that the functions interpolate either addition and subtraction or multiplication and division. We examine performance of this modified representation on the same suite of problems in Section 7.5.1. The chapter concludes with a summary of all the results and a discussion.

7.2 Interpolated Arithmetic Operators

Our choice of Boolean test functions in the previous two chapters was motivated to a considerable degree by the ease with which functions could be represented and interpreted. The link between representation and function (i.e between genotype and phenotype of the function node) was close because each node's representation was a truth table specifying the value returned for each of its input cases. This was only possible because the range of inputs was finite and comparatively small. This is not the case for arithmetic operators, of course, and an alternative representation must be used.

The aim of this section is to develop arity-2 function nodes that interpolate between the four arithmetic operators of addition, subtraction, multiplication and protected division. We do this by encoding the function node as a vector of three parameters – one that interpolates addition and subtraction, one that interpolates multiplication and division and a final parameter that interpolates the two interpolated results.

We interpolate addition and subtraction using a single real parameter, α in the range [0, 1] to return a result, i_{as}, as follows:

$$i_{as} = \alpha(arg1 + arg2) + (1 - \alpha)(arg1 - arg2)$$

Similarly we use a second real parameter, β, to interpolate multiplication and protected division, giving the result i_{md}:

$$i_{md} = \beta(arg1 \times arg2) + (1 - \beta)(pdiv(arg1, arg2))$$

Finally we use a third parameter, γ, to return a final result that interpolates all four operators:

$$i_{tot} = \gamma i_{as} + (1 - \gamma)i_{md}$$

The value thus obtained, i_{tot}, is the value returned by the node. In this way, the behaviour of an interpolated arithmetic node is completely specified by 3 independent parameters. This method is known as bilinear interpolation and is the technique used in all the experiments reported in this chapter. A simplified form of this interpolation is described in Section 7.5, and other techniques are discussed in Section 9.2.1.

Figure 7.2 shows how an interpolated node represented in this way could be constructed from a syntax tree with the function set $\{+, -, *, pdiv\}$ and the terminal set $\{\alpha, \beta, \gamma, a1, a2\}$, where $a1$ and $a2$ are the left and right arguments associated with the node. The behaviour of these functions is clearly complex, and we would not generally expect standard GP to be able to evolve them at multiple locations throughout a tree.

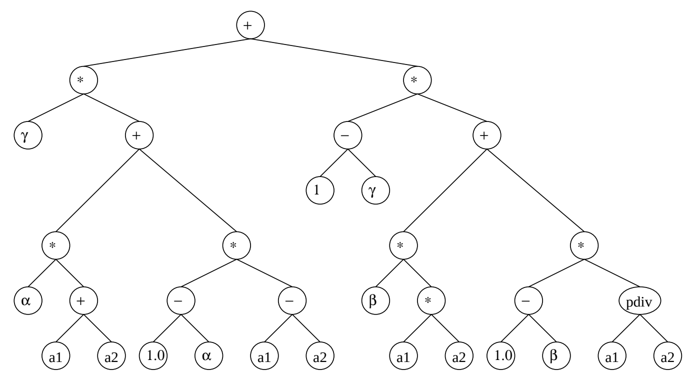

Figure 7.2: A syntax tree equivalent to a single interpolated arithmetic node constructed from pure arithmetic operators. The terminal set includes the three interpolating parameters, and the symbols $a1$ and $a2$, which are the arguments to the interpolated node.

Note that if one of the parameters is set to 1 or 0, one of the terms in its equation evaluates to 0 and does not contribute to the result. Table 7.1 illustrates this, showing the operators interpolated when α, β and γ are set to 0, 1 or some non-integer value in between. Thus, for example, when $\alpha = 1$, the result returned by i_{as} is simply the sum of its two arguments. If, in the same function, $\gamma = 1$, the contribution of i_{md} is ignored, irrespective of the values of the two arguments or β. The function therefore behaves as a pure addition node.

Table 7.1 shows that there are a number of nodes in which the value of α or β is irrelevant in determining the final result. This representation is therefore redundant, in that more than one instance of these functions will occur in the function set. This matter will be discussed further in the next section.

This section has described a general methodology for interpolating the values returned by arithmetic function nodes. In the next section we describe the implementation of these function nodes using a binary representation.

Table 7.1: The effects of varying α, β and γ on which operators are interpolated to produce the return value. The value `non-int` indicates that the parameter is a non-integer value in the range $0 < p < 1$. The value `any` indicates that the parameter can take *any* value without affecting the value returned by the function

Operator	α	β	γ
+	1	any	1
-	0	any	1
*	any	1	0
pdiv	any	0	0
+, -	non-int	any	1
*, pdiv	any	non-int	0
+, *	1	1	non-int
+, pdiv	1	0	non-int
-, *	0	1	non-int
-, pdiv	0	0	non-int
+, *, pdiv	1	non-int	non-int
-, *, pdiv	0	non-int	non-int
*, +, -	non-int	1	non-int
pdiv, +, -	non-int	0	non-int
+, -, *, pdiv	non-int	non-int	non-int

7.3 Representing Interpolated Operators with a Binary Encoding

Encoding the three parameters necessary to specify an interpolated arithmetic node is straightforward. Each parameter is encoded as a binary integer of n bits, which is then mapped onto the range [0, 1]. The interval, and the consequent precision possible, of the interpolation is $\frac{1}{2^n-1}$, i.e. a higher value of n provides a lower resolution of interpolation although at the expense of an explosively expanding search space. Each node therefore consists of $3n$ bits and the function set contains 2^{3n} functions. Clearly, even for quite modest values of n, the size of the search space may rapidly become too large to be manageable.

An example tree in which $n = 2$ is shown in Figure 7.3.

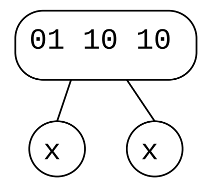

Figure 7.3: Example tree using 2-bit binary encoding. The three parameters encoded on this tree are $\alpha = \frac{1}{3}$, $\beta = \frac{2}{3}$ and $\gamma = \frac{2}{3}$.

In this tree, α, β and γ are, respectively $\frac{1}{3}$, $\frac{2}{3}$ and $\frac{2}{3}$. Both arguments are the input variable x, and the tree therefore evaluates to $\frac{2}{9}x^2 + \frac{4}{9}x + \frac{1}{9}$.

A number of properties of the binary encoding are immediately apparent. Firstly, with $n = 1$, each of the 3 parameters must evaluate to either 0 or 1. As a consequence, the function set contains only the four arithmetic operators. The redundancy of the representation gives two instances of each operator and a total function set size of 8.

For $n > 1$, the function set is biased, in that the functions possible under the representation are not equally represented. An encoding of n bits gives 2^n instances of each of the 4 uninterpolated arithmetic operators. Referring to Table 7.1, we see that interpolating addition and subtraction and multiplication and division (i.e when $\gamma =$ 0 or 1, and either α or β are non-integers) also gives rise to redundancy. For example, the $n = 2$ function $\{$`10##11 x x`$\}$ returns $\frac{4}{3}x$ irrespective of the middle 2-bits (which are represented here by the "don't care" symbol, #). A function set encoded with $n > 1$ bits will therefore include 2^n functions that interpolate addition and subtraction to some degree, and a further 2^n that interpolate multiplication and division. The remaining functions interpolated functions are each included once.

In the next section, we report a number of experiments we conducted using this encoding with various values of n. Amongst other things we will see that performance is sensitive to the value of n used, and that this constitutes another parameter to be optimised by the programmer. We are not, of course, constrained to use a binary encoding for the interpolating parameters, and we may instead represent them directly on the chromosome as floating point values in the interval [0,1]. An illustration of this, a description of GP-SUX and GP-SPM operators modified to work on such a representation, and the results of a number of experiments conducted with these operators and representation will be reported in Chapter 8.

7.4 Experimental Studies

We compared GP-SUX with GP-SPM, GP-UX with GP-PM and standard GP on a number of symbolic regression problems using the arity-2 arithmetic node representation described above. For each set of operators, we compared performance with 3 resolutions, specifically $n = 1, 2, 3$. The function sets therefore comprised 8, 64 and 512 functions respectively. We considered polynomial functions of a single independent variable. Whilst some of these functions included rational coefficients, the idea was that the interpolated functions would prove useful in generating approximately equivalent expressions and random constants were *not* included in the terminal set.

The parameters common to all the experiments are summarised in Table 7.2.

Table 7.2: Parameters used in all the experiments reported in this chapter.

Number of Runs	50
Generations	100
Population Size	200
Max. depth	6
Tourn. Size	4
Std Cross prob	0.9
Unif. Cross prob	0.7
Std Mut. prob	0.01
Unif Mut prob	0.1
Fitness Cases	20
Test Data Range	[-1.0,+1.0]

Most of these parameters are the same as those used in the experiments of the previous chapter. The tournament size and uniform crossover probability parameters were changed as a result of pilot studies on the functions used. The maximum initial depth was not varied in either of the uniform crossover conditions.

20 real fitness points were selected at random from within the range [-1.0,+1.0]. Fitness was defined as the root of the sum of the squared differences between the value returned by the tree and that returned by the target function for each of the 20 cases. If the difference was within 0.01 of the target value, the result was judged a hit. Whilst selection for reproduction was made on the basis of raw fitness, a solution was judged satisfactory if it scored 20 hits.

Many of the problems tested proved too difficult for any of the three algorithms to solve. For this reason, we recorded the fitness of the best (sub-optimal) solution found over the 50 runs, and the average fitness of the best solutions found in each run. In almost all our experiments, standard GP generated populations of very large trees, and this made a heavy impact on processing times. To give some indication of this, and to draw contrasts with the processing required by GP-UX and GP-SUX, we also recorded the total number of function and terminal nodes evaluated by each implementation over the 50 runs of each problem.

7.4.1 Order 3 and 5 Polynomials with Integer Coefficients

We compared the performance of standard GP, GP-UX with GP-PM and GP-SUX with GP-SPM on two polynomial functions without rational coefficients. These were the quintic polynomial, x^5-2x^3+x, and a scaled-down version of the problem, x^3-x, which we call the triptic polynomial.

The quintic polynomial is the function used by Koza in [59][page 110], which his implementation of standard GP without ADFs was able to solve with an effort of 396,000. This function factorises to $x(x-1)^2(x+1)^2$, giving it three roots at $x = 0, -1, 1$. The graph of this function over the interval [-1.0,+1.0] is shown in Figure 7.4.

In [58], Koza scales up the problem to the sextic polynomial, to give a problem on which GP with ADFs outperformed GP without. Pilot studies suggested that GP-UX

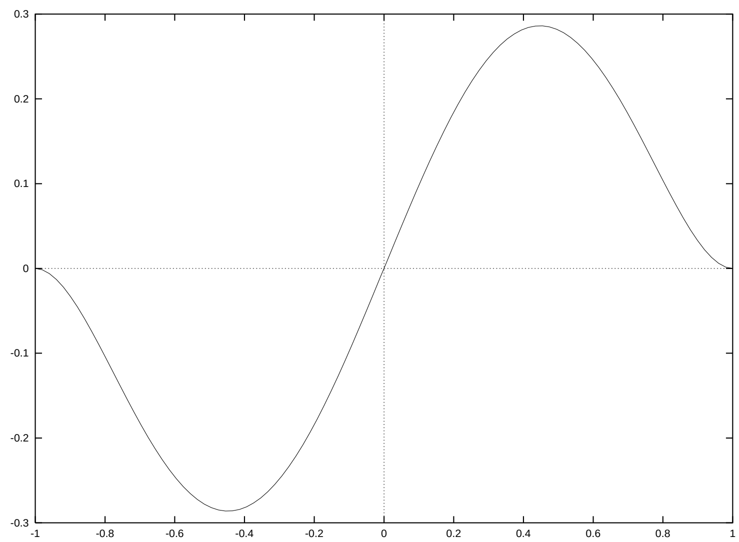

Figure 7.4: Graph of the quintic polynomial function $x^5 - 2x^3 + x$ over the interval [0, 1].

and GP-SUX had difficulty solving higher-order polynomials, and for this reason we used the scaled-down version of the problem given above. The reasons for GP-UX and GP-SUX's poor performance on high-order polynomials will be discussed in full below. The triptic polynomial factorises to $x(x-1)(x+1)$. The graph of this function over the interval [-1.0,+1.0] is shown in Figure 7.5.

We tested standard GP, GP-UX and GP-SUX on both the problems using the three function sets. The minimum efforts required to solve the triptic polynomial and average complexity of correct solutions for standard GP, GP-UX and GP-SUX are shown in Table 7.3. Table 7.4 shows the best and average fitnesses of the best sub-optimal solutions returned at the end of each run, the average complexity of these solutions and the number of nodes evaluated over all 50 runs.

The scaled-down triptic polynomial is, unsurprisingly, an easy problem to solve. Its lack of real-valued coefficients means that interpolated functions are extraneous, and the deterioration in performance for $n > 1$ indicates that they impair search. This is confirmed when we examine the correct solutions found when $n > 1$.

The majority of correct solutions found by GP-SUX and GP-UX with the 2- and 3-bit function sets were small – typically between 7 and 11 nodes – and consisted entirely

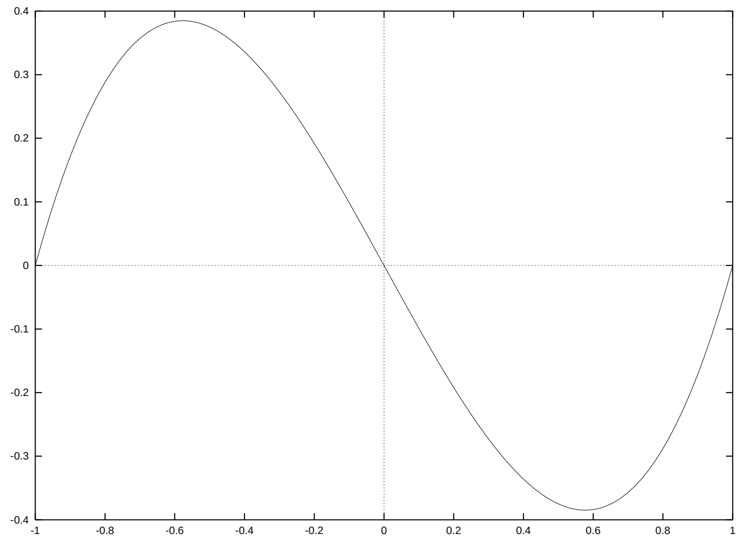

Figure 7.5: Graph of the triptic polynomial function $x^3 - x$ over the interval [0, 1].

Table 7.3: Minimum effort required by standard GP, GP-UX and GP-SUX to solve the triptic polynomial.

	Bits	Effort	Successes	Comp.
Standard	1	4,800	49	27.52
Standard	2	414,000	3	68
Standard	3	—	0	—
GP-UX	1	4,200	48	17.8
GP-UX	2	58,800	17	8.29
GP-UX	3	193,600	7	8.43
GP-SUX	1	4,800	47	20.32
GP-SUX	2	32,000	24	8.7
GP-SUX	3	140,600	9	8.11

of pure arithmetic nodes. There were a few exceptions on the 2-bit representation, however. Figure 7.6 shows an 11 node solution to the triptic polynomial solution containing 2 interpolated arithmetic functions. One of these – `001010` – interpolates subtraction, multiplication and division. With its arguments x^2 and 1, this node evaluates $x^2 - \frac{2}{3}$. The other interpolated function is the root node, which interpolates

Table 7.4: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated over all 50 runs of the triptic polynomial. With the exception of the total nodes evaluated, these data are based on runs where GP failed to find a solution, and the number of runs (out of 50) is also given.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	1	0.088	0.088	1755	1	17
Standard	2	0.019	0.16	694.3	47	238
Standard	3	0.026	0.166	394.7	50	212
GP-UX	1	0.488	0.588	34	2	8
GP-UX	2	0.072	0.385	9.97	33	9
GP-UX	3	0.098	0.349	9	43	9
GP-SUX	1	0.445	0.668	15	6	
GP-SUX	2	0.030	0.257	12.23	26	9
GP-SUX	3	0.029	0.269	11.73	41	9

addition and subtraction. With its arguments $x^3 - \frac{2}{3}x$ and x, this node evaluates the expression $(\frac{1}{3}x^3 + \frac{1}{9}x) + (\frac{2}{3}x^3 - \frac{10}{9}x) = x^3 - x$.

All the solutions found by GP-UX and GP-SUX with the 3-bit function set consisted entirely of pure arithmetic functions with one exception – a 13 node tree found by GP-SUX which is shown in Figure 7.7.

This tree includes 3 interpolated function nodes. One of these – `011111001` – interpolates addition, subtraction and multiplication and evaluates the expression $\frac{6}{49}x + \frac{6}{7}x^2$. The second – `011000001` – is strikingly similar but interpolates addition, subtraction and protected division and evaluates $\frac{6}{49}x + \frac{6}{7}$. These nodes provide the two arguments to a pure subtraction node, which returns $\frac{6}{7}(x^2 - 1)$. The third interpolated node – `000000001` – interpolates division and subtraction. However, since its arguments are both the input variable x, the subtracted component evaluates to 0 and is ignored. The node therefore returns the constant $\frac{6}{7}$. The root node evaluates the expression $\frac{6}{7}(x^2 - 1) \times \frac{7}{6}x = x^3 - x$.

