An Investigation of Supervised Learning in Genetic Programming

Chris Gathercole

Ph.D.
University of Edinburgh
1998

Abstract

This thesis is an investigation into Supervised Learning (SL) in Genetic Programming (GP). With its flexible tree-structured representation, GP is a type of Genetic Algorithm, using the Darwinian idea of natural selection and genetic recombination, evolving populations of solutions over many generations to solve problems. SL is a common approach in Machine Learning where the problem is presented as a set of examples. A good or fit solution is one which can successfully deal with all of the examples.

In common with most Machine Learning approaches, GP has been used to solve many trivial problems. When applied to larger and more complex problems, however, several difficulties become apparent. When focusing on the basic features of GP, this thesis highlights the immense size of the GP search space, and describes an approach to measure this space. A stupendously flexible but frustratingly useless representation, Anarchically Automatically Defined Functions, is described. Some difficulties associated with the normal use of the GP operator Crossover (perhaps the most common method of combining GP trees to produce new trees) are demonstrated in the simple MAX problem. Crossover can lead to irreversible sub-optimal GP performance when used in combination with a restriction on tree size. There is a brief study of tournament selection which is a common method of selecting fit individuals from a GP population to act as parents in the construction of the next generation.

The main contributions of this thesis however are two approaches for avoiding the fitness evaluation bottleneck resulting from the use of SL in GP. To establish the capability of a GP individual using SL, it must be tested or evaluated against each example in the set of training examples. Given that there can be a large set of training examples, a large population of individuals, and a large number of generations, before good solutions emerge, a very large number of evaluations must be carried out, often many tens of millions. This is by far the most time-consuming stage of the GP algorithm. Limited Error Fitness (LEF) and Dynamic Subset Selection (DSS) both reduce the number of evaluations needed by GP to successfully produce good solutions, adaptively using the capabilities of the current generation of individuals to guide the evaluation of the next generation. LEF curtails the fitness evaluation of an individual after it exceeds an error limit, whereas DSS picks out a subset of examples from the training set for each generation.

Whilst LEF allows GP to solve the comparatively small but difficult Boolean Even N parity problem for large N without the use of a more powerful representation such as Automatically Defined Functions, DSS in particular has been successful in improving the performance of GP across two large classification problems, allowing the use of smaller population sizes, many fewer and faster evaluations, and has more reliably produced as good or better solutions than GP on its own.

The thesis ends with an assertion that smaller populations evolving over many generations can perform more consistently and produce better results than the 'established' approach of using large populations over few generations.

Acknowledgements

I'd like to take this opportunity to thank the many people who have assisted, coerced, guided, bullied, ridiculed, encouraged, or otherwise contributed to my completing this thesis. Many thanks to my supervisor, Dr. Peter Ross, for knowing the answers to many questions. Many thanks to Dave Corne for lending an ear, and likewise to his better-groomed replacement, Emma Hart. Many thanks to the attendees and organisers of the GP96 and GP97 conferences for many inspiring conversations and talks, and especially to Bill Langdon for his many helpful comments. Many thanks to the denizens of E17 and E19 for the endlessly diverting chats. Many thanks to many other people. And last but not least, many thanks to SERC, who became EPSRC, for funding nearly the whole of my PhD with grant number 93314680.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

C. S. Gathercole
Edinburgh
March 22, 1998

Contents

Abstract ii

Acknowledgements iii

Declaration iv

List of Figures xi

List of Tables xii

I Overview of Genetic Programming 1

1 Introduction 2

1.1 Why Look At GP 2

1.2 Why Read this Thesis 3

1.3 What is in this thesis 3

1.4 Search Algorithms 4

1.5 GP: The What and the How 11

1.6 Basics of Supervised Learning 18

2 Ramifications of a Large and Messy Problem 20

2.1 Why Choose the Thyroid Problem 21

2.2 Starting Point 23

2.3 First Impressions 25

2.4 Early Snags and Decisions 25

2.5 Longer Term Snags, Workarounds, and Hindsight 34

2.6 Applying GP to a problem . . . 37
2.7 Summary . . . 42

II A Closer Look At Genetic Programming 43

3 GP Tree Representation 44
3.1 The standard GP tree . . . 44
3.2 Counting Trees . . . 46
3.3 Extending the Function and Terminal Sets . . . 52
3.4 Summary . . . 58

4 GP Tree Recombination and Selection 59
4.1 Crossover and the MAX problem . . . 60
4.1.1 Why Restrict Tree Size . . . 61
4.1.2 The MAX Problem . . . 66
4.1.3 Crossover in GP . . . 77
4.1.4 Experiment Details . . . 78
4.1.5 Results . . . 80
4.1.6 Analysis of Crossover . . . 90
4.1.7 Discussion of MAX problem . . . 92
4.1.8 Summary . . . 96
4.2 Tournament Selection . . . 98
4.2.1 Various Selection Methods . . . 98
4.2.2 Some Effects of Tournament Selection . . . 101
4.3 Discussion . . . 106

III Genetic Programming and Supervised Learning 108

5 Making use of the Training Set in GP 109
5.1 Training Sets in Machine Learning . . . 110
5.2 Selecting Training and Test Sets . . . 113

5.3 Approaches for Evolutionary Algorithms . . . 114
5.4 Approaches for GP in this thesis . . . 116

6 Dynamic Subset Selection **117**
6.1 Subset Selection Methods . . . 118
6.2 Historical Subset Selection (HSS) - the algorithm . . . 118
6.3 Dynamic Subset Selection (DSS) - the algorithm . . . 119
6.4 Random Subset Selection (RSS) - the algorithm . . . 122
6.5 GP Details . . . 123
6.6 The 'Large and Messy' Thyroid Problem . . . 123
6.7 Thyroid Results . . . 126
6.8 A Smaller Problem: TicTacToe Endgames . . . 132
6.9 TicTacToe Results . . . 133
6.10 A quick summary of results from other runs . . . 133
6.11 Smaller Populations over More Generations . . . 134
6.12 DSS Discussion . . . 135

7 Limited Error Fitness **139**
7.1 LEF - the algorithm . . . 140
7.2 GP Details . . . 145
7.3 The Even N Parity problem . . . 146
7.4 Results . . . 147
7.5 LEF Discussion . . . 155

8 Small Populations, Many Generations **160**
8.1 Solving the TicTacToe problem with a small population . . . 161
8.1.1 GP parameters . . . 161
8.1.2 LEF parameters . . . 163
8.1.3 DSS parameters . . . 165
8.1.4 Comparison of GP, GP+LEF, and GP+DSS, on TicTacToe . . . 165
8.2 Solving the Thyroid problem with a small population . . . 167
8.3 Discussion . . . 169

IV Summary and Conclusion **172**

9 Further Work **173**

10 Summary **176**

11 Conclusion **179**

Glossary **182**

Bibliography **184**

List of Figures

1.1 Linear String $\rightsquigarrow$ tree 15

1.2 LISP-like Program $\rightsquigarrow$ tree 15

1.3 Algebraic Expression $\rightsquigarrow$ GP tree 15

2.1 A 'Large and Messy' Problem 24

2.2 Easy Thyroid subproblem 29

3.1 Structure of IFLTE subtree – *(If Less Than Or Equal to)*, arity=4 . . . 45

3.2 Example of an ADF tree 53

3.3 example AADF tree 56

4.1 Spike and Decay with Parsimony 64

4.2 Optimal Tree for MAX-depth-4-{+}{1} 68

4.3 Optimal Tree for MAX-depth-4-{*,+}{1} 68

4.4 Optimal Tree for MAX-depth-4-{*,+}{0.5} 68

4.5 Optimal Tree for MAX-depth-5-{*,+}{0.25} 68

4.6 Optimal Tree for MAX-depth-4-{*,/}{0.9} 69

4.7 An optimal tree for MAX-nodes-81-{*,+}{0.25} 70

4.8 Avg gens needed by MAX-depth-D runs using Crossover 81

4.9 StdDev of gens needed by MAX-depth-D runs using Crossover 81

4.10 Failure of MAX-depth-D runs using Crossover 82

4.11 Failure of MAX-depth-D runs using Crossover & Mutations 82

4.12 Sub-Optimal tree for MAX-depth-5-{*,+}{0.5} 83

4.13 Sub-Optimal tree for MAX-depth-5-{*,/}{0.9} 83

4.14 Sub-Optimal tree for MAX-nodes-81-{*,+}{0.25} 84

4.15 Gens needed for MAX-nodes-N-{*,+}{0.25} with Crossover 86
4.16 Gens needed for MAX-nodes-N-{*,+}{0.5} with Crossover 86
4.17 Gens needed for MAX-nodes-N-{*,+}{1} with Crossover 86
4.18 Gens needed for MAX-nodes-N-{*,+}{0.25} with Crossover & Mutations . 87
4.19 Gens needed for MAX-nodes-N-{*,+}{0.5} with Crossover & Mutations 87
4.20 Gens needed for MAX-nodes-N-{*,+}{1} with Crossover & Mutations . . 87
4.21 Success of MAX-nodes-N-{*,+}{0.25} runs with Crossover 88
4.22 Success of MAX-nodes-N-{*,+}{0.5} runs with Crossover 88
4.23 Success of MAX-nodes-N-{*,+}{1} runs with Crossover 88
4.24 Success of MAX-nodes-N-{*,+}{0.25} runs with Crossover & Mutations . 89
4.25 Success of MAX-nodes-N-{*,+}{0.5} runs with Crossover & Mutations . . 89
4.26 Success of MAX-nodes-N-{*,+}{1} runs with Crossover & Mutations . . . 89
4.27 Average Parent Selection Frequency 103
4.28 Average Distribution of Repeated Selections 103
4.29 Average Likelihood of Non-Selection 104
4.30 Average Number of Unchecked Parents 104

6.1 Simple GP tree for class 1 and class 2 cases 125
6.2 GP tree for class 3 cases 127
6.3 Errors made on Thyroid test set by DSS & RSS 128
6.4 Errors made on Thyroid test set by GP & GP+HSS 128
6.5 Errors made on Thyroid training set by GP & GP+DSS 129
6.6 Errors made on Thyroid training set by GP & GP+DSS 130
6.7 Dynamics of DSS: showing the varying difficulty and age weights 130

7.1 Best-of-gen Fitness of GP on Even N Parity, N=6, Pop=400 148
7.2 Best-of-gen Fitness of GP+LEF on Even N Parity, N=6, Pop=400 . . . 148
7.3 Best-of-gen Bushiness of GP on Even N Parity, N=6, Pop=400 149
7.4 Best-of-gen Bushiness of GP+LEF on Even N Parity, N=6, Pop=400 . . 149
7.5 Fitness Std Dev of standard GP on Even N Parity, N=6, Pop=400 . . . 150
7.6 Fitness Std Dev of GP+LEF on Even N Parity, N=6, Pop=400 150

7.7 Error Limit of GP+LEF on Even N Parity, N=6, Pop=400 151
7.8 Evals per Gen of GP+LEF on Even N Parity, N=6, Pop=400 151

List of Tables

3.1 The number of possible GP trees with N nodes 51

4.1 Details of the optimal trees for MAX-nodes-N-{*,+}{0.25} 74

4.2 Details of the optimal trees for MAX-nodes-N-{*,+}{0.5} 75

4.3 Details of the optimal trees for MAX-nodes-N-{*,+}{1} 76

6.1 Distribution of Classes in Thyroid Data 119

6.2 Best results by GP (and NN) on Thyroid Problem 131

6.3 Best results by GP on TicTacToe problem 133

6.4 Further Thyroid Training and Test Results 135

7.1 Summary of results from runs on the Even N Parity problem 152

8.1 Looking for a good Population Size for the TicTacToe problem 163

8.2 Looking for a good Tournament Size for the TicTacToe problem 163

8.3 Looking for good LEF Pause Parameters on TicTacToe problem 164

8.4 Comparison of GP, GP+DSS, GP+LEF on TicTacToe Problem 166

8.5 Results for GP and GP+DSS on Thyroid Problem 168

Part I

Overview of Genetic Programming

Chapter 1

Introduction

1.1 Why Look At GP

Genetic Programming is an evolutionary-based search technique which can be applied to many different types of problems, and it has had some notable successes. Gruau has developed a method for encoding 'growth' instructions for Neural Networks, evolving both the structure and weights of networks, [Gruau *et al.* 96]. Howley has used GP to "discover near optimal control laws for the minimum time reorientation for a spacecraft", [Howley 96]. Garces-Perez *et al* have used GP to produce "superior" results in the facility layout problem, [Garces-Perez *et al.* 96]. Walsh and Ryan have used GP to develop a technique for the "Autoparallelisation of Sequential Programs", [Walsh & Ryan 96].

As can be seen in these examples, GP is a flexible, widely applicable algorithm. It is simple, easy to explain and understand, with a huge potential for solving large and complex problems beyond the ken of the programmer. The fact that GP does not often achieve this potential gives rise to an intriguing area of study.

A large part of GP's appeal stems from its simplicity. Using its basic but flexible tree-structured representation, GP is capable of solving some difficult problems with little input or external knowledge. However, it soon becomes clear that GP can be made faster, more efficient, and more effective, by relaxing the simplicity requirement and instead looking at how to allow GP to take more advantage of the problem structure, or properties of its own evolving population of solutions. In particular, in Supervised

Learning problems, the speed and capability of GP is highly dependent on the size and make up of the training data, i.e. the way the problems are presented to GP as a set of examples.

1.2 Why Read this Thesis

This thesis is not aimed at convincing the ignorant that GP is best in certain situations. Instead it dodges that issue in favour of looking at ways of improving standard GP and the use of GP in the particular area of Supervised Learning classification problems. GP has many obvious but circumventable weaknesses, most notable among these are its rapacious demands for computer memory and CPU time. There is lots of scope for improvement. This thesis highlights some weaknesses of GP with Supervised Learning, and describes some procedures which can improve GP's performance. In effect, this thesis adds some utilities to a tool-box which can be used to repair GP when it doesn't work well, or needs a boost.

1.3 What is in this thesis

This chapter introduces GP and the entire thesis. It looks at GP from two viewpoints: how it fits in with other search algorithms, in Section 1.4, and the mechanics of how GP works, in Section 1.5. Section 1.6 looks at the idea of Supervised Learning. Chapter 2 takes a look at some of the lessons learned whilst using GP, highlighting several difficulties that could have been avoided, and some of which are dealt with in this thesis.

Chapter 3 looks at some aspects of the tree-based representation in GP, including the huge size of the search space, and Automatically Defined Functions (ADF) where GP can evolve its own representation. Chapter 4 looks at some aspects of GP operators, including restrictions on tree size, the adverse interaction of Crossover (a standard GP operator) and restricted tree size, and the use of Tournament Selection for choosing parents from the population.

Chapters 5 to 8 contain most of the research done for this thesis, looking at a variety of methods for dealing with the training set in supervised learning, taking advantage of the current abilities of the GP population to reduce the computational effort necessary to find good solutions. The two main methods are Dynamic Subset Selection (DSS), where a different subset of the training set is selected for each generation of GP, and Limited Error Fitness (LEF), where each individual in the population is only allowed to make a limited number of errors before its fitness evaluation is curtailed. Chapter 8 concentrates on the beneficial use of much smaller population sizes to tackle the same problems as those in earlier chapters, needing many more generations but requiring less computational effort.

Chapter 9 describes what could or should be done to follow up the work in this thesis. There is a summary in Chapter 10, and a conclusion in Chapter 11. Following the concluding chapter there is a Glossary of many of the terms and acronyms used in this thesis, including a page reference to where each is first mentioned or defined in the thesis.

But first, looking at GP in the context of other computer-based search algorithms...

1.4 Search Algorithms

For a given problem, a computer-based search algorithm may be needed to find good or optimal solutions. Problem solving is equivalent to searching for solutions. The problem may be beyond the capabilities of current knowledge of direct, calculus-based solutions, or perhaps the problem owner is simply lazy.

Choosing or designing a representation, along with a method for moving from one solution to another or a description of the interrelationships between solutions, defines a Search Space, which is simply the collection of all possible 'reachable' solutions to the given problem. To be sure, the vast majority of possible solutions are likely to be very bad ones. But, for a problem to be solvable, the representation has to be sufficiently powerful that the search space contains at least one good or acceptable solution, and it must be possible to 'reach' such solutions.

A measure of the merit of each particular solution has also to be codified in some way. The computer-based search algorithm has to be given a method for identifying acceptable solutions when they are discovered by the search process, i.e. a measure of how good or 'fit' a particular solution is relative to other solutions, or perhaps an absolute measure of the fitness of a solution. By the same token, there should be some way of identifying unacceptable solutions.

A search algorithm can make use of earlier results to guide its search through the space. The simplest two, Enumeration and Random search, shown below, use no feedback from earlier search results. Alpha-Beta pruning allows Enumeration to use earlier results to shorten the search. Hill Climbing and Simulated Annealing keep a record of the current best individual and use it as guide to further search. More complex approaches, such as TABU search, and Evolutionary Algorithms such as GP, maintain a record of several individuals to help guide their later searches.

Without Feedback

The most straightforward approach for finding an optimal solution is to enumerate all possible solutions to a problem, i.e. check every possible solution in the search space, and pick the best one. If the problem is to find the largest of a small set of numbers, the Enumeration approach is perfectly satisfactory but, alas, it should be obvious that this isn't going to be the approach of choice for most other problems. If the search space is finite, the Enumeration approach will be able to find the optimal solution, but it will only know which it is when it has completed the enumeration. All but the simplest of problems have infinite, or finite but extremely large search spaces. It will be either impractical (i.e. it will take too long) or impossible (i.e. it will take forever) to search them in their entirety. With GP, the search space is often finite but enormous, increasing massively in size with every increase in permitted tree size (see Section 3.2).

Random search is perhaps the simplest option for searching large search spaces. Random solutions are generated and tested until a sufficiently good one is found or a sufficiently large sample of the search space has been made. There is no guarantee that this approach will find an optimal solution unless it is allowed to run for an infinite time. It is highly susceptible to the content of the search space. If there are very few non-bad solutions, Random search is unlikely to find any useful solutions. However it can be used to form an impression of the distribution of solutions throughout the space, giving an idea of the possible effectiveness of searching via other methods.

With Feedback

One common use of the Enumeration approach is in game-playing programs. The computer looks ahead in the game tree at all possible moves and their consequences and picks the best move, i.e. the one least likely to lead to a loss, or most likely to lead to a win. The addition of Alpha-Beta pruning is required to allow this approach to work on any but the most trivial of games. In this specialised problem area, Alpha-Beta pruning can reduce the size of the search space by allowing certain regions to be ignored on the basis that they can't possibly contain acceptable solutions, using the information gained in an earlier part of the search. Even with Alpha-Beta pruning, it is not possible to exhaustively search the entire game tree for chess. The addition of some human expert chess knowledge encoded in the program allows the search to be curtailed still further with some educated guesswork.

The Enumeration approach with Alpha-Beta pruning and some expert chess knowledge, running on a very fast computer with dedicated chess circuitry, known as Deep Blue, [Tan 95, Hsu *et al.* 95], has successfully taken on, drawn with, and beaten the human world chess champion, Gary Kasparov (considered to be the best ever player in the history of chess) over several competitive games, though it lost the tournament overall. The chess world awaited with interest the next incarnation of Deep Blue (Deeper Blue) with essentially the same algorithm but even faster hardware, which it was generally acknowledged would overcome even Kasparov to become world chess champion (unless the powers-that-be in the World Chess Organisations got their act together in time to change the rules to prevent non-humans from winning the title). Sure enough, Deeper Blue was better and did cause Kasparov all sorts of problems, even going on to win the competition on points, winning several games along the way. Interestingly, the main difficulty Kasparov had was not that Deeper Blue was playing especially wonderful chess (it wasn't bad, though), but that Deeper Blue was deciding on its moves so quickly that Kasparov wasn't able to much thinking during his opponent's clock time, and consequently was put under a great deal of pressure leading to numerous major un-Kasparov-like mistakes.

Hill Climbing, a form of neighbourhood search, is perhaps the simplest modification to random search which makes use of previous search results to guide its current search. An initial random solution is generated and tested. Instead of then generating a completely new solution, as would be the case with purely random search, one or more close variants of the current solution are generated and tested. The best variant (or perhaps an equally good variant, if none are actually better) then becomes the focus of the search, and one or more variants are generated from this new solution, and tested, and so on. If after several iterations no improvement is found, the search is considered to have ended, and the current solution is the best one found during the entire search. This method can be very fast at finding the optimal solution in certain search spaces. Some modifications to Hill Climbing, described below, have produced benchmark solutions (i.e. the best solutions found so far by any method) to some quite difficult problems, [Ross & Corne 95]. However, in its simple form described here, Hill Climbing can quickly become trapped on sub-optimal hills, i.e. it reaches a region in the search space where all variants of the current focus solution are worse, but the focus solution is not the optimal solution. Another way of looking at this situation is that there is no way, using the idea of close variants, to move from the current solution to the optimal solution, where each variant is no worse than the last.

Simulated Annealing (SA), [Press *et al.* 92], is an approach which attempts to overcome the difficulty just mentioned for Hill Climbing. As with Hill Climbing, a random solution is generated and tested. Rather than only choosing as-good or better variants to be the next focus, SA can accept worse variants with some probability. The probability and (in some versions of SA) the randomness of the variations change with time, so that SA becomes less likely to accept worse variants after the algorithm has been running for a while. Eventually the size of the variations reaches zero, and the search is considered to have ended at the current focus solution. This more complex approach is much less likely to 'get stuck' on sub-optimal hills than simple Hill Climbing, and has been used by O'Reilly, [O'Reilly & Oppacher 94a], to match GP on several problems, using the same tree-based GP representation, described below in Section 1.5.

Hill Climbing (HC) can be made more flexible by removing the requirement that the best variant becomes the new focus solution. With Stochastic Hill Climbing (SHC), a better variant becomes the new focus with a certain probability, and it is even possible that a worse variant might become the focus solution with a certain (smaller) probability, as with SA. This allows the algorithm to 'travel' across the search space without necessarily getting 'trapped' on small sub-optimal hills.

A comparison of GP, Stochastic Iterated Hill Climbing (SIHC, described below), and SA, in [O'Reilly & Oppacher 96], shows that the simple hill climbing algorithms work well with the tree-based GP representation and powerful mutation operators. Hybrids of GP and SIHC and SA can improve upon standard GP. A comparison of Genetic Algorithms (GA, described below), SA, and SHC on several real timetabling problems, using equivalent operators, [Ross & Corne 95], shows that SA and SHC can outperform GA.

With Feedback and Memory

TABU search is a metaheuristic which can be added to algorithms such as Hill Climbing. One weakness of Stochastic Hill Climbing is that the search might traverse the same part of the search space repeatedly, wastefully re-visiting and re-testing solutions. This is especially true when reaching the top of a 'hill' in the search space; the search has nowhere to go except back upon its earlier route. With TABU, a list is kept of forbidden moves. When the underlying search algorithm makes an allowed move, that move or something abstracted from it is added to the TABU list. The TABU list is normally of fixed length and so loses a move from its end which then becomes as acceptable move again. The list of forbidden moves helps stop the Hill Climbing search from getting stuck on sub-optimal hills, enabling it to explore other regions of the search space.

Although not really a modification of the Hill Climbing algorithm, Iterated Hill Climbing is nonetheless a powerful approach. As its name suggests, it consists of repeated runs of the Hill Climbing algorithm, starting from a different random solution each time, likewise Stochastic Iterated Hill Climbing (SIHC). This is another way of avoiding getting stuck on sub-optimal hills. The best solution can then be taken from several independent searches. As already mentioned, Hill Climbing is very quick. Iterated Hill

Climbing can carry out a reasonable search of the search space in much less time than that required by the Evolutionary Algorithms described below.

Take the simple Stochastic Iterated Hill Climbing algorithm, and expand its one focus solution into a collection of several focus solutions, called a population. Test all the individuals in the population, using a fitness function, which assigns a score to each individual based on how well it solves the problem in hand. Then construct some variants of the better solutions in the population, test them, and insert them into the population by replacing some of the existing worse solutions. What you now have is a simple model of evolution through Natural Selection (or unnatural selection, if you prefer). Better solutions are likely to 'survive' (i.e. remain in the population) long enough to reproduce (i.e. have variants made), and worse solutions are likely to be 'killed off' (i.e. replaced by new solutions). This approach is known generally as an Evolutionary Algorithm (EA).

The 'child' solutions, i.e. variants of the existing 'parent' solutions, can be produced in several ways. A simple random variation of the parent can be made, known as Mutation, i.e. a copy of the parent with small random changes. Copies of two or more parent solutions can be combined in a simple way known generally as Crossover to produce one or more child solutions which contain a mixture of features copied from the parent solutions, though this term covers many different types of combination. The solutions in the population can be regarded as a very simplistic form of genetic material, analogous to chromosomes.

EAs come in many different flavours. Evolutionary Programming (EP), [Fogel 92, Fogel 93, Fogel 95, Fogel & Fogel 96] typically use Mutation on representations of Finite State Machines. Evolution Strategie (ES), [Bäck *et al.* 91, Hoffmeister & Back 91], originally only used Mutation, but now also incorporate Crossover. ES can perhaps be characterised by the use of real-value encodings and 'strategy vectors' which guide the way Mutation is carried out on each individual. Evolutionary Algorithms which make some use of Crossover are commonly known as Genetic Algorithms (GA), [Holland 75, Goldberg 89a]. GA and ES were developed more or less simultaneously, though the proponents of ES would insist that GA and ES are distinctly different.

A GA is essentially a simple model of the theory of Darwinian evolution by Natural Selection and Genetic Recombination. The theory was proposed by Charles Darwin [Darwin 59] and Alfred Russell Wallace (whose surprise letter to Darwin detailing his own thoughts on evolution, arrived at independently, kick-started Darwin's publication of 'Origin of the Species' and the ensuing shake-up of the Creationist orthodoxy of the time) during the last century. There is little sensible argument against its descriptive and explanatory power of what occurs in the natural world. In its much simplified (and much more recent) form as a GA, for computer-based search, the model of evolution performs very well. Crossover, sometimes considered the principle distinguishing feature between a GA and other search strategies, seems to be a help in some situations and a hindrance in others.

The typical representation used in a GA is linear. This stems from Holland's early attempts to duplicate natural genetic evolution by mimicking the linear chromosomes found in natural DNA. Linear solutions are simple to work with, and can be applied to many different problems. They are easy to code if the solutions are all of a fixed length, with simple ways of modifying or recombining solutions together. However, not all problems are amenable to this fixed-size representation, where the size is set before the search begins. In many cases the size of an acceptable solution is not known in advance. A more flexible approach known as Messy GAs, [Goldberg 89b, Goldberg *et al.* 93], allows variable length linear solutions, though requires different methods of mutating and recombining solutions. The increased expressive power of the Messy GA adds to the algorithm's complexity.

An alternative approach is to drop the linear format, instead adopting a tree-structured representation. This is a natural variation of the standard GA. There are many obvious and straightforward ways of mutating and recombining solutions of this form, described below, such as the exchange of subtrees. Genetic Programming (GP) is a GA using this particular tree-based representation.

The term 'Genetic Programming' is a little misleading, implying that the algorithm's sole use is to generate programs, although that may have been the original aim during its development. There are many approaches to the evolution of computer programs including tree-based GA (aka GP), [Kinnear Jr. 93, Langdon 95, Crosbie & Spafford 96, Brave 96], machine-code GA, [Nordin & Banzhaf 95], and Artificial Life methods, [Ray 91, Thearling & Ray 94], etc. Unfortunately the term 'Genetic Programming' is used to refer to both tree-based GAs and the evolutionary generation of programs. This thesis concentrates on the aspect of GP which is a tree-based GA.

1.5 GP: The What and the How

Perhaps the best introduction to GP can be found in "Genetic Programming: on the Programming of Computers by means of Natural Selection", [Koza 92]. Whilst not the first or only proponent of the automatic generation of programs by computers, (Cramer, amongst others, did some earlier work, [Cramer 85]), Koza's book helped popularise the field. A large and weighty, but easy to read, tome, it describes and delves into many aspects of GP. His next GP book, [Koza 94], similarly weighty, takes GP a bit further, expanding on some of the themes in the previous book, notably Automatically Defined Functions (described in this thesis in Section 3.3). The collection "Advances in Genetic Programming", [Kinnear 94], provides a very good snapshot of the wide range of GP-related research, as does the more recent "Advances in Genetic Programming 2", [Angeline & Kinnear, Jr. 96], which also includes Langdon's extensive GP bibliography, [Langdon & Koza 95, Langdon 96]. Another good collection of GP research can be found in the GP-96 conference proceedings, [Koza *et al.* 96], and GP-97 proceedings, [Koza *et al.* 97]. Since GP is just a simple, natural variation of a GA, there are any number of relevant GA-related books and papers available. The classic, seminal GA book is "Adaptation in Natural and Artificial Systems" by Holland, [Holland 75]. There are proceedings of numerous annual and biannual conferences which have focussed on aspects of Evolutionary Computation such as "International Conference on Genetic Algorithms", ICGA, "Parallel Problem Solving from Nature", PPSN, "IEEE Conference on Evolutionary Computation", "Genetic Algorithms in Engineering Systems: Innovations and Applications", Galesia.

GP can be considered to have two main components:

- an evolutionary search algorithm
- a tree-structured representation

The two components are quite separate. The evolutionary algorithm which searches for solutions is independent of the way the solutions are represented.

Evolutionary Search

The underlying search algorithm, known as a Genetic Algorithm (GA), uses a simple model of Darwinian evolution to search for good solutions. The success (or not) of natural selection and genetic recombination relies on the potential for offspring to occasionally be 'better' than or improve upon their parents. A GA starts with an initial group or population of individuals, generated at random. These individuals can be thought of as chromosomes, if the biological analogy is taken far enough. Each individual is tested to see how good it is at solving the problem in hand, using a fitness function. The problem in hand could be to design a bridge that is both light and strong, or construct a timetable that satisfies as many constraints as possible, or to design a Neural Network to classify phonemes, or to predict the next major fluctuation in share prices given the previous five days worth of trading figures. In short, almost anything goes.

Each individual in the population is a possible solution to the problem. Whilst all of these initial individuals will almost certainly be very poor or unfit solutions, some of them will be slightly less unfit than others. These fitter solutions are selected to act as parents for the next generation of solutions. Individual parent solutions are copied or mutated, or pairs of parent solutions are mixed together (in a process known as Crossover), to produce new or child solutions. The child solutions can be placed back into the population, replacing the most unfit individuals, (in a process known as steady-state replacement), updating the population, or can be collected together to create a completely new generation of individuals which then replaces the previous generation. The latter option (generational replacement) is the one described more fully here, and used throughout this thesis.

The choice of which method (or operator) to use in constructing each child is made at random for each child. The user specifies 'operator selection frequencies', i.e. the bias given towards choosing Mutation, Crossover, and any other operators which may have been defined to create new individuals.

Once the new generation of individuals has been created, they are all tested, and the fitter ones are selected to act as parents of the next generation. This generational cycle is repeated until one of four things occurs:

- an individual is produced which solves the problem completely but not necessarily optimally and the search process ends (in some cases it is possible to know when an optimal solution has been discovered)
- an individual is produced which solves the problem sufficiently well that the user decides to end the search
- it becomes apparent that the search process will not produce suitable individuals and the user decides to end the search
- something goes wrong and the process has to be debugged and restarted

(There are no prizes for guessing which two of these possibilities occur most often in practice.)

A GA is an extremely flexible algorithm, and there are a great many variations upon the basic theme. Some common ones are:

- Elitism. The best individual(s) of one generation are explicitly copied into the next generation, ensuring that the GA doesn't lose or 'forget' good individuals.
- Seeding the initial generation with some possibly good solutions. This allows the GA to take advantage of any extra domain knowledge the user might have, or to make use of earlier results, allowing the population to start evolving with fitter individuals.
- Parallel populations and Migration. Several populations are evolved simultaneously, all working on the same problem, with occasional individuals transfered

or migrated from one population to another. This allows the efficient use of fast parallel computers. A GA is eminently parallelise-able.

Each aspect of a GA is subject to minute and seemingly never-ending adjustments, and many many parameters. The design of particular GAs and their assorted parameter values is still very much an intuitive process, based on experience and feedback from earlier runs.

Representing Solutions in GP

The standard GA uses a linear representation, such as a string of bits, or numbers, or characters, for each solution. Ideally, it should be able to represent all possible solutions to a problem using the particular representation. The linear representation is simple, concise, and easy to modify. For example, a linear solution string can be mutated by randomly changing one or more of the characters in the string. Two parent strings can be combined via Crossover by exchanging substrings between them to produce two child solutions which contain a mixture of material from both parents, in a process analogous to genetic crossover. The linear representation is suited to many problems, especially when the structure and maximum size of likely good solutions are known in advance.

However, when solution sizes are open-ended, or solutions are likely to have some kind of hierarchical structure, the linear representation can be restrictive. Genetic Programming is a GA which uses a tree-structured representation. This flexible representation can be used to encode LISP-like programs (Figure 1.2), algebraic expressions (Figure 1.3), hierarchical relationships between different parts of a solution, and linear solutions (Figure 1.1), among other things (see Section 3.1 for more examples).

GP is given a set of functions (i.e. nodes which support subtrees) and terminals (i.e. leaf nodes which do not support subtrees). A subtree can be a single terminal node or consist of functions and terminals. Standard GP ensures that there is closure, i.e. any function can have any subtree(s) and the whole tree is still valid. A simple example of a supervised learning problem should make this clear.

Imagine the problem is to find an equation for mapping between two input vari-

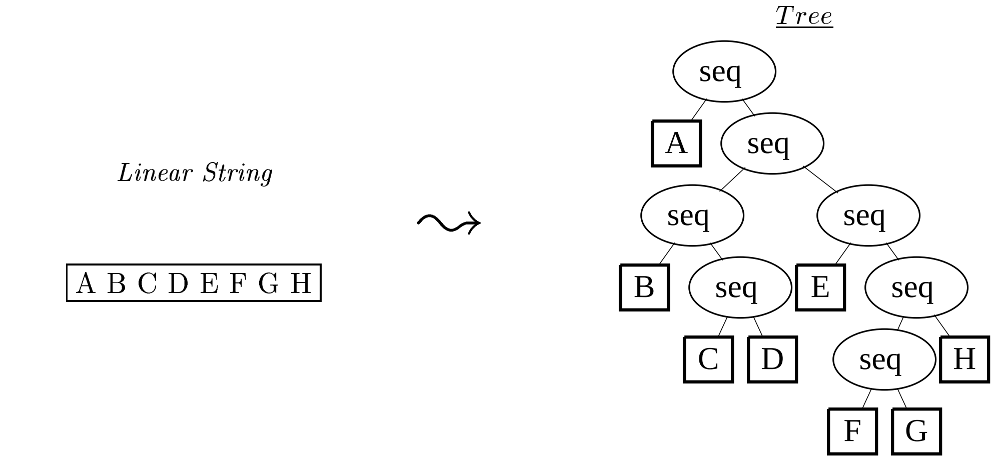


Figure 1.1: Linear String ⇝ tree

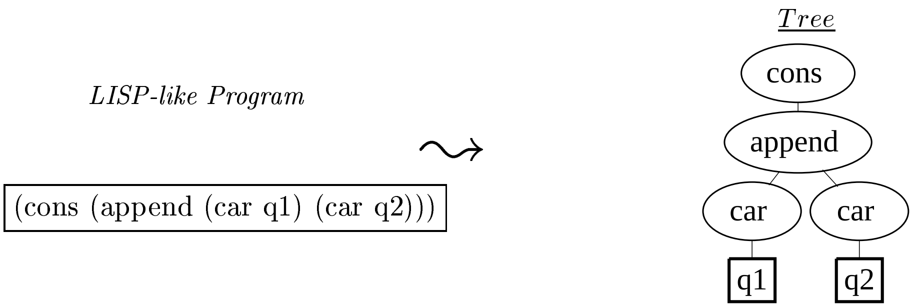


Figure 1.2: LISP-like Program ⇝ tree

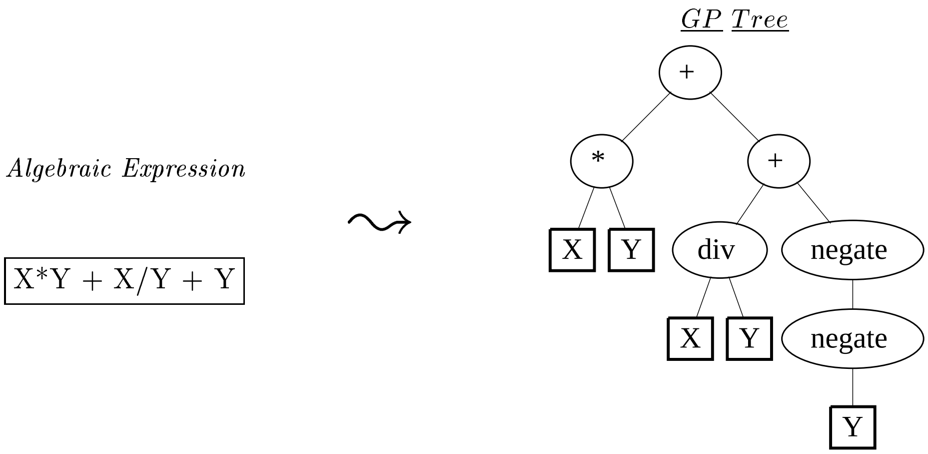


Figure 1.3: Algebraic Expression ⇝ GP tree

ables X and Y and an output variable. GP is allowed to use the functions { +, *, div, -, negate }, and the terminals { X, Y }. The functions are the basic arithmetic operations 'plus', 'times', 'divide', and 'multiply' which take two arguments each, and the function 'negate' which negates its one argument (i.e. multiplies by -1). The terminals are the two problem-specific variables, and have real number values.

An example GP tree is shown in Figure 1.3. The binary arity functions, '*', '+', 'div', each have two subtrees, whilst the single arity function 'negate' has one subtree. The subtree on the left containing '*', 'X' and 'Y', represents the arithmetic expression 'X*Y'. The tree as a whole represents '(X*Y) + (X/Y + (- (- Y)))' or, more simply, 'X*Y + X/Y + Y'. With the standard closure condition, any function node can have any tree as a subtree. In the case of the division function, this presents a problem. If Y ever has a value of zero, there would be a division by zero, which is mathematically undefined, and would normally result in a computer error. In such an instance, a GP function is 'protected', i.e. it is defined to produce a legal value whenever it would not normally do so, and thus the function is protected from values it cannot handle. The divide function is often defined to return 1 (or perhaps 0) if the denominator (i.e. the value returned by the right-hand subtree) is zero. The two 'negate' nodes in the same subtree are obviously redundant, but there is no requirement for GP trees to be sensible or efficient.

New or child trees can be created from parent trees very easily. A parent tree can be mutated by replacing a randomly selected node with a different one of the same arity, e.g. by replacing the 'div' (a function node of arity 2) by a '-' node (also of arity 2), or by replacing one of the 'Y' nodes (arity 0) by an 'X' node (also arity 0). Another form of Mutation is to replace a randomly chosen subtree by a new, randomly generated subtree. Two parent trees can be combined by exchanging a randomly selected subtree from one parent with a randomly selected tree in the other parent in a process known as Crossover. There are many variations of these operations for producing new trees. The closure constraint means that all of these operations on GP trees only ever produce valid trees.

In such a problem, there would usually be a set of examples of input values (X and Y) and their associated output value. A GP individual would be tested or evaluated on each example by instantiating the variables X and Y in the tree to their respective values, calculating the return value of the tree, and comparing this value with the correct or target output value. The sum of absolute differences over all the examples could be used as a measure of how good or fit the solution is, i.e. the smaller the better. This method of evaluating the fitness of an individual is known as Supervised Learning (or perhaps supervised training).

Several useful variations of basic GP are:

- Strongly-Typed GP

  The closure constraint is removed, and constraints based upon data-types are used instead. This means that not all possible trees will be valid trees. Function nodes can only have subtrees as arguments which return the correct data-type. [Montana 95, Haynes *et al.* 96].

- Mutation-only GP

  Some research suggests that the standard crossover operator may not necessarily be *A Good Thing*, [Gathercole & Ross 96], (see Section 4.1). Some modifications to Crossover have worked well, [Angeline 96a, Angeline 96b], as has an assortment of mutation operators, [O'Reilly & Oppacher 94b].

- Hill Climbing and Simulated Annealing

  Ignoring the population aspect of GP, but making use of the tree-structured representation, [O'Reilly & Oppacher 94b]

- Automatically Defined Functions

  A more powerful representation, allowing GP to evolve hierarchical function definitions, especially suited to problems whose solutions have a strong hierarchical structure, [Koza 92, Koza 94, Kinnear, Jr. 94] (see Section 3.3).

- "A Compiling Genetic Programming System that Directly Manipulates the Machine Code", with an awesome speedup in evaluation times, and only a few restrictions, [Nordin 94]

- Representing the GP population using a Directed Acyclic Graph

  This highly efficient method for representing a collection of trees can lead to massive savings in memory usage, and huge speedups in evaluation time by caching earlier results of subtree evaluations, [Ehrenburg 96].

1.6 Basics of Supervised Learning

Supervised Learning can be any one of a variety of ways for presenting a problem to a computer-based learning algorithm. The simplest analogy is that of a Pupil-Teacher arrangement. The teacher presents the pupil with a problem (or sub-problem). The pupil works out an answer and returns it to the teacher. The teacher compares the pupil's answer with the correct answer, and then gives the pupil a reward or punishment (or an error score) accordingly. The pupil can use this feedback to try an improve its method for calculating answers in future.

In terms of GP, the teacher is the fitness function. Knowledge of the problem in hand is encoded in the fitness function, enabling it to assess the worth (i.e. fitness) of all the candidate solutions produced by GP (the pupil). Good solutions get a good score, i.e. a reward, and bad solutions get a bad score, i.e. effectively a punishment.

A problem can sometimes be defined in terms of a set of examples. The learning algorithm has to come up with a mapping between input and output values that correctly deals with the training set. The (often unspecified) hope is that this mapping will transfer well to previously unseen examples and be able to cope with all possible examples of this type. In such a problem, the fitness of an individual would relate to the sum of errors it makes on the whole training set of examples. It is not possible then to work out from the fitness score exactly what errors were made on each training case. This is known as batch-learning. If the problem is to win at chess, the only feedback the learning algorithm would normally receive is a notification of whether it had won or lost a game, known as delayed feedback. There is no way of working out from the feedback what aspects of the way the game was played were good or bad. An even more difficult situation would be when the chess learning algorithm is presented with feedback only after several games had been played.

The main problems tackled in this thesis are of the batch-learning type, in Chapters 5 to 8. The aim is to correctly classify a training set of, say, 4000 example cases. In one problem there is also a test set which is used as a guide to see how well a solution generalises to cases which were unseen during the training phase.

So that was an overview of Genetic Programming and Supervised Learning. The next chapter looks as actually using GP to try and solve problems...

Chapter 2

Ramifications of a Large and Messy Problem

This chapter illustrates some of the difficulties of using GP in practice by applying GP to a largish messy supervised learning classification problem. The Thyroid Problem, described in Section 2.1, has been extensively tackled elsewhere using Neural Networks, [Schiffmann *et al.* 92a, Schiffmann *et al.* 92b], among other algorithms, and has a benchmark score associated with it. The process of applying GP to the problem is discussed. Of the many difficulties encountered, the fitness evaluation bottleneck is the most fundamental and hardest to avoid. To combat this, Dynamic Subset Selection (DSS), [Gathercole & Ross 94a, Gathercole & Ross 94b, Gathercole & Ross 97a], a modification of the standard supervised learning approach was designed. DSS enables GP to produce good solutions to the Thyroid Problem, and is described in greater detail in Chapter 6.

In general, published papers seem to miss out a lot of detail concerning the difficulties and choices made along the way to the development of a particular method. These unspecified choices, perhaps dead-ends or mistakes, will likely be repeated by later researchers, unaware that some of their difficulties have already been tackled. What follows is a description of how the features of the Thyroid Problem led to the steps taken towards its solution, using GP, giving the reasons for some choices made along the way. This should allow for a re-examination of the approach. Hindsight now shows where several decisions could or should have been made differently.

2.1 Why Choose the Thyroid Problem

Attempting toy problems with GP, such as the Iris Problem, [Fisher 36], (a stalwart of Machine Learning research which is denounced with feeling as being far too simple a problem in [Francone *et al.* 96]), can be a profoundly unsatisfying experience,. If a problem has one or several simple and easily-attainable solutions, it is difficult to learn more about GP, and in particular to explore GP's limitations. The GP literature, e.g. [Atkin & Cohen 94], suggests that it is difficult to scale GP up to work successfully on larger problems.

The Thyroid Problem, available from [Werner 92], is considerably larger than the Iris Problem. It is a Supervised Learning task, like the Iris Problem, but consists of a set of approximately 4000 training cases and a set of 3500 test cases, where each case comprises twenty-one fields. In contrast, the Iris Problem consists of 100 training cases, 50 test cases, and each case comprises four fields. The task in both problems is to correctly assign each case into one of three possible classes. The Thyroid data is messy and noisy, and like the Iris data (measured by hand from an assortment of irises), is based on real measurements. A solution to the Thyroid Problem is of practical importance, since it relates to the identification of hospital in-patients who are likely to go on and develop later complications with their thyroid gland. The data falls into three separate classes, two of which signify a thyroid illness, and are thus the important ones to identify, and one much larger class (92% of all cases) which signifies no thyroid illness.

Described by Schiffmann *et al* is an attempt to use a variety of Neural Network approaches to solve the Thyroid Problem, [Schiffmann *et al.* 92a, Schiffmann *et al.* 92b]. Their results indicate it is possible to solve the problem to a high degree of accuracy, nearly 98% correct, but that it is not easy to do so (92% correct is actually a trivial solution since one of the three classes consist of 92% of all of the cases). The Neural Network results set up a good benchmark, or a target to aim at using GP, enabling a useful comparison between the two different approaches.

Possibly the earliest published report on using the thyroid dataset in Machine Learning research is [Quinlan 86]. Later, in [Quinlan 87], Quinlan reports error rates of 0.3%

and writes,

> "This domain is a good starting point becase it uses 'live' data from which, warts and all, extremely accurate classifiers can be constructed."

A variety of ML algorithms are applied to the thyroid dataset in [Weiss & Kapouleas 89], including an assortment of statistical pattern recognition algorithms such as Linear Discriminant, Quadratic Discriminant, Nearest Neighbour, Bayes Independence, and Neural Networks (back propogation), and some decision tree induction methods. The best results reported are for CART (Classification and Regression Trees, [Breiman *et al.* 84]) scoring 0.21% training and 0.64% test error rates. Weiss and Kapouleas state that the NN runs took by far the longest of all they carried out, requiring up to 11.5 hours per run for the larger networks.

Turney tackles the thyroid problem with a variety of algorithms from the point of view of cost of classification, [Turney 95], where each field in the data, corresponding to a medical test, has an associated cost. The aim is to minimise the error rate and the total cost per classification, and the combined error scores reported make it difficult to compare with work done in this thesis. In [Raymer *et al.* 97], Raymer *et al* also look to minimise classification costs, by attempting to reduce the number of fields used in the classifications. The best error rate reported for the GA, which evolves a weight set for use by a K Nearest Neighbours algorithm, is 2.25% on unbiased holdout tests, but the time taken for a typical run is not reported.

In summary, several different algorithms have been used to tackle the thyroid problem. The decision tree induction algorithms in particular have produced the best results in the shortest time, and there have been several investigations into reducing decision tree sizes or the number of fields used. Neural networks have not managed to perform as well, and the larger networks appear to require training times roughly equivalent to GP (i.e. runs for GP+DSS reported in Chapter 8.2).

There are many other Supervised Learning tasks in the public domain whose datasets are readily available over the Internet, e.g. [UCI 97]. The Thyroid Problem is one of the larger problems. Its data is already split into Training and Testing sets. It is relatively straightforward to apply the standard GP algorithm, although there was a large initial hurdle, described below in Section 2.4, of how to interpret a GP tree's output as indicating one of three categories.

Whilst it is easy to apply GP to solving the Thyroid Problem, it rapidly becomes obvious that there are many hurdles to overcome, such as very slow fitness evaluations, premature convergence, the need for a large population, and many more, for GP to tackle the problem successfully. Although not common to many simple problems, these difficulties are what makes the Thyroid Problem interesting, and will have to be overcome if GP is to be applied to larger, more difficult problems in the future.

2.2 Starting Point

The starting point for the attempt, in this thesis, to tackle the Thyroid Problem using GP consisted of:

- the Thyroid papers, [Schiffmann *et al.* 92a, Schiffmann *et al.* 92b], and data, [Werner 92].

  The Thyroid papers indicate that the Thyroid problem is solvable to a high degree of accuracy, and give a benchmark figure, approx 98% correct on the Test set, enabling a good comparison to be made with GP. The data is already split into Training and Test sets. When viewed using XGOBI, [Swayne *et al.* 91], shown in Figure 2.1, a large degree of overlap is obvious between the three different classes of the Thyroid data. Also, one of the three classes is much more common than the other two.

- Koza's huge book, [Koza 92].

  This opus covers many different example problems, which suggest sensible initial settings for the many GP parameters. It gives an idea of what might be a reasonable set of functions to complement the problem-specific terminals (i.e. the terminals corresponding to the fields in the problem). Koza also strongly recommends the addition of an ephemeral random constant to the terminal set. Each time this terminal is selected as a leaf node in a randomly generated subtree,

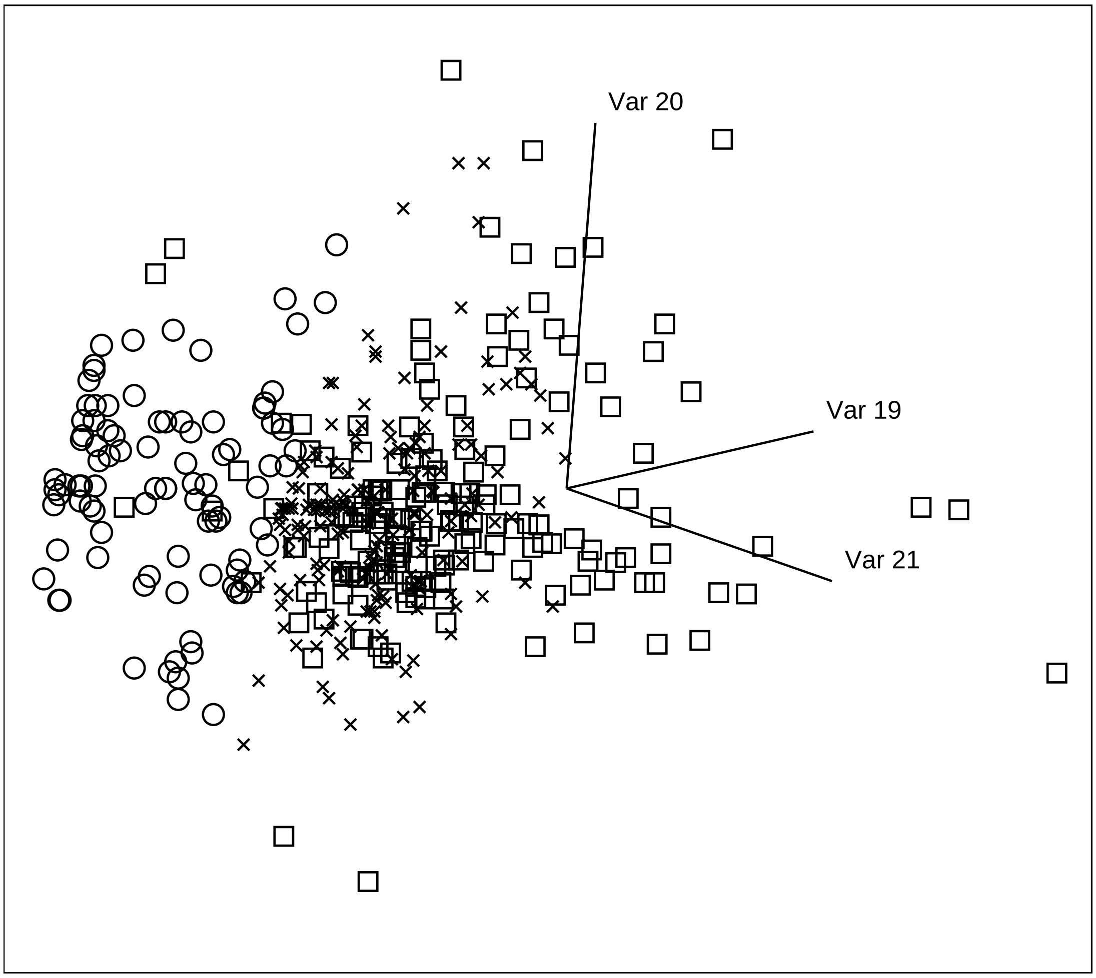


Figure 2.1: A 'Large and Messy' Problem: This figure shows a 3D Slice (of 21 dimensions) of a 500 case subset of the Thyroid training data showing the overlap between the 3 classes. Classes 1 and 2, signifying a thyroid illness, are represented by ◯ and ×. Most of the cases belong to the largest class, signifying no thyroid illness, represented here by □. Only a subset of these cases have been included since they would obscure most of the figure.

The view presented here has been selected by hand on the basis that it shows the three classes more clearly than any other view, though the classes still overlap a great deal. From most other viewpoints, the classes overlap even more and are much less distinct.

for example when a random individual is created at generation 0 or during a mutation operation, it will take on a random floating-point value. This value remains fixed throughout the remainder of that particular node's existence in the population, where it might be spread by the actions of the crossover operator.

- off-the-peg GP code, "Simple Genetic Programming in C" (SGPC), [Tackett & Carmi 93].

  The SGPC code is a well-written implementation of the standard GP algorithm, written in C, and available in the public domain. It is easy to adapt to supervised training problems. In SGPC, a GP tree is represented using C pointers, a flexible approach, but slow and uses up a great deal of memory.

- several Sun and HP workstations.

  Once the GP program was ready to run, there were several Sun and HP workstations available. Each was able to handle jobs of no more than approximately 10Mb or so. There was also a large server available, able to handle much larger jobs up to 100Mb or so. These machine limitations imposed an upper limit on population 'volume', i.e. total number and size of individuals in the population.

2.3 First Impressions

The overriding initial impression from the early runs of GP on the Thyroid Problem was that GP is very slow, inefficient, and impractical. The runs would converge early to bad solutions and then enter long periods of no improvement at all, producing large numbers of unfit individuals. Population sizes of several thousand were needed to improve solution quality, which had a severe impact on the turnaround time for GP runs, taking many days to achieve non-trivial solutions.

2.4 Early Snags and Decisions

Choosing Parameter Values One of the most daunting aspects of using GP is the very large number of parameters that need to be specified before the algorithm can be used. Parameters include the function and terminal sets, population size and structure, replacement method, operator type and selection frequencies, etc. All of these affect GP's performance to varying degrees. More often than not, wrong choices will lead to GP performing very badly.

To some extent, Koza's book answers all of the questions raised above. Among the many examples in the book are practical suggestions for parameter settings. This is a great help when starting out on a problem, but it quickly becomes clear that each problem is unique, and GP responds differently to different parameter settings on each problem. What is left after Koza's book is educated guesswork, and feedback from previous runs.

Perhaps the most important GP parameters to 'get right' are the terminal set, the function set, population size (and structure and replacement strategy), in combination with fairly 'standard' settings for other parameters such as the operators and operator selection frequencies.

For the Thyroid problem, the terminal set is specified by the 21 fields in the problem, where each variable in the terminal set refers to one of the fields in the Thyroid data. Koza recommends including an ephemeral random constant, described above in Section 2.2, but early runs indicated that this had no significant effect (for the Thyroid problem) and was replaced with a small group of fixed constants, { 0, 1 }, which also, it later appeared, had no significant effect. The selection of sets of functions and terminals, constant or otherwise, is a topic of much debate within the GP community. It wasn't the focus of work done in this thesis and is not pursed further here.

Constructing the function set is something of an art form. It has to be powerful enough to allow GP to construct good solutions, flexible enough to allow for variety, and not too large that it reduces the efficiency of GP's search. The function set is { IFLTE, +, -, *, %, tanh, log, minimum_of_3, negate, sqrt }, where the functions behave as follows:

- IFLTE (If arg1 is Less Than or Equal to arg1 then the answer is arg3 else it is arg4)
- +,-,* (plus, minus, multiply)

- % (protected division where division by zero is defined to be 1)
- tanh
- log (protected natural logarithm, where log(0) is defined to be -1000)
- minimum_of_3 (returns the smallest of its three arguments)
- negate (multiplies by -1)
- sqrt (protected square root of the absolute value of its argument)

The function set seems to have most of these properties, and was found through a combination of Koza's book, guesswork, and feedback from early runs.

Koza makes no bones about recommending that population size should be set as large as possible, i.e. at least in the 1000's. Whilst this is certainly more likely to enable GP to find non-trivial solutions it is not very practical from the point of view of computing resources (see below).

Population structure can take many forms. The simplest, used in this thesis, is panmitic, where any individual can be combined with any other individual during the breeding stage. Another structure involves the population topologically separated into demes or islands, where only neighbours (in some sense) can be combined during breeding.

There are many proponents of the two different replacement strategies: generational, where an entirely new generation is constructed from the previous generation, or steady-state, where new individuals replace old individuals in the same population. Generational replacement requires the addition of Elitism, where the best individual(s) from the previous generation is included in the next generation, to ensure that the population does not lose its best individuals. Steady-state replacement is implicitly elitist since the best individual will never be replaced. Early results indicated that steady-state replacement was perhaps a bit more susceptible to prematurely converging to bad solutions, so generational replacement was used throughout this thesis. A panmitic population requires fewer parameters than a distributed population, so for this reason alone it was used throughout this thesis. Population structure and replacement, as with the choice of functions and terminals, are topics of much current debate within the GP (and GA) community, and are not pursued further here.

How to distinguish between more than two classes The task in the Thyroid Problem is to construct a solution which distinguishes between three classes of examples in the training data. For GP, this means that when an individual tree is evaluated on a particular case, the tree's output must be interpreted as indicating which one of the three classes the case belongs to. It quickly becomes clear that this is not a trivial step.

The obvious approach is to subdivide possible GP tree outputs into ranges such as

- $output < 0$, signifies class 1
- $output = 0$, signifies class 2
- $output \geq 0$, signifies class 3

Thus a tree output of 7.4, say, would be interpreted as class 3. Early results with this choice of ranges and others, such as

- $output < 0$
- $0 \leq output < 100$
- $100 \leq output$

were discouraging. GP seems unable to cope with this extra step, instead fixating on the largest class, 3, with trivial trees. To be successful, GP individuals would have to cope with the extra translation step for their outputs to be interpreted correctly, as well as distinguishing between the different classes using the information in the fields of each example.

Fortunately, with the Thyroid Problem, there exists a 'natural' division into two subproblems, where each subproblem is simpler than the whole problem, and one of the subproblems involves a much smaller training set. This natural division creates two

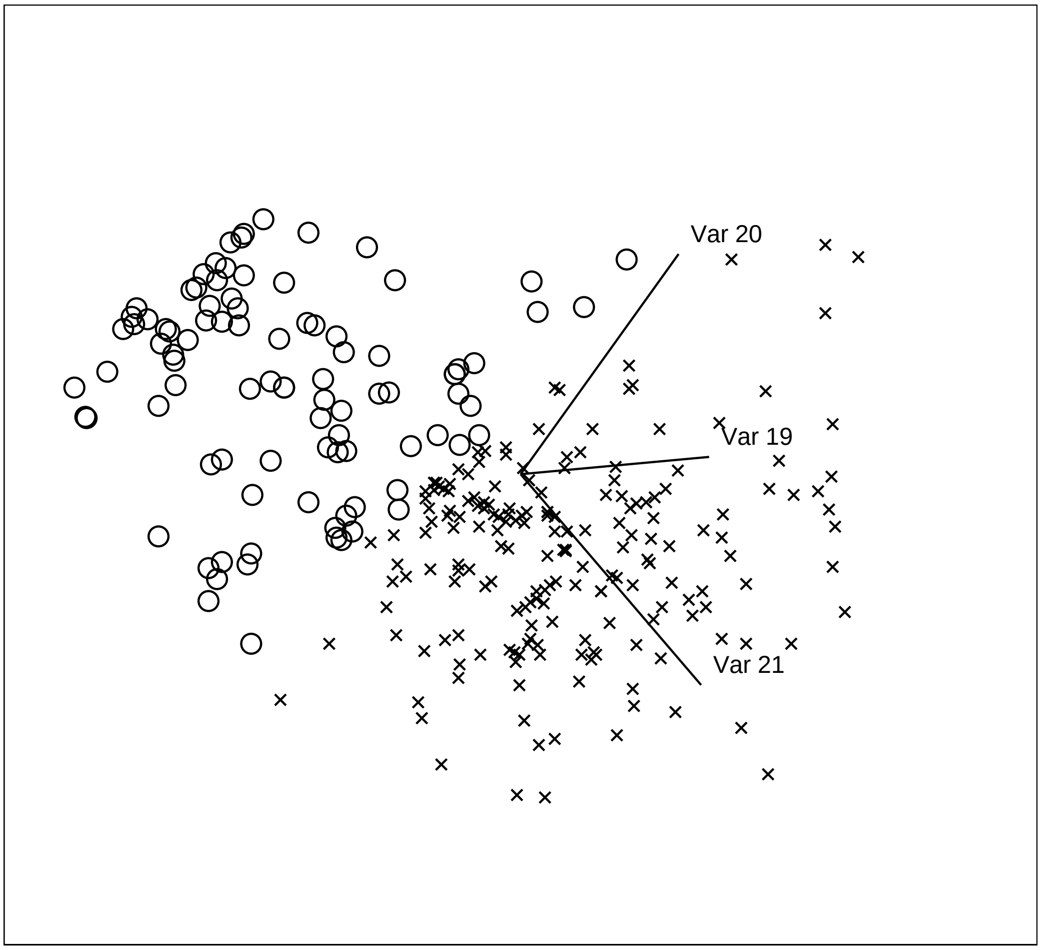


Figure 2.2: Easy Thyroid subproblem: 3D Slice (of 21 dimensions) of all 83 class 1 cases, represented by ◯, and 191 class 2 cases, represented by ×, showing the distinct split between the two classes. This subproblem is simple to solve.

binary classification problems. The first is to distinguish between cases from class 3 (the largest class, which signifies that the patient has no thyroid trouble) and all the others (much smaller classes, which signify that the patient has some form of thyroid trouble), and then if the case is not in class 3, to distinguish between cases from class 1 and from class 2 (two distinct thyroid ailments). Even more fortunate is the fact that the smaller subproblem, distinguishing between classes 1 and 2, is very easy. The split between these two classes is obvious in Figure 2.2. Using the same setup as used to tackle the larger subproblem, it is easy for GP to discover a 100% correct solution for distinguishing between classes 1 and 2. This means that most of the effort can be focussed on the one binary classification task of identifying class 3 cases. Interpreting a GP tree's output as identifying one of two possible classes is much easier than the situation involving three classes. For this thesis,

- $output < 0$, signifies class 3
- $output \geq 0$, signifies not class 3

The approach of splitting a large, multi-class (i.e. more than two) classification problem into smaller binary classification subproblems can, in principle, be expanded to cope with any number of possible classes in the training data. A group of cases belonging to N classes could be classified using $\log_2 N$ binary classification steps. Experiment and/or pre-processing for dependencies would be needed to find the best way to subdivide the main problem. The initial approaches of translating from ranges of possible tree outputs to an indication of particular classes emerged as being too great a hurdle for GP to overcome, and although an interesting topic for further study, was not pursued further in this thesis.

Fitness Measure One of the simplest fitness measures possible in this type of classification problem is the error count, i.e. the number of misclassifications made, but it does suffer from several deficiencies. It is unable to distinguish between two trees which make different errors but make the same total number of errors. It is also heavily affected by the relative sizes of classes in the data. In the Thyroid data, class 3 is much more prevalent than the other two classes combined, so it is possible for a tree to score well (only 8% errors !) by always choosing class 3. In hindsight, an obvious approach is to re-scale the size of an error corresponding to a misclassification, according to the relative size of the classes. Thus choosing the largest class incorrectly would incur a higher penalty than choosing one of the other smaller classes incorrectly. However, the simple error count was used in this thesis, until the addition of Dynamic Subset Selection, described below in Section 6.

Bad Solutions All the early runs failed to produce anything like good solutions, i.e. they failed to evolve individuals which significantly outperformed the randomly generated trees in generation 0 of each run. Runs quickly fixated on solutions which classified all cases as belonging to the largest class, producing small trees, and rapidly converging in a few generations to close copies of one individual. In such a situation the population quickly loses most of the variety in the function and terminal sets. An obvious approach to the problem of small trees is to forbid the addition of small trees to the population, by imposing restrictions on the operators. But, as with rescaling the error count, it was not used here on the Thyroid problem.

Need for many runs to provide good Statistics In order to get useful information about GP performance, say the effect of some parameter changes, many runs are needed to provide adequate statistics about their effectiveness since, in some circumstances, GP can produce widely differing results using the same parameters. This is very difficult to achieve with GP since the runs are very slow. When starting on a problem, there are so many parameter choices that need to be made and tested that a large element of guesswork is necessary, leaving later choices vulnerable to being affected by spurious results. Unfortunately, GP seems quite sensitive to a variety of parameters. Many decisions taken during this thesis were based on a very few successful runs, and have no doubt led to some bad choices for parameter settings.

Bugs GP is a robust algorithm. It is very effective at hiding errors in the program, or even taking advantage of them in the fitness evaluation stage, producing poor but extremely ‘fit’ individuals! The C language used in SGPC is susceptible to many subtle bugs, in particular the pointer representation used can easily produce obscure bugs that are difficult to track down. To deal with bugs in C, the 'assert' function is extremely helpful. It is best used at every stage in the program to confirm that things are as they should be. A good programming methodology involving frequent testing is needed right from the start to prevent later GP research from descending into an ever more desperate search for wily and elusive bugs.

GP is *slow* As indicated in [Koza 92], GP may need a large population size. This, plus the needs of supervised training (i.e. evaluation of individuals on each training case), combined with the flexible but inefficient C-pointer approach used by SGPC to represent GP trees, leads to a major bottleneck at the fitness evaluation stage of the GP algorithm. The GP program becomes very large, requiring many megabytes of computer run-time memory, and very slow, due to the large number of evaluations. Runs can take many days. The large process size exacerbates CPU use, making it much less efficient. This is due to thrashing, where a CPU spends most of its time swapping pages of memory in and out of swap space, rather than allocating processing time to its processes. A further problem is that GP produces a large number of unfit trees.

There are several ways of tackling the fitness evaluation bottleneck.

Increasing the speed of the algorithm can be done through better coding. This involves a major rewrite of the code, which is a lengthy process, and a detailed look at efficient representations such as in [Keith & Martin 94]. The approach of efficient coding has been taken to a successful extreme in [Nordin 94, Nordin & Banzhaf 95, Francone *et al.* 96], producing linear GP individuals which are evaluated directly as raw machine code. Nordin *et al* seem to have surmounted the obvious difficulties which might occur when allowing GP to produce and execute raw machine code. They report speedups in the region of 100x faster than traditional C-based programs. Such a speedup would allow a much faster turnaround time for GP runs, allowing a great deal more study to done on optimising parameters. However this approach does have several limitations such as lack of flexibility, and the functions and terminals can have no side-effects (which means they do not affect the context of any subsequent function calls or terminals, e.g. a node whose evaluation causes a robot to turn left has a side-effect, but a node which simply returns a value does not). A more expensive speed increase can be gained through the purchase of larger and faster computers, but this is perhaps not an option available to the average researcher. A more feasible approach might be to make use of smaller computers in parallel, since the basic GP algorithm can be easily adapted to work in parallel. This is probably 'the way of the future', but unfortunately it brings up a large number of new parameters. Andre and Koza seem to have the best of both worlds, using a parallel network of powerful computers, [Andre & Koza 96].

Looking once again at the bottleneck of fitness evaluation, another approach is to reduce the need for so many fitness evaluations. Reducing the size of the population leads immediately to worse solutions, but as can be seen in Chapter 8, when allowed to run for many more generations, a small population can outperform a large population using fewer fitness evaluations overall.

It would be nice to be able to reduce the size of the training set, since it is directly proportional to the length of the fitness evaluation step. A closer inspection of the performance of a GP population on the Thyroid training set reveals that many of the training cases are easy, given that most of the population can correctly classify them. This leaves a core of more difficult cases which are frequently misclassified. Using only this core of difficult cases (545 out of 3772) as a training set leads to Historical Subset Selection (HSS), described in Chapter 6. HSS allows much faster fitness evaluations, whilst still producing good solutions. A more flexible approach is to select a subset of the training set dynamically. If each case in the training set is assigned a weight based on its difficulty, i.e. how often it was misclassified when it was last part of a fitness evaluation stage, and the number of generations since it was last selected, a subset of cases can be selected and used to evaluate the fitness of each generation. With the subset size around 10% of the full Thyroid training set, Dynamic Subset Selection (DSS), described in Section 6, leads to roughly a 10x speed increase in the GP generation rate, and produced better solutions than when the whole set was used to evaluate each generation.

What runtime information to look at and store The aim in the Thyroid problem is to construct a solution which performs well at classifying the training data. Consequently, the ultimate measure of success of a run is the fitness of the best individual it produces. The next most important measure of a run is how long to let it continue running, which could be for a fixed number of generations or, more usually, until the run shows no signs of further improvement. A measure of improvement can be taken from the change in fitness of the best individual in successive generations. However, this approach ignores any dynamics within the population. Fitness diversity is a useful guide but it is hard to measure. Average population fitness does not indicate how the trees themselves are changing. Tracking the frequencies of nodes in the population involves the output of a lot of information, especially when runs can take many thousands of generations. Tracking tree structure, and the frequencies of subtrees can be CPU intensive as well as requiring even more output, and is representation independent. By storing a unique seed for the random number generator for each run, it should be possible to reproduce earlier runs exactly, and extract more details at a later stage. This saves memory use for storing output but runs the risk of missing important information first time round, and is very slow.

2.5 Longer Term Snags, Workarounds, and Hindsight

Longer term difficulties and decisions can be divided into two main categories: coding strategy, and research method.

Coding Strategy

Old Bugs One of the most disheartening aspects of using a computer program to produce data in a series of runs over an extended period of time is uncovering old bugs. These well hidden monsters have remained incognito until the most recent modification to the program, or perhaps an inspired test run. Having found a bug it is necessary to check its impact (if any) on previous runs. If the bug is sufficiently serious there might be nothing for it but to go back and repeat all the previous runs. Obviously, the best approach is to avoid or prevent bugs in the first place but, as mentioned above, the GP algorithm is very robust, and its output and performance can be very deceptive.

The best strategy is to assume bugs will arise, to pepper the code with error checks right from the start, and to carry out frequent and varied tests.

Tweak Parameters or Add New Code Tweaking parameters allows you to improve the existing setup, but there is the possibility that tweaking will be a never ending process, especially if the solution is not achievable with the current setup. Adding new code or extending the algorithm, however, is an excellent way to add more parameters to the system, adding complexity to the code and the algorithm. No matter what high hopes there are for getting these new parameters right first time, they and the others will still need tweaking.

Reproduce-ability One of the major problems which arises after adding modifications to the code is that the code is likely not to be backwards compatible, unless great efforts are made every step. This causes difficulties when it comes to reproducing old results, which means it is more important to store key information from each run instead of relying on being able to reproduce the data later from re-runs. A version control system such as RCS [Tichy 85] is (and indeed would have been) extremely useful.