These two solutions demonstrate that GP-SUX and GP-UX are able to combine

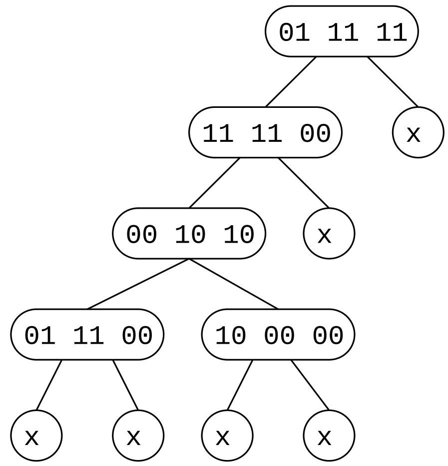

Figure 7.6: An 11 node solution to the triptic polynomial function $x^3 - x$ found by GP-UX with the 2-bit function set. This tree contains 2 interpolated arithmetic nodes

interpolated functions in surprising and complex ways to generate the desired result. However, it is also clear that this was a difficult way of solving the problem – GP-SUX and GP-UX each found 4 solutions that used interpolated operators with the 2-bit function set, and GP-SUX found one with the 3-bit function set. None of the solutions included a function that interpolated all four operators. Having analysed the behaviour of the trees shown in Figures 7.6 and 7.7, it is easy to see why. The interpolated nodes introduce undesirable coefficients that must be removed from the final result by further interpolation. This requires sensitive interplay between the

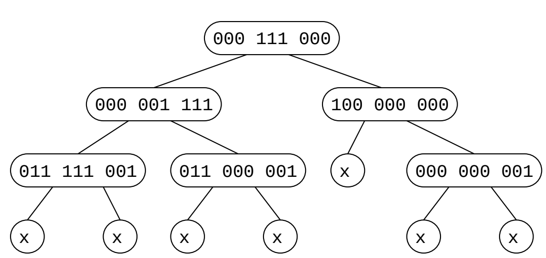

Figure 7.7: A 13 node solution to the triptic polynomial function $x^3 - x$ found by GP-SUX with the 3-bit function set. This tree contains 3 interpolated arithmetic nodes

interpolating nodes and will generally be very difficult to achieve with trees much larger than the ones shown.

It is hardly surprising in the light of this that of the 3 solutions found by standard GP with the 2-bit function set, 2 were approximations. The correct solution contained 11 nodes and solved the problem in a similar manner to the GP-UX tree shown in Figure 7.6. Standard GP's search operators are predictably poor at navigating a space of this complexity.

Examining the total number of nodes processed over each of the 50 runs, we see that standard GP took significantly longer than either GP-UX or GP-SUX. In almost all the experiments reported in this chapter, standard GP solutions showed a marked tendency to bloat, as can be seen from the average complexity of the incorrect solutions, and this, of course, had a serious impact on processing time.

The results for the quintic polynomial are shown in Tables 7.5 and 7.6. It is immediately clear that the scaled-up quintic polynomial is significantly more difficult to solve than the triptic polynomial. Moreover, the addition of extraneous functions to the function set in the 2 and 3-bit conditions rendered the problem all but insoluble to all the operators.

Table 7.5: Minimum effort required by standard GP, GP-UX and GP-SUX to solve the quintic polynomial.

	Bits	Effort	Successes	Comp.
Standard	1	36,000	40	143.1
Standard	2	—	0	—
Standard	3	—	0	—
GP-UX	1	813,600	5	23.4
GP-UX	2	—	0	—
GP-UX	3	—	0	—
GP-SUX	1	891,200	8	24
GP-SUX	2	—	0	—
GP-SUX	3	—	0	—

Table 7.6: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over all 50 runs of the quintic polynomial. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	1	0.018	0.252	1233.5	10	186
Standard	2	0.032	0.111	837.7	50	212
Standard	3	0.061	0.142	844.5	50	287
GP-UX	1	0.079	0.374	17	45	14
GP-UX	2	0.068	0.29	11.12	50	12
GP-UX	3	0.115	0.34	9.4	50	15
GP-SUX	1	0.074	0.36	19.67	42	15
GP-SUX	2	0.054	0.256	12.72	50	12
GP-SUX	3	0.072	0.343	10.9	50	12

The effort required for standard GP using the uninterpolated ($n = 1$) function set to solve the quintic problem is, at 36,000, about an order of magnitude better than the estimate given by Koza in [59]. Koza used populations of 4,000 individuals and terminated each run after 50 generations. We used populations of 200 individuals and ran them for 100 generations. He also included random ephemeral constants – unnecessarily for this function – in the terminal set whereas we did not. We saw in the previous chapter that standard GP is frequently able to solve problems using smaller populations and that this enhances performance since less effort is wasted evaluating trees unnecessarily. It is also apparent that including extraneous terminals in the terminal set impairs performance in much the same way as the inclusion of extraneous functions in the function set.

Examining the best solution trees returned by GP-UX and GP-SUX on each run of the 1-bit condition, we were struck by the fact that many exhibited the same fitness of 0.4416. GP-SUX found 22 such solutions and GP-UX found 29. These varied in size from 7 to 39 nodes, but all evaluated the same expression: $x - x^3 = x(1-x)(1+x)$.

This expression differs from the target function only to the extent that x^5 and the coefficient are missing. It is also clear that the x^5 component accounts for a tiny amount of the variance over the test interval. Given the implicit pressure towards smaller trees generated by the point mutation operator, and the simplicity of this approximation, it is apparent that $x - x^3$ is an attractor from which GP-UX and GP-SUX populations with the 1-bit function set have difficulty escaping.

Other such attractors were also observed. GP-UX and GP-SUX each found two instances of the function $\frac{x-x^3}{2x^2+1}$. With a fitness over the 20 points of 0.079, this is a better approximation of the quintic polynomial, but it is more complex and presumably therefore more difficult to find. GP-UX and GP-SUX each found a tree of 19 nodes that evaluated this function. Additionally, GP-SUX found one of 31 nodes and GP-UX one of 33 nodes.

We did not observe attractors of this sort for $n > 1$, presumably because of the difficulty of evolving semantically equivalent trees using interpolated functions. Nor was it apparent that attractors were a problem for standard GP with any of the function sets. As we saw on the 6-multiplexer function in the last chapter, the pressure towards larger solutions ensures that standard GP populations are able to grow their way out of these regions of the search space, refining solutions with the addition of ever-more complex sub-branches. That standard GP frequently solved the quintic polynomial in this way (at least in the 1-bit condition) is clear from the fact that the average complexity of correct solutions was about 143 nodes.

Performance with the interpolated function sets in both these problems was comparatively poor. This is unsurprising given that neither test function contained any real coefficients and that the interpolated functions were therefore extraneous. In the next section we test the operators, representations and function sets on a function that, without the inclusion of random constants in the terminal set, would seem to benefit from such function sets.

7.4.2 Polynomials with Real-Valued Coefficients

In this subsection we consider a test function where function sets containing interpolated arithmetic functions may outperform those without. The function we considered was the low-order polynomial with real-valued coefficients used by Koza in [58]: $2.718x^2 + 3.1416x$. The graph of this function over the interval [-1.0,+1.0] is shown in Figure 7.8.

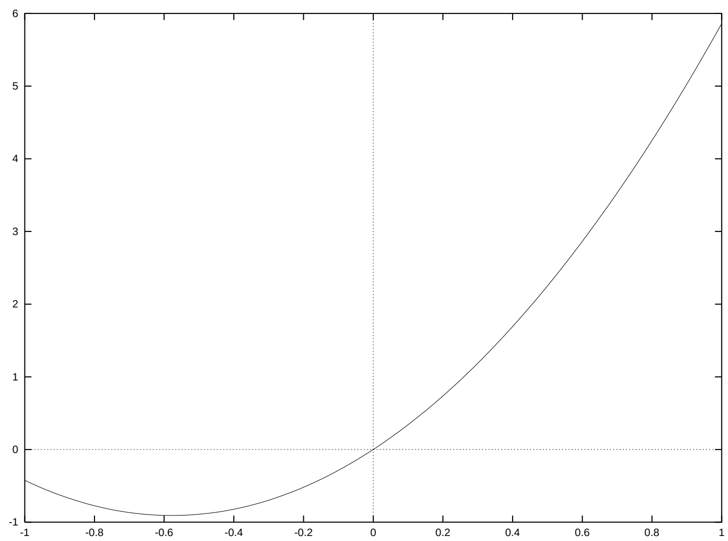

Figure 7.8: Graph of the Koza function $2.718x^2 - 3.1416x$ over the interval [-1.0,+1.0].

The results for the Koza test function $2.718x^2 + 3.1416x$ are given in Tables 7.7 and 7.8.

In [58], Koza used a population of 500 individuals to solve this problem and included the random ephemeral constant in the terminal set. He reported a minimum effort under these conditions of 305,000. He noted that omitting random constants from the terminal set forces GP to construct sub-expressions that approximate the coefficients in the target function, and that this would almost certainly result in a deterioration in performance. Our results with the 1-bit function set, whilst confirming this, indicate that the problem is nevertheless not insoluble without constants. Standard GP found 7 solutions that satisfactorily approximated the target function, and did so with an effort of 504,000. The best, and smallest, of these is listed in Figure 7.9 – a 67 node solution with a fitness of 0.0198531 that evaluates $2.7333x^2 + 3.2x$.

In the 2-bit condition, standard GP actually improved on Koza's estimate. The solutions were large and complex, containing numerous instances of the interpolated functions. No solutions were found in the 3-bit condition, where the function set

Table 7.7: Minimum effort required by standard GP, GP-UX and GP-SUX to solve the Koza function.

	Bits	Effort	Successes	Comp.
Standard	1	504,000	7	168
Standard	2	244,400	15	215.3
Standard	3	—	0	—
GP-UX	1	—	0	—
GP-UX	2	—	0	—
GP-UX	3	—	0	—
GP-SUX	1	—	0	—
GP-SUX	2	—	0	—
GP-SUX	3	1,333,400	2	17

Table 7.8: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and total number of nodes evaluated over 50 runs of the Koza polynomial. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Node ($\times 10^6$)
Standard	1	0.023	0.233	835.235	43	238
Standard	2	0.017	0.16	487.1	35	161
Standard	3	0.073	0.227	536.9	50	188
GP-UX	1	0.397	0.397	17.8	50	13
GP-UX	2	0.061	0.272	15.92	50	10
GP-UX	3	0.066	0.243	15.44	50	12
GP-SUX	1	0.29	0.395	18.4	50	12
GP-SUX	2	0.023	0.179	17.44	50	12
GP-SUX	3	0.035	0.181	15.625	50	14

Figure 7.9: A 67 node solution to the Koza function $2.718x^2 - 3.1416x$ found by standard GP with the 1-bit function set.

```
{ 111 { 000 { 110 X0 { 101 { 110 X0 { 000 { 101 { 101 X0 X0  } { 101 {
010 X0 X0  } X0  }  } { 110 { 101 X0 { 000 { 110 X0 { 110 X0 { 101 X0
X0  }  }  } { 110 X0 { 101 X0 { 101 X0 X0  }  }  }  }  } { 101 X0 {
101 X0 { 101 X0 X0  }  }  }  }  }  } { 101 X0 X0  }  }  } { 100 { 101
X0 { 101 X0 X0  }  } X0  }  } { 110 { 100 X0 X0  } { 101 X0 { 101 X0 {
101 { 010 X0 { 101 X0 X0  }  } X0  }  }  }  }  }
```

apparently became too unwieldy for GP to use effectively. This is reflected in the fact that the best and average fitnesses of the best sub-optimal solutions found are inferior to those found in the 1 and 2-bit conditions.

GP-UX on the 1-bit condition converged on the same solution in *every one* of the 50 runs. Whilst the structure of each solution varied, every tree was equivalent to the expression $3x^2 + 3x$. With the 1-bit function set, GP-SUX found 49 solutions that evaluated to this expression, and one that evaluated the expression $0.5x^3 + 2.5x^2 + 3x$. Whilst marginally superior to $3x^2 + 3x$ over the test interval, the inclusion of the x^3 term means that this solution is in fact a poorer general approximation of the test function. This solution is therefore a result of slight overfitting of the test data.

It is clear that the function $3x^2 + 3x$ constitutes a powerful deceptive attractor within the space of smaller programs that GP-UX and GP-SUX search. This reflects a relative dearth of competitive expressions with real-valued coefficients in these regions. At 67 nodes, the 1-bit standard GP solution given above was theoretically accessible to GP-UX and GP-SUX, but the pressure towards smaller programs meant that this space was never really searched. It would seem from the functions considered so far that problems in which solutions are more common in, or exclusive to, the space of larger programs will prove difficult for GP-SUX and GP-UX.

Neither GP-SUX nor GP-UX found solutions using the 2-bit function set. They did, however, come close. Moreover, the solutions were frequently surprisingly easy to simplify. The best solution found by GP-UX, shown in Figure 7.10, scored 12 hits on the target function. Of its 10 function nodes, only 3 interpolated the arithmetic operators and none incorporated division. This tree evaluates $0.037x^3 + 2.741x^2 + 2.963x$.

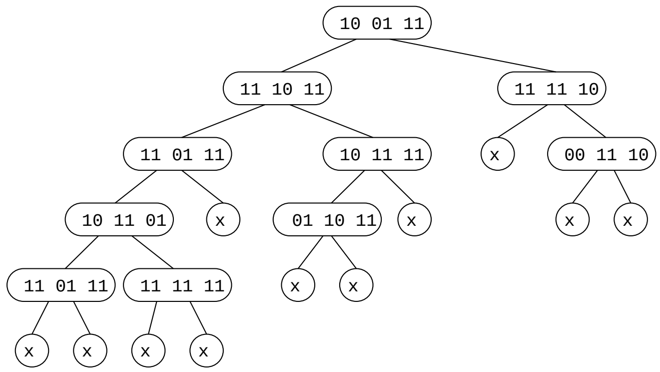

Figure 7.10: Sub-optimal solution to the Koza polynomial found by GP-UX with the 2-bit function set. This tree evaluates $0.037x^3 + 2.741x^2 + 2.963x$.

The best solution found by GP-SUX, shown in Figure 7.11, scored 19 hits. Of its 9 function nodes, only 2 interpolated the arithmetic operators. One of these – the function `010110` – interpolated between multiplication and division, but since both arguments were the input variable x, this did not complicate the expression too severely. The tree evaluates to $-0.037x^3 + 2.741x^2 + 3.148x$.

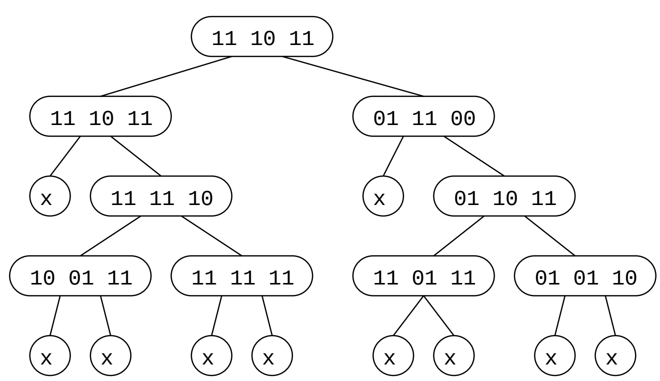

Figure 7.11: Sub-optimal solution to the Koza polynomial found by GP-SUX with the 2-bit function set. This tree evaluates $-0.037x^3 + 2.741x^2 + 3.148x$.

If we ignore the x^3 component, which is an artefact of the interpolated operators and in any case accounts for a tiny proportion of the variance over the input data interval, then both these expressions provide reasonable approximations of the target function. Again, it must be emphasised that the processing required, in terms of the number of function nodes evaluated, to complete 50 GP-UX or GP-SUX runs was a fraction of that required to complete 50 standard GP runs. These results are therefore far more encouraging than consideration of the effort would seem to suggest.

GP-SUX found 2 solutions scoring 20 hits with the 3-bit function set, one of which is shown in Figure 7.12.

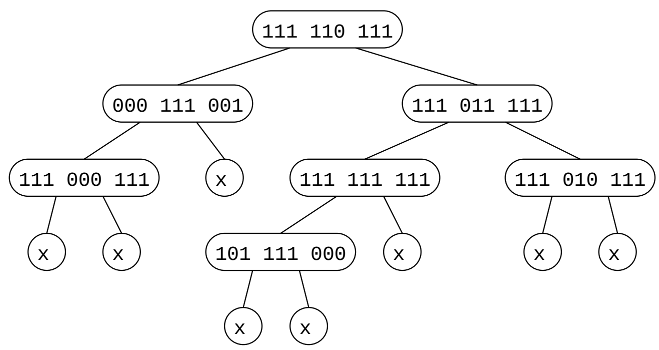

Figure 7.12: Solution to the Koza polynomial found by GP-SUX with the 3-bit function set. This tree evaluates $2.714x^2 + 3.143x$.