Research Method

Being led astray An insidious consequence of very slow runs is the idle time between starting a run and viewing its results. It is very easy to extrapolate from earlier and partial results to make changes to parameters and start new runs, following up the assumptions made. This can lead to dead-ends, where parameter changes do not improve GP performance. By the time this is realised, a great deal of time can be wasted. It is important to base modifications on good statistics, involving many runs using the same parameters but different random number seeds. Unfortunately, given that GP is very slow, this is somewhat difficult to achieve.

Use of Test set as a Training set Ideally, for comparisons with other algorithms, or assessment of an algorithm's performance on a problem, there should be a test set of unseen data. Only after all the development and training has finished should the algorithm be checked on the unseen data. There are many established methods for selecting representative test and training sets from one large data set of examples, some of which are described in Section 5.2. For the Thyroid problem, the data was already split into training and test sets by Schiffmann *et al.* In this thesis, every effort was made to 'ignore' the test set when making modifications to improve GP's performance. The test set was certainly never explicitly used to guide modifications towards making GP better at generalisation from the training set.

Data Explosion Completing many long runs, each with a large population, over many generations, produces a vast volume of data to be processed and/or stored. There are many details which may or may not be important later. The approach of making runs reproduce-able runs into the difficulties mentioned above, and still requires the storing of all parameter settings and random seed numbers for each runs, and makes data mining impossible.

Parameter Explosion Extending the GP algorithm throws up a huge number of parameters. It is important to document each one, use clear names, make the defaults clear, and to assume idiocy on the parts of the user and especially the programmer by including extensive error checks on the bounds of all the parameter ranges. If not, chaos could well ensue. At the end of the programming done for this thesis, there were 170 input parameters, 55 special data types, 500 function definitions, and over 30,000 lines of C code.

Reputation with other non-GP users Given that GP is CPU intensive, memory intensive, sometimes increasing greatly in size during a run, when runs can last several days, and many runs are needed to produce adequate statistics, the GP user won't win any popularity awards on multi-user computers. There seems to be no way of avoiding this (apart from obtaining your own computer), so it is best to get used to the idea of receiving hate e-mail.

2.6 Applying GP to a problem

The procedure for using GP can be divided into four main parts:

- Knowledge Acquisition
- Knowledge Representation
- GP Tuning
- GP Runs

Knowledge Acquisition is the first stage in problem solving. For the purposes of this thesis, the problems were chosen in order to investigate the performance of GP, rather than from any particular urge or need to actually solve them. Having selected a problem, the next step is to analyse it and extract any salient features. Armed with this information, it should be possible to make an informed choice about what might be the best approach to use in order to try and solve the problem. Once again, for the purposes of this thesis, the approach is always GP.

Knowledge Representation is a key stage. With a good representation, a problem can be made much more amenable. With GP, the underlying representation is obviously a tree structure, but that is only the start of the process of designing a representation. Most test problems come prepackaged with data fields. These can often be taken directly as the terminal set for GP, but in many cases some pre-processing, such as Principal Components Analysis [Jolliffe 86], is needed to identify the relevant parts of the problem data, to remove extraneous data, or to construct more useful combinations of the data. An example of this process is well described in [Tackett 93], where GP is used to classify feature vectors extracted from infrared images containing images of tanks (or not, as the case may be).

Though the terminal set is often easily decided upon, the function set is often not as simple to construct. It is the glue which binds the terminals into useful expressions, and needs to be sufficiently powerful to allow GP to construct good solutions. This step is still a 'black art'. Despite many different researchers using GP, there is no straightforward formulae which can be applied to decide upon suitable function sets. It is clear that Automatically Defined Functions, described and used in [Koza 92, Koza 94], can and probably should be added to the GP representation when a problem contains inherent hierarchy of small solutions forming part of larger solutions, or when there is a great deal of similarity between different parts of the problem. Both of these characteristics are apparent in the Even-N Parity problem, described in Chapter 7. However Koza has demonstrated that ADF can be an impediment if such characteristics are not present. Other more powerful features such as indexed memory, [Teller 94], only seem useful in certain specially constructed problem areas. Successful GP applications have involved incorporating as much problem knowledge as possible into the function and terminal sets. If there are known links between terminals, then those links, e.g. square root, or log_{10}, should be included in the function set. It is not necessarily the case that reducing the volume of input data is the best approach to take, since key relationships within the data may be lost.

Along with the basic tree-based representation, GP comes with some standard operators, i.e. ways of changing or recombining existing trees to produce new and different and possibly better trees. Crossover is often considered the main GP operator, where subtrees are exchanged between two parent trees to produce one or two child trees containing a mixture of nodes from each parent. Along with Crossover is usually some form of mutation, where nodes or subtrees are replaced by randomly generated nodes or subtrees. Usually the sites in the parent trees where these operators work are chosen at random, without any regard to which parts of the parent trees are in some way important or essential to the functioning of the tree or causing the tree to produce incorrect answers. This blind action of the operators results in high percentage of child trees performing worse than their parent trees. Using a specific classification-tree representation, [Vere 95], the benefits of more targeted operator actions are obvious, where leaf decision analysis can make use of the fact that the "fitness (error) contribution of each subtree is localised and independent of other disjoint subtrees". O'Reilly and Oppacher, using various mutation operators and Simulated Annealing, have shown that Crossover is not necessary for the successful use of the GP tree representation in solving problems, [O'Reilly & Oppacher 96]. Lang has shown how mutations and simple hill climbing can perform better than GP, calling into question the effectiveness of Crossover, [Lang 95]. All in all, it is never usually obvious what are the best operators or combination thereof for particular problems.

Associated with the set of operators is the set of operator selection probabilities, which establish how frequently each operator is used to generate individuals for the next generation. With very low operator success rates, i.e. the children are usually worse than the parents, it is not obvious how to balance the operator selection. Research in GAs, [Tuson & Ross 96b, Tuson & Ross 96a], has shown that dynamically altering the selection probabilities can be difficult to do well as it is both problem- and representation-dependent, and in fact can hinder the GA. Section 4.1 provides some food for thought when constructing operators and choosing selection probabilities.

Given a particular GP representation, the fitness function needs to be specified in such a way that it can identify the relative merit of solutions expressed using this representation. In combination with the GP operators, the fitness function defines a search space for GP to traverse in search of good solutions. The fitness function encodes a great deal of the knowledge the user has about the problem. Ideally the fitness function should facilitate an easy path from bad solutions via a series of easy steps (i.e. operator actions) to optimal solutions, where each solution along the path has a better fitness than the ones before. Unfortunately, most problems do not have such well-behaved search spaces. A great deal of effort has gone into looking at the behaviour of search spaces in GA, also known as fitness landscapes, [Jones 95], but rather fewer studies have been published on GP search spaces. Needless to say, GP search spaces are hideously complicated. Especially in supervised learning problems, the fitness evaluation of the population is the main bottleneck in the GP algorithm. Chapters 5 to 8 look at ways of alleviating this bottleneck and extracting more information from the supervised training set.

Tuning GP i.e. selecting initial or new settings for its assorted parameters, is a 'black hole' into which a great deal of time and effort disappears. The number of aspects of a GP program which can be tweaked in a desperate attempt to improve its performance is nothing short of phenomenal. Perhaps the single most important parameter is population size. Too large and GP takes forever to complete each generation. Too small and, well, a thought-provoking part of this thesis shows that GP can, in certain situations, perform better with a very small population than a very large population, i.e. it finds better solutions in a much shorter time (see Chapter 8). Taking population size to the extremes: infinite – means that GP should be able to randomly generate an optimal tree in generation 0; one – means you have a form of Hill Climbing or, with a few extra features, Simulated Annealing, which have both been shown to perform well.

Although not conclusive, the impression gained from work done during this thesis is that the effective population size is dependent on the type of problem being tackled in the following way. If the search space contains a wide range of fitness values (ignoring the addition of parsimony), where it is possible to produce a succession of trees with small increments in their fitness values, such as the TicTacToe and Thyroid problems, small populations over many generations perform better. If the problem is difficult, and the search space contains only a small number of distinct fitness values, such as the Even-N parity problem, described in Chapter 5, a larger population is necessary to allow GP the chance to construct better solutions. Experience has shown that it is worthwhile trying GP first with a small population running over many generations. If there are still signs of improvement in fitness after many generations, then a larger population is probably unnecessary, and would perhaps hinder rather than help. It appears that a large population might be more prone to converging prematurely to sub-optimal solutions (perhaps it finds and fixates upon local optima too rapidly, whereas a smaller population might not even find most of these local optima and wouldn't move towards local optima as quickly).

One debate which occurs in the GP community but not the GA community concerns restrictions on tree size. Given GP's propensity to 'bloat', [Blickle & Thiele 94], where the size of individuals in the population increases as they accumulate garbage, running into practical limits on the availability of computer memory, it has become common practise to impose some limits on tree size, or to use parsimony, a bias in the fitness function against larger trees. Not all reports have been in favour of such restrictions. Rosca has indicated that GP trees tend to grow to a certain (large) average size and then oscillate around this size, [Rosca 96]. A more general study of the principle of Occam's Razor, the idea that 'smaller is better' which is used in much Machine Learning literature, indicates that such a bias leads to solutions which are less able to generalise successfully on unseen data, [Webb 96]. Section 4.1 takes a look at an adverse interaction between the Crossover operator and restricted tree size. There is no consensus as yet on what is the best approach to take, except perhaps that GP tends to use up too much memory, especially with large populations, and parsimony seems to hold down tree size quite effectively without appreciably hindering GP.

Since, for many problems, the optimum solutions are not known (and also, often, their fitness values), a decision must usually be made about when to end GP runs. Too few generations and GP might not have had sufficient chance to evolve good solutions. Too many generations and much time might be wasted as GP shows no sign of improvement, with its population having converged to become copies or damaged copies of the best individual, unable to produce any better solutions. If left for long enough, the mutation operators can in theory generate all possible trees, but this isn't perhaps the most efficient way to use GP. The stopping criteria in this thesis are usually when a known optimum is found, or after a certain number of fitness evaluation or generations have passed, or the computer has crashed. In most cases, trial runs are needed to establish a baseline performance for GP.

As new features are added to GP, the programmer experiences what can only be described as a parameter explosion. Each new parameter can affect all the original parameter settings. There is usually no way of knowing what is the best setting for a particular parameter. Guesswork, some testing, and reading the literature, are the only options available.

Once a particular GP design has been decided upon, the decision of what to record as output is relatively simple. Usually the fitness of the best individual in each generation is sufficient, along with, perhaps, the average population fitness. Deciding upon a particular GP design is usually quite challenging. During the design, much experimentation is needed to find the best parameter settings, and much data needs to be examined, processed and stored. A balance has to be struck between recording all information about the GP run that might be useful, and not filling up gigabytes of disk computer space with millions of numbers. If the runs are made repeatable, it should be possible to recover any data if it is later deemed necessary.

GP Runs: The phrase "A watched kettle never boils" must have been thought of with GP in mind. GP can be very slow. Although Nordin *et al*, [Nordin 94, Nordin & Banzhaf 95], seem to have hit upon an impressively fast GP implementation, where the individuals consist of directly evaluated raw machine code, most implementations, such as [Andre & Koza 96, Tackett & Carmi 93, Implementations 97], are compiled from a high level language such as C, or C++, or are even interpreted, e.g. Lisp. Large populations, many generations, difficult problems, all conspire to produce long run times.

The single most important bottleneck in GP is the fitness evaluation stage. In particular with Supervised Training, tree evaluation gets carried out many millions of times. Larger training sets mean more evaluations. Chapter 5 looks at ways of alleviating this bottleneck. In particular, Dynamic Subset Selection (DSS), has proved to be a very effective and robust method for speeding up run times and enabling GP to solve the difficult Thyroid problem to a high degree of accuracy.

2.7 Summary

There is a morass of parameters and possible variations of the GP algorithm. Without much useful theory as a guide, all that remains is re-use of suggested parameters settings by other practitioners, or seat-of-the-pants twiddling by trial and error.

There is as yet no satisfactory way of getting GP to produce trees which can successfully classify cases from more than two classes, though there is always the option of splitting the problem into a series of binary decisions.

GP is approaching its current practical limits with the Thyroid problem. Concentrating on the fitness evaluation bottleneck has produced several approaches for speeding up GP evaluations, reducing run times, and producing better solutions than GP using the standard Supervised Learning method.

Part II

A Closer Look At Genetic Programming

Chapter 3

GP Tree Representation

This chapter looks at GP's tree-based representation, with an eye towards boosting the performance of GP. Following a short reprise of the standard GP tree representation, Section 3.2 investigates the size of the GP search space, which is really very large indeed. Section 3.3 looks at variations of the standard GP tree representation, concentrating especially on the approach of Automatically Defined Functions (ADF), where GP can develop its own functions, potentially more powerful and useful than those in the original function set. The summary in Section 3.4 highlights the speed of the machine code GP implementation, and the power of ADFs, but indicates that tackling the fitness evaluation bottleneck, as in Chapters 5 and 8, provides more immediate and widely applicable improvements in GP for supervised learning problems.

3.1 The standard GP tree

The standard GP tree is a simple structure, consisting of a mixture of terminal (or leaf) nodes, and non-terminal (or function) nodes with branches. The terminal and non-terminal nodes are drawn from a set of permitted nodes. Each node, when it is evaluated, returns a value. For a terminal node, this value could be the current instantiation of a variable represented by that node, or a constant number, or it could represent an action (also known as a side-effect) such as "rotate the left wheel forward by 90 degrees". If a node does have such a side-effect, its value could be simply a constant, or it could be a value which indicates the success (or not) of the action. A function node's evaluated value usually depends on the evaluations of its subtrees (also known as arguments). Such a function node could represent the simple operation of addition, in which case its evaluated value would be the sum of the values of its two subtrees, or the function node could represent a sequence of actions, in which case each of its subtrees would be evaluated in turn, and the function node's value might be the value of its last subtree. One of the commonly used function nodes is IFLTE *(If Less Than Or Equal to)*, with four subtrees, i.e. an arity of four, shown in Figure 3.1.

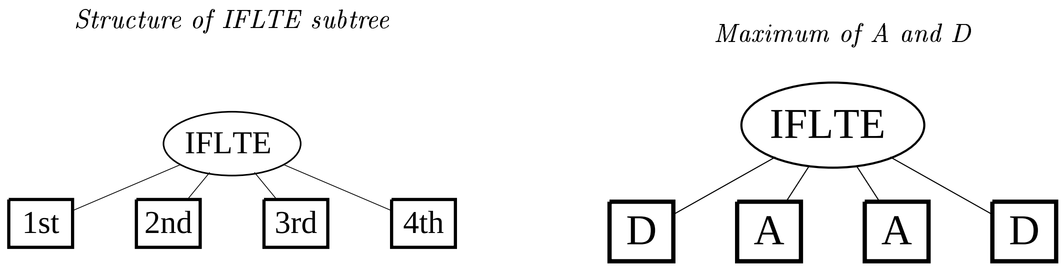


Figure 3.1: Structure of IFLTE subtree – *(If Less Than Or Equal to)*, arity=4

The tree on the left shows the structure an IFLTE subtree, and the tree on the right gives an example of IFLTE subtree in practice where it returns the maximum value of the variables A and D. If the value of the first subtree is less than or equal to the value of the second subtree, the function node's value is taken to be the value of its third subtree, otherwise it is taken to be the value of its fourth subtree. If the first subtree always has a value which is less than the second subtree, the fourth subtree of the function node IFLTE will never be evaluated.

Any and all combinations of function and terminals are permitted. To ensure that all combinations of nodes produce sensible values, i.e. closure, the function nodes are 'protected' to be able to cope with any possible value. This is easily demonstrated by the divide function. In normal arithmetic, division by zero is not defined, and would lead to a fatal error in the GP program if a division by zero was attemped. In GP, this special case is covered by defining the value of division by zero to be one, or perhaps zero. Thus, if the second argument of a division node returns a value of zero, the division function still evaluates to a sensible value. This generality is very flexible and robust, allowing any subtree to be replaced by any other subtree, whilst the overall tree can still be evaluated successfully. Very often, GP trees will not 'make sense', and consist in effect of mathematical junk. However, it is often possible to construct very powerful expressions using GP trees. Function nodes can be nested to any depth, though there is usually some restriction on overall tree size.

The question of choosing what function and terminal nodes GP is allowed to use is not straightforward. If function and terminal sets are not sufficiently powerful, GP will not be able to construct trees which can perform well on the particular problem. If the sets are too large, the search space is very large, and GP can be made even less efficient than usual. Choosing function and terminal sets for each problem is an art form, often requiring some experimentation, good knowledge of the problem, and luck, and this thesis makes no attempt to take this aspect of GP any further.

3.2 Counting Trees

The number of GP trees which can be constructed from given function and terminal sets and even with a size restriction can be very very large (obviously the number of trees is infinite without such a size restriction). This section looks at just how large that is.

It is not a simple task to count the number of trees possible with given function and terminal sets, and, of course, a restriction on tree size. Without such a restriction, the question of how many trees are possible becomes rather easy to answer. There are two main type of tree size restriction:

maximum number of nodes –
: unlimited depth, but an overall limit on the number of nodes.

maximum depth –
: a limited number of levels below the root node, though all subtrees are allowed to fill out to this depth. It is a much coarser control on tree size than a restriction on the number of nodes.

A literature search produced no easy method for calculating the number of trees possible for a given set of nodes, however it is quite straightforward to design a recursive search algorithm to do the calculation quickly. Such an algorithm for calculating the number of trees possible with a restriction on the number of nodes is as follows:

```
# Algorithm for calculating the number of trees possible
# with a restriction on the number of nodes:
#
#
# Given a maximum number of nodes, N
# Given a list of function node arities, L
# Given a number of terminals, T
#
# Store the results for the number of possible trees in a 2-D array,
# indexed by the number of subtrees and the maximum number of nodes,
#   Tree_Count[subtrees, nodes]
#
# Define the recursive COUNT_TREES_WITH_MAX_NODES algorithm, to
# calculate the number of trees with exactly N nodes, with these
# arguments:
#
# Remaining_subtrees,
#   the number of subtrees to be filled out with at least one node
# Remaining_nodes,
#   the number of nodes still to be included in the tree

define
   COUNT_TREES_WITH_MAX_NODES( Remaining_subtrees,
                               Remaining_nodes )    :-

if( Tree_Count[Remaining_subtrees,Remaining_nodes] is defined )
then
  {
    return( Tree_Count[Remaining_subtrees,Remaining_nodes] )
  }

# otherwise calculate it as follows...

if( Remaining_subtrees == Remaining_nodes )
#                                  i.e.\  can only use terminal nodes
then
  {
    Tree_Count[Remaining_subtrees,Remaining_nodes]
       = T ** Remaining_subtrees
    return( Tree_Count[Remaining_subtrees,Remaining_nodes] )
  }

if( Remaining_subtrees == 1 )
#                          have to use a function node at this point
then
  {
    Subtotal = 0
    foreach function arity F in the list L, where F < Remaining_nodes
```

```
      {
        Subtotal
           = Subtotal
             + COUNT_TREES_WITH_MAX_NODES(F,Remaining_nodes-1)
      }

    Tree_Count[Remaining_subtrees,Remaining_nodes] = Subtotal
    return( Tree_Count[Remaining_subtrees,Remaining_nodes] )
  }

# ...otherwise divide nodes amongst subtrees.
# The algorithm divides the remaining nodes between the first subtree
# and the rest of the subtrees, with the rest of the subtrees getting
# at least Remaining_subtrees-1 nodes, and the first subtree getting
# at least one node. R is the number of nodes allocated to the rest of
# the subtrees. For any particular allocation of nodes, the number of
# possibilities is the product of First_subtree_count and
# Rest_subtree_count. The total of all possible allocations gives the
# number of ways of distributing Remaining_nodes amongst
# Remaining_subtrees.

Subtotal = 0

for( R = Remaining_subtrees -1;
     R < Remaining_nodes;
     R = R + 1 )
  {
    First_subtree_count
       = COUNT_TREES_WITH_MAX_NODES(1,Remaining_nodes-R)
    Rest_subtree_count
       = COUNT_TREES_WITH_MAX_NODES(Remaining_subtrees-1,R)

    Subtotal
       = Subtotal + (First_subtree_count * Rest_subtree_count)
   }

Tree_Count[Remaining_subtrees,Remaining_nodes] = Subtotal
return( Tree_Count[Remaining_subtrees,Remaining_nodes] )

-: End of definition of COUNT_TREES_WITH_MAX_NODES
```

The number of trees with exactly N nodes is the result returned by COUNT_TREES_WITH_MAX_NODES(1,N). The algorithm works by recursively calculating the number of possible trees with less than N nodes before using that information to calculate the final value for N. It will return 0 for a particular N if it is impossible to construct a tree with exactly N nodes, e.g. with binary arity functions and even N. A similar algorithm can be constructed to calculate the number of possible trees of a particular maximum depth (i.e. where no nodes exceed the maximum depth) by using the ideas of Remaining_depth instead of Remaining_nodes.

The number of possible trees for a variety of maximum numbers of nodes N, and a variety of function and terminal sets, are given in Table 3.1 below. As can be seen, the number of possible trees with a maximum number of nodes N increases exponentially with N. The search space for the Thyroid problem increases in size by a factor 30, approximately, for each increment of N. The search space for the TicTacToe problem increases more slowly, by a factor 12, approximately. This is due to there being more variety of function and terminal nodes in the Thyroid problem.

If the trees are restricted by depth, then the largest function arity becomes the most important factor to consider. For each increment in allowed depth, the number of possible trees increases enormously quickly, much faster than with the restriction on numbers of nodes. If $Trees_D$ is the number of possible trees filled out to depth D, and A_i is the arity of function i, then

$$Trees_{D+1} = \sum_i (Trees_D)^{A_i}$$

$$Trees_0 = NumberofTerminals$$

It can be seen by inspection that the most important contribution to the increase of trees with depth comes from the largest function arity, $A_{largest}$. For the Thyroid problem, this means for each increment in depth, the number of possible full trees increases by at least a power of 4, and the number of nodes in these trees 'only' increases by a factor of 4.

It has to be said that the GP search space is rather large, rapidly reaching ten to the power of several hundred even for quite simple problems - far too large to even consider

The number of possible GP trees with N nodes		
N	Thyroid Problem, in Section 6 arities={1,1,1,1,2,2,2,2,3,4} 23 terminals	TicTacToe Problem, in Section 8.1 arities={1,2,2,2,2,2} 10 terminals
1	2.3e+01	1.0e+01
2	9.2e+01	1.0e+01
3	2.5e+03	5.1e+02
4	3.9e+04	1.5e+03
5	1.1e+06	5.3e+04
6	2.2e+07	2.6e+05
7	5.8e+08	7.0e+06
8	1.4e+10	4.6e+07
...		
50	2.7e+71	1.7e+56
51	8.2e+72	2.4e+57
52	2.4e+74	3.6e+58
53	7.3e+75	5.3e+59
54	2.2e+77	7.8e+60
55	6.5e+78	1.1e+62
56	1.9e+80	1.7e+63
57	5.8e+81	2.5e+64
58	1.7e+83	3.7e+65
...		
200	4.4e+293	2.2e+232
201	1.3e+295	3.4e+233
202	4.1e+296	5.1e+234
203	1.2e+298	7.6e+235
204	3.8e+299	1.1e+237
205	1.2e+301	1.7e+238
206	3.5e+302	2.6e+239
207	1.1e+304	3.9e+240
208	3.3e+305	5.8e+241

Table 3.1: The number of possible GP trees with N nodes

an exhaustive search. The brief calculations above merely hint at its complexity. On the face of it, GP is presented with a daunting task when it is required to search the space for useful trees. In this context, efforts to reduce the size of the search space through the use of parsimony (a penalty against large trees) and more powerful and compact functions (described below) seem well worthwhile.

3.3 Extending the Function and Terminal Sets

Data-Typing One consequence of the flexibility of the standard GP representation is that a function node can take any function or terminal nodes as arguments, i.e. function arguments, function node values, and terminal node values, all have the same data-type. This allows GP operators to combine subtrees indiscriminately and still produce legal (though not necessarily effective) trees. [Montana 95] looks at strongly-typed GP (STGP), eliminating the closure constraint. Instead of just one data type, the functions and terminals have a variety of data types, and the GP operators are restricted in what combinations of nodes are allowed, restricting the search space. Montana introduces generic functions, generic data types, and local variables. This results in GP producing a higher percentage of 'sensible' trees, though with a larger overhead for the operators. Montana looks with some success at a variety of "moderately complex problems involving multiple data types", but highlights the difficulty of defining good evaluation functions, and shows that STGP (as with most versions of GP) has difficulty scaling up to much larger problems. [Haynes *et al.* 96] extends STGP by allowing more data types.

Linear Representation [Perkis 94] looks at the use of a stack, and a linear program representation instead of the usual tree based representation. Terminals are a class of function which push preset variables onto a stack. Functions pop their arguments off the numerical stack and return their result by pushing it onto the stack. Function calls that occur with too few items on the stack simply do nothing. Perkis reports that stack-based GP can be implemented very efficiently, and works well with simple problems, but again has difficulty scaling up to larger problems.

[Nordin & Banzhaf 95] describes a compiling GP system that directly manipulates

SPARC machine code. As with Perkis's Stack-Based approach, Nordin and Bahnzaf use a linear program representation. Although this machine code approach has some limitations in that the functions are not allowed any side-effects, the GP algorithm can run two orders of magnitude faster than the usual approaches of manipulating tree representations, and with much smaller memory requirements. This increase in speed allows Nordin and Bahnzaf's GP to be successfully applied to much larger and more complex problems than before. Francone *et al* indicate that rather than being a hindrance, when compared with the less restricted standard GP tree structure, the compiling GP's linear representation performs very well on a variety of sparse data problems, [Francone *et al.* 96]. Extending their representation, Nordin and Bahnzaf have demonstrated that their compiling GP system can successfully use a close analogue of Koza's Automatically Defined Functions.

Automatically Defined Functions [Koza 92, Koza 94] introduces the idea of Automatically Defined Functions (ADF). ADF imposes a high-level structure on each tree in the GP population. Each tree thus has a result-producing branch, which is evaluated to determine the tree's fitness, and the one or more other branches provide the definitions of the one or more functions which can be referred to in the result-producing branch, all of which evolve together. Each main branch has its own function and terminal sets, and a structure-preserving crossover can only occur between the result-producing subtree of the same main branch in each parent. An example ADF tree is given below in Figure 3.2.

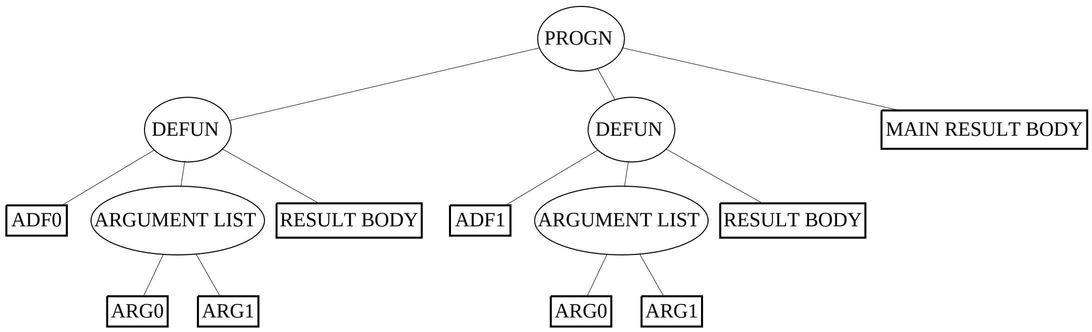


Figure 3.2: Example of an ADF tree

Only the subtrees labelled 'RESULT BODY' can be changed during the evolutionary process. In the case of this example, the tree consists of two ADF branches and a main results-producing branch. The main results-producing branch has an extra function ADF1 in its function set. The ADF1 branch has an extra function ADF0 in its function set, and two extra, local, terminals ARG0 and ARG1 which refer to the arguments of any occurrence of ADF1 (which is defined here to takew two arguments) in the main results-producing branch. The ADF0 branch has two extra, local, terminals ARG0 and ARG1 which refer to the arguments of any occurrence of ADF0 (which is also defined here to take two arguments) in the ADF1 RESULT BODY subtree. This structure enforces a hierarchical arrangement of ADFs. Care is taken to avoid recursion by not allowing an ADF branch to refer to itself i.e. not including an ADF in its own function list. Any tree-combining operations such as crossover are only permitted between equivalent branches in different trees, e.g. the ADF1 result body branch of one tree can only be crossed with the ADF1 result body branch of another. In effect, for the example above, there are three separate breeding populations of branches: the main result body, the ADF0 result body, and the ADF1 result body.

Koza demonstrates that the ADF approach is effective on problems which contain a hierarchical structure, in particular where solutions to the main problem can be constructed through a combination of solutions to easier subproblems. This is demonstrated on the Even-N Parity problem, (also covered in Section 7), where solutions for large N can comprise combinations of solutions for smaller N. The whole hierarchy can be represented by a single GP tree, though the number of ADFs must be specified in advance. For problems where there isn't a hierarchical structure for ADF to exploit, it is less clear how much ADF is of benefit to GP. The extra overheads necessary for ADF mean that it can cause GP to run more slowly and less efficiently.

Anarchically Automatically Defined Functions Early experiments in this thesis with a more flexible form of ADF, Anarchically Automatically Defined Functions (AADF), indicate that part of the success of ADF is the structure it imposes on the form of the solution. With AADF, automatic function definitions can occur anywhere and more than once within the GP tree, unlike in ADF where the tree structure places strict limits on possible function definitions. An example AADF tree is shown in Figure 3.3 below, where the 'div' function node is being redefined. Any of the functions in the function set or terminals in the terminal set could be redefined in this way including, strangely enough, the key REDEFUN function node by which the redefinitions take place.

The REDEFUN function takes three arguments:

- Left branch: the name of the function being redefined, taken to be the root node of that subtree ('div', in this example). If this subtree consists solely of a terminal node, then that terminal is being redefined.
- Middle branch: the new definition of this function (or terminal)
- Right branch: a result-producing branch in which the new definition of the function (or terminal) takes effect

Each function has a default definition that can be overridden by REDEFUN. In this example, the default definition of 'div' is protected division, i.e. its value is the result of dividing its first argument by its second argument, with checks to ensure that division by zero does not occur. The 'div' function has an arity of 2, i.e. it takes two arguments. Thus the redefined version must also have two arguments. (All the functions in this example have an arity of two or one.) In the example, where 'div' occurs in the rightmost, result-producing branch of REDEFUN, it has two arguments 'B' and 'X'. The values of these two arguments are passed to the new definition of 'div' by temporarily changing the values of 'X' (to be the first argument, i.e. the value of 'B') and 'Y' (to be the second argument, i.e. the value of 'X') within the middle, redefinition branch of REDEFUN. In the example, 'X' is being used within the redefinition, but 'Y' is not, so the value of the second argument of the 'div' node in the result-producing branch has effectively been ignored within the redefinition branch.

'X' and 'Y' have been added to the terminal set specially to allow REDEFUN to define functions up to an arity of two. Redefinitions of any of the arity two functions will involve the use of 'X' and 'Y' in this way. Redefinitions of any arity 1 functions, e.g. 'SQRT', will only make use of 'X'. In other function sets, with higher arity functions, more terminals would need to be used to pass the values of the functions' arguments into the redefinition. The 'X' and 'Y' (and however many other argument-related) nodes would normally be initialised with some simple default values, e.g. 1.

As can be seen in the example tree in Figure 3.3, the new behaviour of 'div' is to multiply its first argument by the value of 'A', ignoring the value of its second argument. Thus, in the result branch of the REDEFUN node in the example, the value of the 'div' subtree is B multiplied by A, rather than B divided by X.

The same approach can be used to redefine terminal nodes. In this case, the left-most REDEFUN subtree would consist simply of one terminal node, the middle subtree would be its new definition, possibly making use of the original value of the terminal. Unlike in the redefinition of function nodes, as described above, the two terminals X and Y would not be redefined, since the terminal node takes no arguments. The right-most subtree would be the result-producing branch in which the new definition of the terminal takes effect.

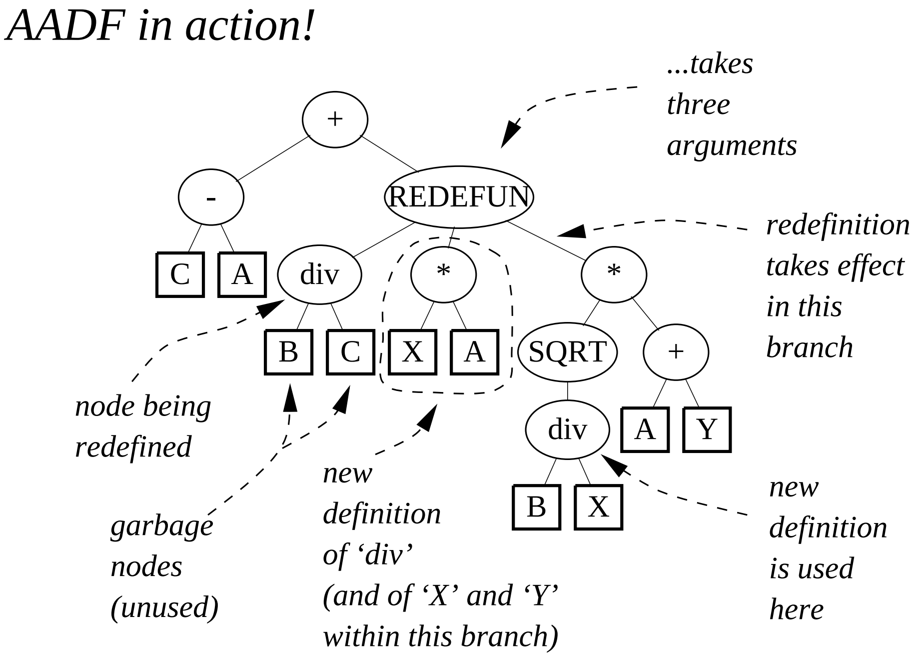


Figure 3.3: example AADF tree

This AADF representation is extremely flexible and almost totally useless. Like the basic GP representation, it has closure, so that any function node can take any function or terminal nodes as arguments. It can allow recursion, iteration, redefinition of any function or terminal, and hierarchical definitions, all at any location within the tree, and as often as any restrictions on tree size permit. Recursion can be avoided or controlled by providing a default behaviour if a new definition refers to itself, e.g. the definition branch of the REDEFUN node can only make use of an earlier definition of the node being redefined. It makes sense to put a block on the REDEFUN node being redefined.

Extremely powerful and flexible trees can be constructed using AADF. By adding a REDEFUN at the top of a tree, all instances of the redefined node have their behaviour changed simultaneously. It is possible to create function hierarchies of arbitrary depth, whereas in ADF the hierarchy is defined at the start of the run. However, such AADF trees are also very unlikely to occur during the evolution of a population, since several parts of a tree have to be right simultaneously for an instance of a REDEFUN node to be effective. This is extremely unlikely to say the least. Early, discouraging, experiments with several variations of AADF indicate that GP is totally unable to take advantage of such flexibility. The REDEFUN nodes and their associated extra subtrees behave much like spurious junk nodes, with no impact on the trees' fitnesses, except perhaps bringing about a larger parsimony penalty. Much care and much more thought is needed to enable GP to take advantage of AADF. One possibility is to cause certain links between nodes to be made inviolate, i.e. the reproduction operators prevented from splitting the trees at those points. Another possibility is to include rewards in the fitness function for making use of the extra features. This might enable GP to retain the extra complications long enough to make use of them.

Adaptive Representation (AR) [Rosca & Ballard 94] looks at the discovery of useful subtrees (building blocks) in a population, generalising them, and adding them to the function set, in effect 'adapting the problem representation on-the-fly'. The next generation of trees can then make use of these new and hopefully more powerful functions, allowing GP to construct a hierarchy of new function definitions in the function set for the entire population to exploit. Although this approach requires lots of extra processing of the population, Rosca and Ballard state that "all new building blocks can be discovered in O(population size) time". Each time a new function is added, the population goes through 'considerable' changes as it evolves to take advantage of it. Rosca and Ballard use this adaptive representation to tackle the Even-N-Parity problem (also covered in Chapter 7) up to $N = 11$, showing that AR compares well with ADF in terms of computational effort, scaling up well to the larger N. As with ADF, AR is an effective hierarchical approach to problem solving with GP.

Both ADF and AR enable GP to explore a search space of smaller trees, by allowing GP to use smaller, more powerful trees, rather than larger unwieldy trees with less powerful components that are more prone to being split up during reproduction.

3.4 Summary

This chapter has taken a brief look at GP's tree-based representation, with an eye towards boosting the performance of GP. The number of possible GP trees is huge, and is most dependent on the largest arity of the functions in the function set. The basic speed of the GP algorithm can be boosted by two orders of magnitude by directly manipulating machine code segments, though with several restrictions such as using a linear representation with a limited number of possible instructions, and the functions and terminals not having any side-effects. However, it still performs very well and is, of course, exceedingly fast, so that it seems worthwhile persevering despite the limitations it might have. Koza's Automatically Defined Functions and Rosca and Ballard's Adaptive Representation can both take advantage of hierarchical structure inherent in several difficult problems, allowing GP to solve problems of far greater complexity than it could manage with the standard representation. However, it is not clear how well ADF and AR would perform on similarly difficult problems without such an exploitable hierarchical structure.

Although obviously powerful additions to the GP toolkit, the compiling GP system, ADF, and AR, all would require a fairly substantial modification or rewrite of an existing implementation of GP. This thesis concentrates on some modifications to standard GP which are easier to implement, such as the more complex fitness function of Dynamic Subset Selection, in Chapter 5, which should work well with all of the extensions to GP mentioned above.

Chapter 4

GP Tree Recombination and Selection

There have been many studies of the performance and effects of operators in GAs, and rather fewer for GP. [Koza 92] looks at the standard operators, insisting that Crossover is essential to GP performance. O'Reilly and Oppacher in several studies, [O'Reilly & Oppacher 96, O'Reilly & Oppacher 92, O'Reilly & Oppacher 94b, O'Reilly & Oppacher 95a], concentrate on designing powerful Mutation operators which function with a variety of Hill-Climbing techniques, (i.e. they do not require either a population or Crossover), or looks at other hybrids involving Crossover. [O'Reilly & Oppacher 95b] looks at a GP version of the GA Schema Theorem, [Holland 75], the main backbone of GA theory, and finds that it does not transfer well to GP, concluding amongst other things that it "constitutes a narrow and imprecise account of GP search behaviour."

The most common GP operators are simple, and inefficient, i.e. they have a low likelihood of producing children which are as fit or fitter than their parents. This thesis does not look any further at ways of improving GP operators, which would be highly problem specific (but see [Vere 95] for work on efficient operators working on decision trees, and [Montana 95, Haynes *et al.* 96] for work on Strongly-Typed GP, where operators are restricted in how they can alter trees). The use of a restriction on tree size is also common to many GP implementations, and this can be seen in Section 4.1 to interact adversely with the main GP operator, Crossover.

Tournament Selection is a widely used method for picking individuals from the population as parents for the operators to work on, and is the selection method used throughout this thesis. Section 4.2 takes a look at some of the consequences of using Tournament Selection and some reasons why it was used in this thesis.

4.1 Crossover and the MAX problem

The Crossover operator is common to most implementations of GP, providing a simple but powerful method for recombining genetic material in a population. Crossover seems to be in widespread use in its simplest form, described in Section 4.1.3, mixing two parent trees through the exchange of randomly selected subtrees to produce one or two child trees. Mutation operators are often used in combination with Crossover. Also common to most GP is some form of upper limit on tree size, necessary to prevent the population expanding to exceed available computer resources.

This section introduces the MAX problem for GP, a convenient mechanism for looking at the machinations of Crossover. The task is to produce the largest possible value for a given function and terminal set and maximum tree depth or maximum number of nodes. Ostensibly an easy problem for GP to solve, results for several variations of the MAX problem, given in Section 4.1.5, confirm some inadequacies of the crossover operator in normal use. These are highlighted in an analysis in Section 4.1.6. Even with the mitigating effects of some mutation operators, described in Section 4.1.3, a loss of diversity in the upper levels of trees in the population due to Crossover, discussed in Sections 4.1.5 and 4.1.7, leads to premature convergence to sub-optimal solutions. This is made irreversible through the interaction of Crossover and the restriction on tree depth.

The tendency of Crossover to ignore the upper tree levels should be well known, but the extent of its negative impact on population diversity and premature convergence are made more apparent here through the use of the MAX problem.

This section is an extension of the paper [Gathercole & Ross 96], in which the MAX problem was first published, which looked solely at a restriction on tree depth. With a restriction on the number of nodes, the MAX problem is more complex for GP, experiencing more subtle interactions between the action of the operators and the tree size restriction. Langdon and Poli take the MAX problem further in [Langdon & Poli 97], concentrating on the restriction on tree depth, but considering bigger trees, different selection pressures, different initialisations of the population, measuring population variety, and the number of steps required to solve the MAX problem.

4.1.1 Why Restrict Tree Size

GP effortlessly takes computers beyond their limits both in terms of memory and CPU use. Ways of reducing CPU use are investigated in Chapters 5 to 8. Two of the easiest ways for reducing GP's memory requirements are restricting the population size, also investigated in Chapter 8, and restricting individual tree sizes within the population. Both of these methods limit GP's use of memory. This section looks at some consequences of imposing restrictions on tree size.

Trees in a GP population have a tendency to 'bloat'. This phenomenon, noted in [Blickle & Thiele 94], might be explained by the fact that larger trees (i.e. ones which contain more garbage or redundancy in the form of superfluous subtrees) are more likely to survive the actions of Crossover undamaged. Smaller trees are likely to result in damaged, unfit trees after Crossover. Whilst nice from a perspective of wishing the trees well, the bloat phenomenon can be a hindrance to the GP user. Another factor could be 'hitch-hiking', where superfluous subtrees benefit from their proximity in fit trees to fit subtrees. Crossover and selection are quite likely to copy and spread the associated non-contributory subtrees along with the fit subtrees. The trees in the GP population expand with each generation, requiring a larger memory allocation, and can result in a reduction in CPU efficiency.

The open-ended nature of bloating is questioned in [Rosca 96]. Rosca suggests the existence of "size attractors", where trees in a population will expand to a certain size range and then fluctuate within this range without continuing to expand indefinitely. Rosca also questions the bias commonly introduced into GP runs in favour of small trees, suggesting that the generalisation capabilities of such small trees are less than those of larger trees. To avoid these difficulties, Rosca proposes an Adaptive Representation, explicitly evolving and selecting code modules instead of entire trees (described in Section 3.3).

In theory, for a classification problem, it is usually possible to construct a huge GP tree which can perform 100% successfully on the training set by explicitly dealing with every case in the training set. In effect, the tree memorises the entire training set - an extreme form of overfitting. Such a tree is unlikely to perform well on a different test set, where the individual cases are not the same as those in the training set. This is the dilemma of generalisation versus memorisation (overfitting). Ideally, GP should produce small trees which contain the essence of what is needed to solve all possible cases, having generalised from the training set to all possible cases. One of the simplest methods of biasing GP towards generalising rather than memorising is to prefer smaller trees to larger trees in the selection process, and is sometimes known as the principle of Occam's Razor. A standard approach used in Machine Learning is to train using just the training set, and regularly test the best individual using the test set, stopping when the performance on the test set begins to worsen. It has been hypothesised in discussions within the GP community that GP is quite resistant to overfitting. Such overfitting was not apparent during the runs of GP in this thesis. Best-of-generation trees' performances on test sets were always still improving towards the end of runs if the training performance was improving. Perhaps the runs never ran long enough to reach a situation where the test performances would begin to worsen.

Looking further at the generalisation capabilities of smaller versus larger decision trees (i.e. not specifically GP trees), [Webb 96] questions the "utility of Occam's Razor" as a guiding principle in Machine Learning. Starting with small, simple decision trees, Webb adds complexity to them without affecting their performance on the training sets, using only the training sets as a guide, then compares their performance on the associated test sets. The larger trees generalise better. Whilst not demonstrating that larger GP trees generalise better than smaller GP trees, Webb's study does suggest that the bias towards smaller trees should be considered carefully.

Nevertheless, despite possible good reasons for allowing GP trees to grow unbounded, it is often impractical to let them do so. Some form of tree size restriction is necessary. There are four main types of size restriction reported in the literature: two of which set explicit size limits, one which edits trees to remove superfluous nodes, and one which biases GP against selecting larger trees as parents.

- A limit on the number of nodes in a tree

  - each time a new tree is generated in the breeding stage, it is scanned, and the number of nodes counted.

- A limit on the depth of a tree

  - each time a new tree is generated in the breeding stage, it is scanned, and the depth calculated, i.e. the longest path from the root node to any leaf node.

If the child tree size exceeds the limit, some form of default is applied, e.g. the child tree is made into a simple copy of its parent, or the breeding step is repeated until it generates a 'legal' tree, or the too-large tree is pruned in some way down to size. Each of these methods for dealing with the size limit has consequences for the direction of the evolution of the GP population.

- Tree editing

  - there are several ways of explicitly reducing the size of trees. [Soule *et al.* 96] looks at the removal of known and obviously superfluous subtrees. Such subtrees can be calculated in advance using the properties of the function set. However, Soule *et al* then noted the spreading of super-superfluous subtrees, i.e. superfluous subtrees which avoided the culling process.

  Another, more subtle and directed form of editing is described in [Blickle & Thiele 94, Blickle 96]. The edges of each GP tree traversed during evaluation are marked. Unmarked edges are those which did not contribute to the fitness of the tree, and those subtrees are replaced by randomly chosen terminals.

- Parsimony

  - a penalty is added to the fitness value of a tree proportional to its size, i.e. trees with more nodes have a larger penalty included in their fitness values. This method can function without any explicit upper limit on tree size. Instead, it biases the selection of parents towards smaller trees. In effect it turns a GP problem into a multi-objective optimisation problem, where the fitness of a tree

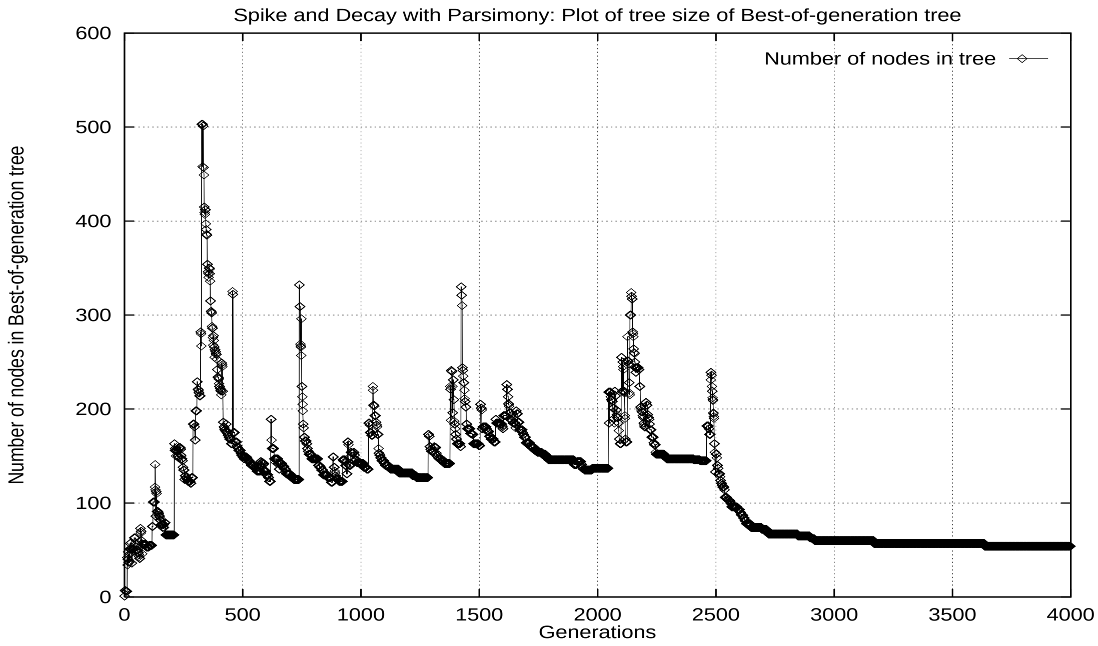


Figure 4.1: Spike and Decay with Parsimony - shows a typical profile of the tree size of the best of generation individual, as parsimony provides a bias towards the selection of smaller but equally fit variants of the best of generation individual (showing a decay in tree size), without hindering the discovery of new, fitter, but much larger trees, (showing a spike in tree size).

depends both on its performance on a problem and on its size. The bias against large trees can be made very weak when the size of the penalty is always less than the smallest unit of fitness. In classification problems this is easy to implement, since the smallest unit would be 1, corresponding to a misclassification of a single case. With a penalty factor of $0.001 * NumberOfNodesInTree$, the parsimony factor would only have an effect on two equally fit but different sized trees, as long as the trees never exceeded 999 nodes in size. For other problems with a more fine grained fitness function, a decision has to be made on the impact of the parsimony on the fitness value.

[Zhang & Muehlenbein 95] looks at an "adaptive balancing of accuracy and parsimony", a method for automatically varying the size of the parsimony penalty according to the quality of the current best solution. The parsimony penalty starts low, and is increased as the quality of the best solution increases.

In use, parsimony has the desired effect of restricting tree growth. During a typical run, the trees expand quickly to a certain, problem (and parameter)-related size range. New, fitter trees which are the result of Crossover are often much larger than the population average. These fitter trees will continue to be selected, but any child trees which have the same fitness but fewer nodes, perhaps through the action of a mutation operator snipping out a superfluous subtree, will be preferentially selected.

Looking at Figure 4.1, a typical run taken from Chapter 8 on the TicTacToe problem, showing the size of the best of generation tree as the generations progress, a spike-decay curve is evident several times during the course of this typical run. The 'spike' corresponds to the discovery of a fitter but very large individual. The 'decay' corresponds to the discovery of equally fit but smaller variations of the tree. The final spike which occurs just before generation 2500, coincides with the discovery of an optimum tree which scores 100% on the training set. As the run continues, selection pressure favours smaller but equally fit variants of this tree, resulting in a classic decay curve.

In practise, if the difficulty of the problem is not known, and it is likely that the population will expand to exceed the memory allocation even with the bias towards selecting smaller trees, parsimony is used in combination with one of the explicit limits on tree size mentioned above.

In this thesis, every effort was made to allow the GP trees to expand without any explicit limits, but with a weak parsimony factor biasing selection towards smaller trees. With large populations, e.g. 5000, in Section 6, this was not possible, and an explicit size restriction was needed. With smaller populations, e.g. 400, and 50, in Sections 7 and 8, there was sufficient memory to allow the trees to grow unbounded to their 'natural' size.

The MAX problem, described below, looks at the consequences of a strict upper limit on tree size interfering with the actions of the Crossover operator when the trees in the population have expanded close to the limit.

4.1.2 The MAX Problem

The MAX problem was constructed specifically to investigate what happens when the trees in a GP population expand to reach an explicit restriction on tree size. The task is to find a tree which returns the largest possible value for a given terminal and function set, with a depth limit, D, or a limit on the number of nodes, N. No trees are allowed to exceed the size restriction. GP is given a maximum number of generations (the cutoff) in which to find an optimal tree, after which it is considered to have failed. The cutoff is made sufficiently large so as to give GP a good chance of finding the optimal tree before reaching the generation limit. In every successful run of the MAX problem the number of generations needed to find the optimal tree was much less than the cutoff.

The MAX problem for GP is analogous to the Ones-Max problem for GAs, [J.D.Schaffer *et al.* 91], where an individual consists of a fixed length binary string (since GP uses a non-linear representation, this analogy can only be a very loose one), and its fitness is simply the sum of its bit values. The optimal solution has all of its bits set to on. Although a simple problem, in practice a GA's population often converges to a state where every individual has some bits set to off in the same positions as every other individual, if the chromosome is sufficiently long. Thus GA Crossover, the usual method for recombining individuals where substrings are taken from two or more parents to create a child, will result in new individuals with the same bits set to off. GA Mutation is then the only operator which can change the off bits to on, resulting in only a slow progression of the population towards discovering the optimal solution. The process by which off-bits turn up at the same positions in each individual in the population is known as hitch-hiking, and is a consequence of using the crossover operator and selection. When substrings are recombined by Crossover to produce fit individuals, the fit individuals get favoured by the selection process and any off-bits get carried along for the ride. Soon all individuals in the population are the same as or close copies of the fittest individuals, all sharing the same off-bits.

For GP and the MAX problem, since the optimal trees will, by necessity, need to extend to the maximum depth (or number of nodes), the size restriction in the MAX problem is obviously a more important factor than in most other GP problems. It becomes apparent below that the interactions between the function and terminals sets, size restriction, Crossover, and selection pressure, can combine to make it very difficult for GP to find an optimal tree. In fact, the problem can be considered deceptive for GP, where the fitness contributions of subtrees discovered early on in the search lead GP astray, away from discovering the necessary subtrees later on.

There are several advantages of the MAX problem for looking at GP as stated: results are known quickly; the solution space is easy to visualise; the optimal trees are known in advance; the problem can be varied and made more difficult in small steps.

The MAX problem can be expressed as

- MAX-depth-D-{Function Set}{Terminal Set}
- MAX-nodes-N-{Function Set}{Terminal Set}

representing the two different size restrictions. The different versions of the MAX problem covered in this thesis are as follows:

MAX-depth-D-{Function Set}{Terminal Set}

The simplest form of the MAX problem is **MAX-depth-D-{+}{1}**, where the only optimal solution (shown in Figure 4.2, for $D = 4$) is a full tree of '+'s and '1's. For a given depth D, where the root node is counted as depth 0, the maximum possible tree value is 2^D. The depths tested are from 3 to 8.

In **MAX-depth-D-{*,+}{n}**, the '*' (times) function is of no use in terms of returning large values unless both its arguments are greater than 1. This requires the use of the '+' function to add the small terminals together, creating values greater than 1. So the '+' function is most effective near the leaves of the tree. The '*' function is most effective at the top of the tree, if sufficiently large values are returned by the subtrees.

For **MAX-depth-D-{*,+}{1}**, the optimal tree, shown in Figure 4.3 for $D = 4$, consists of '1's in the bottom layer, '+'s in the penultimate layer, '+'s or '*'s in the

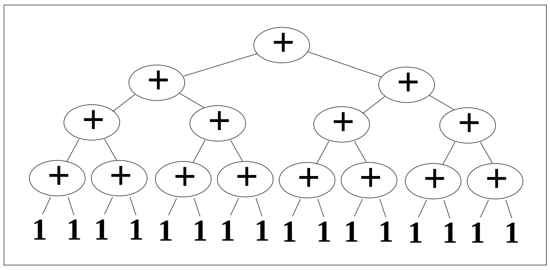


Figure 4.2: Optimal Tree for MAX-depth-4-{+}{1}

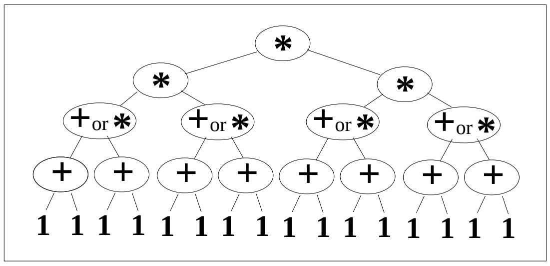


Figure 4.3: Optimal Tree for MAX-depth-4-{*,+}{1}

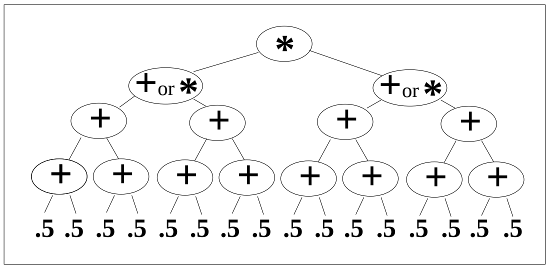


Figure 4.4: Optimal Tree for MAX-depth-4-{*,+}{0.5}

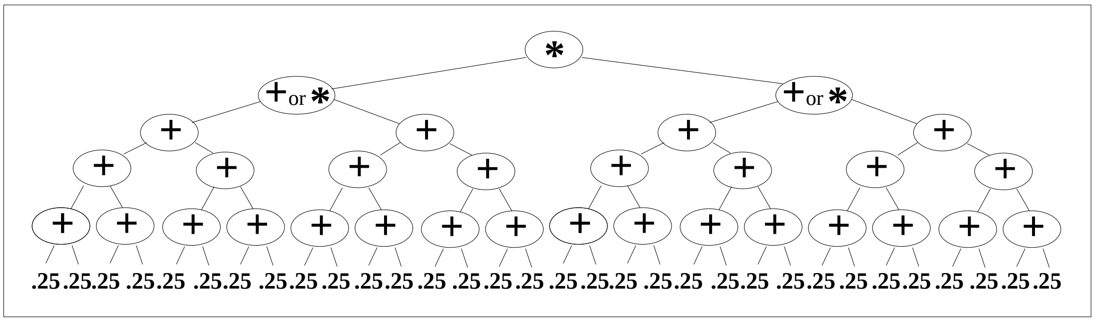


Figure 4.5: Optimal Tree for MAX-depth-5-{*,+}{0.25}

next-to-penultimate layer, and '*'s in any layers above that. For a given depth D, the maximum possible tree value is $4^{2^{D-2}}$, where $D >= 2$, and there are 2^{D-2} distinct optimal trees. The depths tested are from 3 to 6.

The MAX problem can be made progressively more difficult for GP by decreasing the size of the constant terminal (the reasons for this are described below in Section 4.1.7). To keep the arithmetic simple, the constants have been kept to inverse powers of 2. With **MAX-depth-D-{*,+}{0.5}**, the constant has been reduced from 1 to 0.5, and now an optimal tree consists of one more layer of '+'s and one less layer of '*'s. The generic optimal tree is shown in Figure 4.4 for $D = 4$. For a given depth D, the maximum possible tree value is $4^{2^{D-3}}$, where $D >= 3$, and there are 2^{D-3} distinct optimal trees. The depths tested are from 3 to 7.

With **MAX-depth-D-{*,+}{0.25}**, the constant has been reduced again to 0.25, with an extra layer of '+'s in the optimal tree as a consequence, shown in Figure 4.5 for $D = 5$.

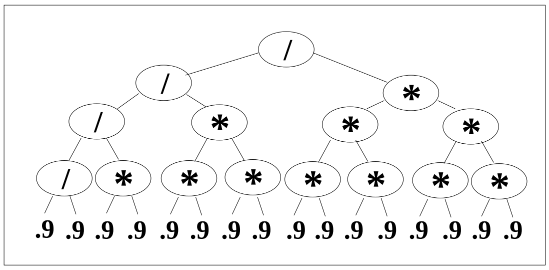


Figure 4.6: Optimal Tree for MAX-depth-4-{*,/}{0.9}

The variation **MAX-depth-D-{*,/}{0.9}** involves the '/' (divide) function instead of the '+' function. The two functions, '*' and '/', must be combined together asymmetrically to create an optimal tree. The '*' function's role becomes that of providing as small a value as possible which, as the second argument of the '/' function, is turned into a large value. The optimal tree for MAX-depth-D-{*,/}{0.9}, shown in Figure 4.6 for $D = 4$, has a different structure from the optimal tree for the MAX-depth-D-{*,+}{n} problems. Most of the tree layers consist of an unbalanced mix of the two functions.

MAX-nodes-N-{Function Set}{Terminal Set}

A subset of the variations described above are attempted with a restriction on the number of nodes instead of depth. The optimal trees for the MAX-nodes-N variations are more complex than for MAX-depth-D. For this reason, MAX-nodes-N-{*,/}{0.9} is not investigated further here, since it isn't immediately obvious what an optimal tree would look like. For the other variations, MAX-nodes-N-{*,+}{n}, the optimal trees have been established. Unlike with the MAX-Depth-D versions, there are many optimal trees for each problem, and the search spaces are much much larger.

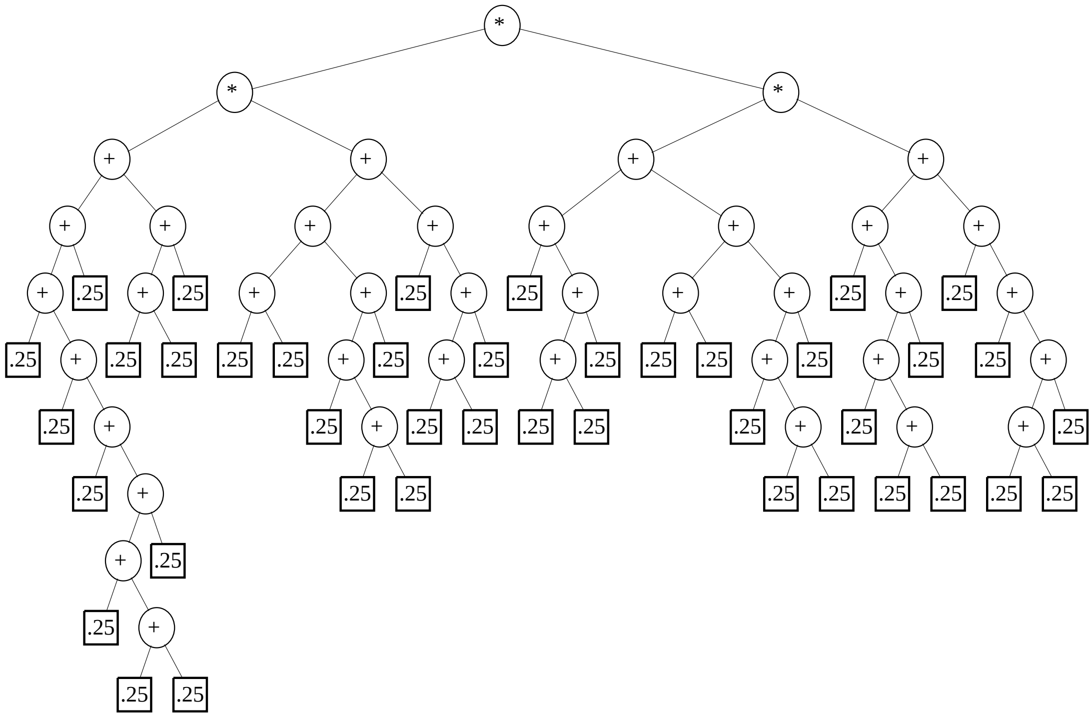


Figure 4.7: An optimal tree for MAX-nodes-81-{*,+}{0.25}

One of the many optimal trees for MAX-nodes-81-{*,+}{0.25} is shown in Figure 4.7. As with the optimal trees for MAX-depth-D, the tree has '*' nodes near the root joining subtrees consisting entirely of '+'s and the constant 0.25, in this case. There are 41 terminal nodes, 38 '+'s, and 3 '*'s. Three of the four {+,0.25} subtrees have ten constants and nine '+'s each, with a return value of 2.5. The other subtree has eleven constants and ten '+'s, with a return value of 2.75. The four subtrees are joined by three '*'s to give an optimal return value of $2.75 * 2.5 * 2.5 * 2.5 = 42.96875$.

No set formula is given here for the tree structure or return value of an optimal tree for each MAX-nodes-N problem. The proof of such a formula is not straightforward, though it is made easier by sticking with the binary arity functions '+' and '*'. The problem of constructing provably optimal trees by hand is an interesting topic in its own right. Instead, a quick enumeration algorithm was constructed to generate the structure and hence return value of an optimal tree for a given N. The following guidelines can be used to speed up the algorithm to find an optimal tree.

A MAX-nodes-N-{*,+}{n} tree can be viewed as

$$(n+n+n+...)*(n+n+..)*(n+n+n+n...)*...$$

made up of some subtrees consisting only of '+'s and 'n', combined together by some '*'s. This can be written as

$$TreeValue = \prod_{i=1}^{t} S_i$$

where t is the number of subtrees, S_i, containing only '+'s and 'n's, and

$$S_i = \sum_{1}^{C_i} n = n * C_i$$

where 'n' is the value of the constant node, and C_i is the number of constant nodes in subtree S_i, with the following constraint on the total number of nodes to satisfy the size restriction

$$t - 1 + \sum_{i=1}^{t}(C_i + C_i - 1) = N$$

where N is the maximum allowed number of nodes, i.e. the number of '*' nodes joining the t subtrees (i.e. t-1) plus the sum of the number of '+' and 'n' nodes in each subtree must be no greater than N. An optimal tree would use all N nodes, assuming N is odd, otherwise it could only use N-1 nodes, being unable to incorporate the remaining node into the binary tree (a binary tree can only have an odd number of nodes). More specifically, when N is odd,

$$\sum_{i=1}^{t} C_i = \frac{N+1}{2}$$

since the number of terminal nodes is one greater than the number of function nodes (with the proviso that the function nodes are binary). All the remaining $\frac{N-1}{2}$ nodes are function nodes. It is simple to show that if $C_i > C_j + 1$, for some i and j, (i.e. one subtree has at least two more constant nodes that another subtree), the overall tree value can be increased by decrementing the larger C_i and incrementing the smaller C_j. Thus all the {+,n} subtrees in an optimal tree have the same number of nodes, or differ only by one '+' and one 'n'. For sufficiently small N, there is a cutoff point where the optimal tree need contain no '*' nodes. For the values of the constant n used here, i.e. 1, 0.5, and 0.25, this cutoff point can be calculated from

$$n * \frac{N+1}{2} \leq 4$$

i.e. when there are insufficient nodes to construct a {+,n} subtree with a return value greater than 4. If a {+,n} subtree has a return value greater than 4, it can be split into two smaller {+,n} subtrees, and combined using '*' to produce a greater return value than before. For larger values of the constant n, an extra constraint would be needed.

Using all the above criteria, a quick enumeration of the few remaining possibilities for a given N results in a list of the number of terminal nodes in each {+,n} subtree. From here it is straightforward to calculate the optimal tree value, and the various permutations of these subtrees to give all the optimal tree structures.

A MAX-nodes-N-{*,+}{n} tree can be represented by a list of numbers, where each element in the list is the number of constants in a {+,n} subtree. The length of the list gives the number of such subtrees. These subtrees are then combined by '*' nodes to produce the final tree. For example, the list (11,10,10,10) represents all the optimal trees for MAX-nodes-81-{*,+}{0.25}, one of which is displayed in full in Figure 4.7. The total number of trees for a given list, such as (11,10,10,10), can be calculated as follows (making use of the algorithm for calculating the number of possible tree with a given number of nodes, and for given function and terminal sets, shown in Section 3.2):

Given the list of numbers of constants in subtrees, (11,10,10,10):

$subtree_{11} = 16796$, the number of subtrees with 11 'n' nodes (and 10 '+' nodes)

$subtree_{10} = 4862$, the number of subtrees with 10 'n' nodes (and 9 '+' nodes)

The number of ways of combining 4 subtrees and 3 '*' nodes, where 3 subtrees are the same size, is the number of permutations of 4 objects (with 3 being identical) times the number of ways of combining 4 subtrees (where each subtree can be considered as a terminal node) and 3 '*' nodes, $subtree_4$.

$SubtreePermutations = 4 * subtree_4 = 4 * 5 = 20$

Thus the total possible number of trees from {11,10,10,10} is

$$NumberOfTrees = SubtreePermutations * subtree_{11} * (subtree_{10})^3 \simeq 8 * 10^{15}$$

In Table 4.1, the optimal tree value, the number of possible optimal trees, the number of trees possible up to and including the size limit, the list of subtree sizes, and the list of subtree values, are given for a range of values of N for the problem MAX-nodes-N-{*,+}{0.25}. A periodic variation is apparent in the list of subtree sizes whose period increases as N increases. Table 4.2, for MAX-nodes-N-{*,+}{0.5}, and Table 4.3, for MAX-nodes-N-{*,+}{1}, show similar features to that for MAX-nodes-N-{*,+}{0.25}. The period of the cycle is shorter for $n = 0.5$, and shorter still for $n = 1$, due to the fact that the {+,n} subtrees with larger constants need fewer nodes to produce a sufficiently large return value for the '*' nodes to be effective.