Of its 7 function nodes, only one interpolates between the arithmetic operators. This node – `000111001` – interpolates subtraction and multiplication, and with its arguments of $2x$ and x returns the expression $\frac{12}{7}x^2 + \frac{1}{7}x$. The root node adds this to the expression returned by the right branch – $x^2 + 3x$ to give the expression $2.714x^2 + 3.143x$. With a fitness of 0.0057, this was the best solution found in any of the runs on any of the conditions. This was to a large degree due to the resolution of interpolation in the 3 bit function set which permitted good approximations of the desired coefficients (multiples of $\frac{1}{7}$) to be generated easily. This was certainly better than the coefficients that could easily be generated by the smaller 2-bit function set (multiples of $\frac{1}{3}$), and explains why GP-SUX performed better with the 3-bit set, in spite of its increased size. The implication is that the 3-bit function set provided useful building block functions for GP-SUX on this problem. However, only GP-SUX was able to take full advantage of these building blocks. The best solutions found by GP-UX and standard GP using the 3-bit function set were slightly worse than those found with the 2-bit set, indicating that both had difficulty negotiating the larger search space.

From the experiments reported so far, it is apparent that GP-SUX and GP-UX have difficulty constructing large solutions, especially when competitive attractors exist within the space of smaller programs. It is also clear that performance is sensitive to the resolution of the interpolating functions. However, the problems we have examined so far seem not to have required interpolated functions, as can be seen from the fact that standard GP succeeded in solving all three with the 1-bit function set. We address this issue in the next section, deriving 3 test functions from small syntax trees constructed at least partially from interpolated arithmetic operators.

7.4.3 "Building Block" Functions

The results reported so far in this chapter have been somewhat disappointing, but there is a suggestion that interpolating functions may prove useful in solving certain specific functions, most notably low-order polynomials with real-valued coefficients. GP-SUX's ability to find a solution to the function $2.718x^2 + 3.1416x$ with the 3-bit function set was due to the fact that both coefficients could be approximated by multiples of $\frac{1}{7}$ and that such fractions are readily generated by the 3-bit function set. However, the problem is comparatively simple and was in any case solved with significantly less effort by standard GP. Are there, then, any complex test functions on which GP-UX or GP-SUX with an interpolated function set can outperform standard GP? A possible candidate is an expression derived from a small syntax tree containing one or more interpolated arithmetic nodes of the appropriate resolution (i.e. 2 or 3 bits in this case). We call test functions created in this way *building block* functions since they are constructed, at least in part, from primitive building blocks specific to one of the interpolating function sets.

We constructed two syntax trees of differing size using functions drawn from the 2-bit function set, and a third small tree from functions drawn from the 3-bit set. We evaluated these trees over 20 points drawn at random from the interval [-1.0,+1.0] and used these values in the fitness functions.

The Small 2-Bit Building Block Function

The smaller 2-bit function is shown in Figure 7.13. This is a full tree of depth 3, containing 3 nodes that interpolate fully between the four arithmetic operators.

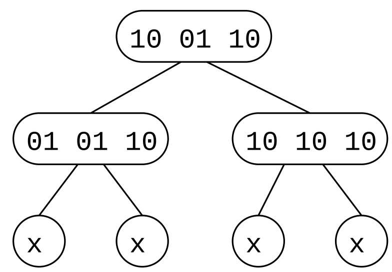

Figure 7.13: The target tree from which the smaller 2-bit building block function is derived.

The presence of interpolated division at the root node means that even this small tree does not reduce to a simple expression. This was also true of many of the solutions found by GP, and made analysing solutions and comparing them with the target function very difficult. The tree given in Figure 7.13 evaluates the expression:

$$\frac{1}{729}\frac{4x^6 + 48x^5 + 384x^4 + 1792x^3 + 3585x^2 + 2052x + 452}{2x^2 + 8x + 1}$$

The graph of this function is shown in Figure 7.14. The polynomial denominator of this function, $2x^2 + 8x + 1$, has roots at $x = -0.129, -3.87$. The first of these two values occurs within the test interval and accounts for the discontinuity observed in the graph shown in Figure 7.14[2].

The results for the smaller 2-bit function are given in Tables 7.9 and 7.10.

GP-SUX found a single solution to this problem using the 2-bit function set, whilst GP-UX found 8. No solutions were found under any of the other conditions. Of the 8 solutions found by GP-UX, 6 replicated the target tree exactly, as did the solution found by GP-SUX. One of the remaining two solutions found by GP-UX is shown in Figure 7.15. We can see that the target tree is contained within this tree, together

[2]None of our test points sampled this discontinuity, however.

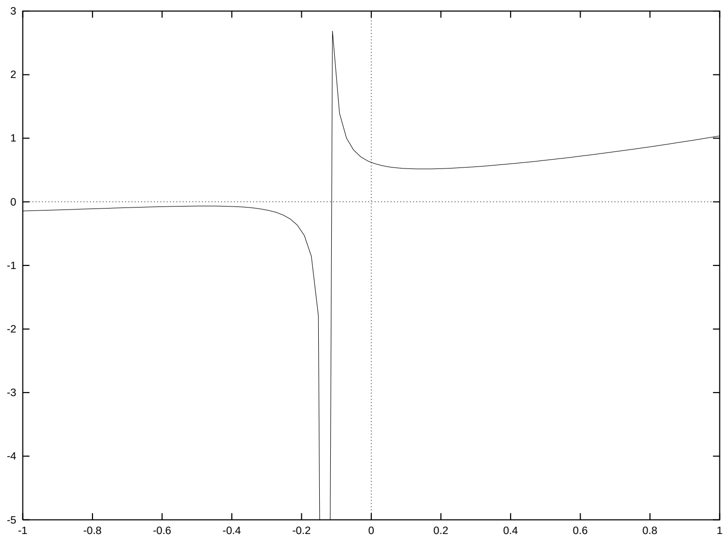

Figure 7.14: Graph of the smaller 2-bit building block function over the interval [-1.0,+1.0].

Table 7.9: Minimum effort required by standard GP, GP-UX and GP-SUX to solve the smaller 2-bit building block function.

	Bits	Effort	Successes	Comp.
Standard	1	—	0	—
Standard	2	—	0	—
Standard	3	—	0	—
GP-UX	1	—	0	—
GP-UX	2	203,400	8	8
GP-UX	3	—	0	—
GP-SUX	1	—	0	—
GP-SUX	2	729,000	1	7
GP-SUX	3	—	0	—

with a pure multiplication node and a pure division node. These two cancel each other out to return the target tree unchanged. The other solution also contained 11 nodes and used essentially the same method. Equivalent expressions using different

Table 7.10: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated complete 50 runs of the smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	1	0.1	0.729	1016.4	50	323
Standard	2	0.091	0.337	707.8	50	298
Standard	3	0.157	0.367	465.9	50	251
GP-UX	1	1.488	2.957	16.48	50	14
GP-UX	2	0.077	0.482	12.47	42	12
GP-UX	3	0.108	0.443	9.76	50	10
GP-SUX	1	0.78	2.912	16.4	50	14
GP-SUX	2	0.067	0.726	12.6	49	13
GP-SUX	3	0.095	0.497	11.4	50	12

functions, if they exist at all, are obviously so rare as to be virtually impossible to find. It follows that the search space contains a tiny number of solutions and that these will be most densely distributed within the region of small trees. The pressure to bloat in standard GP meant that its populations moved out of these regions almost immediately, and consequently stood very little chance of finding a solution.

Despite the fact that it generally converged on reasonably small trees, GP-SUX with the 2-bit function set was markedly inferior to GP-UX on this problem. With the representation used, the search space is undoubtedly highly epistatic, since neither bits, parameters nor nodes can be optimised independently of others elsewhere in the tree. Kauffman [53] has shown that epistatic landscapes in GAs are highly multimodal and tend to be difficult to search. Such spaces will also be very difficult for GP-SUX to search since it will almost always converge on local optima from which, by virtue of its tendency to perform local search, it will be unable to escape. The problem may actually be more severe than it is for a GA because the GP-SPM operator can only flip one bit per node at a time. In contrast, GP-UX with GP-PM maintains a

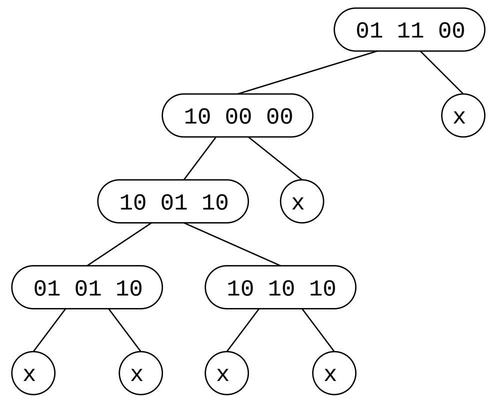

Figure 7.15: 11 node solution to the smaller 2-bit building block function found by GP-UX.

much higher genetic diversity and is not so susceptible to convergence at local optima. Furthermore, the size of the 2-bit function set appears to be small enough to give it a reasonable chance of locating the optimal functions. It would appear, then, that this highly epistatic function is an example of a problem for which the sort of prolonged global search performed by GP-UX represents a better strategy.

The Larger 2-bit Building Block Function

Given the difficulty of finding a function that could be constructed from 7 nodes, it seemed very unlikely that GP-UX or GP-SUX would be able to solve more complex problems made up entirely of fully interpolated arithmetic nodes. In constructing our larger tree, we therefore included a number of partially interpolated and uninterpolated arithmetic functions. The larger 2-bit tree is shown in Figure 7.16.

This tree comprises two nodes that interpolate fully between the four arithmetic operators, one that interpolates addition and subtraction, one that interpolates multiplication and division, one that interpolates addition and division, and two pure multiplication nodes. It thus includes four functions where some redundancy may be exploited, as indicated in Figure 7.16 by the "don't care" symbol #. The expression evaluated by this tree is

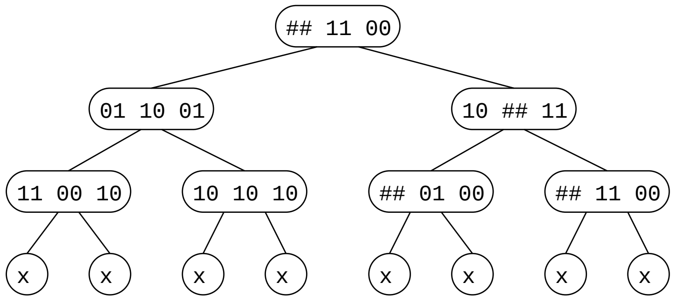

Figure 7.16: Target tree from which the larger 2-bit building block function is derived.

$$\frac{4}{729}\frac{(x^2+1)(32x^5+258x^4+668x^3+621x^2+502x+95)}{2x^2+8x+1}$$

We can see immediately that this expression contains the same polynomial denominator as the smaller 2-bit function and will therefore exhibit discontinuities at the same points. The graph of this function over the interval [-1.0,+1.0] is shown in Figure 7.17.

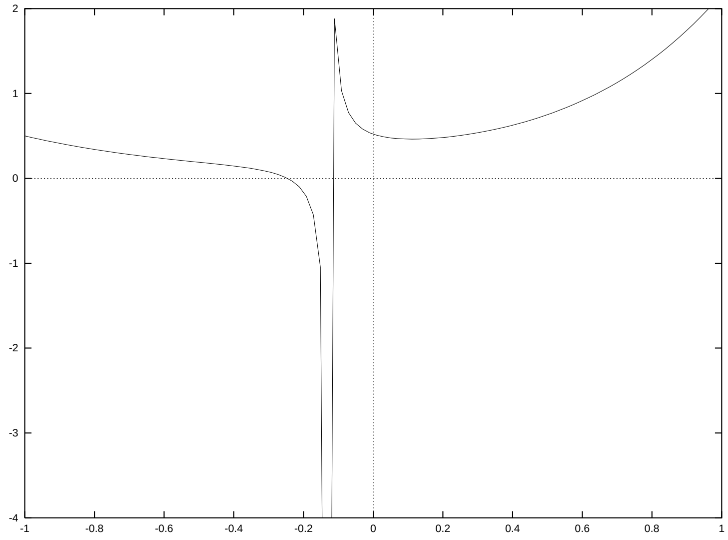

Figure 7.17: Graph of the larger 2-bit building block function over the interval [-1.0,+1.0].

No solutions to this problem were found in any of the conditions. The best and average fitnesses of the best individual found on each of the 50 runs, the average complexities of these individuals, and the time taken to complete 50 runs are shown in Table 7.11.

Table 7.11: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the number of nodes evaluated over 50 runs of the larger 2-bit building block function.

	Bits	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	1	0.1526	4.33	831.2	212
Standard	2	0.112	0.49	944	238
Standard	3	0.159	0.616	744.7	207
GP-UX	1	5.85	7.08	14.96	13
GP-UX	2	0.148	1.18	14.32	13
GP-UX	3	0.385	1.09	13.24	12
GP-SUX	1	4.5	7.066	10.76	10
GP-SUX	2	0.165	1.184	19.4	12
GP-SUX	3	0.151	1.038	16.04	13

Performance is notably poorer than on the smaller 2-bit function. Standard GP and GP-UX performed best when the 2-bit function set was used, with standard GP outperforming both GP-UX and GP-SUX. GP-SUX exhibited its best performance with the 3-bit function set, although this was not superior to standard GP with the 2-bit set. In both this and the smaller 2-bit function, standard GP's syntax trees became very large, a fact reflected in both the average complexity of solutions and the node evaluations performed over 50 runs, and it is almost certainly this which allowed it to find slightly better approximations than either GP-UX or GP-SUX.

On average GP-UX and GP-SUX found solutions of roughly equivalent quality with all 3 function sets, and both were severely impaired by the 1-bit set. The best solution found by GP-UX was slightly better than GP-SUX with the 2-bit function set, and somewhat poorer with the 3-bit. Whether this reflects differences in the efficiency of the search of these problem spaces performed by each operator on either representation is unclear. Whilst obvious attractors such as those observed in the Koza function are not present, it was also clear that GP-UX and GP-SUX rapidly converged on very small trees. These problems were exacerbated by the small populations used, and it is conceivable that they may be overcome by using larger population sizes.

Even with the presumably simplifying influence of the pure arithmetic operators, we can see that this is a very difficult problem to solve. With the appropriate 2-bit function set, GP-UX and GP-SUX frequently converged on trees of roughly the right size – as can be seen from the average complexities – but were nevertheless unable to find the right functions. The problem is not only epistatic but also more complex than the previous function, and these two factors put it beyond the capabilities of any of the operators with the small populations used.

The 3-bit Building Block Function

The results of the previous two experiments implied that if solutions were to be found to an expression constructed from 3-bit interpolated functions, the expression would have to be small. The 3-bit tree we used is shown in Figure 7.18. Its three function nodes all interpolate the four arithmetic operators.

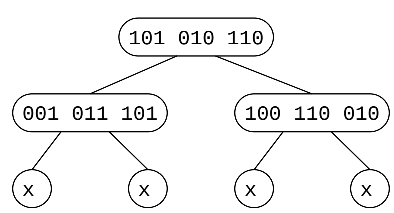

Figure 7.18: Target tree from which the 3-bit building block function is derived.

The function evaluated by this tree is

$$\frac{2}{117,649}\,\frac{5,400x^6 + 14,760x^5 + 602,256x^4 + 845,044x^3 + 729,879x^2 + 315,277x + 100,405}{30x^2 + 16x + 5}$$

The dividing polynomial in this expression, $30x^2 + 16x + 5$, has no real roots and discontinuities are therefore not observed. The graph of this function over the interval [-1.0,+1.0] is shown in Figure 7.19.

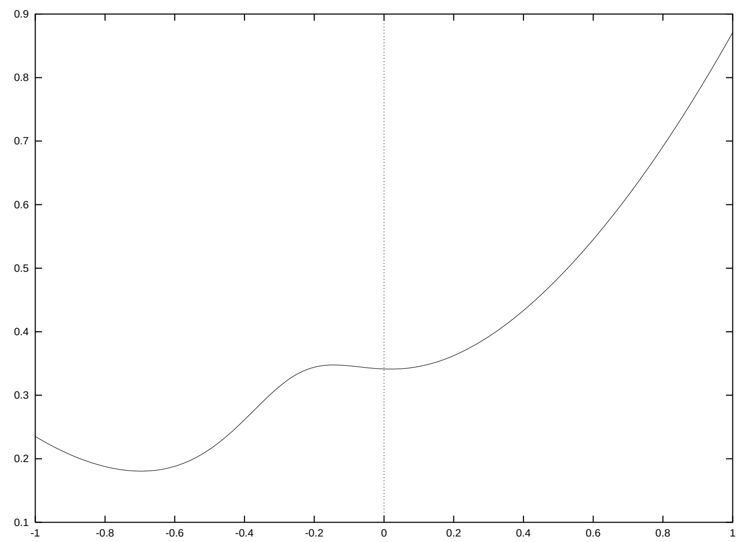

Figure 7.19: Plot of the 3-bit building block function over the interval [-1.0,+1.0].

No solutions to the 3-bit tree function were found on any of the runs. The results pertaining to the sub-optimal solutions are given in Table 7.12.