Details of the optimal trees for MAX-nodes-N-{*,+}{0.25}, where $27 \leq N \leq 97$					
N	MAX Value	Optimal trees	Possible trees	Constants per subtree	Subtree values
27	3.5	7.43e+05	7.08e+09	14	3.5
29	3.75	2.67e+06	5.09e+10	15	3.75
31	4.	9.69e+06	3.69e+11	16	4
33	4.5	1.23e+06	2.69e+12	9 8	2.25 2
35	5.0625	2.04e+06	1.97e+13	9 9	2.25 2.25
37	5.625	1.39e+07	1.45e+14	10 9	2.5 2.25
39	6.25	2.36e+07	1.07e+15	10 10	2.5 2.5
41	6.875	1.63e+08	7.95e+15	11 10	2.75 2.5
43	7.5625	2.82e+08	5.93e+16	11 11	2.75 2.75
45	8.25	1.97e+09	4.43e+17	12 11	3 2.75
47	9.	3.46e+09	3.32e+18	12 12	3 3
49	9.75	2.45e+10	2.50e+19	13 12	3.25 3
51	10.5625	4.33e+10	1.88e+20	13 13	3.25 3.25
53	11.390625	2.92e+09	1.42e+21	9 9 9	2.25 2.25 2.25
55	12.65625	2.98e+10	1.08e+22	10 9 9	2.5 2.25 2.25
57	14.0625	1.01e+11	8.16e+22	10 10 9	2.5 2.5 2.25
59	15.625	1.15e+11	6.20e+23	10 10 10	2.5 2.5 2.5
61	17.1875	1.19e+12	4.72e+24	11 10 10	2.75 2.5 2.5
63	18.90625	4.11e+12	3.60e+25	11 11 10	2.75 2.75 2.5
65	20.796875	4.74e+12	2.74e+26	11 11 11	2.75 2.75 2.75
67	22.6875	4.98e+13	2.10e+27	12 11 11	3 2.75 2.75
69	24.75	1.74e+14	1.61e+28	12 12 11	3 3 2.75
71	27.	2.03e+14	1.23e+29	12 12 12	3 3 3
73	29.25	2.16e+15	9.45e+29	13 12 12	3.25 3 3
75	31.6875	7.63e+15	7.26e+30	13 13 12	3.25 3.25 3
77	35.15625	6.57e+14	5.58e+31	10 10 10 9	2.5 2.5 2.5 2.25
79	39.0625	5.59e+14	4.30e+32	10 10 10 10	2.5 2.5 2.5 2.5
81	42.96875	7.72e+15	3.31e+33	11 10 10 10	2.75 2.5 2.5 2.5
83	47.265625	4.00e+16	2.56e+34	11 11 10 10	2.75 2.75 2.5 2.5
85	51.9921875	9.21e+16	1.97e+35	11 11 11 10	2.75 2.75 2.75 2.5
87	57.19140625	7.96e+16	1.52e+36	11 11 11 11	2.75 2.75 2.75 2.75
89	62.390625	1.11e+18	1.18e+37	12 11 11 11	3 2.75 2.75 2.75
91	68.0625	5.85e+18	9.12e+37	12 12 11 11	3 3 2.75 2.75
93	74.25	1.36e+19	7.06e+38	12 12 12 11	3 3 3 2.75
95	81.	1.19e+19	5.47e+39	12 12 12 12	3 3 3 3
97	87.890625	4.00e+18	4.24e+40	10 10 10 10 9	2.5 2.5 2.5 2.5 2.25

Table 4.1: Details of the optimal trees for MAX-nodes-N-{*,+}{0.25}

Details of the optimal trees for MAX-nodes-N-{*,+}{0.5}, where $27 \leq N \leq 99$					
N	MAX Value	Optimal trees	Possible trees	Constants per subtree	Subtree values
27	12.5	2.94e+03	7.08e+09	5,5,4	2.5,2.5,2
29	15.625	2.74e+03	5.09e+10	5,5,5	2.5,2.5,2.5
31	18.75	2.47e+04	3.69e+11	6,5,5	3,2.5,2.5
33	22.5	7.41e+04	2.69e+12	6,6,5	3,3,2.5
35	27.	7.41e+04	1.97e+13	6,6,6	3,3,3
37	31.5	6.99e+05	1.45e+14	7,6,6	3.5,3,3
39	39.0625	3.84e+04	1.07e+15	5,5,5,5	2.5,2.5,2.5,2.5
41	46.875	4.61e+05	7.95e+15	6,5,5,5	3,2.5,2.5,2.5
43	56.25	2.07e+06	5.93e+16	6,6,5,5	3,3,2.5,2.5
45	67.5	4.15e+06	4.43e+17	6,6,6,5	3,3,3,2.5
47	81.	3.11e+06	3.32e+18	6,6,6,6	3,3,3,3
49	97.65625	5.38e+05	2.50e+19	5,5,5,5,5	2.5,2.5,2.5,2.5,2.5
51	117.1875	8.07e+06	1.88e+20	6,5,5,5,5	3,2.5,2.5,2.5,2.5
53	140.625	4.84e+07	1.42e+21	6,6,5,5,5	3,3,2.5,2.5,2.5
55	168.75	1.45e+08	1.08e+22	6,6,6,5,5	3,3,3,2.5,2.5
57	202.5	2.18e+08	8.16e+22	6,6,6,6,5	3,3,3,3,2.5
59	244.140625	7.53e+06	6.20e+23	5,5,5,5,5,5	2.5,2.5,2.5,2.5,2.5,2.5
61	292.96875	1.36e+08	4.72e+24	6,5,5,5,5,5	3,2.5,2.5,2.5,2.5,2.5
63	351.5625	1.02e+09	3.60e+25	6,6,5,5,5,5	3,3,2.5,2.5,2.5,2.5
65	421.875	4.07e+09	2.74e+26	6,6,6,5,5,5	3,3,3,2.5,2.5,2.5
67	506.25	9.15e+09	2.10e+27	6,6,6,6,5,5	3,3,3,3,2.5,2.5
69	610.3515625	1.05e+08	1.61e+28	5,5,5,5,5,5,5	2.5,2.5,2.5,2.5,2.5,2.5,2.5
71	732.421875	2.21e+09	1.23e+29	6,5,5,5,5,5,5	3,2.5,2.5,2.5,2.5,2.5,2.5
73	878.90625	1.99e+10	9.45e+29	6,6,5,5,5,5,5	3,3,2.5,2.5,2.5,2.5,2.5
75	1054.6875	9.96e+10	7.26e+30	6,6,6,5,5,5,5	3,3,3,2.5,2.5,2.5,2.5
77	1265.625	2.99e+11	5.58e+31	6,6,6,6,5,5,5	3,3,3,3,2.5,2.5,2.5
79	1525.87890625	1.48e+09	4.30e+32	5,5,5,5,5,5,5,5	2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5
81	1831.0546875	3.54e+10	3.31e+33	6,5,5,5,5,5,5,5	3,2.5,2.5,2.5,2.5,2.5,2.5,2.5
83	2197.265625	3.72e+11	2.56e+34	6,6,5,5,5,5,5,5	3,3,2.5,2.5,2.5,2.5,2.5,2.5
85	2636.71875	2.23e+12	1.97e+35	6,6,6,5,5,5,5,5	3,3,3,2.5,2.5,2.5,2.5,2.5
87	3164.0625	8.37e+12	1.52e+36	6,6,6,6,5,5,5,5	3,3,3,3,2.5,2.5,2.5,2.5
89	3814.697265625	2.07e+10	1.18e+37	5,5,5,5,5,5,5,5,5	2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5
91	4577.63671875	5.58e+11	9.12e+37	6,5,5,5,5,5,5,5,5	3,2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5
93	5493.1640625	6.69e+12	7.06e+38	6,6,5,5,5,5,5,5,5	3,3,2.5,2.5,2.5,2.5,2.5,2.5,2.5
95	6591.796875	4.69e+13	5.47e+39	6,6,6,5,5,5,5,5,5	3,3,3,2.5,2.5,2.5,2.5,2.5,2.5
97	7910.15625	2.11e+14	4.24e+40	6,6,6,6,5,5,5,5,5	3,3,3,3,2.5,2.5,2.5,2.5,2.5
99	9536.7431640625	2.89e+11	3.29e+41	5,5,5,5,5,5,5,5,5,5	2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5,2.5

Table 4.2: Details of the optimal trees for MAX-nodes-N-{*,+}{0.5}

Details of the optimal trees for MAX-nodes-N-{*,+}{1}, where $15 \leq N \leq 63$

N	MAX Value	Optimal trees	Possible trees	Constants per subtree	Subtree values
15	18	1.20e+01	6.50e+04	3,3,2	3,3,2
17	27	8.00e+00	4.31e+05	3,3,3	3,3,3
19	36	6.00e+01	2.92e+06	4,3,3	4,3,3
21	54	3.20e+01	2.01e+07	3,3,3,2	3,3,3,2
23	81	1.60e+01	1.41e+08	3,3,3,3	3,3,3,3
25	108	1.60e+02	9.93e+08	4,3,3,3	4,3,3,3
27	162	8.00e+01	7.08e+09	3,3,3,3,2	3,3,3,3,2
29	243	3.20e+01	5.09e+10	3,3,3,3,3	3,3,3,3,3
31	324	4.00e+02	3.69e+11	4,3,3,3,3	4,3,3,3,3
33	486	1.92e+02	2.69e+12	3,3,3,3,3,2	3,3,3,3,3,2
35	729	6.40e+01	1.97e+13	3,3,3,3,3,3	3,3,3,3,3,3
37	972	9.60e+02	1.45e+14	4,3,3,3,3,3	4,3,3,3,3,3
39	1458	4.48e+02	1.07e+15	3,3,3,3,3,3,2	3,3,3,3,3,3,2
41	2187	1.28e+02	7.95e+15	3,3,3,3,3,3,3	3,3,3,3,3,3,3
43	2916	2.24e+03	5.93e+16	4,3,3,3,3,3,3	4,3,3,3,3,3,3
45	4374	1.02e+03	4.43e+17	3,3,3,3,3,3,3,2	3,3,3,3,3,3,3,2
47	6561	2.56e+02	3.32e+18	3,3,3,3,3,3,3,3	3,3,3,3,3,3,3,3
49	8748	5.12e+03	2.50e+19	4,3,3,3,3,3,3,3	4,3,3,3,3,3,3,3
51	13122	2.30e+03	1.88e+20	3,3,3,3,3,3,3,3,2	3,3,3,3,3,3,3,3,2
53	19683	5.12e+02	1.42e+21	3,3,3,3,3,3,3,3,3	3,3,3,3,3,3,3,3,3
55	26244	1.15e+04	1.08e+22	4,3,3,3,3,3,3,3,3	4,3,3,3,3,3,3,3,3
57	39366	5.12e+03	8.16e+22	3,3,3,3,3,3,3,3,3,2	3,3,3,3,3,3,3,3,3,2
59	59049	1.02e+03	6.20e+23	3,3,3,3,3,3,3,3,3,3	3,3,3,3,3,3,3,3,3,3
61	78732	2.56e+04	4.72e+24	4,3,3,3,3,3,3,3,3,3	4,3,3,3,3,3,3,3,3,3
63	118098	1.13e+04	3.60e+25	3,3,3,3,3,3,3,3,3,3,2	3,3,3,3,3,3,3,3,3,3,2

Table 4.3: Details of the optimal trees for MAX-nodes-N-{*,+}{1}

4.1.3 Crossover in GP

Crossover is part of the standard GP 'package'. It is the most obvious and 'natural' method for recombining two trees to produce child trees. It is considered by some to be the main power underlying the GP algorithm, and by others to be at best just another form of Mutation and at worst an actual hindrance to GP, [Angeline 97]. Whatever its effectiveness, Crossover appears in virtually every description of GP, and every implementation of GP. It is the operator with which most newcomers to GP begin.

The standard crossover operator in GP, as described in Section 1.5, given two parent trees, can in theory bring a subtree from anywhere in one parent tree and swap it with a subtree anywhere in the other parent tree to produce two new child trees containing a mixture of genetic information from both parent trees. There are no restrictions on the selection of subtrees. Any node in either tree can be chosen as a crossover point, though occasionally there are biases in favour of non-terminal nodes. There is no direct equivalent of a static location as with genes in a chromosome in a GA. Thus, if a particular node, e.g. a '+'-node, appears anywhere within any tree in the GP population, then it is possible for GP Crossover (assuming such a tree is ever selected to be a parent) to spread that particular node to anywhere in a tree in the next generation. Only when that particular node disappears from all trees in the population, will GP Crossover be unable to spread it into the next generation.

In a typical GP setup, operators such as Crossover work with some idea of a maximum allowed depth or maximum number of nodes allowed for each tree in the population, as described above in Section 4.1.1. A consequence of this restriction on tree size is that the crossover operator may no longer be able to swap all possible pairs of subtrees between two parents and still produce 'legal' trees. This effect is minimal when the population consists only of trees much smaller than the size limit, but becomes more marked as GP trees usually tend to increase in size in later generations.

There are many simple ways of dealing with Crossover's propensity for producing illegal trees, as mentioned above in Section 4.1.1. The simplest is to select just one of the two child trees, at least one of which is guaranteed to be legal, and is the method used here. Other methods include reselecting the crossover points in some way until both children are legal, pruning over-large trees, or default to making the children identical copies of their parents. Thus, in practice, Crossover in GP is not free to move just any subtree to any part of another tree. The tree size restrictions limit the effectiveness of Crossover and, as discussed below, can hinder GP's discovery of better trees via Crossover.

In normal use, Crossover is used along with other operators, usually some forms of Mutation, creating one offspring from one parent (ascertained after many discussions and examining the literature). One oft used type of Mutation operator randomly generates a new subtree in a parent tree to produce a child. This method is useful since it can introduce small effective subtrees to a population but is not likely to introduce large effective subtrees. Another form of mutation operator replaces an individual node in a parent tree with another node of the same arity to produce a child. This method is useful for reintroducing nodes back into a population. But, unless the population can make immediate use of them, Selection will probably wipe them out in the next generation. In the case of MAX-depth-D-{+,*}{0.5}, (see Figure 4.4) this would speedily bring about the discovery of the optimal tree since replacing a high-level '+'-node with a '*'-node creates a fitter tree. Looking at MAX-depth-D-{/,*}{0.9}, (see Figure 4.6), the left-hand side of the trees often requires the introduction of large effective subtrees which are unlikely to be created by mutating a subtree and would not be created by mutating an individual node. Thus incorporating either or both of the mutation operators as described above would not help GP find the optimal tree.

4.1.4 Experiment Details

The MAX-depth-D variations discussed in this section are {+}{1}, {+,*}{1}, {+,*}{0.5}, {+,*}{0.25}, and {*,/}{0.9}. The depths are between 3 and 8, where the upper depth limit depends on the size of the optimal tree value overflowing the floating-point representation, or complete failure by GP at smaller depths. Fifty runs were made for each variation, with the only operator being Crossover. These runs were then divided according to whether or not they were successful, as detailed below in Section 4.1.5. For comparison, MAX-depth-D-{+,*}{0.5} and {*,/}{0.9} were repeated with the addition of the mutation operators mentioned above in Section 4.1.3. Their selection frequencies were: 60% Crossover; 20% Mutate Node; 20% Mutate Subtree.

An effort was made to keep the GP used here simple since the aim was to demonstrate some qualitative aspects of the crossover operator, and not to strive for optimal GP performance. The implementation used generational replacement with elitism on a population size of 200. A tournament of size 6 was used with the MAX-depth-D runs. With hindsight this tournament size is probably too large with a population size of only 200 for GP to perform as well as it could, but many of the runs were successful despite this. A tree's fitness is the maximum possible return value minus the tree's return value. Thus a fitness score of zero is optimal, with worse fitnesses being large and positive. Standard crossover was used, with two parents producing one child, where all trees were limited to a maximum depth D. A crossover point was chosen by randomly selecting one node from all of the nodes in a tree. Crossover could have been made more likely to produce a child different from its parents by requiring that at least one of the crossover points in the two parents was a function (i.e. non-leaf) node, since all terminal nodes were identical. Runs were stopped after 500 generations as failures, or earlier as successes if the optimal tree was produced.

For the MAX-nodes-N variations, {*,+}{1}, {*,+}{0.5}, {*,+}{0.25}, the population size was kept at 200, but the tournament size was reduced to 3, after early results indicated that GP was often failing to find the optimal trees. Reducing the selection pressure in this way did improve GP performance by a small amount. Just the results involving tournament size 3 are described in detail below. Since the MAX-nodes-N problems are apparently substantially harder for GP than the MAX-depth-D problems, GP was set a later cutoff at 3000 generations. Most of the successful runs finished well before the cutoff generation. Those that discovered an optimal tree close to the cutoff had shown no signs of improvement for many hundreds of generations before that, making it fairly safe to conclude that the discovery of the optimal tree was a chance event due to an extremely unlikely but successful mutation of all or most of a tree. As with the MAX-depth-D variations, the size restriction was raised as high as possible so that the optimal tree values did not exceed the machine-specific limits of the floating-point representation used. For sufficiently small N, the MAX-nodes-N problem is trivial, with GP usually discovering an optimal tree in generation 0, i.e. when trees are initially generated at random. Also, as with the MAX-depth-D variations, the runs were repeated with the addition of the Mutation operators, leading to over 20,000 runs being carried out.

4.1.5 Results

Performance of GP on MAX-depth-D

The average number of generations for the successful runs for each of the variations of the MAX-depth-D problem, in Figure 4.8, shows the MAX problems increasing in difficulty for GP with increasing depth, unsurprisingly.

The percentage of runs which ended in failure, shown in Figure 4.10, increases rapidly with depth for all but MAX-depth-D-{*,+}{1} and {+}{1}. The populations in these failed runs reached a state where it was impossible or extremely unlikely to produce the optimal tree. The populations in successful runs were, in effect, 'lucky' to discover the optimal trees before the disappearance of crucial subtrees from the populations.

Adding the mutation operators, shown in Figure 4.11 has helped MAX-depth-D-{+,*}{0.5} to achieve 100% success at all the depths tested (and similarly for MAX-depth-D-{+,*}{0.25}, though it is not shown on the graph). The MAX-depth-D-{/,*}{0.9} runs, on the other hand, still have difficulty in finding the optimal trees. Looking at the runs which failed, the entire population had usually converged to be a duplicate (or very close copy) of the 'best of run' tree.

Typical sub-optimal trees found for MAX-depth-D

The following trees exemplify the sub-optimal trees discovered by GP in the variations of the MAX problem (where depth 0 refers to the root node):

For **MAX-depth-5-{*,+}{0.5}**, the 'best of run' tree shown in Figure 4.12 is not very different (in terms of node changes) from the optimal tree. Each node at depth 1 is a '+' instead of a '*'. However all the fit trees in the population also had no '*'-nodes in this layer, and were all the same size and shape (i.e. full to depth 5). The only '*'-nodes in the population were at depth 0. Given this situation, Crossover is no

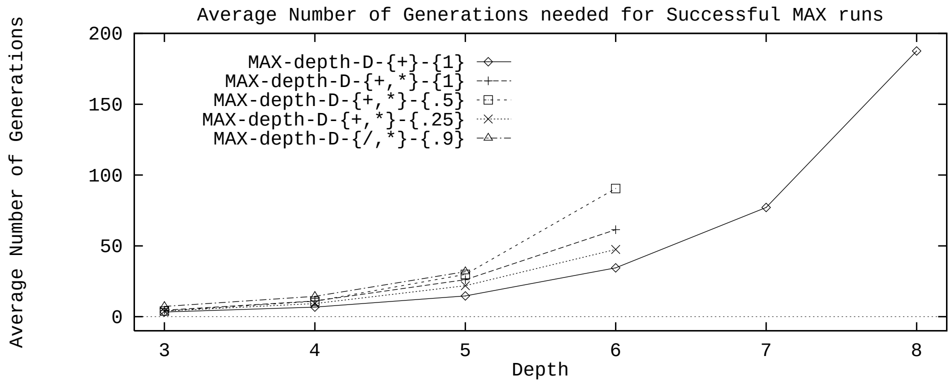


Figure 4.8: Average number of generations needed by the successful MAX-depth-D runs using Crossover, showing difficulty increasing with depth

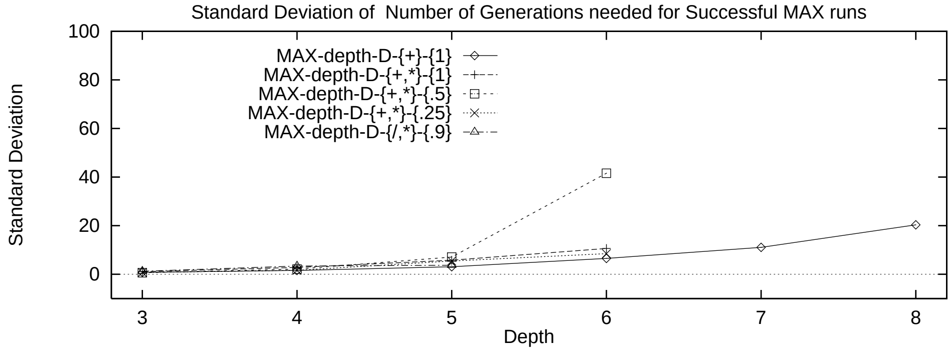


Figure 4.9: Standard Deviation of number of generations needed by the successful MAX-depth-D runs using Crossover, showing difficulty increasing with depth

longer able to improve on 'best of run' tree. An operator which mutated individual nodes could easily construct a fitter tree from this one.

For **MAX-depth-5-{*,/}{0.9}**, the 'best of run' tree shown in Figure 4.13 needs two more '/'-nodes down the left-most side to become the optimal tree. Given that all the fit trees in the population came to be duplicates of this tree, Crossover could not improve upon it. Any subtree not containing a '/'-node would have a smaller value than the '.9'-node at depth 3, and thus would result in the whole tree having a smaller fitness. The only '/'-nodes are to be found at depths 0, 1, and 2, and Crossover cannot bring those subtrees down to start at depth 3 since that would create an illegal tree. An operator which mutated subtrees might be able to construct a fitter tree from this one.

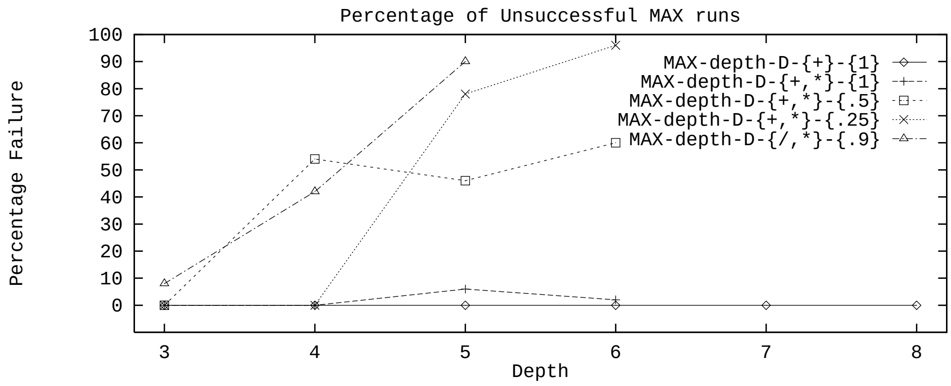


Figure 4.10: Percentage of MAX-depth-D runs failing, using Crossover

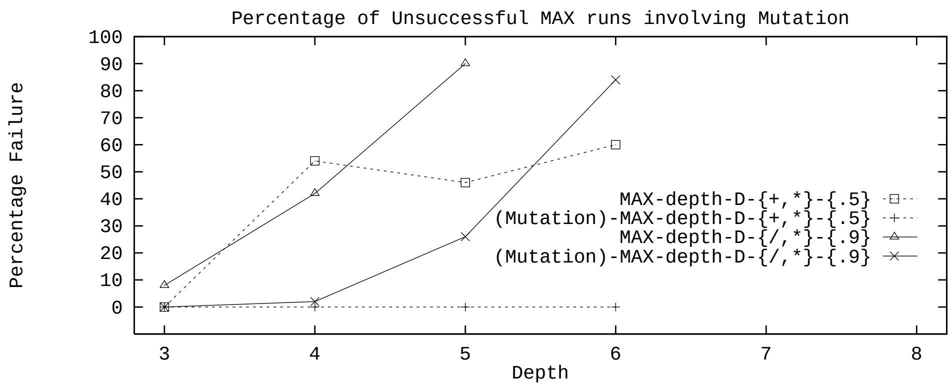


Figure 4.11: Percentage of MAX-depth-D runs failing (dashed lines), using Crossover only (□) and Mutations (+), and Percentage of MAX-depth-D runs failing (solid lines), using Crossover only (△) and Mutations (x)

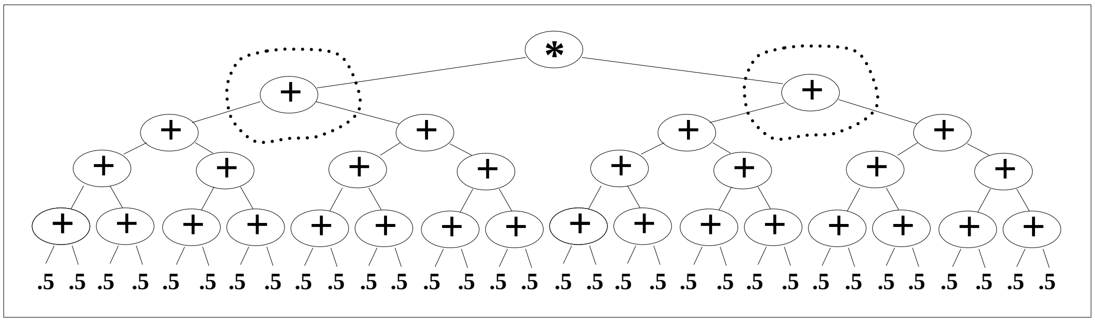


Figure 4.12: Sub-Optimal tree for MAX-depth-5-{*,+}{0.5}

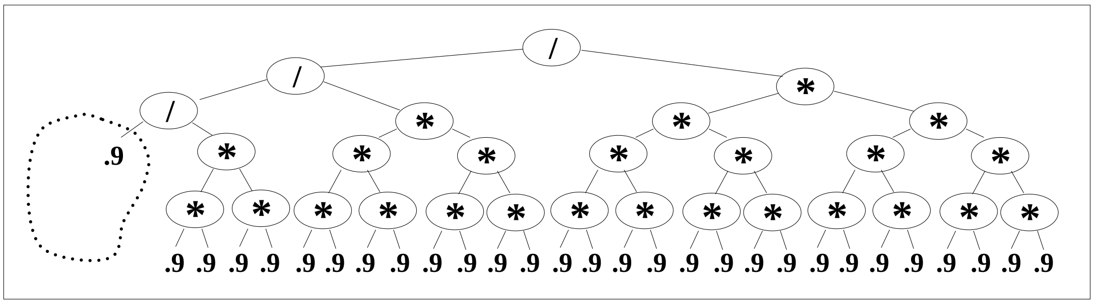


Figure 4.13: Sub-Optimal tree for MAX-depth-5-{*,/}{0.9}

Performance of GP on MAX-nodes-N

As with the results for MAX-depth-D above, the results for the variations of the MAX-nodes-N are in two main sets of figures. The average number of generations needed by the successful runs are shown in Figures 4.15 to 4.20. Although there were 50 runs for each N and sets of operators with and without Mutation, the averages shown for MAX-nodes-N-{*,+}{0.25} and MAX-nodes-N-{*,+}{0.5} become very erratic and effectively meaningless for larger values of N, since so few runs were successful. If there were no successful runs for a particular N, then that point doesn't appear in the graphs. Many of the points for large N which do appear, correspond to very few successful runs, often only one successful run. The averages for MAX-nodes-N-{*,+}{1} show both that few generations were needed, and that the addition of the Mutation operators actually hinders GP for this particular variation, sometimes doubling the number of generations needed. The averages for MAX-nodes-N-{*,+}{0.5} without Mutation show a rapid increase in the number of generations needed up to N=100, whereas for MAX-nodes-N-{*,+}{0.5} with Mutation the generations needed by suc-

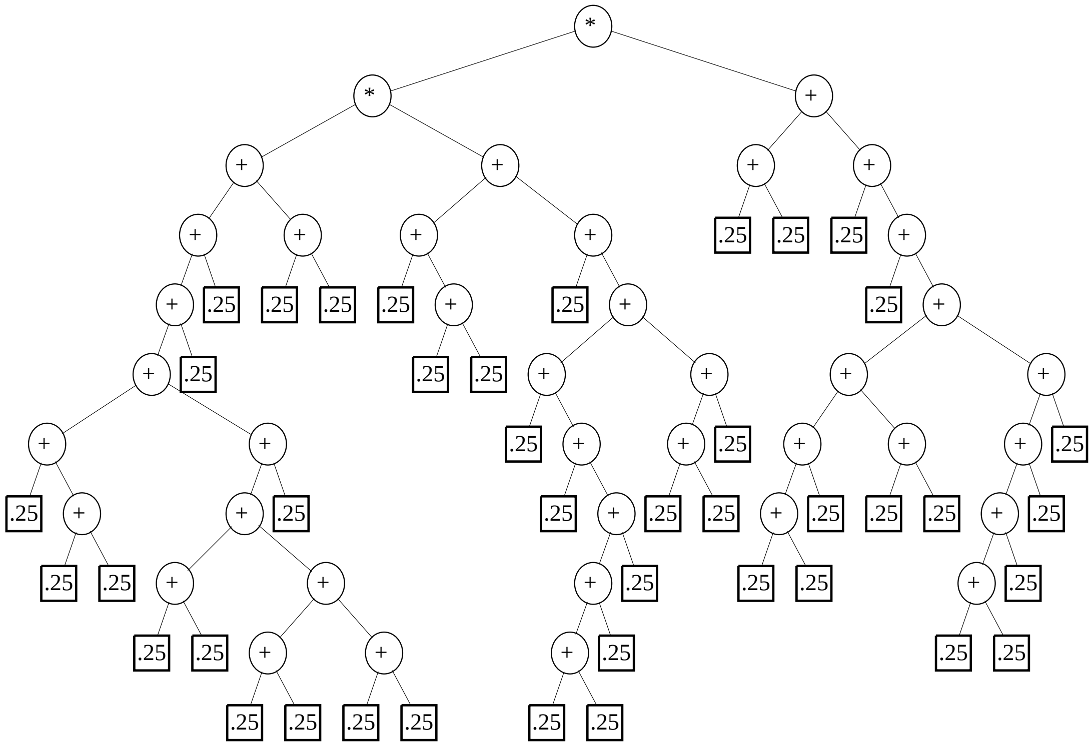


Figure 4.14: A Sub-Optimal tree for MAX-nodes-81-{*,+}{0.25}, with error 3.156250

cessful runs rises much more slowly. The averages for the MAX-nodes-N-{*,+}{0.25} runs both with and without Mutation also rise slowly.

Figures 4.21 to 4.26 show the percentage of runs which ended in failure. These graphs are statistically more reliable than the ones mentioned above for average generations to success. Each point is a value averaged over 50 runs. The two graphs for MAX-nodes-N-{*,+}{1} in Figures 4.23 and 4.26 show that this version of the MAX problem is very easy for GP both with and without the Mutation operators. The graphs for the average generations to success, in Figures 4.17 and 4.20, show that GP needs few generations to successfully find an optimal tree. The optimal trees for this problem consist of many small {+,n} subtrees joined by many '*' nodes. GP quickly discovers these small subtrees, and the subproblem of putting them together optimally is quite simple.

The sets of graphs for MAX-nodes-N-{*,+}{0.25} and MAX-nodes-N-{*,+}{0.5}, on the other hand, show some distinctive features. Both the graphs showing failures and the graphs showing average generations to success show a periodic variation in difficulty as N changes. Looking at both the graph in Figure 4.18, showing the percentage failures of the MAX-nodes-N-{*,+}{0.25} runs, and Table 4.1, indicating the structures of the optimal trees, it is apparent where the periodic variation comes from. As N increases, each sudden jump in the percentage failure corresponds to an increment in the number of {+,n} subtrees in the optimal solution. Reading from the table, for N=53, the first optimal subtree structure with three {+,n} subtrees is (9,9,9), and the next time the number of {+,n} subtrees increases (to four) is for N=77. These two values of N correspond to the first two spikes in the graph. Similarly, each spike after that corresponds to the next increase in the number of {+,n} subtrees in the optimal solution. The relative difficulty of the MAX problem is related to the distribution of sizes of {+,n} subtrees in the optimal tree and the sizes of {+,n} subtrees which GP is likely to construct.

Regarding the impact of the Mutation operators on GP's performance, for small N (up to around 50), the addition of the Mutation operators corresponds to a much lower failure rate for both MAX-nodes-N-{*,+}{0.25} and MAX-nodes-N-{*,+}{0.5}. For larger N, GP can be seen to have benefited greatly from the addition of Mutation operators in MAX-nodes-N-{*,+}{0.25}. The periodic variation in difficulty is more obvious, but there are many more successes as N increases all the way to 249. Somewhat confusingly, the same improvement for large N is not apparent in MAX-nodes-N-{*,+}{0.5}. Instead, it appears that the addition of Mutation operators has actually hindered GP for larger N, with GP performing worse than without Mutation.

Typical sub-optimal trees found for MAX-nodes-N

The sub-optimal trees discovered for MAX-nodes-N-{*,+}{0.25} and MAX-nodes-N-{*,+}{0.5} follow a common theme. A typical sub-optimal tree is shown in Figure 4.14 for MAX-nodes-81-{*,+}{0.25}. Whereas the optimal tree should be of the form (11,10,10,10), shown in Figure 4.7 and discussed in 4.1.2, i.e. with four subtrees containing 11, 10, 10, and 10 '0.25' nodes, the sub-optimal tree shown in 4.14 is of the form (14,14,13). It contains fewer but larger subtrees, producing a slightly smaller return value than the optimal tree. In fact, all of the sub-optimal trees discovered by GP were of this form, containing one fewer subtree

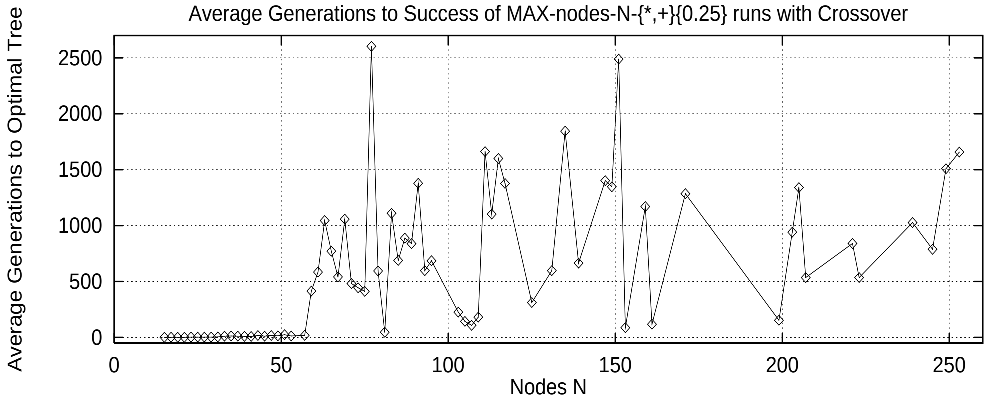


Figure 4.15: Average number of generations needed by the successful MAX-nodes-N-{*,+}{0.25} with Crossover

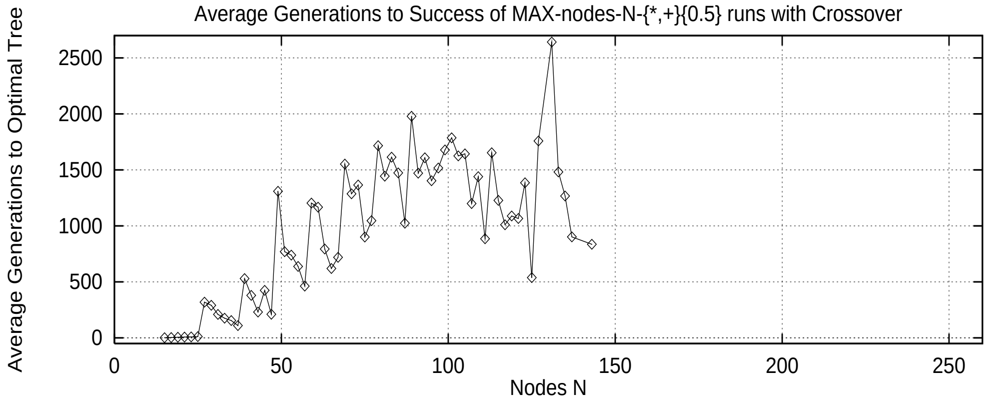


Figure 4.16: Average number of generations needed by the successful MAX-nodes-N-{*,+}{0.5} with Crossover

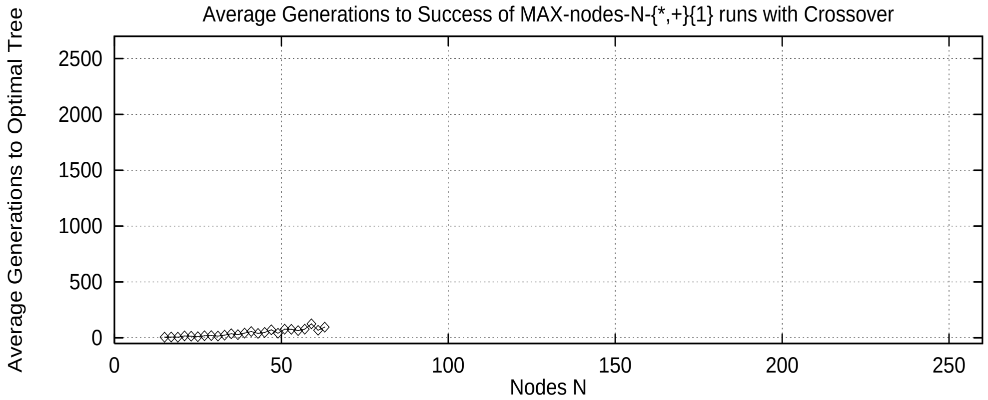


Figure 4.17: Average number of generations needed by the successful MAX-nodes-N-{*,+}{1} runs with Crossover

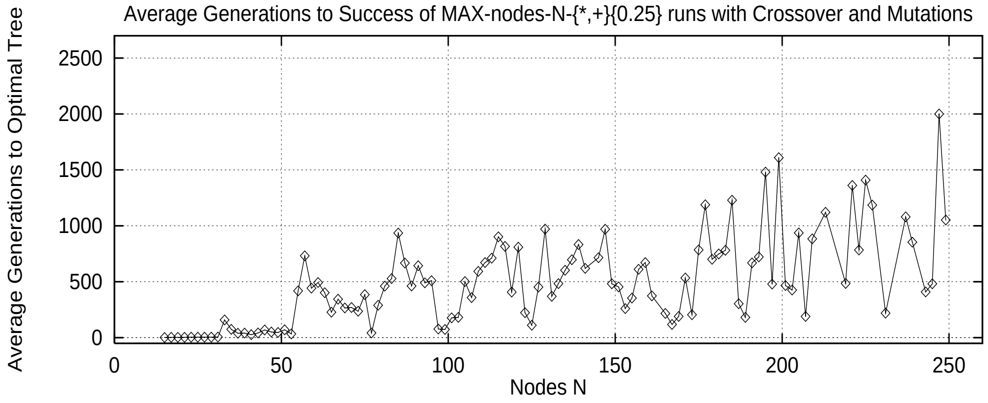


Figure 4.18: Average number of generations needed by the successful MAX-nodes-N-{*,+}{0.25} runs with Crossover & Mutations

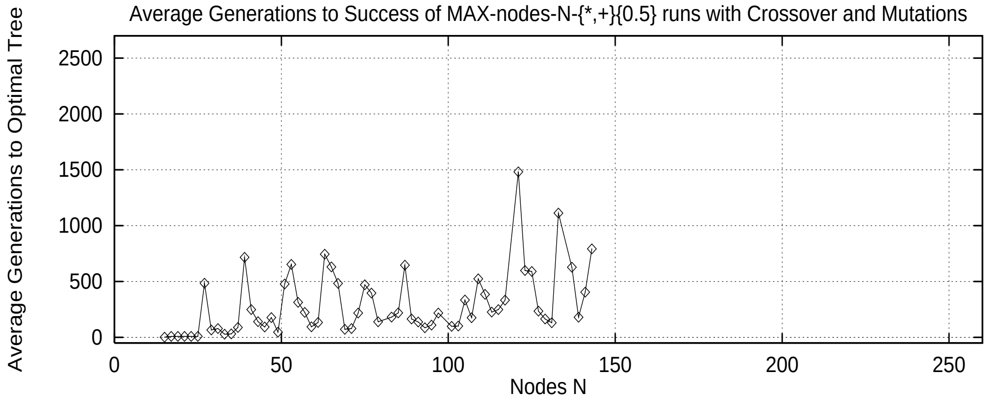


Figure 4.19: Average number of generations needed by the successful MAX-nodes-N-{*,+}{0.5} runs with Crossover & Mutations

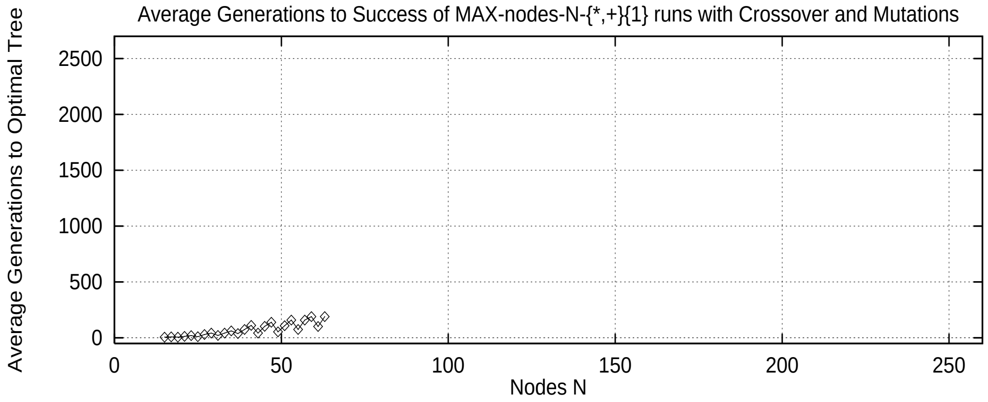


Figure 4.20: Average number of generations needed by the successful MAX-nodes-N-{*,+}{1} runs with Crossover & Mutations

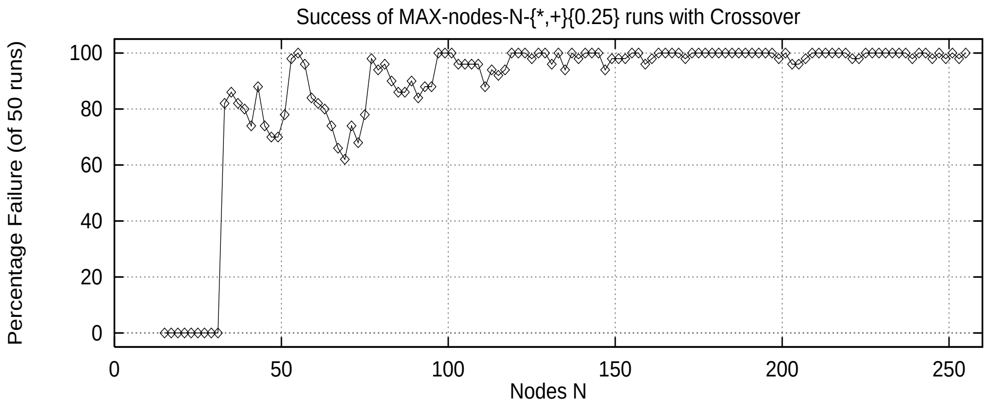


Figure 4.21: Success of MAX-nodes-N-{*,+}{0.25} runs with Crossover

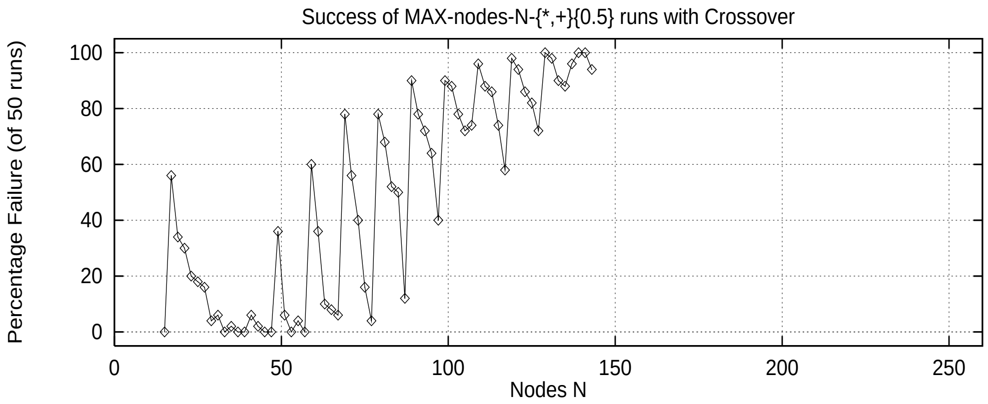


Figure 4.22: Success of MAX-nodes-N-{*,+}{0.5} runs with Crossover

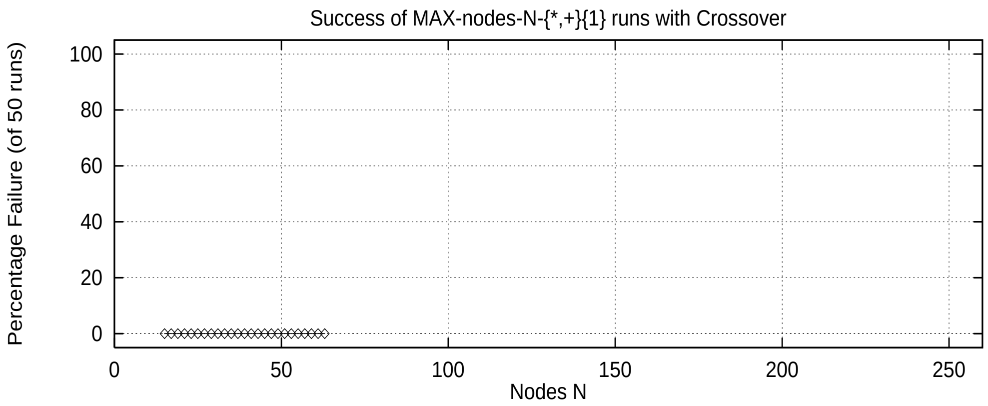


Figure 4.23: Success of MAX-nodes-N-{*,+}{1} runs with Crossover

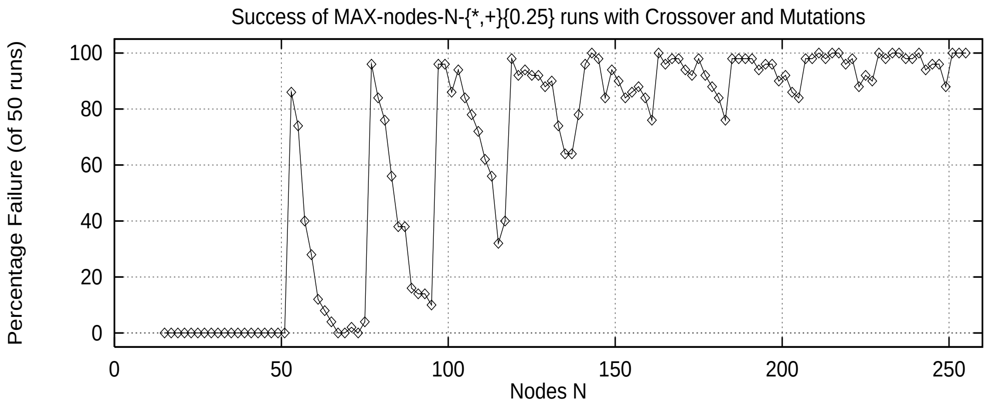


Figure 4.24: Success of MAX-nodes-N-{*,+}{0.25} runs with Crossover & Mutations

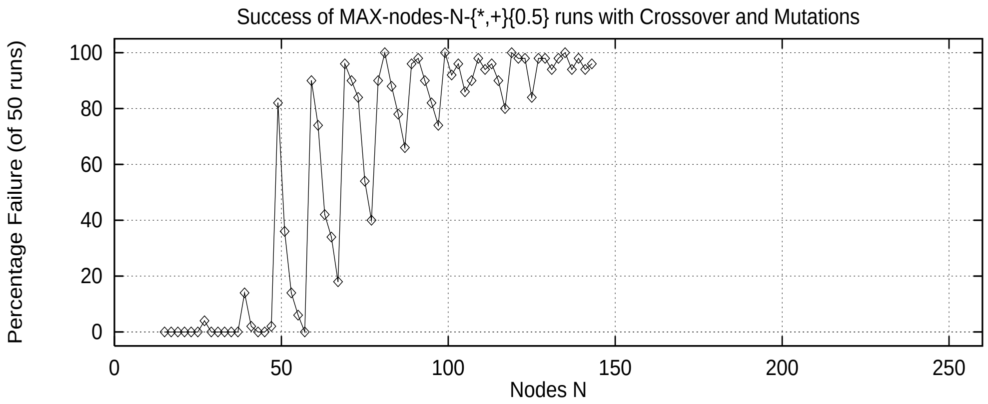


Figure 4.25: Success of MAX-nodes-N-{*,+}{0.5} runs with Crossover & Mutations

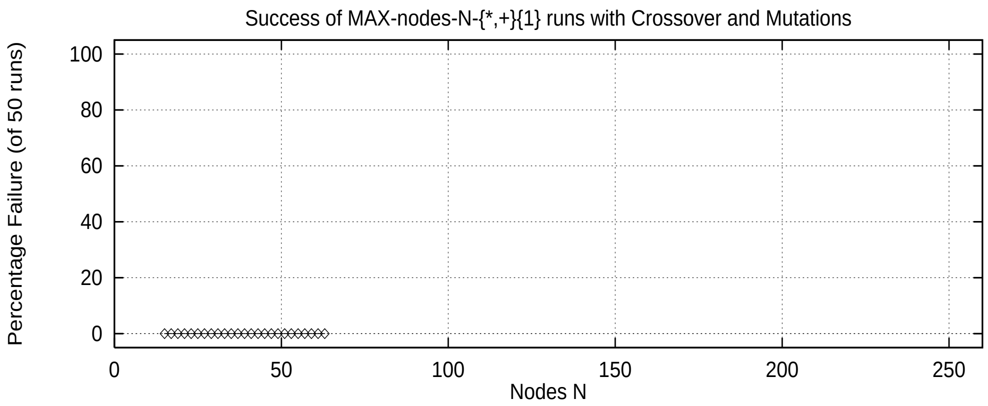


Figure 4.26: Success of MAX-nodes-N-{*,+}{1} runs with Crossover & Mutations

than the optimal tree. It is not possible for Crossover to produce a fitter child tree from two trees of this form. It is not possible for mutating a single node to produce a fitter child. It is possible but extremely unlikely that mutating an entire new tree could produce an optimal tree, but that applies in all situations where such a mutation is being used. To construct a fitter tree, using the sub-optimal tree as a basis, each of the large {+,n} subtrees would have to be modified (i.e. pruned) and a new subtree added (using the pruned nodes), joined to the main tree via a '*' node.

4.1.6 Analysis of Crossover

This section presents a simple analysis of the crossover operator in GP, showing its likely impact on GP trees where there is a restriction on tree depth, and the population consists of 'full' trees, i.e. trees which have filled out to the maximum allowed depth. This situation is frequently reached whilst tackling the MAX-DEPTH problems. The following calculations apply to GPs with function sets involving functions of arity of two, e.g. like the MAX problems. If the largest function arity is greater than 2, the problem is exacerbated.

D - the maximum tree depth

$\boldsymbol{l}$ - tree layer, from 0 (the root node) to D

$\mathbf{L}_l$ - the number of nodes in layer l

$= 2^l, \quad where \quad 0 \leq l \leq D$

$\mathbf{N}_l$ - the number of nodes in all layers from 0 to layer l inclusive

$= \sum_{j=0}^{l} L_j = \sum_{j=0}^{l} 2^j = 2^{l+1} - 1$

$\mathbf{N}_D$ - the total number of nodes in a tree of depth D

$= 2^{D+1} - 1$

C(any)$_l$ - the likelihood of any node(s) in a tree, in layer l, experiencing some crossover activity, i.e. when a crossover point occurs in any layer from 0 to l

$= \frac{N_l}{N_D} = \frac{2^{l+1}-1}{2^{D+1}-1} \simeq \frac{2^{l+1}}{2^{D+1}} = \frac{1}{2^{D-l}}$

C(layer)$_l$ - the likelihood of crossover occurring within a layer l, i.e. when the two crossover points are in the same layer

$= \frac{L_l^2}{N_D^2} = \frac{2^{2l}}{(2^{D+1}-1)^2} \simeq \frac{2^{2l}}{2^{2D+2}} = \frac{1}{2^{2(D-l+1)}}$

C(2 legal offspring) - the likelihood of crossover, based on a random choice of crossover points, producing two legal offspring

$= \frac{legal}{total} = \frac{\sum_{j=0}^{D} L_j^2}{N_D^2} = \frac{\sum_{j=0}^{D} (2^j)^2}{(2^{D+1}-1)^2} \simeq \frac{\sum_{j=0}^{D} 2^{2j}}{2^{2(D+1)}}$

$\simeq \frac{2^{2D}}{2^{2(D+1)}} = \frac{1}{4}, \quad for\ large\ D$

Looking at **C(any)**$_l$ shows that the upper layers receive relatively little attention from the crossover operator, especially as the trees grow large. In the MAX problem as used here, the tree sizes in question are fairly small (e.g. **D**=6), even so the upper tree layers are mostly unaffected by Crossover.

Looking at **C(layer)**$_l$ shows how an exchange of subtrees within the upper layers is much less likely again. This implies that there is little or no spread of subtrees within the upper layers in later generations.

Looking at **C(2 legal offspring)** shows that Crossover will frequently produce illegal offspring once the trees in the population fill out. For each pair of offspring produced by Crossover in this way, there will always be at least one legal offspring, but there is a high probability that the other will be illegal. Either this illegal offspring has to be modified in some way or simply take just one offspring from each crossover operation.

Given two full trees, selecting just one offspring produced by Crossover will mean either that subtrees were swapped between the same layer in each tree, or that a subtree from a lower level has been raised to a higher level. It is impossible for a subtree from a higher level to be brought down to a lower level and still produce a legal tree, since it would exceed the depth limit. Thus Crossover, producing a single offspring from each pair of parent trees, operating on a population of full trees, will produce offspring in the next generation in the following manner:

- mostly through the exchange of low level subtrees
- some through the raising of a low level subtree to a higher level
- very few through the exchange of high level subtrees
- none through the lowering of a high level subtree to a lower level

If subtree discovery takes place in the lower levels, Crossover would be very effective at spreading these new subtrees through the population. If, on the other hand, improvements in subtrees solely or largely take place in the higher levels, Crossover will be very slow to spread these new subtrees through the population.

4.1.7 Discussion of MAX problem

The analysis in Section 4.1.6 highlights one of the main biases that Crossover brings to GP, even without considering populations of full trees. Subtree discovery and the spread of subtrees takes place at lower levels, mostly involving the leaf nodes. The effectiveness of this is highly dependent on the problem in hand. Immediately beneficial subtrees are quickly spread within the trees and through the population, at the expense of other subtrees of less immediate benefit. For MAX-depth-D-$\{+\}\{1\}$, this results in a speedy discovery of the optimal tree.

The situation mentioned above for Crossover in general is made worse when a tree depth restriction is incorporated. Whilst the GP population consists of trees which are shallower than the depth restriction, it is still possible, though unlikely for Crossover to move subtrees around freely. When the trees have expanded to reach the depth restriction, the situation changes. The only movement of subtrees possible via Crossover is from lower levels to higher levels or within the same level. In the case of the other MAX problems, such as MAX-depth-D-{+,*}{0.5}, this can result in a purging of '*'-nodes from the population, perhaps only leaving a few of these function nodes in the higher levels of some of the trees, where they will remain relatively untouched. The benefit of these nodes only emerges after the trees reach an appreciable size, by which time it is unlikely or impossible that they can be spread by Crossover. Once the trees have reached the depth limit, the only way the higher levels are affected is through the promotion of lower level subtrees, which contain no '*'-nodes, and the movement of subtrees within the same level. If there are no '*'-nodes in any tree at a particular high level, it is now impossible for Crossover to introduce a '*'-node to this level; the population has effectively converged to being duplicates of a sub-optimal tree, and no further improvement is possible. Langdon and Poli's study of the MAX-DEPTH problem "... show[s] that this can happen even when the population retains a high level of variety and show that in many cases evolution from the sub-optimal solutions to the solution is possible if enough time is allowed", [Langdon & Poli 97].

Results from the MAX-nodes-N runs show some characteristics similar to the MAX-depth-D results. Early loss of '*' nodes from the lower levels hinders the search for the optimal tree later on. Until the trees reach an appreciable size, the fitter subtrees are those which contain more '+'-nodes. Only after gaining large '+'-subtrees does it become worthwhile to have high level '*'-nodes. After gaining high level '*'-nodes, it then becomes worthwhile to have more but smaller '+'-subtrees which would involve altering all of the large '+'-subtrees simultaneously. Making just one '+'-subtree a little smaller would simply lower the fitness of the tree. Thus the trees are trapped in a sub-optimal form.

A scan of published papers has indicated that this restriction on the number of nodes is used more often than a maximum tree depth, since it lends itself well to various memory and CPU-efficient GP implementations. The trees in a GP population still expand with the generations, quickly reaching their size limit. At this stage, Crossover affects mainly the peripheral (i.e. leaf) nodes, and becomes unable to modify upper levels of the trees effectively, although perhaps not as often as with the depth restriction.

Further runs, tracking the number, distribution, and location of '*' nodes throughout the population should make the pattern of loss of '*' nodes more explicit.

Modifications to Crossover Several other investigations have focussed in some way on Crossover in GP. [Rosca 95] looks at causality in GP, relating the changes in the structure of GP trees caused by Crossover with changes in the properties of the trees. [D'haeseleer 94] looks at context preserving Crossover in GP, attempting to ensure that swapped subtrees will still be effective. This moves away from the more fluid approach proposed by Koza. Instead of allowing any and all combinations of functions and terminals, D'haeseleer removes the closure constraint, allowing several different data-types, only allowing nodes of the same data-type to be brought together. With less flexibility, GP has fewer opportunities to construct inappropriate trees. [O'Reilly & Oppacher 95a] looks at hybrids of operators such as Crossover and Hill Climbing, maintaining that these mutation-based operators can match or outperform basic Crossover. [Lang 95] shows how mutations and simple hill climbing can perform better than GP with Crossover, calling into question the effectiveness of Crossover and the population-based approach of GP.

There is no doubt that it should be possible to modify Crossover or its use so that it is more likely to result in the discovery of optimal (or at least better) solutions in the MAX problem, even if this is at the expense of speed on those problems where it does well already. In practice, where the focus of a paper is not on the operators themselves, 'standard' Crossover still seems to be used as one of the main GP operators, usually in combination with other operators such as Mutation. More attention should be paid to ensure that the other operators are capable of overcoming Crossover's shortcomings. Simple mutation operators have been shown here to be insufficient. Using very large population sizes to boost Mutation's chance of constructing large useful subtrees only obscures the problem.

The method of selecting crossover points could be modified to ensure that the tree layers get a more even spread of crossover activity. [Koza 92] uses Crossover where internal nodes have a 90% chance of being selected as a crossover point compared with 10% for leaf nodes. However, the higher tree levels will still experience much lower crossover activity than the lower levels. It might be better to scale the selection probability according to depth or the number of nodes in each level, perhaps targeting sections of tree which have a low node variance across the population, but this requires extra processing, and the identification of important tree sections would likely be somewhat problematic.

For the purposes of enabling GP to solve the MAX-depth-D problem more easily, there is one operator action in particular which will obviously be highly effective. It could be considered as a form of Mutation, or incorporated into a form of Crossover. Either way, the operator would select a point near the root of a parent tree, copy the subtree below this point, and insert it at a point further away from the root, replacing the subtree there. Excess nodes would be pruned off, so that the resulting child tree satisfied the size constraints. The consequences of this operator would be to counter the main weakness of standard Crossover in the MAX problem, i.e. its inability to bring subtrees down to the lower levels, and would enable the downward spreading of the '*' nodes. This would certainly result in a very much higher success rate (if not 100% success) in the MAX-depth-D problems, and would probably help in MAX-node-N. This operator, whilst undoubtedly effective with the MAX-depth-D problems, is also likely to be useful for other GP problems where nodes near the root are crucial at lower levels, as well counteracting Crossover's main weakness mentioned above.

However, with MAX-node-N, there is another difficulty to overcome, where the sub-optimal trees have several large {+,n} subtrees, but the optimal trees have one or more {+,n} subtrees all of which are smaller. To move from such a sub-optimal tree to an optimal tree, all of the large subtrees have to be modified and a new subtree added, all in a single operator step, otherwise the resulting child tree, though 'nearer' to being an optimal tree, will have a lower fitness, and is thus unlikely to be chosen as a parent. An operator which explicitly corrected for this occurrence is unlikely to be useful for other GP problems, unless they are known to have similarly deceptive sub-optimal trees.

The simplest approach for avoiding the effects of Crossover is not to use it, instead relying on an assortment of Mutation operators. It would be a worthwhile extension of the MAX runs so far to see what happens without Crossover. A hypothesis is that the MAX-depth-D problems would be solved more slowly, but with a much higher success rate, and that the success rate on the MAX-nodes-N would also increase, though there might still be the difficulty with discovering the sub-optimal trees with fewer and larger subtrees than the optimal trees.

For problems where tree size is part of the solution, allowing unlimited tree sizes is obviously not applicable. For others, where the optimal tree sizes are unknown, parsimony (penalising large trees) appears to work well, especially when it is used only to discriminate between trees which would otherwise have the same fitness (this impression has been gleaned from many experiments and communications with other GP practitioners). Thus trees can grow as large as is needed to do better on a problem, and Crossover can operate in an unrestricted fashion, but there is continual selection pressure for smaller trees which do just as well.

4.1.8 Summary

The MAX problems for GP show how the crossover operator and selection can be directly responsible for loss of diversity of nodes within the upper tree levels within a population, leading to premature convergence to a sub-optimal solution or very very slow improvement in solutions. When used in combination with a restriction on tree depth, the premature convergence becomes irreversible. Subtree discovery and movement takes place mostly near the leaf nodes, with nodes near the root left mostly untouched. Diversity drops quickly to zero near the root node in the tree population, resulting in GP being unable to create 'fitter' trees via the crossover operator. The addition of simple mutation operators is not sufficient to overcome these problems. Care should be taken to ensure that the spread of subtrees throughout the GP population is not stifled by Crossover.

When used in combination with a restriction on the number of nodes in a tree, the population converges on trees with a sub-optimal structure. It is this structure rather than the loss of '*' nodes which renders GP from getting any closer to discovering an optimal tree. The sub-optimal structure involves fewer and larger {+,n} subtrees, which are rapidly spread through the population, at the expense of the smaller {+,n} subtrees which are needed for constructing an optimal tree. For MAX-nodes-N-{*,+}{0.25 or 0.5}, the failure of GP is not necessarily due to Crossover and selection. This MAX problem is quite deceptive, with larger {+,n} subtrees returning larger values.

This should not be taken as an attack on the crossover operator and GP, or a claim that the MAX problem epitomises all GP problems. Instead it gives a better understanding of how Crossover works in practice, often in combination with tree size restrictions, and enables the user to be aware of its potential failings.

4.2 Tournament Selection

This section looks at Tournament Selection, the method used in this thesis for picking individuals from the GP population to be used as parents in creating the next generation of individuals. The aim is to find ways of speeding up GP for supervised learning tasks, perhaps by reducing the population size, memory requirements, or number of fitness evaluations. A brief survey of a variety of selection methods in Section 4.2.1 is followed by an explanation of why Tournament Selection was chosen here. Section 4.2.2 looks at some consequences of using Tournament Selection. A much more substantial but somewhat opaque study of a variety of selection methods, including Tournament Selection, can be found in [Blickle & Thiele 95]. Blickle and Thiele take an in-depth look at the behavioural characteristics of the various selection methods, proving numerous theorems along the way. An earlier study of several common selection schemes can be found in [Goldberg & Deb 91]. The study below is much simpler and more straightforward.

4.2.1 Various Selection Methods

Reasons for Selection

There are two stages in the GP algorithm where individuals are selected from the current population: selecting parents, and selecting individuals to be replaced.

Selecting for Replacement

There are two main methods of replacing individuals in GP:
Steady-State, and Generational.

In Steady-State Replacement, once a new child has been created and evaluated, a decision is made on whether to insert the child into the population, displacing an existing individual, or to discard the child. Several alternatives have been used in the literature:

- replace worst - discard the worst existing individual

- replace at random if child is better - choose an individual at random and discard it if it is worse than the child, otherwise discard the child
- replace parent if no worse - discard the parent if the child is no worse
- etc ... There are numerous other variations

In Generational Replacement (the method used in this thesis), on the other hand, the entire population is replaced by a completely new generation of individuals, avoiding all the replacement decisions mentioned above, but usually requires the addition of Elitism. Elitism is simply the explicit copying of the best individual in the current generation into the next generation, ensuring that the population does not lose its previous best individual. Steady-State Replacement is implicitly elitist since the best individual would never be discarded. For GP, [Koza 92] recommends the use of Over-Selection, a more extreme version of Elitism. This is considered necessary since GP tends to produce a large proportion of very unfit individuals. The top 50%, say, of individuals in the current population are explicitly copied into the next generation, and only they are used as parents to create the rest of the population, discarding the worst 50% before the breeding stage. This approach is similar to that used in some Evolution Strategies, [Bäck *et al.* 91].

Selecting Parents

There are numerous approaches documented in the literature for selecting parents from a population in GA-type algorithms. Typically there is a bias towards selecting fit individuals more frequently than unfit individuals. Three of the main types are as follows:

Roulette-Wheel Selection was one of the earliest methods, described in [Holland 75], where an individual's chance of being selected is related to its fitness (a form of Fitness-Proportionate Selection). This method has fallen out of favour due to the fact that it is easily swayed by 'super'-fit individuals in a population. If an individual has a substantially higher fitness than the rest of the population it will dominate the breeding process. Likewise, if the population has a high average fitness, there will be little difference between individuals over likelihood of selection, so there will be no effective bias in favour of fitter individuals. In order to avoid these problems, extra processing is needed to re-scale the fitness values.

Rank-based Selection involves sorting the entire population in terms of fitness. Now an individual's chance of being selected is proportional to its rank in the population, i.e. a higher ranked individual is more likely to be selected than a low ranked individual. Rank-based Selection does not suffer from the two inadequacies of Roulette-Wheel Selection mentioned above. Both Roulette-Wheel and Rank-based Selection involve processing or sorting the fitnesses of the entire population. Each selection can then involve scanning the entire population again.

Tournament Selection is much simpler and less computationally intensive. A fitness tournament is held to select a parent, where the best individual picked from a small set chosen at random from the population is taken to be the parent. Varying the tournament set size varies the selection pressure, i.e. the likelihood that the best individuals in a population will be selected as parents. No pre-processing of the population's fitnesses is needed. Tournament Selection is amenable to parallel implementations and spatially-biased selection, where parents are chosen within a certain neighbourhood of a specified location in a spatially distributed population.

For the Crossover operator, two or more parents are needed to produce offspring. Usually, both parents are selected as described above. However, sometimes, as in [Ratford 96], the choice of the second parent is affected by the choice of the first parent. This could be to ensure that the two parents are substantially different (or similar). Another alternative is for the second parent to be selected completely at random.

There have been some studies of selection methods, mentioned above, looking particularly at GAs. It is not certain that these studies can or should be applied to GP, but the assumption is usually made that they can. In general, it seems the previously popularised approach of Roulette-Wheel Selection quickly fell from favour, and has been replaced with a form of Rank-based or Tournament Selection, with low selection pressure, i.e. without a strong bias in favour of the fittest individuals.

Tournament Selection in this Thesis

For the purposes of this thesis, Tournament Selection was used, without over-selection. Early experiments indicated that Tournament Selection was marginally less likely to result in the GP population converging prematurely to variations of the best but sub-optimal individuals. These results could have been spurious, but since Tournament Selection has appeared to perform adequately, and there didn't seem to be any other apparent advantages to using over-selection or other selection methods, it seemed easier to stick with simple Tournament Selection. Varying other parameters and modifications to GP had a much bigger impact on GP performance.

In Summary, Tournament Selection is a very simple method to implement. It works quickly, since it does not require an initial scan of all population fitnesses (though one is needed to find the best individual for elitism), sorting of the population by fitness, or repeated scans of the entire population for each selection. It is easy to modify the selection pressure in small steps using the tournament size.

What follows is a brief investigation into Tournament Selection, looking at the effects of varying the tournament size, and the distribution of parent selections amongst a population.

4.2.2 Some Effects of Tournament Selection

The following graphs reproduce the effects of Tournament Selection on a generation of a population. Generational Replacement is used, i.e. an entirely new population is generated to replace the old one. Although just one population size, 50, is shown here, the shape of the graphs would be the same for other population sizes, though the scales on the axes would change.

For simplicity, the sample population used is ranked in order of fitness, starting with member 0 having the best fitness. Each selection of a parent involves randomly picking a tournament set of individuals, and then selecting the fittest of these as the parent. The effect of different tournament sizes is shown in the graphs. A tournament size of 1 corresponds to a randomly selected parent with no bias towards greater fitness. As the tournament size increases, through 2, 4, and 6, the bias towards greater fitness increases.

The number of parents which would be selected to create the next generation is taken here to be $1.4 * PopulationSize$, i.e. 70. This corresponds to the operators and operator selection probabilities used in most of the runs in this thesis: Crossover with 40% probability, requiring two parents, and various Mutation operators with a total probability of 60%, requiring one parent. Thus the number of parents which are selected on average from each generation to produce one child is $0.4 * 2 + 0.6 = 1.4$.

The selection of parents is simulated for one generation. This is repeated 1000 times and the results averaged to produce these graphs. The graphs show the frequency of selection, in Graph 4.27, the distribution of repeated selections, in Graph 4.28, the likelihood of not being selected, in Graph 4.29, and the likelihood of not being checked (i.e. the individual is never part of a tournament), in Graph 4.30.

Parent Selection Frequency Graph 4.27 shows how, unsurprisingly, the fitter individuals get selected more often, with the plot for tournament size=1 showing the distribution of selections if they are completely random and not based on fitness at all. Only with tournament size=2 are the least fit individuals in with any substantial chance of being selected when fitness is used as the selection criteria. Increasing the tournament size increases the bias abruptly in favour of the fittest individuals. With tournament size=6, the plot indicates that something very similar to 50% over-selection is occurring, as mentioned in Section 4.2.1.

The fittest individuals are repeatedly selected, as can also be seen in Graph 4.28. This suggests that there would be much repetition of subtrees within the population. A method for storing the entire GP population as a single directed acyclic graph (DAG), instead of as individual trees is discussed in [Handley 94, Keijzer 96, Ehrenburg 96]. As long as the components of the trees have no side-effects, earlier subtree evaluations can be cached and do not have to be re-evaluated when they appear in other trees. Handley reports a 15- to 28-fold reduction in node storage requirements, and 11- to 30-fold reduction in the number of nodes evaluated per run, for populations of size

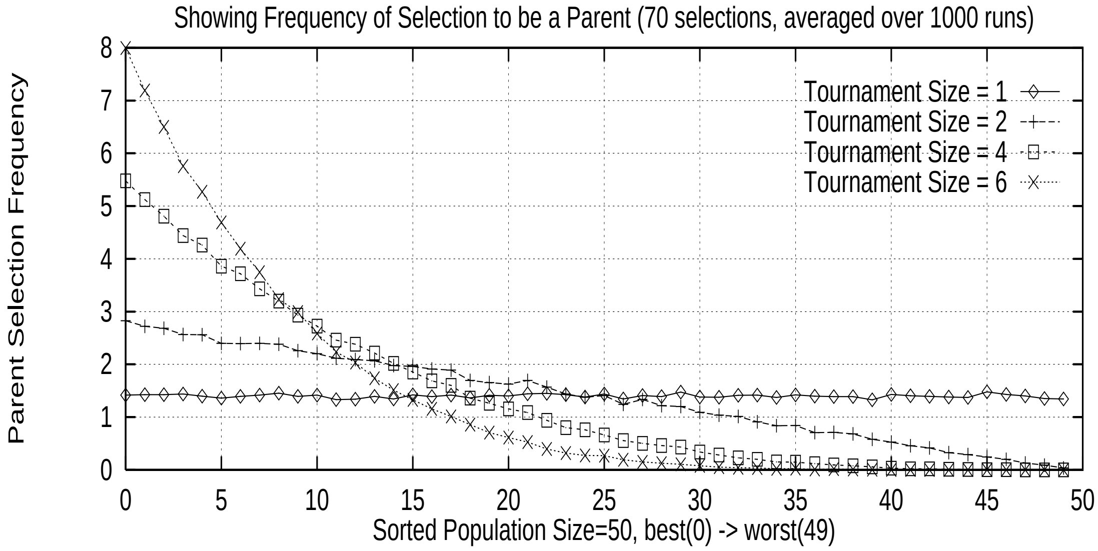


Figure 4.27: Average Parent Selection Frequency

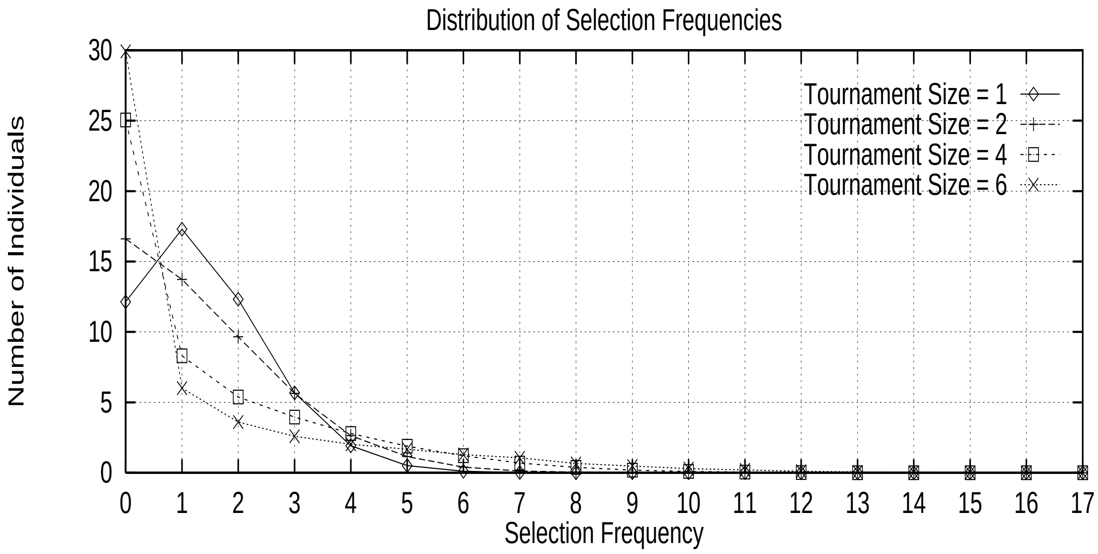


Figure 4.28: Average Distribution of Repeated Selections

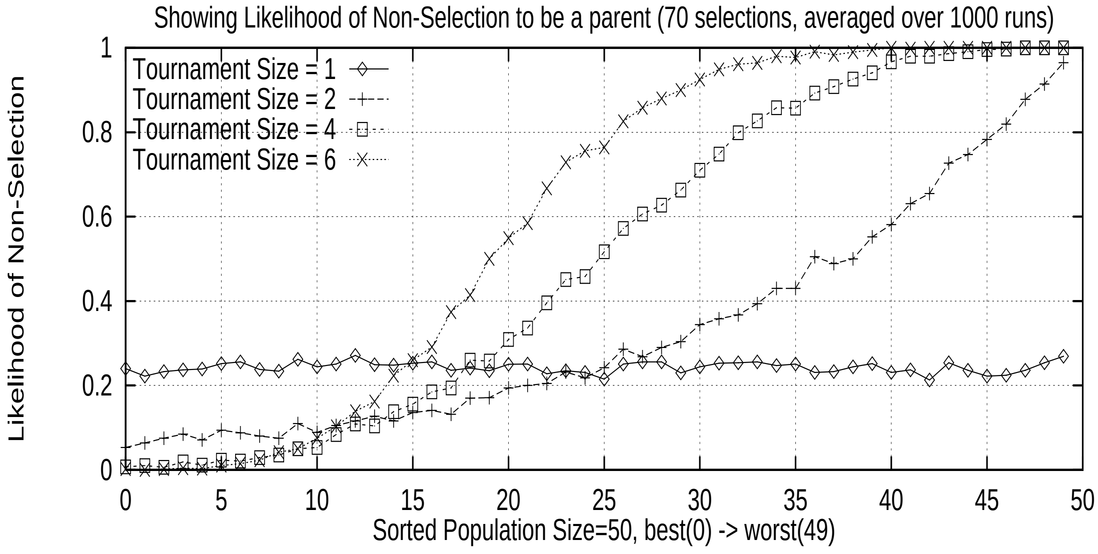


Figure 4.29: Average Likelihood of Non-Selection

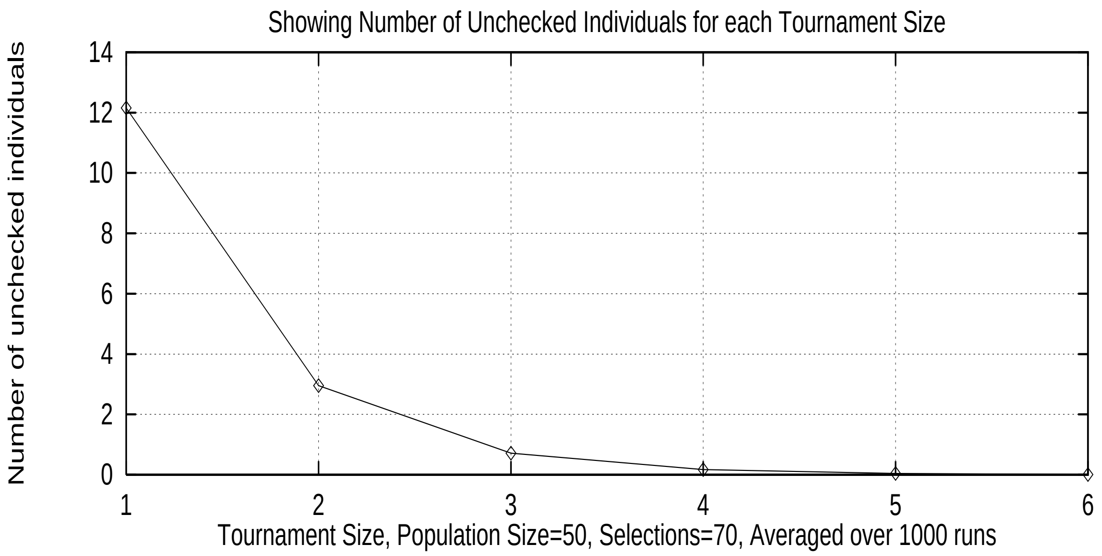


Figure 4.30: Average Number of Unchecked Parents

500. Speedups of this magnitude would obviously be a huge benefit for supervised training problems with large training sets, as discussed in Section 5, but it is not clear if it would work quite as well with the Dynamic Subset Selection method, discussed in Section 6, where the evaluations of trees would differ for each generation. Keijzer looks at the impact of DAGs on Automatically Defined Functions (ADFs, discussed in Section 3.3), suggesting that in certain situations the memory needed for node storage can be massively reduced.