Table 7.12: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the 3-bit building block function.

	Bits	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	1	0.041	0.102	1137.4	336
Standard	2	0.029	0.151	690.3	214
Standard	3	0.037	0.163	321.4	187
GP-UX	1	0.139	0.467	15.48	10
GP-UX	2	0.105	0.242	7.4	12
GP-UX	3	0.089	0.189	5.32	11
GP-SUX	1	0.139	0.454	14.96	11
GP-SUX	2	0.078	0.229	8.52	10
GP-SUX	3	0.056	0.197	5.8	10

Standard GP found better approximations with all the function sets than anything produced by either GP-UX or GP-SUX but did so, particularly in the 1-bit condition, at the expense of very large trees.

On the 1-bit function set, the best solutions found by GP-UX and GP-SUX scored a fitness of 0.139. GP-UX found 10 such solutions, whilst GP-SUX found 6. Every one of these trees evaluated to $\frac{1}{3-2x}$. A more frequently found solution scored 0.395 and evaluated the linear expression $\frac{x+1}{3}$. GP-UX found 12 such trees, whilst GP-SUX found 18. These two expressions therefore account for almost half the solutions found by GP-UX and GP-SUX. Attractors were observed in the 2-bit test functions, but with nowhere near the same frequency. This is presumably because of the discontinuity present in those functions, which is not a feature of the 3-bit target tree.

GP-UX and GP-SUX performance, both in terms of the best solution found and the average fitness of solutions, improved as the resolution of the function nodes increased. The best 3-bit solution found by GP-UX is shown in Figure 7.20. Despite being structurally isomorphic to the target tree, its 3 function nodes are some 7, 6 and 5 Hamming points distant from the optimum. The expression evaluates to a highly complex order 7 polynomial, divided by an order 3 polynomial. Whilst this function is too complex to be worth showing in full, it did feature the dividing order 3 polynomial $90x^3+166x^2+71x+21 = (5x+7)(18x^2+8x+3)$. This function has one root which, occurring at $x = -1.4$, falls just outside the test interval. The function will therefore produce a discontinuity at this point, and is the result of overfitting.

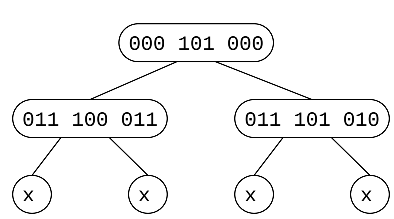

Figure 7.20: Sub-optimal solution to the 3-bit building block function found by GP-UX with the 3-bit function set.

The best 3-bit solution found by GP-SUX is shown in Figure 7.21. This is a better approximation of the target function than the best solution found by GP-UX, and evaluates:

$$\frac{2}{16,807}\frac{25,000x^6 + 52,125x^5 + 71,510x^4 + 56,580x^3 + 72,536x^2 + 54,918x + 31,812}{25x^2 + 12x + 10}$$

We can see that whilst the coefficients bear little superficial resemblance to those in the target function, the two polynomials are of the same order. Furthermore, we can see that the dividing polynomial has no real roots and, unlike the GP-UX solution given above, will not produce a discontinuity. The graph of this function is shown in Figure 7.22.

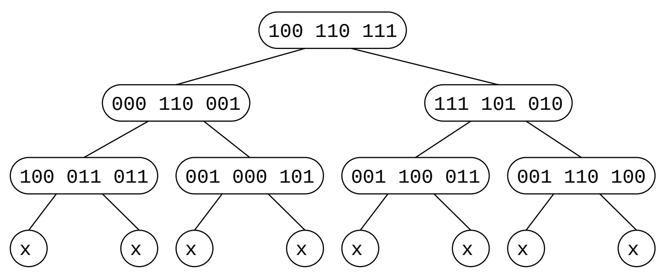

Figure 7.21: Sub-optimal solution to the 3-bit building block function found by GP-SUX with the 3-bit function set.

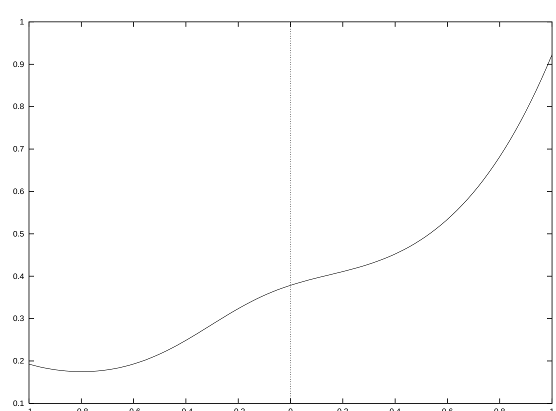

Figure 7.22: Plot of the function generated by the tree shown in Figure 7.21 over the interval [-1.0,+1.0].

The failure of GP-UX and GP-SUX to solve this function may seem surprising given that the target tree contained just 3 nodes and that in many of the runs the population converged on structurally identical trees. The problem in all three functions described in this sub-section appears to be that there are epistatic relationships both between the function nodes and between the parameters within the function nodes. Thus the fitness of a given function node is dependent on the other functions in the tree, and the fitness of a given parameter is dependent on the other parameters in the node. The problem is further compounded by the fact that equivalent solutions constructed from different functions are so rare as to be almost impossible to find.

The fact that GP-SUX found better solutions than GP-UX on the 3-bit building block function probably indicates that it was better at compensating for errors in the values of its other parameters rather than that it was closer to finding a solution per se. Put another way, once a GP-SUX population had converged on a region, it was only able to perform a very local search of the area and locate a nearby optimum. GP-UX, by virtue of the large function set, was probably unable even to do that.

7.5 Partially-Interpolated Function Nodes

After the impressive gains reported earlier in this thesis, the results of this chapter have been rather modest. However, the results of these experiments have highlighted a number of possible flaws in the interpolated representation that could be improved upon. These are:

- The representation is highly epistatic, particularly at the intra-node level. Generating a fit node at a given position in the tree requires optimising 3 parameters and since optimising a single parameter often cannot be done independently of the other two, the task facing GP-UX and GP-SUX is formidable.
- The size of the function set grows extremely rapidly with the number of bits encoding each parameter such that, given the epistatic nature of the representation, the size of the search space becomes unrealistic for all but the coarsest of interpolations.
- On the other hand the representation is (perhaps unnecessarily) redundant. An encoding using n bits per parameter contains 2^n instances of each pure arithmetic function. A single application of the GP-SPM operator to such a

node will therefore leave the function semantically unchanged in $\frac{1}{3}$ of cases on average.

- Solutions to many problems may be possible without recourse to fully interpolated functions, as can be seen in the solution to the Koza function shown in Figure 7.12 which contains a single node interpolating subtraction and multiplication.

The last point is central to the studies reported in this section, where we describe and test functions that interpolate either addition and subtraction or multiplication and division. There are a number of ways that such a scheme could be implemented. We could, for example, reduce the number of bits encoding the γ parameter to 1, allowing either of the remaining parameters to be turned on or off. However, the redundancy of the function set would actually increase (since there would be 2^n instances of each function) and the impact on the size of the search space would not be especially significant.

A simpler and more efficient technique would be to dispense with the redundant parameter altogether. In this approach each node comprises a single interpolating parameter encoded over n bits, plus an extra bit specifying whether the node interpolates addition and subtraction or multiplication and division. At 2^{n+1}, the size of the function set compares well with the 2^{3n} functions generated by the fully-interpolated representation. This technique also addresses the remaining two points in the above list – there is no redundancy in the representation, and the parameters can be optimised with a greater degree of independence. For obvious reasons, we call these functions *partially interpolated*.

The partially-interpolated function set does not require any modifications to be made to any of the operators used in this chapter.

7.5.1 Experiments with the Partially-Interpolated Function Set

We tested standard GP, GP-UX with GP-PM and GP-SUX with GP-SPM on the Koza Polynomial $2.718x^2 + 3.1416x$ and the 3 Building Block functions described in Section 7.4. The performance of the fully-interpolated functions on the triptic and quintic polynomial problems did not seem to justify further investigation, and they were omitted from the current suite. We encoded our functions with $n = 2, 3, 4$ bits (for $n = 1$, the function set is simply the four pure arithmetic functions). The remaining parameters were the same as those given in Table 7.2.

The Koza Polynomial, $2.718x^2 + 3.1416x$

The minimum efforts, number of successful runs and average complexities of correct solutions for standard GP, GP-UX and GP-SUX running partially-interpolated function sets on the function $2.718x^2 + 3.1416x$ are given in Table 7.13

Table 7.13: Minimum efforts required by standard GP, GP-UX and GP-SUX to solve the Koza function using the partially interpolated function set.

	Bits	Effort	Successes	Comp.
Standard	2	118,800	20	67.75
Standard	3	231,000	10	181.0
Standard	4	319,200	10	446.0
GP-UX	2	—	0	—
GP-UX	3	115,200	17	19.58
GP-UX	4	2,781,600	1	15.0
GP-SUX	2	—	0	—
GP-SUX	3	118,800	20	20.0
GP-SUX	4	4,560,000	1	21.0

Comparing these results with those for the fully interpolated set, given in Table 7.7, it is clear that all 3 techniques profit from the use of the partially-interpolated function set on this problem. Standard GP's performance is improved by a factor of 2 on the 2-bit encoding, and it succeeded in finding solutions with higher values of n where previously it had failed. Examining the solutions found with the 2-bit encoding, we noticed that the overwhelming majority of nodes were uninterpolated – that is they evaluated one of the four pure arithmetic operators – and that there was a pronounced bias towards addition-subtraction functions. This is not surprising since reasonable approximations to this function can be found with minimal use of 2-bit interpolated functions, and the multiplication operator need only be used once to obtain the 2nd order polynomial. It was much more difficult to effect such a strategy as n increased, however, and we see a corresponding increase in both effort and average solution complexity, in spite of the fact that the 3-bit encoding almost certainly contains the most appropriate building blocks for this problem. In these cases, standard GP consistently moved into the space of larger programs where the search space is presumably smoother.

Neither GP-UX nor GP-SUX succeeded in solving the problem with the 2-bit encoding, but both performed very well with 3 bits. Recall that with the fully interpolated function set, GP-UX failed to find a solution at all, whilst GP-SUX found just 2 solutions with the 3-bit encoding. GP-SUX's performance with the 3-bit partially-interpolated function set represents an improvement of over an order of magnitude compared with the same fully-interpolated encoding.

GP-UX and GP-SUX evidently utilised entirely different strategies to standard GP in locating solutions. A fairly typical solution, found by GP-SUX, is given in Figure 7.23. The root node's right branch indirectly interpolates the x^2 term to return $\frac{12}{7}x^2$ which, when added by the root node to the x^2 term returned by the left branch, gives $\frac{19}{7}x^2$. The right branch contains a single node interpolating addition and subtraction that, with its two terminal arguments, returns $\frac{8}{7}x$. This is term is added twice to x further up the tree to give $\frac{22}{7}x$. The final result is therefore $\frac{19}{7}x^2 + \frac{22}{7}x$, which is a very good approximation of the test function. The vast majority of the solutions found by GP-SUX and GP-UX that we analysed were semantically equivalent to this tree, and it was clear that the search space is rich in small trees that evaluate this expression. It is also clear that the function set is not too large to impair GP-PM.

GP-UX and GP-SUX each found a single solution with the 4-bit encoding. Both solutions were, unsurprisingly, of inferior quality to those found with the 3 bit encoding and in the case of the GP-SUX function, were the result of overfitting. The GP-UX solution (not shown) evaluated $\frac{43}{15}x^2 + \frac{12002}{3825}x = 2.87x^2 + 3.138x$.

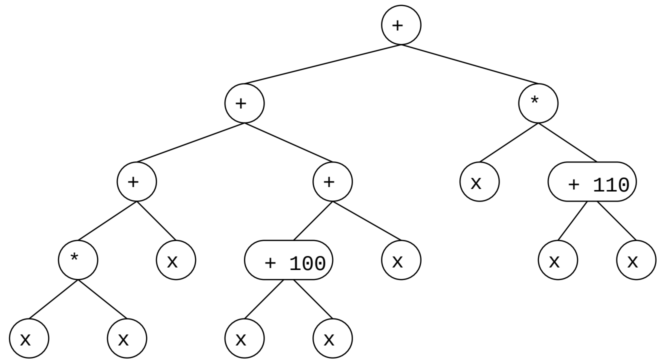

Figure 7.23: Solution to the Koza polynomial found by GP-SUX with the 3-bit partially interpolated function set. Both operator and parameter are given for the two interpolated functions in this tree; the remainder are uninterpolated. This tree evaluates to $\frac{19}{7}x^2 + \frac{22}{7}x$.

The Building Block Functions

The improved performances of the partially-interpolated function set on the order-2 polynomial, $2.718x^2 + 3.1416x$ are encouraging, but the problem is relatively simple. The building block functions are likely to prove far more difficult, particularly for GP-UX and GP-SUX, not least because they are not derived from trees made up of partially-interpolated nodes. Nevertheless, we are seeking to develop efficient algorithms capable of tackling complex test functions, and it will be interesting to see how well each of the operators fare on these problems.

The results for standard GP, GP-UX and GP-SUX on the small 2-bit building block function are given in Tables 7.14 and 7.15.

Standard GP solved the problem once, running the 3-bit encoding. At 697 nodes, this is a very large solution, and seems to indicate that small approximations to this function, if they exist at all, are very sparse. Neither GP-UX nor GP-SUX succeeded in solving the problem with any of the function encodings. As can be seen from Table 7.15, their populations consistently converged on small trees, and the quality of their best sub-optimal solutions generally fell some way short of that of standard GP.

Neither Standard GP, GP-UX nor GP-SUX succeeded in solving the Large 2-bit building block function with any of the encodings. Table 7.16 shows that standard GP

Table 7.14: Minimum efforts required by standard GP, GP-UX and GP-SUX to solve the Small 2-bit building block function using the partially interpolated function set.

	Bits	Effort	Successes	Comp.
Standard	2	—	0	—
Standard	3	2,280,000	1	697.0
Standard	4	—	0	—
GP-UX	2	—	0	—
GP-UX	3	—	0	—
GP-UX	4	—	0	—
GP-SUX	2	—	0	—
GP-SUX	3	—	0	—
GP-SUX	4	—	0	—

generally yielded closer approximations although these were large and complex. On average, GP-UX and GP-SUX performed much more poorly with the 4-bit encoding, yet paradoxically both found their best solutions with this function set.

GP-UX and GP-SUX also failed to solve the 3-bit building block function, but standard GP did find a single solution. As can be seen from Table 7.17, this was found with the 3-bit function set and, at 1189 nodes, was very large. We should be wary of concluding on the basis of this single success that the partially interpolated representation offers standard GP any pronounced advantage on this problem and indeed, comparing Table 7.18 with Table 7.12 we do not note any striking differences. These three functions are clearly very difficult for all the algorithms, regardless of the function set or encoding used.

7.6 Summary of Results

The six experiments reported in this chapter have provided a number of interesting findings, of which the main points are:

Table 7.15: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	2	0.08	1.09	1074.0	50	347
Standard	3	0.05	0.53	797.2	49	248
Standard	4	0.11	0.21	861.0	50	318
GP-UX	2	0.21	2.27	14.8	50	12
GP-UX	3	0.21	2.12	16.4	50	13
GP-UX	4	0.17	1.28	16.0	50	13
GP-SUX	2	0.36	2.43	14.7	50	11
GP-SUX	3	0.27	2.9	15.72	50	12
GP-SUX	4	0.23	3.12	16.9	50	12

- In all the experiments, standard GP populations tended to bloat rapidly. In contrast, and more interestingly, GP-UX and GP-SUX populations converged on trees significantly smaller than the largest possible given the maximum depth at initialisation. A consequence of this was that standard GP outperformed GP-UX and GP-SUX on problems that, like the quintic polynomial, appear to be characterised by a high density of solutions in the space of larger programs.

- A contributing factor to this phenomenon is the presence of highly competitive, but sub-optimal, trees within the space of smaller programs. These trees are quickly found by GP-UX and GP-SUX with the result that the populations converge around them and are then unable to escape.

- The inclusion of interpolated arithmetic operators in the function set was only justified when the test function contained non-integer coefficients. When the function could be solved without them, they were demonstrably extraneous, and even deleterious. This point, whilst predictable, is by no means obvious as the results with the arity-3 function set in the previous chapter showed.

Table 7.16: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Large 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	2	0.03	0.32	770.6	50	380
Standard	3	0.06	0.67	681.4	50	328
Standard	4	0.11	0.21	861.0	50	328
GP-UX	2	0.19	1.88	13.8	50	12
GP-UX	3	0.24	1.24	12.9	50	11
GP-UX	4	0.08	3.43	13.7	50	12
GP-SUX	2	0.29	1.88	13.8	50	11
GP-SUX	3	0.19	1.35	17.32	50	11
GP-SUX	4	0.08	5.58	15.6	50	13

- GP-SUX is capable of using the fully-interpolated function set to find good, parsimonious approximations to low-order polynomials such as the function $2.718x^2 + 3.1416x$. To do so, however, the interval should allow close approximations of the coefficients to be generated easily.

- There is some evidence that more epistatic problems, such as the building block functions described in Section 7.4.3, benefit from a prolonged global search with the fully-interpolated function set as performed by GP-UX. This remains a hypothesis, however, as GP-UX was unable to find a solution when the complexity of the target tree or the size of the optimal function set increased.

- The use of a partially-interpolated function set, whereby the nodes interpolate either addition and subtraction or multiplication and division significantly enhanced the performance of all three search operators on the low-order polynomial $2.718x^2 + 3.1416x$. However, these gains did not generalise to the building block functions.

Table 7.17: Minimum effort required by standard GP, GP-UX and GP-SUX to solve the 3-bit building block function using the partially interpolated function set.

	Bits	Effort	Successes	Comp.
Standard	2	—	0	—
Standard	3	1,778,400	1	1189.0
Standard	4	—	0	—
GP-UX	2	—	0	—
GP-UX	3	—	0	—
GP-UX	4	—	0	—
GP-SUX	2	—	0	—
GP-SUX	3	—	0	—
GP-SUX	4	—	0	—

- GP-UX and GP-SUX, whilst occasionally outperformed by standard GP in terms of effort, were nevertheless frequently able to generate reasonable, and far more parsimonious solutions. In most cases standard GP had to evaluate over an order of magnitude more nodes than GP-UX and GP-SUX to complete 50 runs, and produced solutions that were over an order of magnitude larger.

- Increasing the population size in GP-SUX and GP-UX runs will almost certainly enhance performance on these problems. Since bloat is not a problem in these conditions, we could increase the population to a significant degree before the processing demands, as measured by program evaluation, became comparable with those of standard GP.

7.7 Discussion

Throughout this thesis, we have taken pains to stress that the operators described here will not outperform standard GP on all problems. In this chapter, we have presented a class of problems in which GP-UX and GP-SUX display at best modest

Table 7.18: Best and average fitnesses of the best sub-optimal solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Smaller 2-bit building block function. The number of runs on which the fitness and complexity data are based is also given for each condition.

	Bits	Best Fit.	Av. Fit	Comp.	Runs	Nodes ($\times 10^6$)
Standard	2	0.05	0.06	1174.2	49	364
Standard	3	0.05	0.53	1167.4	50	338
Standard	4	0.11	0.21	671.4	50	288
GP-UX	2	0.11	0.3	14.8	50	10
GP-UX	3	0.09	0.21	13.0	50	10
GP-UX	4	0.08	0.17	12.8	50	10
GP-SUX	2	0.15	0.31	14.7	50	10
GP-SUX	3	0.09	0.22	13.24	50	10
GP-SUX	4	0.09	0.19	13.72	50	10

gains over standard GP, and in some cases perform worse. We have also examined a number of reasons for this. The principal reason appears to be that many of these problems appear to require the ability to generate large solutions since the space of smaller programs contains many attractors whose influence it is difficult to escape without some pressure towards bloat. Standard GP operating on large trees appears to have been able to make many small refinements of the behaviour of fit trees and it is clear that the fitness landscape in these regions is smoother [71]. It is also clear that including beneficial building block primitives in the function set may not help if the probability of using them appropriately is low. Trees that make efficient use of these building blocks may be difficult to find because, as with GP-UX with GP-PM, it is unlikely that the right functions will be found at all or, as seems to be the case with GP-SUX and GP-SPM, there is no clear fitness gradient leading search towards the regions they occupy.

It was clear from every experiment we ran, including a number not reported here, that GP-UX and GP-SUX populations quickly converged on comparatively small trees, typically of between 7 and 20 nodes. This was the case for both the fully- and partially-interpolated function sets. In the case of the quintic polynomial, this convergence could be clearly premature and lead to poor performance on what was for standard GP a very easy problem to solve. It was also clear from pilot studies that increasing tree depth did not affect this – the population still converged on trees of the same size. This is in contrast to our observations with the even-n-parity functions, where we saw that GP-SUX and GP-UX with arity-2 functions seemed to prefer trees of around 40 nodes.

What is it about these problems that causes GP-UX and GP-SUX populations to converge so consistently on small trees? The work reported here is a small subset of a sizeable body of pilot studies we conducted with GP-UX and GP-SUX with a view to inducing reasonable parameters and creating a representative suite of test functions. During the course of these experiments we noticed that while GP-UX and GP-SUX yielded reasonable results on order-2 polynomials with suitable coefficients (i.e. appropriate to the encoding of the interpolation), performance dropped sharply when the order was increased to 3 and higher-order polynomials became all but impossible. Some idea of this can be obtained by comparing performances on the triptic and quintic polynomials – whilst standard GP's performance with the 1-bit encoding deteriorated by about about a factor of 7, GP-SUX and GP-UX both witnessed an increase in effort of over 2 orders of magnitude. The explanation is found in the interval from which sample points were drawn. Since the absolute values of these points were all less than 1.0, higher-order terms in a given expression accounted for an increasingly tiny proportion of the variance in determining the final result. GP-UX and GP-SUX populations rapidly identified small trees that accounted for most of the variance generated by the lower-order terms and converged on them. Whilst they were theoretically large enough nevertheless to allow higher-order terms to be constructed, the fitness landscape in the space of these small trees was apparently quite rugged, and the populations generally became stuck on local optima. In contrast, standard GP was able to construct trees of great complexity which allowed it both to generate higher-order polynomial terms and to refine the coefficients it applied to them.

An obvious solution to this problem is to broaden the sample range such that values greater than 1.0 and less than -1.0 are considered. We did try this and whilst it did result in larger GP-UX and GP-SUX trees, it evidently introduced other problems since performance did not improve on any of the test functions. In the end we opted for the sample interval [-1.0, +1.0] simply to be consistent with that used by Koza in his symbolic regression studies[58][59].

A notable difference between the arithmetic and Boolean function sets we considered was that the former resulted in almost no inviable branches, whereas inviable code fragments arose frequently with the latter and were a pervasive feature of solutions. Inviable sub-trees appear to be useful to GP-UX and GP-SUX for a number of reasons. Firstly, they provide locations on which the search operators may act without affecting the individual's phenotype. Depending on the ratio of coding to non-coding material in the genome, they may therefore offer a means of fine-tuning the crossover and mutation operator probabilities at run-time. Secondly, inviable code allows the size and structure of a tree to vary in the sense that the coding branches of a tree may be associated with inviable sub-trees of varying size. For GP-UX and GP-SUX, this will be significant because it means that a population converging on trees of an overall given structure still has many options available to it in terms of the structure of the coding segments. This almost certainly contributed to the superior performances of deeper trees on the parity and multiplexer problems reported in the last chapter. Thirdly, introns allow trees to perform the function of small programs whilst avoiding the space of small programs from which, once converged, GP-UX and GP-SUX populations are unable to escape.

We observed very few inviable branches in any of the GP-UX or GP-SUX solutions we examined in these studies, and indeed the only way in which they could arise is via multiplication by 0. In their absence, GP-UX and GP-SUX are clearly far more constrained in the ways in which they attempt to solve the problem. In the problems we considered here, the space of smaller programs contained numerous trees that were able to compete successfully with their larger counterparts, with the result that the population converged prematurely around them. This was clearly demonstrated on the Koza function where, with the 1-bit binary function set, 99 of the 100 trees returned by GP-SUX and GP-UX at the ends of their runs were equivalent to $3x^2 + 3x$. Whilst expressions containing non-integer coefficients may not have been particularly rare even within the space of smaller programs, it is clear that almost none of them could compete with the function $3x^2 + 3x$, which therefore constitutes a powerful attractor.

How then should we address this problem? Four possible approaches are presented here:

- Specifying minimum tree depth.
- Explicitly coded introns.
- Combining standard and uniform crossover.
- "Smooth ADFs".

The first, and simplest, approach is to specify a minimum initial tree depth. There is a lower, as well as an upper, limit on the depth of tree that GP uniform crossover can generate and this, of course, is given by the depth of the shallowest tree in the population. If we specify a minimum tree depth at initialisation, it follows that no trees below this depth will be generated during the run, and the population is forced to search spaces of larger trees.

The minimum tree depth can be increased iteratively on the basis of performance on previous runs. If early runs with no minimum specified depth indicate that the system is frequently converging on small trees and failing to find a solution as a result, we increment the minimum depth (and possibly also the maximum depth) on later runs. The space of programs searched thus remains constrained but shifts over time towards larger trees. At the limit where $mindepth = maxdepth$, we have a system that is very similar to the GA-based algorithm described in Chapter 5. Although this will increase the wallclock time required to perform a run, an upper limit on bloat is nevertheless enforced and we should still be able to increase the tree depths by quite some way before the processor demands become equivalent to those of standard GP.

Our second alternative is to code introns explicitly. One way of doing this with arithmetic operators would be to add 2 bits controlling which, if any, of the arguments were considered. If only one of the bits is set, the value returned by the specified subtree is simply passed on unchanged. If neither argument is to be considered, the sub-tree could be replaced with a random ephemeral constant. As stated above, the use of introns allows small coding programs to be encased within larger ones without affecting phenotypic behaviour. It would be interesting to see whether the benefits that introns offer GP-UX and GP-SUX are sufficient to justify the increase in size of the search space that would result from the use of the extra bits.

Standard GP has proved impervious to all the attractors associated with small trees described in this thesis by virtue of its ability to bloat its way out of their vicinity. There is no reason why we shouldn't include standard GP crossover as a third search operator in our GP-UX and GP-SUX systems. To prevent excessive bloat, we would generally use it with a low probability, and possibly only later in the run. By this stage, GP-UX and GP-SUX may have discovered useful sub-tree schemata that could be reused at other locations throughout the tree. This is significant because whilst we want to escape the attractors present in the space of small trees, we still wish to keep program size as small as possible. In order to do both, it is imperative that we use the standard crossover operator sparingly, and that when we do use it, the larger trees generated stand a good chance of competing with the smaller trees in the population.

Perhaps the most promising area for development lies in the combination of GP-UX and GP-SUX with ADFs. We will return to this when we consider possible areas of further research in the final chapter, so only a brief outline will be given here. The efficiency with which GP-UX and GP-SUX have been shown to search the spaces of small trees suggests that they may prove very effective in generating small, useful sub-routines that may be used by a larger program. We could use different function sets, with varying degrees of interpolation, in each of the ADFs, allowing the main program to select the ones best suited to the task. We could also vary the search operators and representations used in the same way. This represents a large and potentially fruitful area of research that needs to be investigated.

7.8 Conclusion

This chapter has described one means of applying the sub-symbolic representation to continuous domains. This offers GP many more possibilities in terms of solutions that may easily be constructed. Of course, we cannot be sure that the number of solutions will increase as fast or faster than the size of the search space, and in most of these experiments we have paid the price since many of the functions proved to be extraneous.

The interpolated arithmetic functions reported in this chapter were developed primarily with GP-SUX in mind, and it is ironic that in some cases it was GP-SUX that seemed to encounter the most problems with them. That GP-SPM was only able to flip one of the bits in a function node per application will undoubtedly have led to difficulties in escaping from local optima, particularly when higher values of n were used. There are other problems with using a binary representation for parametric encoding such as this that have long been recognised in the GA community (see, for example, [79]) and that have led to the proposal of a number of alternatives such as Gray encoding and floating point parameters. In the next chapter, we examine the use of floating point parameters in interpolated arithmetic functions. In particular, we will be interested to see whether this allows GP-SUX to search more efficiently the space of small programs.

Chapter 8

Interpolating Arithmetic Functions with Real-Encoded Parameters.

In this, the final chapter of this thesis in which experimental results are reported, we demonstrate that the use of subsymbolic search operators is not restricted to binary representations. In these studies, we again attempt to tackle a number of symbolic regression problems using interpolated arithmetic functions, but in this case we use real-valued parameters to control the interpolation together with, in the case of GP-SUX, specialised crossover and mutation operators. And in keeping with the bulk of the work reported in this thesis we draw heavily on GA research for inspiration.

A number of researchers investigating parametric optimisation in GAs have argued for the use of real-encoded parameters [21, 85]. This requires the use of modified search operators that in many cases improve performance in spite of the potentially infinite search space associated with such a representation. Moreover, if we can obtain even reasonable performance from any of our three systems we will have derived an added benefit, since the burden of attempting to select a suitable degree of binary interpolation will be removed from the programmer.

In this chapter, then, we examine performance of real-encoded fully and partially-interpolated functions sets on some of the problems reported in the previous chapter. After the modest displays observed with the binary representation, we do not expect any spectacular results but it will be interesting to see whether similar improvements to those reported in GAs by Davis [21, 85] can be realised by our GP operators. In the next section we describe the manner in which such a representation is applied to our function nodes and the modifications necessary to allow the search operators to work on them. We then report the results of experiments conducted with partially and fully-interpolated function sets on the Koza polynomial and the three building block functions described in the previous chapter. The chapter concludes with a brief summary and a discussion of the results.

8.1 Encoding Interpolated Arithmetic Operators with Real-Valued Parameters

Encoding parameters for fully-interpolated arithmetic function using floating point values is very simple. At initialisation, each function node is specified by 3 floating point values generated randomly in the range [0, 1]. The tree is then evaluated as before. An example tree is shown in Figure 8.1.

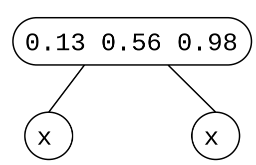

Figure 8.1: Example tree using real-valued parameter encoding.

The tree shown in Figure 8.1 evaluates the expression:

$$0.0112x^2 + 0.2548x + 0.0088$$

Encoding partially-interpolated functions is, of course, equally simple. Each node comprises a single real-valued encoding parameter plus an extra bit specifying whether the function interpolates addition and subtraction or multiplication and division.

Some modification of the uniform crossover and point mutation operators used by GP-UX and GP-SUX is necessary to allow them to operate on this representation. In both cases uniform crossover is applied to each node with 50% probability and then applied to each parameter (including the controlling bit in the case of partially-interpolated functions) *within* the node with 50% probability. In GP-UX the selected parameters are then exchanged between the two parents.

An operator analagous to GP-SUX acting on a real-valued representation should interpolate between the two exchanged parameters to generate offspring. In the experiments reported here, the algorithm is as follows. We first select a value, z, at random from within the interval $[p_1 - \epsilon, p_2 + \epsilon]$ where p_1 and p_2 are the two crossed parameters such that $p_1 \leq p_2$, and ϵ is a small value (we used $\epsilon = 0.05$ in these studies). The use of the ϵ term means that the offspring parameters generated by crossover will, in a small minority of cases, fall outside the range $[p_1, p_2]$ and this will reduce the rate of convergence. Our crossover operators all generated two offspring, and to do this with this implementation, we derived a second crossing parameter, z', such that $z' = p_2 - z$. Our offspring parameters, o_1 and o_2 are then given by $o_1 = z$ and $o_2 = p_1 + z' = p_1 + p_2 - z$. Obviously, this operation is only applied to real-valued parameters – in the case of partially-interpolated functions, the controlling bit is simply exchanged as before.

A single GP-PM operation is achieved simply by replacing the mutated parameter with another drawn at random from the interval [0, 1]. In the case of GP-SPM the mutated value is perturbed by a value drawn at random from a Gaussian distribution with mean 0 and standard deviation 0.325.

Both GP-SUX and GP-SPM may generate interpolating parameters outside the interval [0, 1]. If this happened, the parameters were set to the maximum or minimum values of 1 or 0.

Standard GP required no modifications to its search operators using this representation.

8.2 Experiments with the Real-Encoded Representation

We compared the performance of standard GP, GP-UX and GP-SUX on the Koza function $2.718x^2 + 3.1416x$ and the 3 building block functions described in Section 7.4.3.

We used almost the same parameters in these experiments as we did with the binary representation, the exception being the crossover and mutation probabilities for GP-UX and GP-SUX. These were, respectively, 0.6 and 0.3 which pilot studies indicated were close to optimal.

8.2.1 The Koza Polynomial, $2.718x^2 + 3.1416x$

The results with the fully-interpolated functions on the Koza polynomial $2.718x^2 + 3.1416x$ are given in Tables 8.1 and 8.2.

Table 8.1: Minimum effort required by standard GP, GP-UX and GP-SUX running the fully-interpolated real-encoded representation to solve the Koza function.

	Effort	Successes	Comp.
Standard	—	0	—
GP-UX	—	0	—
GP-SUX	291,000	14	12.57

Table 8.2: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Koza function with the fully-interpolated real-valued representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.0299	0.3	611	185
GP-UX	0.103	0.459	12.04	11
GP-SUX	0.02	0.247	12.17	11

The problem is clearly comparatively easy for GP-SUX to solve. It outperformed

GP-SUX using the 3-bit binary function set by a factor of about 4.5. Its performance was roughly equivalent to standard GP using the 2-bit binary function set, although it evaluated a fraction of the nodes over 50 runs and found far more parsimonious solutions. The best solution found by GP-SUX over the 50 runs scored a fitness of 0.0143, and is shown in Figure 8.2. We note that many of the parameters in this tree are close or equal to 1. This tree evaluates to $0.0152x^3 + 2.751x^2 + 3.107x + 0.0004$.