Whilst not pursued in this thesis, the use of DAGs for efficiently representing a GP population looks very promising.

Distribution of Repeated Selections Graph 4.28 shows the distribution of selection frequencies, i.e. the number of individuals selected with each frequency. Looking at the Selection Frequency 0, it is apparent that a sizeable fraction of the population remains unselected. For tournament size=6, 30 out of the 50 individuals (i.e. 60%) are not selected to be parents. Even with the low selection pressure due to tournament size=2, on average 30% of the population remains unselected in each generation. This suggests there could be some practical way of evaluating but not retaining a fraction of the population. Experiments would be needed to ascertain if a current generation's fitness profile could be used as a guide to deciding whether or not to retain certain child individuals created for the next generation. A simple bias against unfit individuals would leave a smaller population that could be 'representative' in some way of the larger, full population. For small population sizes such as 50, shown here, the memory savings would be small, however they could be significant with much larger populations, since the percentage of unselected individuals would be the same regardless of population size.

Likelihood of Non-Selection Graph 4.29 shows the likelihood of individuals not being selected at all in a generation. To all intents and purposes, for larger tournament sizes, the top 10-20% of the population are always selected, and the bottom 10-20% are never selected. However, it can be seen in the plot for tournament size=2 that most individuals in the population have a substantial possibility of being selected.

Number of Unchecked Individuals Graph 4.30 shows the number of individuals in a population which take no part in the Tournament Selection process, i.e. they are never picked to be part of a tournament. As the tournament size increases, although the number of tournaments remains the same, the number of individuals being picked to participate in the tournaments increases substantially. Thus the likelihood of an individual remaining unpicked decreases substantially. This indicates that there isn't much scope for efficiency savings in CPU or memory usage by avoiding the production and evaluation of these 'wasted' individuals. Even with large populations, the number of unpicked individuals would be small.

Looking at the results above, the main area with scope for improvement (in GP using Tournament Selection) is the use of a representation such as a directed acyclic graph, where the entire population is stored compactly, and subtree evaluations can be cached. The Graphs 4.27 and 4.28 indicate that there would be a great deal of redundancy within the population, with many trees coming from only a few individuals. The potential reduction of memory and/or CPU requirements reported for this representation is impressive, but with the caveats that to save CPU time the evaluations of the nodes must be free of side-effects. However, the performance of GP can no doubt be optimised in several different ways (perhaps with different operators as in [Ehrenburg 96]) to run better using such a representation.

Other than the use of DAGs, there seem to be no obvious areas where GP can be improved substantially to take advantage of (or avoid the consequences of) Tournament Selection.

4.3 Discussion

There are two main issues raised in this chapter. The first and perhaps main issue, raised in Section 4.1 as the MAX problem, is that Crossover (and indeed other operators) can have an adverse interaction with restrictions on tree size, resulting in a loss of operator efficiency and, in certain cases, the inability of the operators to improve on a sub-optimal solution. There are no guaranteed fixes for this problem, which is likely to arise with varying degrees of severity whenever GP individuals start approaching their limits on size. However, being aware of the possibilities, combined with a few simple precautionary steps, should minimise its impact. The main precaution seems to be to allow the individuals, if possible, to grow to their preferred size, perhaps applying a gentle parsimony pressure to bias GP towards selecting smaller individuals. Where this is not possible, greater care needs to be taken to ensure that the operators can function well with individuals approaching or at the size limits.

The second issue, raised in Section 4.2, concerns the great potential of storing the entire GP population as a directed acyclic graph, with its huge savings in memory requirements and reduction in subtree evaluation through the caching of earlier evaluations, and the caveat that the GP nodes must be free of side-effects during evaluation.

Part III

Genetic Programming and Supervised Learning

Chapter 5

Making use of the Training Set in GP

It soon becomes obvious that one of the first and biggest hurdles to overcome when using GP to tackle a large supervised learning classification problem is the sheer number of fitness evaluations needed by GP. To calculate the fitness of a GP individual, it is evaluated against each case in the training set; its fitness relates to the total number of errors made. A large training set means a large number of evaluations. A difficult problem may require a large population of individuals, which may take many generations before finding a good solution. Many tens of millions of evaluations might be needed, taking many days of computer time.

This chapter looks at what can be done with the training set, including methods from other fields such as Statistics and Machine Learning, with the aim of speeding up GP, and making it a more practical and reliable tool. There is a great deal of literature for training artificial Neural Networks and constructing Decision Trees for classification problems, some of which is described in Section 5.1. Section 5.2 describes some methods for selecting training and test sets from raw data in a statistically sound fashion. Section 5.3 describes several methods which have been applied to evolutionary algorithms, followed by Section 5.4 which describes some methods applied to GP, including some original work done for this thesis.

5.1 Training Sets in Machine Learning

Without doubt the GP and Evolutionary Algorithm communities in general are rediscovering a lot of work that the wider Machine Learning community has known about for years.

Neural Networks One example from the area of Neural Networks is provided by [Zhang 94]. Zhang looks for:

> "... factors which influence the learning speed and generalisation ability of the networks. One of them is the nature and size of the training set. While there is no guarantee that the generalisation performance is improved by increasing the training set size, the training time increases as the number of examples increases. In general, one should choose those examples which are most likely to help the network solve the problem."

Neural Networks obviously share some of GP's difficulties with supervised learning of large training sets. Back-propagation, perhaps the most common method for training Neural Networks often requires that the training data is presented repeatedly (in epochs) before the network succeeds in solving the problem. Each time a case or batch of cases is presented to the learning network, the error between the network's output and the target output is propagated back through the nodes in the network, so that each node has an error term. The links between nodes (i.e. weights) are adjusted in an attempt to reduce the error.

In the paper, Zhang proposes a criterion for selecting critical examples, and presents an efficient method for selecting examples and scheduling their training order based on this criterion. Bypassing a great deal of mathematics, the method can be summarised as follows: choose the cases on which the network makes the largest errors. The network is then trained on this subset of cases, minimising the errors. And then the process is repeated.

Zhang reports:

> "Our experimental results show that the selective incremental learning finds and uses only a critical subset of given examples, which leads to a considerable enhancement in training speed and generalisation performance."

Cohn *et al* present a different approach to training Neural Networks on a subset of the whole training set, [Cohn *et al.* 94]. They use the idea of "selective sampling" from "regions of uncertainty" to guide the choice of training examples. For a binary classification problem, a network is trained on an initial random sample of training examples. The network's real-valued output (between 0 and 1) is then thresholded into one of three zones: "Class 1" (0.9 or greater'), "Class 2" (0.1 or less), and "Uncertain" (between 0.1 and 0.9). The "Uncertain" points define a "region of uncertainty". As yet unclassified points which fall into this region are selected from the training set.

Several limitations of this approach are highlighted: the "region of uncertainty" can come to encompass the entire training set; it is highly dependent on the initial random sample of examples; and there are difficulties in scaling up to more complex problem domains. Nevertheless,

> "... selective sampling demonstrates significant improvement over passive, random sampling techniques on a number of simple problems."

Decision Trees One example from the area of Decision Trees is provided by [Quinlan 86]. A decision tree is a hierarchical arrangement of small classification rules, where each node represents a 'decision' about one of the fields in the problem. For example, *If Field 1 is Red then A else if Field 1 is Blue then B else if Field 1 is green then C*, or *If Field 2 is TRUE then D else E*. The components *A,B,C,D,E* can represent a final classification, e.g. *is-a-fish*, or could be a further decision subtree. There is a large degree of overlap between decision trees and GP trees, e.g. [Vere 95].

Quinlan describes an approach for synthesising decision trees called ID3:

> "The basic structure of ID3 is iterative. A subset of the training set called the *window* is chosen at random and a decision tree formed from it; this tree correctly classifies all objects in the window. All other objects in the

> training set are then classified using this tree. If the tree gives the correct answer for all these objects then it is correct for the entire training set and the process terminates. If not, a selection of the incorrectly classified objects is added to the window and the process continues. In this way, correct decision trees have been found after only a few iterations for training sets of up to thirty thousand objects described in terms of up to 50 attributes. Empirical evidence suggests that a correct decision tree is usually found more quickly by this iterative method than by forming a tree directly from the entire training set.
>
> ...
>
> While decision trees generated by the above systems are fast to execute and can be very accurate, they leave much to be desired as representations of knowledge."

This last point is one which could also be levelled at much of what is produced by GP. However, Quinlan has demonstrated several methods for simplifying decision trees, [Quinlan 87].

Salzburg demonstrates the effectiveness of feature selection "for improving the speed and accuracy of machine learning programs on large data sets", [Salzburg 93]. "Combined Stepwise Selection" (CSS) is used in combination with four different methods of classification: back-propagation, nearest neighbour, nested hyperrectangles, and multivariate ("oblique") decision trees which use multivariate tests at each non-leaf node of the decision tree.

Rather than look at all possible combination of features, which would take a prohibitively long time, CSS attempts first to reduce the set of features one by one, then selects subsets of features from the reduced feature set. A classifier is evaluated (i.e. trained from scratch) on a set of features. The classifier is then evaluated on each possible subset where one feature has been removed from the initial set. If the classifier works equally well without the deleted feature, it is assumed to be OK to ignore it. The feature which causes the smallest decrease in accuracy is removed. This process is repeated as long as the decrease is below a preset threshold (0.5%).

Using brute-force search, the best pair of features is chosen from the reduced feature set, evaluating the classifier on all possible pairs of features to allow for possible and quite common pairwise interactions between features. Each of the remaining features (from the reduced feature set) is then tried in turn, in combination with the best pair. The feature that gives the best improvement in accuracy is added to the pair, and the process is repeated with the set of three, and so on, trying each remaining feature in turn, until the improvement is less than the preset threshold.

Salzburg reports that the classifier methods were able to produce more accurate results with the smaller feature sets. The smaller feature sets allowed the discovery of new knowledge about the underlying scientific domain. Whilst the CSS method is not perhaps ideal for GP (the number of runs needed would probably be prohibitive), it does indicate again that feedback between the learning method and the way the problem is presented to the learning method can lead to faster and more accurate results.

Feature set selection is a topic which is not explored further in this thesis.

5.2 Selecting Training and Test Sets

The topic of selecting training and test sets from raw classification data has had a great deal of effort and statistics thrown at it. The aim is to choose representative sets which give the particular learning method every opportunity to produce an effective classifier, based on the training set, and an accurate estimate of how well the classifier will generalise to unseen data, based on the test set. Simply partitioning the data randomly into two sets runs the risk of selecting non-representative sets. One method in particular has been established as a popular and effective selection method.

Cross-Validation involves splitting the data into k equal or nearly equal sized sets. One of the k sets is taken as the test set, and the remaining k-1 sets are combined to form the training set. This is repeated, taking each of the k sets in turn as the test set. The learning algorithm is trained and tested on each of the combinations. Choosing the value of k can be done by guesswork or experiment. If k is the number of cases in the whole set, so that the test set is of size one each time, the method is known as

Jack-Knifing.

The data in the Thyroid problem described in Section 6.6 was already divided into training and test sets for earlier Machine Learning studies, and the TicTacToe data, described in Section 6.8, was used as a single training set, so the topic of training and test set selection was not explored in this thesis.

5.3 Approaches for Evolutionary Algorithms

The standard use of a training set in an Evolutionary Algorithm (EA) is to evaluate each individual against each case in the training set in order to establish the individual's fitness. This is repeated for each individual in the population and for each generation. Whilst laudably simple, this approach can obviously lead to a very large number of evaluations for any but the simplest of problems. It would be an advantage if a much smaller subset of the training set could be used in place of the whole training set.

A method for selecting a single representative subset to use as a training set is Historical Subset Selection, described in Section 6.2. It shows that even a very simple selection method can work well. The smaller training set leads to much a faster turnaround time for GP. Variations of this simple approach have been mentioned in several papers and discussions with other machine learning practitioners, and it is an obvious method to try when faced with large training sets.

A more flexible method is to pick a variety of subsets during the course of a training run. There are many ways that different subsets could be selected from the training set. The goal is to pick the right subsets to allow the learning algorithm to proceed as fast and as accurately as possible. The simplest method for picking a different set for each generation is randomly. Random Subset Selection, described in Section 6.4, can perform surprisingly well, though not as well as the more directed methods described below. As with HSS, this method has been mentioned in discussions with other machine learning practitioners as an obvious method to try when faced with large training sets.

HSS and RSS have been described here to provide a context for the following method, Dynamic Subset Selection, which was developed during the course of this thesis to allow GP to use and benefit from large training sets.

One criteria that can be used to guide the selection is the performance of the population. If individuals in the population consistently classify a case correctly, then that case is of limited use in judging the relative performance of the individuals in the population. On the other hand, a case which is often misclassified does provide more useful information for the fitness function.

In Chapter 6, Dynamic Subset Selection makes use of the difficulty of each training case, i.e. how often it is misclassified, and its age, i.e. how many generations since it was last selected. This has worked well on some large classification problems, using less computer resources to produce better results than standard GP. Siegel describes a similar algorithm in [Siegel 94], but does not make use of the age aspect.

A different approach to selecting subsets on which to evaluate the entire population is described in [Hillis 90]. Hillis uses two spatially distributed populations, where the fitness of an individual in one population is based on how well it confounds an individual at the same location in the other population. One population, the 'hosts', is evolving minimal sorting networks, whilst the other population, the 'parasites', is evolving difficult subsets of training cases, where the success of a training subset represents a failure of a sorting network and vice versa. The host/parasite relationship prevents large portions of the population from becoming stuck in local optima. "Successive waves of epidemic and immunity keep the population in a constant state of flux." Only significant training cases show up in the parasite population, so it is sufficient to apply only a few tests to an individual each generation, substantially reducing computation time per generation. These two factors mean the system can be run productively for many more generations.

Rosin and Belew also co-evolve populations, using a globally calculated fitness and "fitness sharing" which depends on the difficulty of an individual's successes, i.e. the more individuals which share a success, the less important or difficult that success is, [Rosin & Belew 95]. This links quite closely with the idea of fitness sharing and 'niches' used by Goldberg and Richardson, where functionally similar individuals have their fitnesses reduced, favouring individuals which have unique abilities, [Goldberg & Richardson 87]. Greene and Smith make a much more explicit use of niches, [Greene & Smith 93]. Individuals are ranked according to their 'discriminability', i.e. the ability to differentiate examples correctly. Moving sequentially through this ordering, each training example is allocated to the first individual which correctly differentiates it. When all the examples have been "consumed", the remaining individuals are discarded, i.e. all the available niches are full. Reproduction takes place randomly within the remaining individuals. Good results and potential are reported.

Angeline and Pollack look at competitive environments where the fitness is related only to the current ability of the population, [Angeline & Pollack 93]. Individuals compete against one another in fitness tournaments playing TicTacToe, amongst other things, rather than a pre-defined 'expert' player. They report that

> "a competitive fitness function requires only a minimal understanding of the search space for a complex task.
> ... [U]sing the population as a reservoir for comparison is preferable to using an exemplar for the task when an objective measure of fitness is unavailable."

5.4 Approaches for GP in this thesis

Chapter 6 presents Dynamic Subset Selection (DSS), and the two simpler methods: Random Subset Selection, and Historical Subset Selection. Chapter 7 presents Limited Error Fitness (LEF). DSS and LEF are approaches which can reduce the number of fitness evaluations needed by GP, and can enable GP to find more accurate solutions. Chapter 8 demonstrates that these two approaches can allow the use of much smaller population sizes in GP.

Chapter 6

Dynamic Subset Selection

When using GP on a difficult supervised learning problem with a large set of training cases and a large population size, a very large number of tree evaluations must be carried out every generation. This chapter describes three approaches, previously published in [Gathercole & Ross 94a, Gathercole & Ross 94b], to reduce the number of such evaluations by selecting a small subset of the training data set on which to actually carry out the GP algorithm.

Dynamic Subset Selection (DSS) -

using the performance of the current GP population to select a new subset of 'difficult' and/or under-selected cases every generation

Historical Subset Selection (HSS) -

using the performance of previous GP runs to construct a single subset

Random Subset Selection (RSS) -

selecting a new subset at random every generation

GP, GP+DSS, GP+HSS, and GP+RSS, are compared on a large classification problem, the Thyroid Problem. GP+DSS can produce better results in less than 20% of the time taken by GP, and produces better results than an attempt using a variety of Neural Networks. GP+HSS can nearly match the results of GP, and, perhaps surprisingly, GP+RSS can occasionally approach the results of GP. GP and GP+DSS are then compared on a smaller problem, the TicTacToe Problem.

6.1 Subset Selection Methods

At present, the potential of Genetic Programming (GP) and Genetic Algorithms (GA) has been demonstrated in many different problem areas. Generally, these experiments have involved solving small, relatively neat problems. The future beckons, however, with large and horribly messy problems, to which the GP method will have to be scaled up.

With supervised learning, a training set of cases is involved and the aim is to learn how to classify these known example cases and hopefully generalise to be able to correctly classify all possible cases. Large problems will require large training sets. In the standard GP algorithm, the entire population of GP trees is evaluated against the entire training data set, and so the number of tree evaluations carried out per generation is directly proportional to both the population size and the size of the training set. This chapter looks at ways of reducing the effective training set size, and shows that this can also allow a reduction in population size.

The simple method of Dynamic Subset Selection (DSS) is described in Section 6.3. DSS reduces the number of such evaluations that need to be carried out before a satisfactory answer evolves and, in fact, can produce a more general answer. Two other selection methods are described for purposes of comparison: the (even) simpler method of Random Subset Selection (RSS), in section 6.4, and Historical Subset Selection (HSS), in Section 6.2, which uses previous GP runs to select a single training subset. A classification problem involving the Thyroid data set, described in Section 6.6 is used as a token 'large and messy' problem. A smaller problem involving TicTacToe endgame positions is described in Section 6.8.

Following on from the results obtained by DSS, a Dynamic Fitness Function (DFF), based on DSS, is proposed for further study.

6.2 Historical Subset Selection (HSS) - the algorithm

For HSS, previous straightforward GP runs are used to establish some measure of how difficult each training case is. Over the course of several runs (say, five or so), the cases misclassified by the best population member in each generation in each run are recorded. These cases then make up the subset used in further GP+HSS runs, and the subset remains static after its initial selection. Due to the rough-and-ready method by which it is selected, the subset contains a mixture of many difficult cases and many which are actually quite easy to classify. Even a best-of-generation population member makes some simple misclassifications early on in its development.

Distribution of Classes in Thyroid Data				
Set	Class 1 (% of set)	Class 2 (% of set)	Class 3 (% of set)	Total
Training	93 (02%)	191 (05%)	3488 (92%)	3772
Test	73 (02%)	177 (05%)	3178 (93%)	3428
HSS	65 (12%)	190 (35%)	290 (53%)	545

Table 6.1: Distribution of Classes in Thyroid Data

Some simple checks showed different runs producing very similar subsets selected by this method. The statistics almost always agreed on which cases were most often misclassified, and only disagreed on some of the easier cases. The subset size used in the runs was 545, consisting of every single case misclassified during seven previous runs of a standard GP. A core of around 300 cases were misclassified more than once or twice, and so were considered to be at least *moderately* difficult cases. The distribution of classes within the set can be seen in Table 6.1. Nearly all of the cases from the two smallest classes are included in the subset, making up nearly 50% of the subset, as opposed to just 7% of the whole training set.

6.3 Dynamic Subset Selection (DSS) - the algorithm

Working with the assumption that supervised learning with GP can proceed effectively even whilst only using a subset of the full training set, this simple idea of DSS is based upon a few premises and a small amount of hindsight. Firstly, it is of benefit to focus the GP's attention onto the *difficult* cases, i.e. the ones which are frequently misclassified. Secondly, it is also of benefit to check cases which have not been looked at for several generations. This leads to the final point that all of the cases in the training set should be looked at, eventually.

The algorithm for DSS involves randomly selecting a target number of cases from the whole training set every generation, with a bias so that a case is more likely to be selected if it is 'difficult' or has not been selected for several generations. In each generation, using a very simple procedure, the subset is selected by the following two passes through the full training set.

- In one pass of the entire training set, of size **T**, in a generation, **g**, each training case, **i**, is assigned a weight, **W**, which is the sum of its current 'difficulty', **D**, exponentiated to a certain power, **d**, and the number of generations since it was last selected (or age), **A**, also exponentiated to a certain power, **a**:

  $$\forall i : 1 \leq i \leq T, \quad W_i(g) = D_i(g)^d + A_i(g)^a$$

  $$\forall i : 1 \leq i \leq T, \quad D_i(0) = 0,\ A_i(0) = 1$$

  ($A_i(0)$ is set to one so that each case has a non-zero weight.)

  The sum of all the cases' weights is also calculated during this first pass.

- Then, in a second pass of the entire training set, each case in turn is given a likelihood (not strictly a probability, more an expected number of such cases), **P**, of being selected to be in the subset. A case's selection likelihood is given by its weight divided by the sum of all the cases' weights and multiplied by the target subset size, **S**:

  $$\forall i : 1 \leq i \leq T, \quad P_i(g) = \frac{W_i(g) * S}{\sum_{j=1}^{T} W_j(g)}$$

  A random number is generated between 0 and 1. If the case's chance P is greater then the random number, it is selected. If a case, i, is selected to be in the subset, then its difficulty, D_i is set to 0, and age, A_i is set to 1 (so that the weights are always greater than zero), otherwise its difficulty remains unchanged, and its age, A_i is incremented. While testing each member of the GP population against each case in the current subset of training cases, the difficulty, D_i, (starting from 0) is incremented each time the case is misclassified by one of the GP trees.

Using this process, if a weight is sufficiently large it will be scaled by S to be greater than 1 and so that case will definitely be selected to be in the subset.

The subset size will fluctuate around the target size **S** each time a new subset is selected. Given that some cases will be selected with a probability of 1 (due to the rough and ready selection process), the average subset size will in fact be slightly larger than the target size. Other selection methods could easily produce subset sizes of exactly **S**, e.g. roulette wheel selection as used in Chapter 8, but it was felt that a varying subset size might contribute more to the efficacy of the GP algorithm, and certainly did not seem to hinder it. The current generation of the GP's population is then evaluated against this subset of cases instead of the entire training set.

The equation for calculating the weights of each case in the training set, $W_i(g) = D_i(g)^d + A_i(g)^a$, is kept as simple as possible. The aim is to find a balance between age and difficulty. The age exponent means that eventually even the easiest case is certain to be reselected as the age contribution to the weight rapidly increases with each passing generation. The exponents allow the relative contribution of age and difficulty to be easily adjusted for different population sizes and training set sizes. As it happens, the fact that exponents are combined in this way means that the equation is quite robust when used, unchanged, with a variety of population and training set sizes. Other, more complicated, combinations of age and weight are possible, but do not appear to be necessary for DSS to function well.

The difficulty ratings of cases depend on the size of the population. Larger populations lead to larger difficulty ratings. This could result in difficult cases being reselected more frequently in runs with larger populations than those with smaller populations. However, since the age weight uses an exponent, it soon (after a few generations more for larger populations) increases sufficiently to achieve a balance with the difficulty weights.

To use this form of DSS, the following three parameters have to be set:

Target Number of cases - subset size

Difficulty exponent - importance given to difficult cases

Age exponent - importance given to unselected cases

Currently (and, it seems, as always), choosing useful combinations of parameter settings is somewhat of a black art. For the purposes of the Thyroid data set, a target size of 400 (out of 3772) was quickly chosen as an effective value after some experimentation, though other values from 200 upwards also worked well. This corresponds to slightly more than the number of moderately difficult cases selected by the HSS method, leaving room for a few easy cases to be included. With the target size set at 400, it was easier to select sensible values for the two exponents. An average difficulty rating for a case, with a population size of 10000, might be around 2000 or so. The most difficult cases could have a rating of up to 10000. With a target size of 400, it would take at least 10 generations to cover all the 3772 training cases. Given this disparity between a very 'difficult' case and an 'old' case, an arbitrary decision was made to keep the difficulty exponent to 1.0 and to set the age exponent to 3.5. With these exponents, the most difficult cases and cases around 15 generations old would have roughly equivalent weights.

Siegel describes an algorithm similar to DSS in [Siegel 94], but does not make use of the age aspect, instead using only a bias towards difficulty.

6.4 Random Subset Selection (RSS) - the algorithm

In RSS, for each generation, each case in turn is selected to be in the current subset of training cases with an equal likelihood, which is scaled to ensure that the subset selected, on average, is of the target size. As with the DSS method, the subset size fluctuates around the target size with each generation.

$$\forall i : 1 \leq i \leq T, \quad P_i(g) = \frac{S}{T}$$

Without any weights biased by difficulty or age, RSS provides an opportunity to distinguish between the effects of using subsets, and the bias introduced by the performance of each generation of the evolving population which affects the subset selection in DSS.

6.5 GP Details

Generational replacement with elitism was used along with tournament selection with a tournament size of 6, and large population sizes of 5000 and 10000. A small parsimony factor was used, in combination with a form of restriction on tree depth to no deeper than 17. The operators were

- 40% Crossover
- 10% Duplicate Parent
- 50% Mutate Subtree

In hindsight, the restriction on tree depth was probably not ideal; a restriction on the number of tree nodes would be preferable. Also, the Mutation operator was quite a blunt operator; perhaps an extra Mutate Node operator might have been useful.

6.6 The 'Large and Messy' Thyroid Problem

The Thyroid data set [Werner 92] represents a hard classification problem; one of several stored at [UCI 97]. The results reported for Neural Networks [Schiffmann *et al.* 92a, Schiffmann *et al.* 92b] provide an useful comparison with the performance of GP, however, the main aim for this investigation was to improve the performance of GP on a hard problem.

The data is based upon measurements of in-patients at a clinic. Each measurement vector consists of 15 binary values (0.0 or 1.0) and 6 floating point values (i.e. 21 fields in all), and falls into one of three classes. Class 3 signifies a 'normal' thyroid gland and is by far the most common class in both training and test data sets, whilst classes 1 and 2 signify that the patient later experienced a thyroid gland problem. To be useful in practise in identifying potential thyroid problems, a classification scheme would have to correctly classify significantly more than 92% of all cases, since over 92% of all patients have a normal (class 3) thyroid gland, as can be seen in Table 6.1.

There are 3772 cases in the training set and 3428 cases in the test set. Examination of the data in graphical form, e.g. using XGobi [Swayne *et al.* 91], reveals that the boundaries between the classes of points are very murky indeed. Points from different classes seem to mingle freely with each other, as can be seen in Figure 2.1 in Section 2.2.

In all runs, only the training set is used by the GP to try to evolve its population to classify the thyroid cases into their correct classes. The test set is only used as a check on each generation's best (or fittest) classifier (with respect to the training set), to see how well it generalises to another set of the same kind of data. A run's best classifier is taken to be the one which performs best when evaluated on the training set. This is not necessarily the one which performs best on the test set. The setup which generates the fittest classifier with respect to the training set which then performs best on the test set in this way is taken to be the most successful one.

The function set, chosen after a great deal of guesswork is:

{ IFLTE, +, -, *, %, TANH, LOG, MINIMUM_OF_3, NEGATE, SQRT }

and terminal set used in this problem is:

{ B_1 to B_{15}, F_1 to F_6, 0, -1, Random_Constant }

where 'B' and 'F' refer to the binary and floating point fields of the Thyroid cases. '0' and '-1' refer to constants added to the terminal set as a possible aid to GP in constructing useful subtrees. There was some experimentation with and without these extra constants, and with and without Koza's recommended 'ephemeral random constant' (each time a new node of this type is created, i.e. by Mutation, it is given a random value which it holds for the lifetime of the node). There was no apparent benefit in using the random constant in the Thyroid problem, making the resulting trees messy and hard to decipher. It was also not clear if the constant nodes '0' and

'-1' had any beneficial effect either. Much more experimentation is needed to establish 'ideal' terminal and function sets.

Modification to Thyroid Problem

To make things easier for the GP (after a few initial, unsuccessful runs), the Thyroid problem was reformulated to classifying cases as class 3 or not class 3. This reformulation allowed the GP tree's outputs to be treated as boolean:

- output $\geq 0 \Rightarrow$ class 3
- output $< 0 \Rightarrow$ not class 3

It proved relatively straight forward, in a separate run, for DSS to produce a tree expression which could distinguish between classes 1 and 2 with 100% accuracy on both the training and test sets. This subproblem can be seen to be quite simple in Figure 2.2 in Section 2.4. In fact, it is linearly separable. The simple tree shown in Figure 6.1 is sufficient to distinguish between class 1 and class 2 cases with 100% accuracy, and was discovered by GP very easily. If this approach were to be used in practise, two GP expression trees would have to be used in two phases: First (and most difficult) distinguish between class 3 cases and the others, then, if it is not a class 3 case, distinguish between class 1 and class 2 cases.

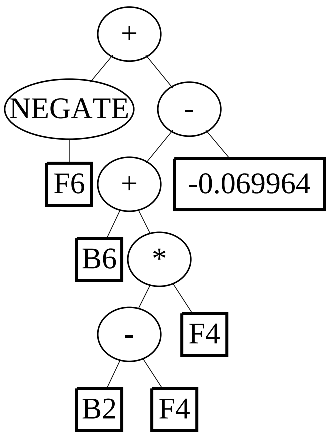


Figure 6.1: Simple GP tree which distinguishes between class 1 and class 2 cases with 100% accuracy.

Experiments were carried out with three methods of Subset Selection and compared against the baseline performance of the standard GP which uses the entire training set in each generation.

6.7 Thyroid Results

This is by far the larger of the two problems attempted in this chapter. Results are given in Table 6.2 for a typical DSS run with a population size of 10000, and for typical GP, DSS, HSS, and RSS runs with a population size of 5000, and for the best Neural Network results reported in [Schiffmann *et al.* 92a]. It was not possible to complete a run of GP with a population size of 10000 in a reasonable time!

Figure 6.3 shows the DSS run easily outperforming RSS, though RSS is still showing signs of improvement after 120 generations. This indicates that subset selection can produce useful results even without any bias used in selecting cases, though the bias used in DSS can be seen to greatly improve subset selection. Figure 6.4 shows the standard GP run outperforming HSS, though only due to a surge around generation 48. These two methods often produce similar scores, but HSS achieves them with many fewer tree evaluations. For this problem, it is thus possible to extract a useful subset of cases using a very simple selection process which allows GP to perform nearly as well (with many fewer evaluations) as with the whole set. Figure 6.5 shows DSS matching GP results using many more generations, but only 20% of the number of tree evaluations.

The best tree produced by the DSS run (with population size = 10000) to distinguish between class 3 and not class 3, was found on Generation 69, giving only 25 errors on the test set, underlined in Table 6.2, and is shown in Figure 6.2. It used only 13 out of the 21 variables available in classifying the Thyroid cases.