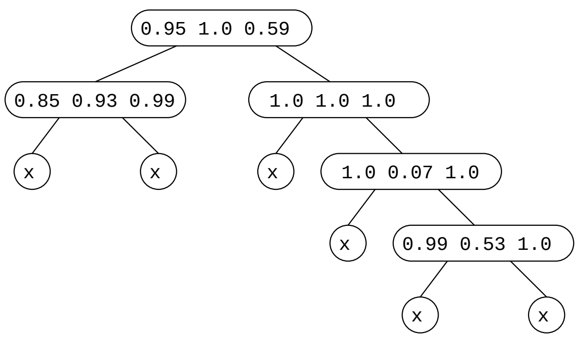

Figure 8.2: Solution to the Koza fitness function found by GP-SUX using the fully-interpolated real-valued parameter encoding. The parameters given in all the figures presented in this chapter are rounded to 2 decimal places for ease of display.

Perhaps predictably, neither standard GP nor GP-UX succeeded in finding a satisfactory approximation to the target function, and in fact GP-UX performed quite poorly. The best solution found by GP-UX scored a fitness of 0.1025 and is shown in Figure 8.3. The function obviously requires some pure arithmetic functions if it is to be solved with a small tree, and indeed many of the parameters in this tree are close to 1 or 0. Unfortunately, GP-UX could only generate new parameter values with point mutation and given that this operator simply replaced the mutated parameter with a value drawn at random from the range [0, 1], the probability of the replacement improving fitness was, in a fit tree, very small. Moreover, the likelihood of obtaining the value 0 or 1, and thereby reducing the degree of interpolation, was vanishingly small.

The results with the partially-interpolated function set – shown in Tables 8.3 and 8.4 – are considerably more encouraging. All three operators succeeded in solving the problem, and with a minimum effort of 46,400, GP-SUX outperformed all the vari-

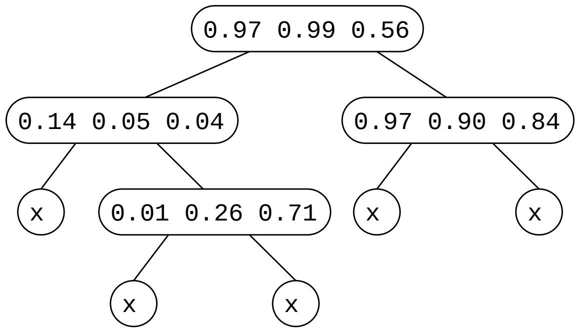

Figure 8.3: Solution to the Koza fitness function found by GP-UX using the fully-interpolated real-valued parameter encoding.

ous operator-representation pairings reported in this thesis by a factor of 2.5. That partially-interpolated functions should outperform their fully-interpolated counterparts on this problem was to be expected from the results of the previous chapter, but the magnitude of the improvement on the part of GP-SUX, especially in view of the vastly increased search space, is particularly striking. A probable cause is that Hamming cliffs, which doubtless pose some problems for GP-SPM as the degree of interpolation increases, are not inherent in these operators, and GP-SUX with GP-SPM is able to make very fine adjustments to highly fit solutions.

Both GP-UX and standard GP, whilst still able to solve the problem with real parameters, are notably impaired compared to when binary partially-interpolated functions are used. Again, it is clear that the operators do not permit them to make directed movements around so large a search space.

8.2.2 The Small 2-bit Building Block Function

The results with the fully-interpolated representation for the smaller 2-bit "building block" function described in Section 7.4.3 are given in Tables 8.5 and 8.6

GP-UX found 6 satisfactory solutions to this problem, whilst GP-SUX found 10. All the solutions found by GP-UX, and 8 of those found by GP-SUX, were isomorphic to the target tree shown in Figure 7.13. The 2 remaining solutions found by GP-SUX contained 9 nodes.

Table 8.3: Minimum effort required by standard GP, GP-UX and GP-SUX running the partially interpolated real-encoded representation to solve the Koza function.

	Effort	Successes	Comp.
Standard	814,400	5	329.0
GP-UX	210,600	16	21.25
GP-SUX	46,400	39	21.46

Table 8.4: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the Koza function with the partially-interpolated real-valued representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.03	0.17	670	238
GP-UX	0.03	0.25	22.2	12
GP-SUX	0.05	0.35	16.09	10

Table 8.5: Minimum effort required by standard GP, GP-UX and GP-SUX running the fully-interpolated real-encoded representation to solve the smaller 2-bit building block function.

	Effort	Successes	Comp.
Standard	—	0	—
GP-UX	592,000	6	7
GP-SUX	329,400	10	7.4

That GP-UX was able to solve this problem and not the Koza function with the fully-interpolated representation is due to the fact that this function is derived from

Table 8.6: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and total number of nodes evaluated over 50 runs of the smaller 2-bit building block function with the fully-interpolated real-encoded representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.018	0.11	586.3	226
GP-UX	0.018	0.098	9.05	11
GP-SUX	0.025	0.142	9.1	10

a tree consisting entirely of fully-interpolated arithmetic functions for which GP-UX has a reasonable chance of constructing approximations. Given the operators with which it did this here, however, it is unsurprising that performance should be impaired relative to when the 2-bit binary function set was used. In contrast, an improvement in performance on this problem was observed for GP-SUX with the real-encoded representation relative to when the 2-bit binary function set was used.

The best tree found by GP-UX scored a fitness of 0.0146 and is shown in Figure 8.4. The parameters in this tree show only the most superficial resemblence to those of the target tree, and the tree evaluates to:

$$\frac{0.041x^6 + 2.01x^5 + 30.22x^4 + 143.73x^3 + 357.47x^2 + 210x + 47.47}{27x^2 + 630x + 73}$$

.

The best tree found by GP-SUX scored 0.015408, and is shown in Figure 8.5. The parameters in this tree are slightly closer to those in Figure 7.13, and the tree evaluates to:

$$\frac{2x^6 + 32.01x^5 + 241.78x^4 + 1105.95x^3 + 2395.78x^2 + 1386.23x + 318.78}{583x^2 + 4056x + 517}$$

.

We note in both cases that the order of the polynomial is the same as the target

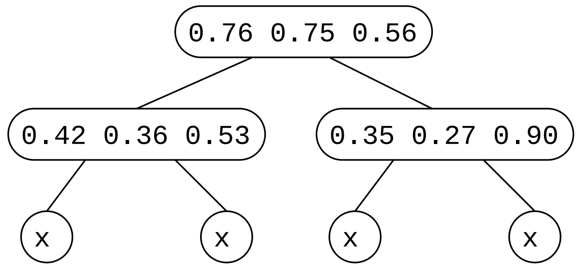

Figure 8.4: Solution to the smaller 2-bit building block function found by GP-UX using the fully-interpolated real-valued parameter encoding.

function given in Section 7.4.3.

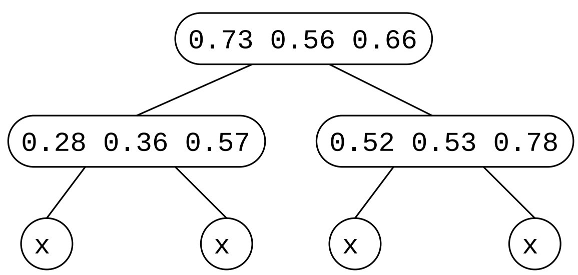

Figure 8.5: Solution to the smaller 2-bit building block function found by GP-SUX using the fully-interpolated real-valued parameter encoding.

None of the operators succeeded in solving the small 2-bit building-block problem with the partially-interpolated function set. The results pertaining to the sub-optimal solutions found are given in Table 8.7.

The results now have a ring of familiarity to them. In terms of fitness, standard GP found better solutions on average than either GP-UX or GP-SUX, but they were all significantly larger and required over an order of magnitude more function node evaluations.

On the other hand, we can see that whilst GP-UX outperformed GP-SUX with the binary encoding, the reverse holds here. This will certainly be due in part to the considerable disadvantages under which GP-UX is operating when using the real-encoded function sets, but GP-SUX's improved performance with this representation

Table 8.7: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the small 2-bit function with the partially-interpolated real-valued representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.04	0.33	627.0	257
GP-UX	0.11	2.61	18.24	16
GP-SUX	0.06	1.28	19.37	16

implies that any problems associated with Hamming cliffs have been overcome.

8.2.3 The Large 2-bit Building Block Function

No solutions were found to this problem on any of the runs using the fully-interpolated representation. The remaining results for the larger 2-bit "building block" function are shown in Table 8.8.

Table 8.8: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the larger 2-bit building block function with the fully-interpolated real-encoded representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.025	0.2	439.2	222
GP-UX	0.037	0.269	5.44	10
GP-SUX	0.0244	0.266	6.12	10

Performance across all three conditions was notably better than when the binary representation was used. GP-SUX found the best solution – a 9 node program that scored 19 hits. This tree, shown in Figure 8.6, evaluates:

$$\frac{470.43x^7 + 12,790.13x^6 + 119,720.49x^5 + 485,542.49x^4 + 835,215.06x^3 + 552,128.82x^2 + 277,789.39x + 40,348.18}{(231x + 475)(279x^2 + 1600x + 171)}$$

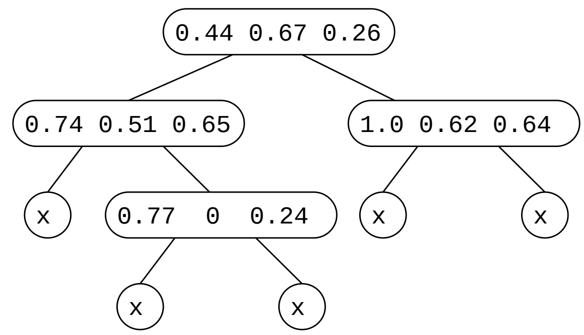

Figure 8.6: Sub-optimal solution to the larger 2-bit building block function found by GP-SUX using the fully-interpolated real-valued parameter encoding.

The best solution found by GP-UX scored 14 hits and is shown in Figure 8.7. This tree evaluates:

$$\frac{0.02x^6 + 10.42x^5 + 1235.05x + 5092.43x^3 + 3000.9x^2 + 2152.8x + 440.57}{27x^2 + 6232x + 873}$$

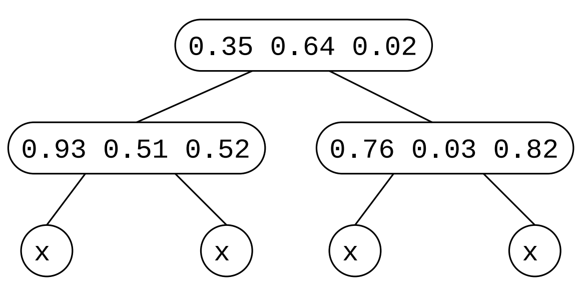

Figure 8.7: Sub-optimal solution to the larger 2-bit building block function found by GP-UX using real-valued parameter encoding.

No solutions were found to the problem with the partially-interpolated function set either. The results for the sub-optimal runs are given in Table 8.9. These results are very similar to those obtained from the smaller 2-bit building block function with the (real-encoded) partially-interpolated representation. Standard GP again found very

Table 8.9: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the large 2-bit function with the partially-interpolated real-valued representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.03	0.25	619.8	227
GP-UX	0.10	5.12	19.5	18
GP-SUX	0.12	1.21	13.84	13

large solutions that evaluated to significantly better approximations of the target function than those found by GP-UX or GP-SUX. Although the best solution found by GP-UX was slightly better than that found by GP-SUX, we can see that on average GP-UX found significantly poorer approximations. Comparisons with the average fitness of sub-optimal solutions presented in Table 7.16 also reveal that GP-UX is impaired by the use of the real-valued encoding whilst, encouragingly, GP-SUX is not.

8.2.4 The 3-bit Building Block Function

No solutions to this problem were found with the fully-interpolated function set on any of the runs. The results for the 3-bit building block function are shown in Table 8.10.

The best solutions found by each of the operators were all better than those found when the binary representation was used, although the differences are not as pronounced as for the larger 2-bit function. GP-UX and GP-SUX tended to converge on trees comprising a single function and two terminal nodes. However, the best solutions in either case were isomorphic to the target solution.

The best solution found by GP-SUX scored 12 hits, and is shown in Figure 8.8. This tree evaluates:

Table 8.10: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total nodes evaluated over 50 runs of the 3-bit building block function with the fully-interpolated real-valued encoding.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.035	0.143	132.6	231
GP-UX	0.048	0.078	3.28	10
GP-SUX	0.049	0.076	3.48	10

$$\frac{149.9x^6 + 309.4x^5 + 512.3x^4 + 451.4x^3 + 541.7x^2 + 360.9x + 254.3}{1328x^2 + 612x + 747}$$

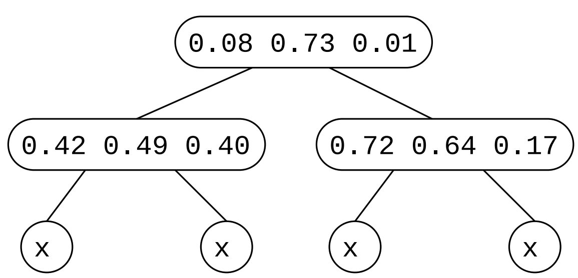

Figure 8.8: Sub-optimal solution to the 3-bit building block function found by GP-SUX using real-valued parameter encoding.

The best solution found by GP-UX scored 11 hits, and is shown in Figure 8.9. This tree evaluates:

$$\frac{460.3x^6 + 953.3x^5 + 2062.4x^4 + 2234.2x^3 + 2092.7x^2 + 1054.7x + 393.3}{5561x^2 + 3036x + 1139}$$

For comparison, the target tree, with its parameters rounded to 2 decimal places, is shown in Figure 8.10.

We can see that, with the possible exception of the right hand node at the lower level, the evolved trees bear little resemblance, in terms of the parameters they contain,

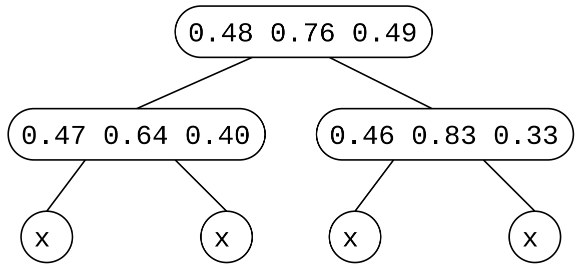

Figure 8.9: Sub-optimal solution to the 3-bit building block function found by GP-SUX using real-valued parameter encoding.

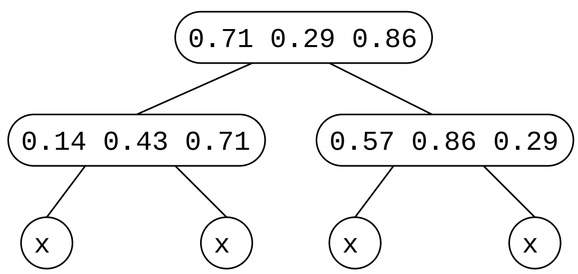

Figure 8.10: The target 3-bit tree with real-encoded parameters.

to the target tree. Even though the correct structure was discovered, the problem remained difficult for both operators. It would appear to be highly epistatic in that fit values for a given parameter will only be recognised as such if the other parameters also happen to be fit. Failing that, the system can only attempt to make the best of what it has (i.e. fine tune the parameters on which it converges) and this will inevitably lead to the population becoming trapped on a local optimum. This in turn results in a function that only approximates the target over the test range – examination of the functions evaluated by the best solution trees shows that they bear little resemblance to the target function and will generalise poorly. To be fair, however, both the larger 2-bit and the 3-bit building block functions are clearly complex and difficult to solve, and perhaps we should not be too surprised that none of the operators were able to locate perfect solutions.

The results of running these experiments with the partially-interpolated function set are given in Tables 8.11 and 8.12.

Table 8.11: Minimum effort required by standard GP, GP-UX and GP-SUX running the partially interpolated real-encoded representation to solve the 3-bit function.

	Effort	Successes	Comp.
Standard	880,000	5	1019.0
GP-UX	—	0	—
GP-SUX	—	0	—

Table 8.12: Best and average fitnesses of the best solutions found at the end of each run, the average complexity of these solutions and the total number of nodes evaluated over 50 runs of the 3-bit function with the partially-interpolated real-valued representation.

	Best Fit.	Av. Fit	Comp.	Nodes ($\times 10^6$)
Standard	0.02	0.07	729.1	267
GP-UX	0.07	0.14	12.36	12
GP-SUX	0.07	0.13	12.52	13

Standard GP, which had previously only succeeded in finding one solution with the 3-bit partially-interpolated binary representation, solved the problem five times and recorded an improvement in performance of a factor of 2. The size of these solutions in both the real- and binary-encoded conditions indicates that acceptable approximations, at least over the sampling range, are easier to find in the space of very large partially-interpolated programs. Quite why standard GP's performance should have profitted from the real-valued encoding is not clear, although it is likely that the wider available range of interpolating parameters offered benefits.

These regions of the search space would have been barely sampled by the initial GP-UX and GP-SUX population, far less explored in later generations, and neither of these operators succeeded in locating solutions in the space of smaller programs where they inevitably converged. On average, however, both found better solutions to this problem than they did to either of the 2-bit building block functions.

8.3 Summary

- The performance of GP-SUX on the Koza polynomial, $2.718x^2 + 3.1416x$ was significantly improved with the real-encoded representation relative to any of the binary encodings. Performance improved by a factor of 4.5 with the fully-interpolated representation, and a factor of 2.5 with the partially-interpolated encoding. With a minimum effort of 46,400, GP-SUX running the partially-interpolated real encoding also recorded the best performance of any of the operators on this problem.

- GP-SUX also recorded a performance gain with the fully-interpolated real-encoding on the small 2-bit building block function, and in general seems to have benefitted from the use of this representation in terms of the quality of the sub-optimal solutions found.

- GP-UX's performance was largely impaired relative to that with the binary encoding. This is not a surprising result – the real-encoded representation was developed with GP-SUX in mind and the GP-UX and GP-PM operators were unable to perform anything like a refined search of what was a very large program space.

- Standard GP performed surprisingly well on all of the problems, particularly when the partially-interpolated functions were used. In every case it employed a radically different strategy to GP-UX or GP-SUX, bloating rapidly into the space of large trees where it was able to make small refinements to fit solutions.

8.4 Conclusion

The experiments reported in this chapter have provided an illustration of an alternative to the binary representation for function nodes used throughout the main body of the thesis. In general, the results for GP-SUX showed some improvements over those when the binary representation was used in Chapter 7 and they suggest that pairing

GP-SUX with a real-encoded representation may serve to further smooth the fitness landscape of many problems. This demonstrates that, with appropriately modified search operators, GP-SUX is capable of searching potentially infinite program spaces very efficiently.

These experiments are intended to provide a brief overview of a single alternative representation, and have therefore not explored many of the possible parameter variations. As with all the studies reported in this thesis, there are a number of parameters, such as the per-node crossover probability, that could be varied to bias the nature of the search performed. With GP-SUX we could also vary the ϵ parameter such that a wider range of values are accessible under crossover. Additionally, the amount by which a value is perturbed under GP-SPM could be changed by altering the standard deviation of the Gaussian distribution, or even by using an alternative distribution of random values.

As in the previous chapter, the experiments reported here are the result of a great many pilot studies we ran with many different functions. The results of both the experiments reported here and the other pilot studies indicate that whilst GP-SUX and GP-UX are able to solve simple, low-order polynomial functions, performance deteriorates rapidly as the problems become more complex. Standard GP found solutions to complex problems such as the building block functions by exploring the space of very large trees where, it has been argued [71], smoother fitness gradients exist. We made a number of attempts to counteract the pressure towards very small trees in GP-UX and GP-SUX including the use of inverse parsimony pressure (i.e. if two trees scored approximately the same fitness, the *larger* was favoured with some probability) and the use of broader sampling intervals for the fitness points so that higher-order polynomial terms accounted for more of the variance. Unfortunately, whilst we succeeded in generating somewhat larger trees, none of these techniques yielded any gains in performance.

This almost certainly reflects a characteristic of the operators that, for these problems at least, will often work against them. Standard GP performs a local search not only of the program space (in that small proportions of the code are exchanged between parents during crossover) but also an increasingly local search of the *phenotype* space since, as the run progresses, changes are on average performed at nodes that are more and more distant from the root. The result is that standard GP is able to inch progressively towards an expression that fits the target function over the sample points. In contrast, GP-UX and GP-SUX continue to modify nodes close to the root of the tree, particularly via their mutation operators, and this will have a sizeable corresponding impact on the values returned. Our experiments suggest that over continuous domains, this is not a particularly good strategy. If correct, our hypothesis also suggests that GP-UX and GP-SUX would benefit from a modification of their mutation operators such that mutations are biased in favour of lower nodes as the run progresses. A full investigation of this theory would require many experiments to establish reasonable biases, whether these biases are problem-specific, and indeed whether such biases could be learned by the system (see chapter 9 for a more extensive discussion of bias-induction). Such experiments are outwith the scope of this thesis, and we offer this hypothesis as a suggestion for future work.

Chapter 9

Conclusions

This thesis has tested performance of two recently proposed search operators for genetic programming – GP uniform crossover and point mutation. We have also developed a novel sub-symbolic node representation that, when used in conjunction with these operators, allows both global search and small, directed movements around the search space. Our results have been, for the most part, very encouraging. Performance on the even-n-parity problem was significantly better than any comparable algorithm of which we are aware. Our operators also found solutions to a number of symbolic regression functions that were at least as good as those found by standard GP. Moreover, the nature of the operators meant that program size was severely curtailed. Consequently many of the solutions generated by GP-UX and GP-SUX were over an order of magnitude smaller than those found by standard GP, and were found with over an order of magnitude fewer node evaluations. The conclusion is that the ability to perform both global and local search of a constrained search space proves beneficial to GP on a number of problems.

However, we have done more than simply describe yet another operator that performs well on the parity problems. A number of properties of the search operators and representation as used here led us to consider a number of aspects of GP search that have hitherto been overlooked, and these produced several surprising results. The principal contributions made by the work reported in this thesis are summarised in the next section.

9.1 Contributions Made by this Thesis

GP-UX is a search operator that performs an initially global search, followed by an increasingly local search. This is in contrast to standard GP crossover, which has been shown to be capable of only a local search [101, 107]. We have argued that GP-SUX performs an initially global search in a manner similar to that of GP-UX, but that later in the run it performs a far more local search, especially when the function set used is large. This allows fine-tuning of highly-fit candidate solutions and if global optima are in the current vicinity, allows optimal solutions to be found rapidly.

However, we have avoided arguing that GP-SUX and GP-SPM are therefore better operators in general. Although a number of interpretations of the No Free Lunch theorems [25] are currently the subject of some debate, we should be very surprised if this were so. GP-SUX and GP-SPM perform a very specific search strategy, namely global search *followed by* very local search. The emphasis is significant because other problems may respond better to other strategies, including an initially local search followed by a more global search as well as sustained global or local search. The primary strength of GP-UX and GP-SUX is therefore that they offer alternative strategies that are simply not provided by standard GP crossover. We will elaborate further on this in the next section.

Our desire to minimise constraints in the use the sub-symbolic representation meant that we were often forced to consider very large function sets that included a number of functions of dubious utility. This was strictly counter to the received wisdom on the subject, which advises the use of small function sets from which extraneous functions are excluded. We might therefore have expected to observe good performance in spite of, rather than because of, the function set used. We believe that the results of many of our experiments at least question the validity of this assumption. GP used the functions it was given in ways that were often surprising, and for reasons that were not immediately clear. The arity-3 Boolean function set used in Chapters 5 and 6 comprised 256 functions, many of which were highly esoteric when viewed against the problem in question. By controlling their arguments, GP was consistently able to modify the behaviour of many of these functions so that they behaved in the manner desired. Nor were improvements in performance restricted to GP-UX and GP-

SUX. Standard GP using small populations and the full arity-3 function set exhibited performance far superior to when many smaller function sets were used. A detailed investigation of this is sadly outwith the scope of this thesis. Instead we argue that the issue of function sets deserves more attention than it has so far been given.

The issue of bloat is also relevant to this research. Of course, it could be argued that bloat was, by definition, never going to pose serious problems in GP-UX and GP-SUX runs. However, other attempts to contain bloat through the use of techniques such as non-destructive crossover and depth-dependent and size-fair operators have produced, at best, modest improvements in overall performance. Again, we must question the efficiency of these operators. With very few exceptions, they ignore homology, exhibit biases towards lower nodes and perform a comparatively local search of the program space. An increasing body of evidence indicates that these are important issues that we ignore at our peril.

Finally, we made a number of hypotheses concerning the structure of the program spaces that were searched. To say that the space of possible programs is large is almost to make a trite observation. As a result, theoretical analysis of these spaces, and the movement of populations around them, has proved far more difficult than it has in, for example, GAs. We should therefore be very careful when applying terms such as unimodal or rugged to GP search spaces.

For example, using Juels and Wattenberg's [52] demonstration that the multiplexer problems are theoretically unimodal to explain the comparative ease with which standard GP solves them is highly suspect. Their proof assumes that the necessary replacement sub-tree is always available, and does not take into account the likelihood of inserting the correct sub-tree at the correct location. Taking the argument to its logical extreme, we could reason that *all* GP spaces are unimodal provided we are allowed to assume a mutation operator that is capable of generating an optimal sub-tree which is then used to replace the correct suboptimal one in the parent tree.

The proof is thus of questionable use in standard GP, and we have shown that its application to GP-UX and GP-SUX search spaces is at best inappropriate. Instead we argue that hypotheses pertaining to the structure of search spaces should take into account the transmission function (i.e. the points accessible given the operators and representation used), the genetic diversity exhibited by the population, and the effects that these factors will have on the relative proximity of improved solutions. We should also bear in mind that whilst GP operators search the program space, the resultant offspring are evaluated in terms of their movements around the phenotype space. Thus, standard GP's good performance on a particular problem (e.g. the 6-multiplexer) may be taken as an indication that fitter solutions, potentially leading to a global optimum, are syntactically close to each other in the program space. On the other hand, good performance by GP-SUX or GP-UX on a problem that is difficult for standard GP may imply that solutions are located in very local regions of the search space. Fitness gradients guiding search towards these regions do exist, but they are too coarse for standard GP's local search operators to follow.

9.2 Further Work

The contributions made by this thesis address diverse issues in GP, and suggest a number of possible directions for future research. We summarise these in this section.

9.2.1 Extending GP-SUX

This thesis has given a detailed analysis of the performance of GP-UX and GP-UX on two classes of problem – Boolean function induction and symbolic regression of polynomials. Ideally we would like to examine the behaviour of these operators using a variety of different problem domains. Examples include, but are not limited to, robot navigation, image analysis and the artificial ant problem. Such problem domains have proved difficult for standard GP in the past, and it would be interesting to see whether the addition of a global search component yields any advantages for search.

Different domains mean different function sets and this calls into question the form of interpolation that we would like the sub-symbolic representation, when used, to perform. For example, implementations tackling the artificial ant problem generally include terminals that advance the ant in a particular direction or change the direction of travel. The sub-symbolic representation could usefully employed in these cases to permit small adjustments to be made to the direction or distance of movement. In general, all we require is a function that deterministically or probabilistically maps parent nodes on to offspring functions in some meaningful way. This approach further constrains the operators and may prove advantageous since the function set need no longer be of a fixed (and generally large) size. Such a technique be worth considering other forms of interpolation if the size of the binary function set is obviously impeding search, and in Appendix A we outline a very simple implementation for the Ant problem.

9.2.2 Alternative Search Strategies

We have seen that GP-UX and GP-SUX perform a particular form of search strategy, namely an initially global search followed by an increasingly local search as the population begins to converge. We have taken pains to avoid concluding from the good performance exhibited on many of the problems described here that such a strategy is necessarily optimal for all problem domains. There would seem to be little point in performing a global search of the fitness landscape if direct paths leading from initial random points to an optimal solution exist. Conversely, if the search space generated by GP-SUX or GP-UX proves to be very rugged, a very local search performed at any stage of the run will fall foul of the numerous local optima (Cf. the 2-bit building block function described in Section 7.4.3, which appears to possess this property). It is conceivable that yet other strategies will be beneficial, for example a local search interspersed with occasional large leaps if the population converges on a sub-optimal solution.

A simple means of tuning the globality of search performed by GP-UX and GP-SUX in the early stages of a run is to vary the per-node crossover probability. If this is perturbed away from the value of 0.5 used in all the studies reported here, offspring probabilistically inherit most of their genetic material from one parent. The result is that offspring more closely resemble their parents. This additional parameter could be tuned using meta-populations or, alternatively, varied in later runs in response to poor performance in earlier runs. The latter strategy can be thought of as "learning to learn", and we explore the idea further below.

9.2.3 Function Sets

Our results, particularly on the parity problems, seem in some cases to contradict the received wisdom on the subject of function set size. GP has proved capable of employing apparently extraneous functions in many surprising ways. The relationship between performance and the function set used is evidently more complex than previously thought. Further work on this area, using different function sets and different problem domains is clearly required.

The role of introns in GP-UX and GP-SUX also needs to be addressed. Inviable and inoperative code fragments may serve a purpose by preventing premature convergence on sub-optimally small programs. They may also provide storing points for terminals that might otherwise be lost from the population. Our Boolean function sets contained many functions that ignored one or more of their arguments and therefore produced inviable regions in the tree. These functions were a prevalent feature of many of the solutions we examined, suggesting that they offer some evolutionary advantage. We also observed that when we used function sets which gave rise to fewer inviable regions (i.e. the interpolated arithmetic functions described in Chapters 7 and 8, programs converged on solutions of a much smaller complexity. Whether this was an effect of the fitness space or the fact that introns were less likely to occur remains unclear. It could, however, be investigated by removing the functions that produce inviable sub-trees from the Boolean function set (and constraining the GP-SUX and GP-SPM operators so that they can work on these representations), or adding functions to the interpolated arithmetic function sets that promote introns.

9.2.4 Bias Induction

Whilst the effects of adding extra functions to the function set remains unclear, it seems reasonable to assume that many of the functions included in an *unbiased* function set such as the full arity-2 or arity-3 Boolean sets will be extraneous on some problems. These function sets are unbiased because they make no a priori assumptions about which functions will be useful and which will hinder search. Nevertheless, it is clear from the experiments reported throughout this thesis that some functions are more useful than others in solving a given problem, since these were consistently observed in the solutions found.

We may make use of this observation by implementing an *initialisation bias* that probabilistically favours those functions frequently found in correct solutions over those that are not. One way of doing this is as follows. On the first run, the population is initialised in an unbiased manner as normal. If the run is successful, each function in the function set is given a point for each time it occurs in the solution. On subsequent runs, trees are initialised with a bias towards the fitter functions found in the solution tree. Successful solutions found on subsequent runs are used to update the fitness table of the function nodes. If necessary, the restriction could be relaxed so that solutions above a threshold fitness are used to update the fitness table, although this may cause problems if these solutions are located in deceptive regions of the space since it may bias search away from the areas where optimal solutions exist. We should expect solutions to be found more rapidly in later runs, as an appropriate bias is gradually learned.

We could also use a similar technique to tune many of the other problem parameters. If solutions are being found regularly, we could try reducing the population size. Alternatively, if search frequently stagnates, we could try running larger populations for fewer generations. In the case of GP-UX and GP-SUX we could also, as stated above, tune the per-node crossover probabilities so that the operators perform a more local search, or progressively bias the application of the mutation operator towards the leaf nodes.

9.2.5 Combining GP-SUX with other Techniques

Our implementations of GP-SUX have been deliberately simple. We have used the search operators on standard tree-based representations without ADFs, module libraries or any other proposed representation modifications. However, there is no reason why GP-SUX should not be used in conjunction with these techniques. An interesting application would be the use of GP-SUX to evolve useful ADFs for symbolic regression problems. Many of the problems in Chapters 7 and 8 seemed to require only a small amount of interpolation. Unfortunately, the use of the interpolated function set generated a bias against the pure arithmetic nodes, and this bias was proportional to the resolution of the function set used. What we have in mind is the use of an interpolated function set in one or more ADFs, and a pure arithmetic function set in the result branch. Small, interpolated building blocks could thus be evolved and combined with pure arithmetic operators in the main body of the program. This should overcome the problem of premature convergence on small trees, and the general bias towards interpolated functions. It may also serve to reduce the epistatic nature of many problems when such representations are used.

We could also use the sub-symbolic representation with other techniques such as the hill-climbing and simulated annealing hybrid approaches proposed by O'Reilly and Oppacher [94]. The representation may improve the quality of the local search performed and, since the number of individuals that are evaluated at each generation is small, enhance performance further on problems that are amenable to local search.

Finally, we could adopt pareto-optimisation and disassortive mating techniques to attenuate the problems caused by premature convergence on small programs. An interesting variant on this would be a modification of Ryan's pygmies and civil servants [117] approach, which maintains two reproductive pools, the first based purely on fitness and the second on fitness and size. Ryan's research was concerned with reducing program size, but we could equally well mate populations of "giants and civil servants" such that larger programs were additionally favoured.

9.3 Summary

This thesis has provided an extensive examination of two search operators for GP – GP uniform crossover and point mutation – and proposed extensions to each based on a novel function node representation to give GP smooth uniform crossover and smooth point mutation. We have demonstrated good performance on a number of problems, most notably the even-n-parity problem which is rendered trivially easy with this approach. These experiments raised a number of additional issues concerning the nature of function spaces, the role of the search operators and representation in determining the structure of these spaces, and the effects of function set size on performance. We have argued that these areas deserve more attention than they have previously been accorded. Further enhancements to the operators are also possible, and we have suggested a number of possible directions for future research.

Appendix A

Applying the Sub-Symbolic Approach to Other Problems – An Example.

One feature inherent in the sub-symbolic representation as it has been used throughout this thesis is large function sets. If a binary representation is used, a function set in which functions are specified by n bits must necessarily contain 2^n functions if the search operators are to be used without constraints. Furthermore, of course, if a real-valued encoding is used the size of the function set is constrained only by the limitations of the machine's representation of floating point values. In Chapters 5 and 6 we saw that, far from impeding GP-SUX, this actually seemed to facilitate search. However, the evidence for this effect in general was rather weakened by the results of Chapters 7 and 8, and it is likely that many problems exist for which the function set should be kept as small as possible.

In this appendix we wish to demonstrate that it is possible to use search operators such as GP-UX in conjunction with a representation that, whilst retaining many of the desirable properties of the sub-symbolic representation, is not constrained *a priori* to be of a given size. Specifically, this approach dispenses with the use of parameters that define a function's behaviour and that, as a side-effect, define the function set's size. We believe this approach will have general application provided that the functions used are in some way related (that is, that attempting to interpolate any two functions from the set is guaranteed to result in a meaningful offspring node). We will illustrate this technique using of a well-known benchmark test – the Ant problem [50] – as an example.

The object of the ant problem is to induce a program that can successfully navigate an artificial agent along a trail placed on a 32x32 toroidal grid. Most GP implementations utilise a function set consisting of the `Prog2`, `Prog3` and `IfFoodAhead` functions. The `IfFoodAhead` function returns true and executes its first sub-branch if the square immediately adjacent to the agent in its current direction of travel contains food (i.e. it is a point on the trail), and returns false and executes the second sub-branch otherwise. `Prog2` and `Prog3` are, respectively, arity-2 and arity-3 functions that execute their sub-branches in sequence. The most commonly-used terminal set consists of the arity-0 `Advance`, `RotateLeft` and `RotateRight` functions. `Advance` moves the agent forward one square in the current direction of travel, whilst `RotateLeft` and `RotateRight` alter the direction by 90 degrees anti-clockwise and clockwise respectively. Each operation takes one time unit, and the fitness function is to minimise the number of time steps required to navigate the trail in its entirety. The problem has been extensively studied and is known to be very difficult for GP to solve. Langdon's extensive work [63, 67, 68] on the problem has established that its fitness landscape is very rugged, and it is for this reason that we think sub-symbolic operators may prove useful.

The technique proposed here is simple – we define a set of complete look-up tables specifying the results of applying each search operator to any given node or nodes. In the case of the ant problem, in contrast to those used elsewhere in this thesis, the most likely candidates for interpolation are the terminal nodes, since it is these that control the movement of the ant around the grid. Ideally, as stated above, these terminals should be related in such a way that the results of interpolation should be meaningful. This arguably does not apply to the canonical terminal set described above (consider, for example, attempting to interpolate between `Advance` and `RotateLeft`, or even `RotateLeft` and `RotateRight`) so we propose the use of an alternative. For simplicity, this comprises the perhaps minimal set of {`AdvanceLeft, AdvanceDiagonalLeft, AdvanceAhead, AdvanceDiagonalRight, AdvanceRight`}. The most notable aspect of this set is the inclusion of the two diagonal movement functions which move the ant one square forward and one square to the left or right.

Interpolating between these terminals is now straightforward. For example, the result of crossing `AdvanceAhead` and `AdvanceLeft` is `AdvanceDiagonalLeft`. A slight problem arises when no exact intermediate exists for two parental functions, e.g. `AdvanceLeft` and `AdvanceDiagonalRight`. If two offspring are being generated, a simple solution is to assign the two closest terminals – `AdvanceAhead` and `AdvanceDiagonalLeft` in this case – to them randomly.

Point mutation is also simple. The node undergoing mutation is replaced with its "nearest neighbour" from the terminal set. If two or more candidates exist, one is selected at random.

This illustration is very simple, but the main points are hopefully clear. The technique described above allows the user far more control over the size and content of the function set than the parametric approach. In particular it permits the use of small, focussed function and terminal sets – indeed, given that the transition table will most likely have to be completely specified by hand, it will be tedious to implement for larger sets. Furthermore, it offers many possibilities for extension. We do not need, for example, to be rigidly deterministic in the application of the transition rules. We may permit other transitions to occur with some probability, particularly if the use of "nearest neighbours" is causing the population to converge prematurely. We whould like to point out that this technique is proposed purely as a suggestion for future research – we have not tested it and therefore cannot comment on its efficacy. Nevertheless, we feel confident that it will provide a useful and much needed addition to the array of methods available to GP users in tackling difficult problems.

Bibliography

[1] R. Aler. Immediate Transference of Global Improvements to all Individuals in a Population in Genetic Programming Compared to Automatically Defined Functions for the Even-5 Parity Problem. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 60–70, Paris, 14-15 Apr. 1998. Springer-Verlag.

[2] L. Altenberg. Emergent phenomena in genetic programming. In A. V. Sebald and L. J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241. World Scientific Publishing, 1994.

[3] L. Altenberg. The evolution of evolvability in genetic programming. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.

[4] P. J. Angeline. Genetic programming and emergent intelligence. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[5] P. J. Angeline. Genetic programming's continued evolution. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 1, pages 1–20. MIT Press, Cambridge, MA, USA, 1996.

[6] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[7] P. J. Angeline. Subtree crossover: Building block engine or macromutation? In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. 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.

[8] T. Baeck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computations*. Oxford Press, 1997.

[9] T. Baeck and H. Schwefel. Evolutionary Computation: an Overview. In *Proceedings of the 1996 IEEE 3rd International Conference on Evolutionary Computation*, pages 20–29. IEEE Press, 1996.

[10] D. Ballard. *Introduction to Natural Computation*. MIT Press, 1997.

[11] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, Jan. 1998.

[12] T. Blickle. Evolving compact solutions in genetic programming: A case study. Tik-report, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, 1996.

[13] T. Blickle and L. 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).

[14] C.Cavalier-Smith. *The Evolution of Genome Size.* John Wiley and Sons, 1985.

[15] C.Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.* Murray, London, UK, 1859.

[16] K. Chellapilla. Evolutionary programming with tree mutations: Evolving computer programs without crossover. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 431–438, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[17] K. Chellapilla. A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 23–31, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[18] X. Chen and S. Hurst. A comparison of universal-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks. *IEEE Transactions on Computers*, 31:140–147, 1982.

[19] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[20] J. Daida, S. Yalcin, P. Litvak, G. Eickhoff, and J. Politom. Of metaphors and darwinism: Deconstructing genetic programming's chimera. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[21] L. Davis and S. Coombs. Genetic algorithms and communication link speed design: Theoretical considerations. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 252 – 256, 1987.

[22] R. Dawkins. *The Selfish Gene.* Oxford University Press, Oxford, 1976.

[23] P. D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[24] T. E. Dunning and M. W. Davis. Evolutionary algorithms for natural language processing. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 16–23, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[25] D.Wolpert and W.Maccready. No free lunch theorems for optimisation. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[26] D. Fogel. *Evolutionary Computation.* IEEE Press, New York, 1995.

[27] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley and Sons, 1966.

[28] J. A. Foster and T. Soule. Comments on the intron/exon distinction as it relates to genetic programming and biology. Position paper at the Workshop on Exploring Non-coding Segments and Genetics-based Encodings at ICGA-97, 21 July 1997.

[29] R. Friedberg. A learning machine, part i. *IBM Journal of Research and Development*, pages 2–13, 1958.

[30] R. Friedberg, B. Dunham, and J. North. A learning machine, part ii. *IBM Journal of Research and Development*, pages 282–287, 1959.

[31] C. Fujiki and J. Dickinson. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In J. J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the second international conference on Genetic Algorithms*, pages 236–240, MIT, Cambridge, MA, USA, 28-31 July 1987. Lawrence Erlbaum Associates.

[32] C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. 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.

[33] C. Gathercole and P. Ross. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference.* MIT Press, 28–31 July 1996.

[34] C. Gathercole and P. Ross. Tackling the Boolean Even N Parity Problem with Genetic Programming and Limited-Error Fitness. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[35] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[36] D. E. Goldberg, K. Deb, and B. Korb. Messy genetic algorithms revisited: studies in mixed size and scale. *Complex Systems*, 4:415–444, 1990.

[37] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[38] S. Handley. Predicting whether or not a 60-base DNA sequence contains a centrally-located splice site using genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 98–103, Tahoe City, California, USA, 9 July 1995.

[39] K. Harries and P. Smith. Exploring alternative operators and search strategies in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. 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.

[40] S. Haykin. *Nueral networks, a comprehensive foundation.* Macmillan, 1994.

[41] J. Holland and J. Reitman. Cognitive systems based on adaptive algorithms. In D. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems.* Academic Press, 1978.

[42] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[43] D. C. Hooper, N. S. Flann, and S. R. Fuller. Recombinative hill-climbing: A stronger search method for genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 174–179, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[44] L. Huelsbergen. Toward simulated evolution of machine language iteration. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[45] H. Iba, H. de Garis, and T. Sato. Genetic programming using a minimum description length principle. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994.

[46] C. Igel and K. Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[47] T. Ito, H. Iba, and S. Sato. Depth-dependent crossover for genetic programming. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 775–780, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[48] T. Ito, H. Iba, and S. Sato. Non-destructive depth-dependent crossover for genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 71–82, Paris, 14-15 Apr. 1998. Springer-Verlag.

[49] T. Ito, H. Iba, and S. Sato. A self-tuning mechansim for depth-dependent crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3.* MIT Press, Cambridge, MA, USA, 1999.

[50] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/ tracker system. In C. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, volume Santa Fe Institute Studies in the Sciences of Complexity, Proceedings Volume 10, Reading, MA, 1991. Addison-Wesley.

[51] T. Jones. Crossover, macromutation and population-based search. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[52] A. Juels and M. Wattenberg. Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. Technical Report CSD-94-834, Department of Computer Science, University of California at Berkeley, USA, 18 July 1995.

[53] S. A. Kauffman. *Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, Oxford, 1993.

[54] M. J. Keith and M. C. Martin. Genetic programming in C++: Implementation issues. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.

[55] K. E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[56] K. E. Kinnear, Jr. A perspective on the work in this book. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

[57] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 Aug. 1989.

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

[59] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[60] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming 3*. MIT Press, Cambridge, MA, USA, 1998. Forthcoming.

[61] K. J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis of Koza's. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, California, USA, July 1995. Morgan Kaufmann.

[62] W. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[63] W. B. Langdon. Better trained ants. In R. Poli, W. B. Langdon, M. Schoenauer, T. Fogarty, and W. Banzhaf, editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 11–13, Paris, France, 14-15 Apr. 1998. The University of Birmingham, UK.

[64] W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. 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.

[65] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pan, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, 23-27 June 1997.

[66] W. B. Langdon and R. Poli. Boolean functions fitness spaces. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[67] W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris, 14-15 Apr. 1998. Springer-Verlag.

[68] W. B. Langdon and R. Poli. Why ants are hard. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. 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.

[69] W. B. Langdon and R. Poli. Why "Building Blocks" Don't Work on Parity Problems. Technical Report CSRP-98-17, University of Birmingham, School of Computer Science, 13 July 1998.

[70] W. B. Langdon and R. Poli. Boolean functions fitness spaces. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 1–14, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[71] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In U.-M. O. L. Spector, W. B. Langdon and P. J. Angeline, editors, *Advances in Genetic Programming 3*. MIT Press, Cambridge, MA, USA, 1999.

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

[73] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. 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.

[74] J. Maynard-Smith. *Evolutionary Genetics*. Oxford University Press, Oxford, 1994.

[75] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:113–115, 1943.

[76] N. McPhee and N. Hopper. Analysis of genetic diversity through population history. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[77] N. F. McPhee and J. 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, 15-19 July 1995. Morgan Kaufmann.

[78] Z. Michalewicz. A hierarchy of evolution programs: An experimental study. *Evolutionary Computation*, 1(1):51–76, 1993.

[79] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, second edition, 1994.

[80] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[81] M. Mitchell. *Introduction to Genetic Algorithms*. MIT Press, 1996.

[82] M. Mitchell and S. Forrest. Genetic algorithms and artificial life. Technical Report Working Paper 93-11-072, Santa Fe Institute, Santa Fe, NM, 1993.

[83] M. Mitchell, S. Forrest, and J. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In F. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992.

[84] M. Mitchell, J. Holland, and S. Forrest. When will a genetic algorithm outperform hill-climbing? In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances In Neural Information Processing Systems 6*. Morgan Kaufmann, 1994.

[85] D. Montana and L. Davis. Training feed-forward neural nets using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, 1989.

[86] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[87] P. Nordin. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[88] P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[89] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995.

[90] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[91] U.-M. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 199–206, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.

[92] U. M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. Working Paper 94-02-001, Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.

[93] U.-M. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem, 9-14 Oct. 1994. Springer-Verlag.

[94] U.-M. O'Reilly and F. Oppacher. A comparative analysis of GP. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, Cambridge, MA, USA, 1996.

[95] D. Patrick and T. York. Forecasting system behaviour under exceptional conditions using symbiotic co-evolution of genetic algorithms and genetic programs. In P. Osmera, editor, *Proceedings of the 3rd International Mendel Conference on Genetic Algorithms, Optimization problems, Fuzzy Logic, Neural networks, Rough Sets (MENDEL'97).*, pages 117–122, Brno, Czech Republic, 25-27 June 1997.

[96] J. E. Perry. The effect of population enrichment in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 456–461, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[97] R. Poli. Genetic programming for feature detection and image segmentation. In T. C. Fogarty, editor, *Evolutionary Computing*, number 1143 in Lecture Notes in Computer Science, pages 110–125. Springer-Verlag, University of Sussex, UK, 1-2 Apr. 1996.

[98] R. Poli. Genetic programming for image analysis. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[99] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*, http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/, 19–30 Aug. 1996. Nagoya University, Japan.

[100] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In T. Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[101] R. Poli. Is Crossover a Local Search Operator? Position Paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.

[102] R. Poli. Sub-Machine-Code GP: New Results and Extensions. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, LNCS, Goteborg, Sweeden, 26-27 May 1999. Springer-Verlag.

[103] R. Poli. Submachinecode GP: New results and extensions. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 65–82, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[104] R. Poli and J.Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 1(1), 2000. to appear.

[105] R. Poli and W. B. Langdon. Genetic Programming with One-Point Crossover. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, 23-27 June 1997.

[106] R. Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. 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.

[107] R. Poli and W. B. Langdon. On the Search Properties of Different Crossover Operators in Genetic Programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. 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.

[108] R. Poli and W. B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3), 1998.

[109] R. Poli and W. B. Langdon. Sub-Machine-Code Genetic Programming. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 13. MIT Press, Cambridge, MA, USA, 1999.

[110] R. Poli, J. Page, and W. B. Langdon. Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. Forthcoming.

[111] I. Rechenberg. Evolution strategy. In J. Zurada, R. Marks, and C. Robinson, editors, *Computational Intelligence – Imitating Life*, pages 147–159. IEEE Press, 1994.

[112] G. Robinson and P. McIlroy. Exploring some commercial applications of genetic programming. In T. C. Fogarty, editor, *Evolutionary Computing*, number 993 in Lecture Notes in Computer Science. Springer-Verlag, Sheffield, UK, 3-4 Apr. 1995.

[113] J. Rosca. Generality versus size in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[114] J. Rosca and D. H. Ballard. Causality in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 256–263, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[115] J. P. Rosca. Analysis of complexity drift in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[116] J. P. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms.* PhD thesis, University of Rochester, Rochester, NY 14627, Feb. 1997.

[117] C. Ryan. Pygmies and civil servants. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press, 1994.

[118] C. Ryan. GPRobots and GPTeams - competition, co-evolution and co-operation in genetic programming. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 86–93, MIT, Cambridge, MA, USA, 10–12 Nov. 1995. AAAI.

[119] C. Ryan and P. Walsh. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *Proceedings of ParCo'95.* North-Holland, 1995.

[120] H. Schwefel. *Evolution and Optimum Seeking.* John-Wiley and Sons, New York, 1995.

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

[122] T. Soule and J. A. Foster. Code Size and Depth Flows in Genetic Programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[123] T. Soule and J. 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.

[124] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. 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.

[125] W. Spears, K. de Jong, T. Baeck, D. Fogel, and H. de Garis. An overview of evolutionary computation. In *Proceedings of the European Conference on Machine Learning*, 1993.

[126] L. Spector and A. Alpern. Induction and recapitulation of deep musical structure. In *Proceedings of the Joint International Conference on Artificial Intelligence (IJCAI'95): Workshop on Music and AI*, Montreal, Quebec, Canada, 1995. Morgan Kaufmann.

[127] G. Syswerda. Uniform Crossover in Genetic Algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic ALgorithms*. MOrgan Kaufmann, 1989.

[128] W. A. Tackett. Genetic programming for feature discovery and image discrimination. In *International Conference on Genetic Algorithms*, 1993.

[129] W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[130] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[131] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[132] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[133] M. L. Wong and K. S. Leung. Evolving Recursive Functions for the Even-Parity Problem Using Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[134] S. Wright. Evolution in mendelian populations. *Genetics*, 16:97–159, 1931.

[135] A. Wu and R. Lindsay. Empirical studies of the genetic algorithm with noncoding segments. *Evolutionary Computation*, 3(2):121–147, 1995.

[136] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[137] B.-T. Zhang and H. Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.