The dynamics of the DSS components can be seen in Figure 6.7, taken from a run with a population size of 5000. The curve for 'average_powered_time_since_used' shows the average weight corresponding to the age (i.e. how many generations since last being selected) of each case in the training set. Rising sharply early on, as only a few are selected and the rest remain unselected, the curve peaks and drops after generation 10,

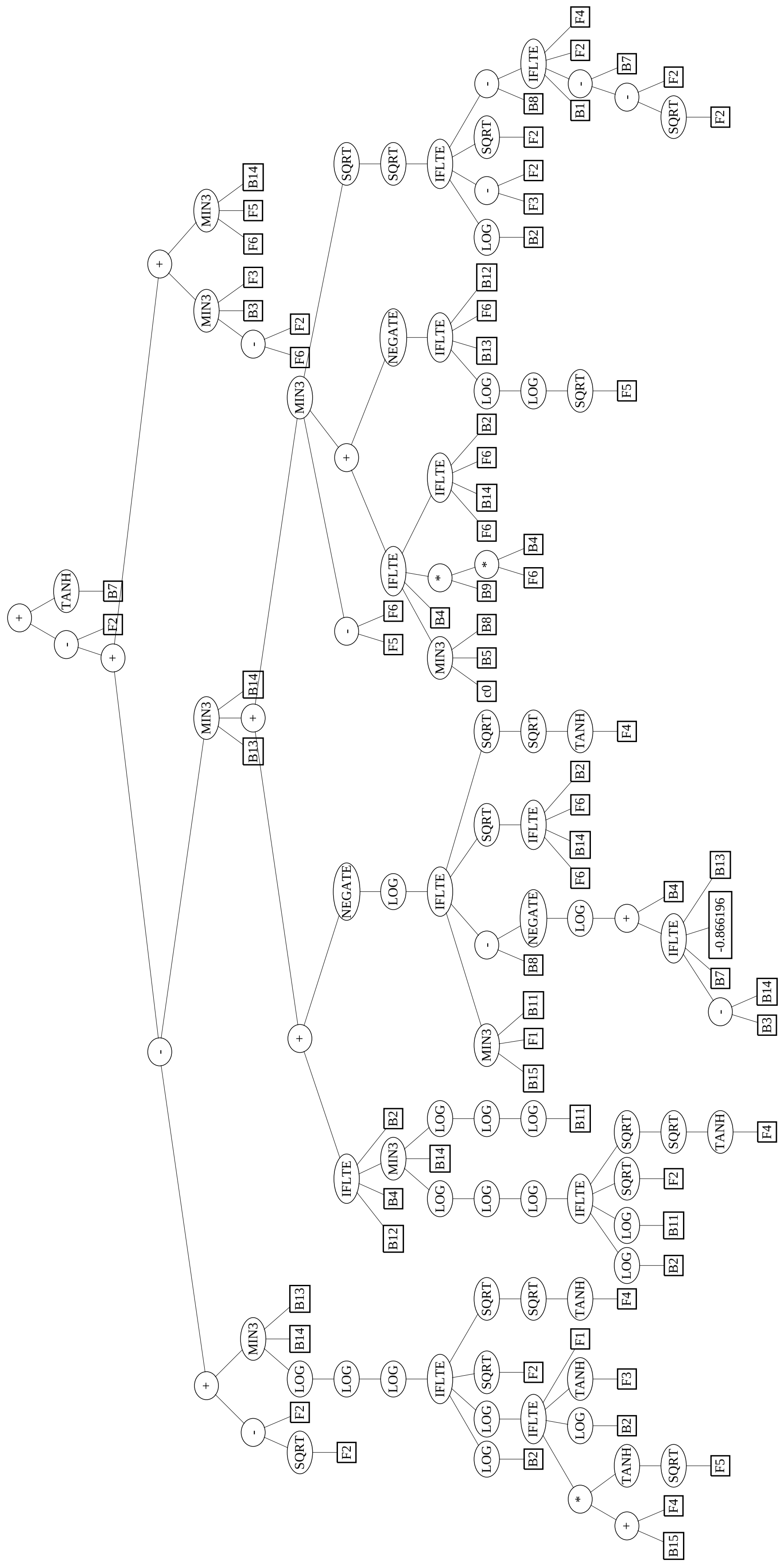


Figure 6.2: GP+DSS tree which distinguishes between class 3 cases and all others with high accuracy.

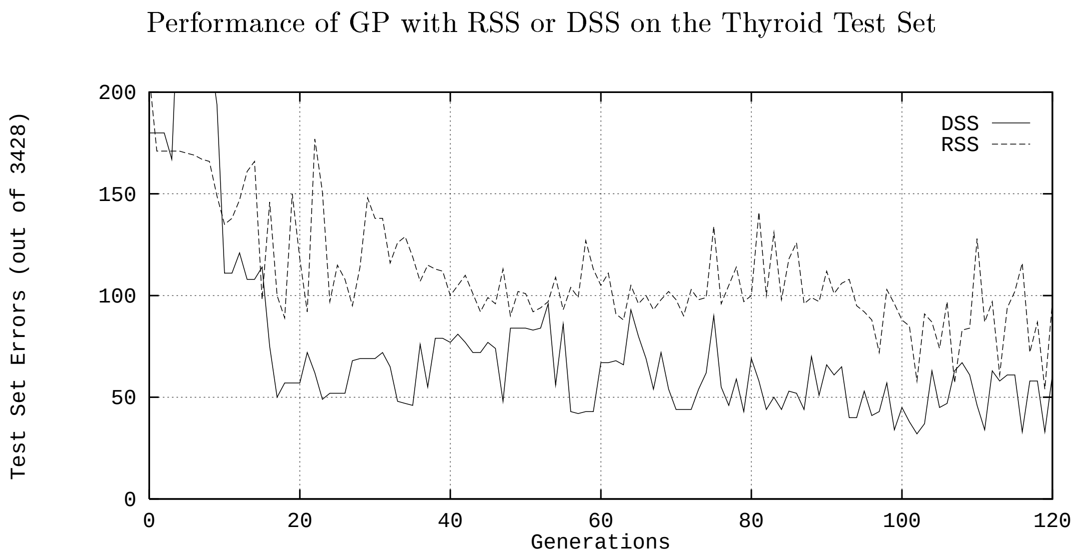


Figure 6.3: The number of errors made on the Thyroid test set by the best-of-generation trees produced during a run of the DSS and RSS Methods for each generation.

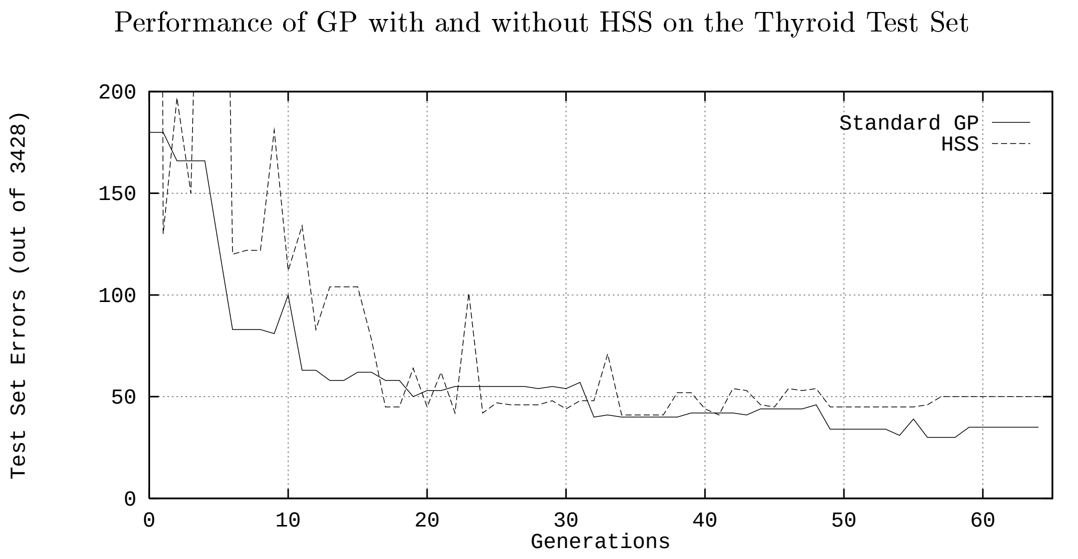
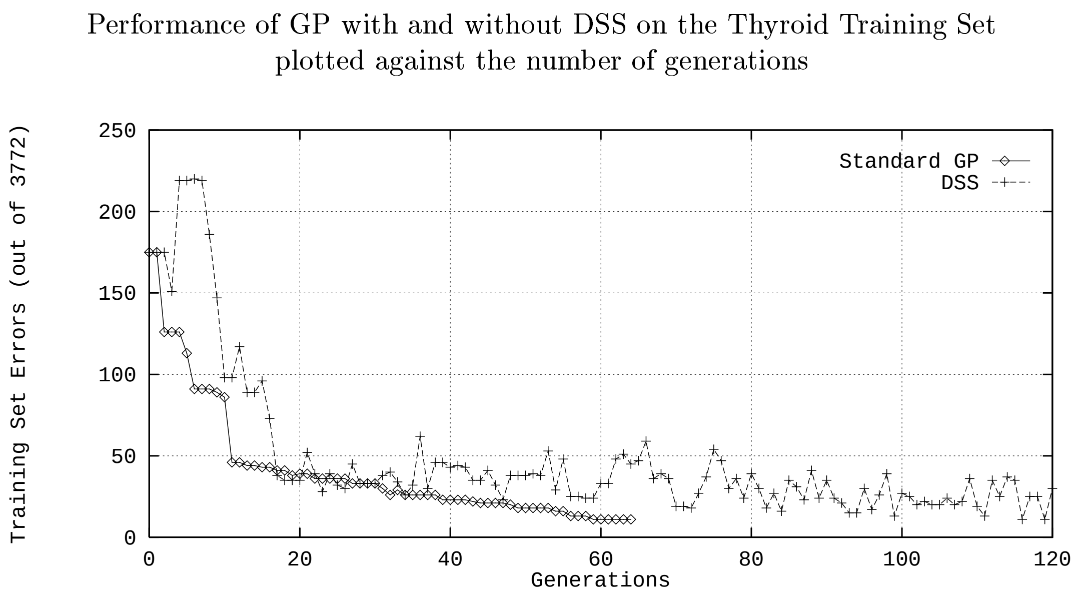


Figure 6.4: The number of errors made on the Thyroid test set by the best-of-generation trees produced during a run of the Standard GP and HSS Methods for each generation.



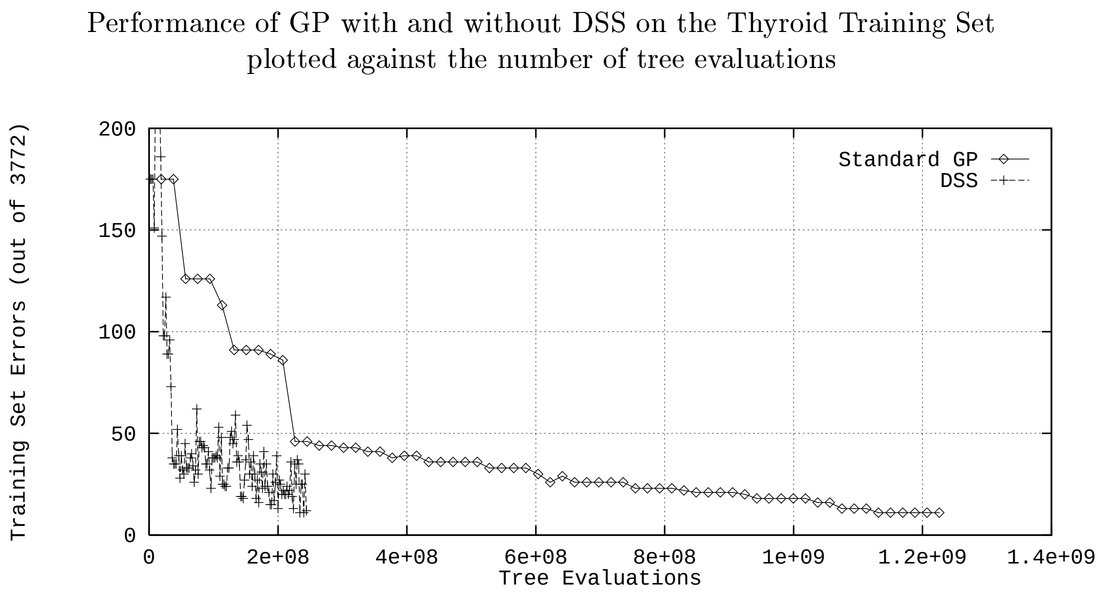


Figure 6.5: The number of training set errors made on the Thyroid training set by the best-of-generation trees produced during a run of GP with and without DSS methods, plotted against the number of generations and tree evaluations.

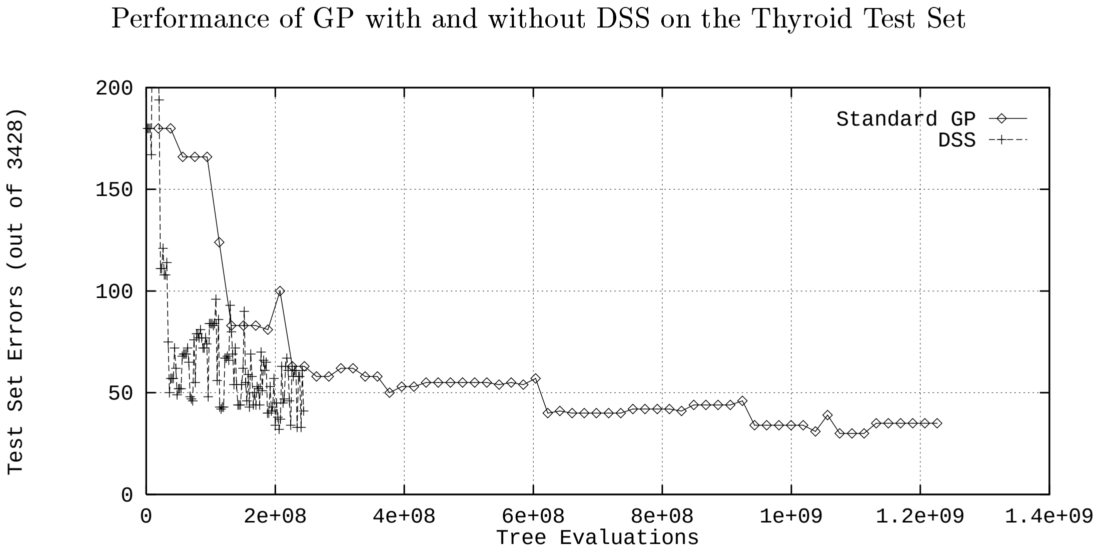


Figure 6.6: The number of errors made on the Thyroid test set by the best-of-generation trees produced during a run of the Standard GP and DSS methods against the number of tree evaluations carried out.

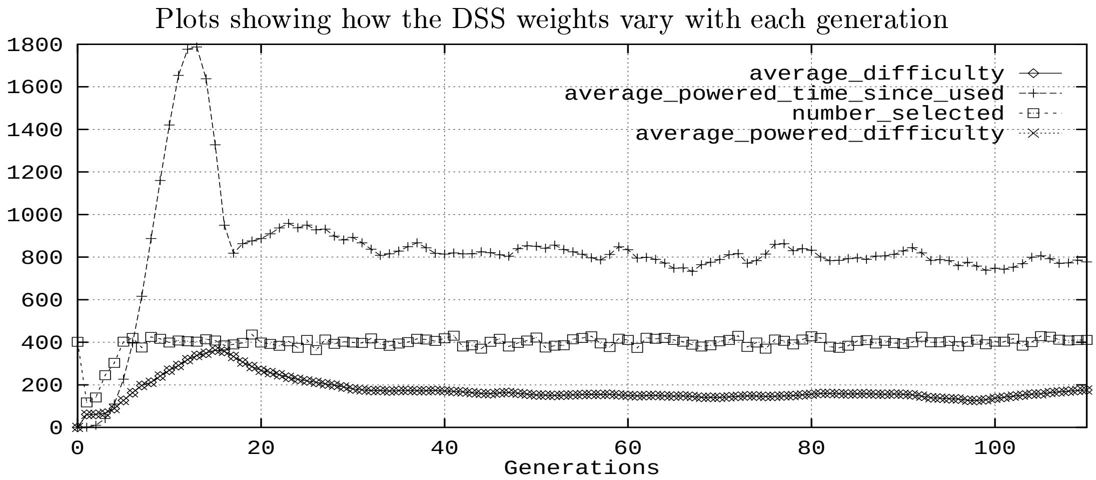


Figure 6.7: Dynamics of DSS: showing the varying difficulty and age weights

Thyroid Training and Test Results							
	Pop.	Subset	Gener-	Avg. Evals	Total	**% correct**	
Algorithm	Size	Size	ations	per Gen.	Evals	Training	Test
GP	10000	3772	n/a	3.8e+07	n/a	n/a	n/a
GP+DSS	10000	400	69	4.0e+06	2.7e+08	99.84	99.27
GP	5000	3772	60	1.9e+07	11.3e+08	99.70	99.00
GP+DSS	5000	400	117	2.0e+06	2.3e+08	99.70	99.00
GP+RSS	5000	400	124	2.0e+06	2.5e+08	99.10	98.40
GP+HSS	5000	545	57	2.0e+06	1.6e+08	99.50	98.70
NN - [Schiffmann *et al.* 92a] - Cascade Correlation						100.00	98.48

Table 6.2: Best results by GP on Thyroid Problem, with best NN results for comparison

as the selection process ensures that all of the cases get selected at least once. The 'average_powered_difficulty' curve shows the average difficulty rating for the cases rising as more and more cases are selected and given a non-zero difficulty rating. This curve can then be seen to drop very slowly over the later generations as the populations evolves to correctly classify more of them. The early dip in the 'number_selected' curve is due to a mistake made when initialising the parameters in early runs which affected the subset selection process in the first few generations. Generally, the number of cases selected can be seen to oscillate close to the subset size of 400. The average age weight can be seen to dominate the average difficulty weight. Cases with above average weights, however, are much more likely to be selected.

GP+DSS seems to perform well with a variety of different DSS parameter settings. The DSS algorithm seems quite robust given that, eventually, all cases will have been selected to participate in several different subsets. If the age weight is too large it can swamp the difficulty weight, and this is perhaps the most likely problem to be experienced with different parameter settings. If the difficulty weight is too large, it will eventually be matched by the age weight, due to the case ages being exponentiated, and the selection process will reach a balance. A variety of different subset sizes all seem to work well.

6.8 A Smaller Problem: TicTacToe Endgames

The TicTacToe problem is smaller and neater than the Thyroid problem. Nevertheless, it is used here to show that DSS can transfer well to other problems.

The data, taken from [Aha 93], consists of the complete set of possible, legal 3x3 board configurations at the end of TicTacToe games (also known as 'Noughts and Crosses'), where player 'x' is assumed to have played first. The target concept is 'win for player x' (i.e. , true when 'x' has one of the 8 possible ways to create a 'three-in-a-row'). There are 958 different board positions (taking into account the board's rotational symmetries), each of which is represented by 9 fields, each of which can take one of three values {1, -1, 0} corresponding to {player 'x', player 'o', blank}. Approximately 65% of the positions are a win for 'x'.

The task for GP is to construct a tree which can correctly classify all possible board positions as to whether or not they are a win for 'x', using the entire set as a training set. This is a variation on the standard method of splitting the cases into training and test sets, but the problem is sufficiently difficult that it still allows a clear comparison between different GP runs, and there was no wish to study the generalisation performance of GP here. The problem has a 'neat' solution, easily constructed by hand, but the resulting tree is fairly large and contains a lot of detail.

The allowed function and terminal sets used in this problem are:

{ AND (both args > 0), OR (either arg > 0), IFGTZ (IF arg is Greater Than Zero)}

{ cornerNW, edgeN, cornerNE, edgeW, centre, edgeE, cornerSW, edgeS, cornerSE }

It can be seen that the function set is not sufficiently powerful to make use of all the possible values of the terminal nodes. The function set cannot distinguish between the two of the possible position values: player 'o' and blank. In effect, the board data has been reduced to indicating whether of not each board position is held by player 'x'. However, even with this reduced level of detail, the problem is still solvable. Other runs tackling the TicTacToe problem are shown in Section 8, looking at the effect of using a much smaller population size, and these runs do have a more extensive function set which can make use of all of the detail available in the board data.

6.9 TicTacToe Results

The following results in Table 6.3 are taken from representative runs. The GP population sizes used here are 1000 and 2000, and the DSS subset size is 200 (out of 958 training cases). The 'difficulty' and 'age' weights are the same as those used in the Thyroid problem. Initial runs indicated that these values produced good results, as did a variety of other values.

It was not possible to get a standard GP run to produce a tree which could classify all of the training cases since the runs were very slow, and always converged to a sub-optimal solution within approximately 100 generations. The GP+DSS runs, with a variety of subset sizes, always achieved close to 100% (approx 95%) if allowed to run for enough generations, and always showed signs of improvement even after many generations.

TicTacToe Training Results						
Algorithm	Pop. Size	Subset Size	Generations	Avg. Evals per Gen.	Total Evals	Training set **% correct**
GP+DSS	1000	200	196	2.0e+05	3.9e+07	100.00
GP+DSS	2000	200	131	4.0e+05	5.2e+07	100.00
GP	1000	958	114	9.6e+05	1.1e+08	90.60
GP	2000	958	94	1.9e+06	1.8e+08	96.20

Table 6.3: Best results by GP on TicTacToe problem

6.10 A quick summary of results from other runs

Different DSS subset sizes were tried on the different problems. As the subset size is reduced, the performance of the GP drops, gradually at first but then rapidly, and seems to mimic that of a much smaller population size. As the subset size is increased towards that of the full training set, the time taken to produce reasonable solutions increases, but, the performance with DSS is still at least as good as that of GP on its own.

Adding in a parsimony factor (i.e. penalising large, 'bushy', trees) speeds up the running of the GP program, since it then uses much less run-time memory to store the whole population of (smaller) trees, and the trees are quicker to evaluate. The standard GP did not seem to perform as well with this restriction as it did without. However, DSS seemed, if anything, to perform better than before. In the TicTacToe problem it was possible to observe the parsimony leading to smaller optimal trees, after the run had discovered its first optimal tree.

Proposed Dynamic Fitness Function (DFF), based on DSS

A fitness function, based on the statistics accumulated during a DSS run was tried. Here, instead of the fitness of a GP tree being the number of training cases it misclassifies, the fitness is instead taken to be the sum of the 'difficulty' ratings of each of the training cases it misclassifies. Again, the difficulty rating of a case refers to the number of GP trees which misclassified the case in the last generation. This fitness function seems to have a somewhat similar effect to DSS in that the GP runs seem to converge more reliably to good solutions, and occasionally produce better solutions. It will need many more runs to try and quantify this, but early indications are that this fitness function works well with both GP on its own and GP+DSS, helping to improve both types of run.

6.11 Smaller Populations over More Generations

An interesting result of using DSS (and DFF) on small populations was noticed over many generations. Large populations (using generational replacement) tend to converge to some best fitness value, and thereafter show no signs of improvement, no matter how many more generations are carried out. On the other hand, smaller populations show a slowly improving best fitness value, even after several thousand generations. The same is not true for smaller populations without DSS. They settle down to a given (often quite bad) best fitness value very quickly.

Table 6.4 contains some indicative runs with different sized small populations. Although they do not achieve as good a peak performance as the large populations, they

Further Thyroid Training and Test Results							
Algorithm	Pop. Size	Subset Size	Generations	Avg. Evals per Gen.	Total Evals	**% correct** Training	Test
GP+DSS	10000	400	69	4.0e+06	2.7e+08	99.84	99.27
GP+DSS	5000	400	117	4.0e+06	2.3e+08	99.70	99.00
GP+DSS	200	400	1711	8.0e+04	1.4e+08	99.34	98.22
GP+DSS+DFF	100	400	1806	4.0e+04	7.3e+07	99.42	98.80
GP+DSS+DFF	100	300	2531	3.3e+04	7.6e+07	99.52	98.98
GP+DSS+DFF	50	400	3870	2.3e+04	7.7e+07	99.47	98.86

Table 6.4: Further Thyroid Training and Test Results

get reasonably close, still using fewer tree evaluations, and using much less computer memory. Using less memory has a knock-on effect with the efficiency of CPU-usage, and in fact increases the speed of tree evaluation. These runs were still (very) slowly improving, but were interrupted when user patience ran out, or a re-boot was scheduled.

The use of small populations is explored further in Chapter 8.

6.12 DSS Discussion

GP, DSS, HSS, RSS, and NNs The GP + DSS method produces results as good as those of the standard GP and in a much shorter time, on the Thyroid Problem at least. DSS can actually produce better answers, as can be seen with the TicTacToe problem, and the population appears to produce a larger variety of solutions in later generations than with standard GP or HSS. The random nature of DSS appears to assist the basic GP algorithm.

HSS out-performed the standard GP in terms of processing time, and nearly matched it in terms of quality of results. HSS was the main contender for *improvement-of-the-week* until DSS was implemented. One big benefit of HSS is the ease with which previous standard GP runs can be cannibalised for information to use in selecting a subset of difficult cases.

RSS performs surprisingly well, and can match the performance of standard GP in certain situations, in a much shorter time. This perhaps indicates one of the benefits of DSS that, in effect, the fitness function is continually being changed, never allowing the GP to settle into a rut.

When compared with the Neural Network results in [Schiffmann *et al.* 92a], the best of which is shown in Table 6.2 above, GP+DSS produced a tree which generalised better from the training set. To be fair, in splitting up the problem into two phases (class 3 or not, then class 1 or 2), the GP has been presented with an easier problem than was presented to the Neural Networks. This could be taken in different ways: splitting up the problem is mildly cheating, or demonstrating the flexibility of the GP approach.

Thyroid Problem For the Thyroid problem, the distribution of errors made by the best tree was split more or less evenly between problem cases (classes 1 and 2) and no-problem cases (class 3). This could be altered by biasing the GP algorithm to erring on the side of problem cases, i.e. more False-Positive errors and fewer False-Negative errors, which would be more useful in a medical environment.

Looking at the trees produced, it was interesting how the best tree used only 13 out of the 21 variables available to classify most of the cases correctly. This could perhaps lead to some useful savings in data collection costs, or it could help focus attention on some key measurements. It might be possible to make some further measurements and split each key measurement into several different, finer measurements. One advantage of GP over NNs is that it is very difficult to obtain such insights from the node weights in a trained NN.

DSS DSS does not seem too sensitive to the choice of subset size, and 'difficulty' and 'age' weights. The ones chosen for the Thyroid Problem carried over successfully to the TicTacToe problem. It is possible to pick bad values, but it seemed just as easy to pick useful ones. A reasonable guess so far (albeit one which needs to be checked on many more and different problems) seems to be a subset size around a fifth to a tenth of the full training set size, with the weights chosen to allow a difficult case and a case five to fifteen generations old to have a roughly equivalent weighting.

There are obviously many factors affecting the optimum choice of these parameters. It appears that a large training set, containing some degree of redundancy, with a core of difficult cases would benefit the most from DSS. However GP, in particular in the TicTacToe problem, seems to suffer from an inability to reach an optimal solution. This could be due to many things, but, applying DSS enables GP to correctly classify all of the training set. Difficult cases are persistently dragged into the subset until the GP population evolves to be able to deal with them. Standard GP does not differentiate between easy and hard cases, and this lack of pressure becomes noticeable near the end of a run when the population fails to find the optimal solution. DSS appears to epitomise this idea of a dynamic fitness function increasing the pressure to solve difficult cases.

At this early stage of investigation, there are strong hints that the method is more widely applicable to general problem solving with GP and GA involving large training sets (for time saving), and to difficult problems (for better and more general answers). What is more, DSS is easily added to the basic GP algorithm. The performance of DSS on the smaller, less messy, TicTacToe problem bodes well for DSS to be applied to many other supervised training problems. Possibly one of the more useful aspects of DSS so far has been its ability to produce results quickly which, for GP, means that different function sets and parameter settings can be experimented with.

DFF DFF is a logical progression from DSS itself, and in many ways has an equivalent effect on the fitness function in supervised learning with a training set. DFF and DSS provide a simple feedback mechanism for focusing a GP population onto its own deficiencies.

Smaller Populations It is interesting that the DSS method which allows GP to be used on large problems in a practical time, also allows GP to be scaled down for use on smaller machines where CPU memory and its usage is more constraining than CPU speed.

Further Research There are myriad lines of investigation to follow up. For instance, how widely applicable is DSS to other problems? How does DSS's randomness influence the behaviour of GP? Would DSS work as well if it was only based on an individual tree's measure of difficulty, e.g. the performance of the best-of-generation tree, or does it need the combined measures from the whole population? Could DSS be applied to other supervised training algorithms, e.g. Neural Networks, where the training cases are continually re-assessed until correctly classified? Could DSS be applied to constraint solving problems? How sensitive is DSS to its parameter settings?

Chapter 7

Limited Error Fitness

This chapter presents Limited Error Fitness (LEF), described in Section 7.1, and first published in [Gathercole & Ross 97b]. LEF is a variation on the standard fitness function for GP on supervised classification problems. LEF enables a simple GP, described in Section 7.2, to solve the previously out of reach Boolean Even N Parity problem for $N > 5$, described in Section 7.3. The test results from runs with N=6 and N=7 are given in Section 7.4, followed by a discussion in Section 7.5.

The Boolean Even N parity problem (finding the parity of N boolean inputs) is a hard one for GP to solve; increasing rapidly in difficulty and solution size with increasing N. Koza has shown that N=5 represents, in effect, an upper limit for standard GP, even with a large population size of 8000. Runs tend to converge rapidly on sub-optimal solutions. Only with the use of Automatically Defined Functions (ADF), a more powerful representation, was Koza able to solve for N=6 and higher, with a large population of 4000, [Koza 92, Koza 94].

> "... the parity functions are the hardest Boolean functions to find via blind random search of the space of S-expressions using the function set F and they are the hardest to learn via genetic programming."

With LEF, standard GP without ADF can readily solve for N=6 and N=7 with a population size of 400, but may require several thousand generations. A smaller population size allows GP to be run on smaller computers at a reasonable speed, in a reasonable length of time. It has the potential to solve for even higher N with larger populations.

LEF is variation on the standard GP fitness function for classification problems. An individual's fitness score is based on how many cases remain uncovered in the ordered training set after the individual exceeds an error limit. The training set order and the error limit are both altered dynamically in response to the performance of the fittest individual in the previous generation.

Evidence indicates that LEF rewards generality, penalises specialists, and maintains diversity in the GP population, preventing premature convergence. After many thousands of generations, if it has not yet found an optimal solution, LEF keeps the GP population in flux. Thus GP is a more effective optimiser, continually emphasising the relative importance of difficult cases, and de-emphasising easy cases. However, LEF is very susceptible to the choice of various parameter values, and often causes the GP population to undergo a catastrophic loss of good individuals. LEF is also used successfully on the TicTacToe problem in Chapter 8. However LEF still hasn't yet been tried on enough problems to identify other potential weaknesses such as over-fitting on the training set.

7.1 LEF - the algorithm

LEF is a variation on the standard method used to evaluate the fitness of a GP individual in supervised learning on classification problems. In effect, it presents a different version of the same problem to each generation of the population, based on how well the population performed on the previous version. The standard method evaluates the GP individual on each case in the set of training cases, compares its 'answer' (or classification) with the correct answer, and the GP individual's fitness score is based on the total number of errors.

With LEF, a GP individual's fitness score is related to how many of the ordered set of training cases it classifies correctly before it makes a certain number of misclassifications. After exceeding the error limit, any cases not yet covered by the individual are counted as misclassified. The fitness score is the total number of misclassified cases. If the GP individual is a poor one, i.e. makes many mistakes, it will not be evaluated on the entire training set. If the GP individual is a good one, it will be evaluated on the entire training set, making fewer mistakes than the number allowed. Thus, in general, it is quicker to find the fitness value for a poor GP individual than a good GP individual, saving CPU time.

At the start of a run, the training set is shuffled into a random order to avoid any biases that may have been introduced in the original ordering. The error limit is set in advance of the first generation, possibly with the benefit of information gleaned from previous runs. The first generation could, however, run without an error limit, and the error limit be set equal to the number of errors made by the best GP individual in the first generation. Later on, the error limit is raised, lowered, or left unchanged, and the training set is re-ordered, depending on the performance of the best GP individual in the preceding generation. The timing of these changes depends on two measures from the best of generation individual (BOGI), and some parameters set at the start of the run. The two measures are the number of cases not covered by the BOGI (because it exceeded the error limit before reaching the end of the training set), and the number of generations since the last improvement in the BOGI (ignoring generations when it got worse). In this instance, the term 'improvement' is taken to mean that the BOGI made fewer errors. With parsimony included in the fitness function, i.e. a penalty for large trees, the BOGI often gets smaller, with a corresponding small decrease in its fitness score, but remains functionally unchanged.

The algorithm for modifying the error limit is as follows:

- BOGI improvement:
  IF the BOGI has improved within the last O-PAUSE generations
  THEN make no parameter changes
- Over-Coverage:
  IF the BOGI makes fewer errors than the error limit
  AND the BOGI hasn't improved for O-PAUSE generations
  THEN
  - reduce the error limit by O-DECREMENT
  - move the O-BUBBLES easiest training cases
    to the end of the ordered training set

- Exact-Coverage:
  IF the BOGI reaches the error limit, but covers all the cases
  AND the BOGI hasn't improved for E-PAUSE generations
  THEN
  - reduce the error limit by E-DECREMENT
  - move the E-BUBBLES easiest training cases
    to the end of the ordered training set
- Under-Coverage:
  IF the BOGI exceeds the error limit before covering all the cases
  AND the BOGI hasn't improved for U-PAUSE generations
  THEN
  - increase the error limit by U-INCREMENT
  - move the U-BUBBLES easiest training cases
    to the end of the ordered training set

These four phases cover all the possibilities for the interaction of the error limit and the performance of the BOGI. Over-Coverage corresponds to the BOGI making fewer errors than the error limit, indicating that the problem could be made harder by reducing the error limit. Under-Coverage corresponds to the BOGI making more errors than the error limit, so none of the population can cover the entire training set. Raising the error limit would give the population a better chance of covering the entire set. The Exact-Coverage phase has been made explicit, even though it could have been incorporated into the other two phases. This phase is quite crucial in LEF; it is when the BOGI only just covers the training set. Reducing the error limit at all will immediately reduce the fitness of the BOGI, possibly by a large amount, allowing other, previously less fit, individuals to the fore.

There are several parameters set at the start of a run. Some typical values are as follows:

- initial error limit - set to allow the first generation BOGI to nearly cover the entire training set. This obviously depends on the training set, and differs for each problem. Experiments have indicated that it is better to start too large than too small, though the algorithm allows it to rise if it is set too low. However in the LEF run for N=6, shown below, the error limit was set quite low at 20 (there are 64 training cases, and a random solution is likely to achieve approximately 32 errors), to demonstrate how the LEF algorithm copes with a BOGI that can't cover the entire training set before exceeding the error limit.

- O-PAUSE - set to 5

- E-PAUSE, U-PAUSE, - set to 15

  These delays can be varied somewhat but experiments have indicated that if E-PAUSE and U-PAUSE are too small, the population doesn't have time to adapt to the new version of the problem, and so doesn't improve very quickly if at all. If U-PAUSE is too long, the population converges too much on the new version of the problem, and loses the diversity needed to solve the earlier versions. In effect, it has to re-learn how later. This loss of diversity is more noticeable with smaller populations, and is usually catastrophic, setting back the population by many generations. O-PAUSE seems less important since it only has an effect when the BOGI makes fewer errors than the error limit, and there are likely to be several other individuals making few errors. Reducing the error limit at this stage still keeps the current BOGI in place, but speeds up the evaluation of the majority of the population.

- O-DECREMENT, E-DECREMENT, - set to 1

  These changes to the error limit are kept small. If they are too large, the change in difficulty of the problem becomes too extreme, the population fails to overcome the change, and bad GP individuals can suddenly become the best of the generation.

- U-INCREMENT - set to 1

  This parameter can be made larger to help counter the effect of the catastrophic loss of good individuals in the population, by allowing a faster increase in the error limit.

- O-BUBBLES, E-BUBBLES, U-BUBBLES, - set to 1

> The change in order of the training set is kept small. The problem is made slightly more difficult, but previous good GP individuals should still perform well. If it is too drastic, as with changes in the error limit, it is detrimental to the development of the population. The 'BUBBLES' refer to one pass of a bubble sort algorithm. Starting with the first case in the set, and moving along the order towards the last case, pairs of cases are swapped if the later one has been misclassified (or left uncovered) more often. This has the effect of moving the easiest case (i.e. the one that was misclassified least often by the previous generation) to the end of the ordered set, and moving the harder cases one place towards the start of the ordered set. There are many other ways of changing the order of the training set, but this is one of the simplest, and appears to have a reasonably good effect. One 'bubble' only reduces the BOGI fitness by at most one, even though, potentially, a case could be moved from the start of the set order all the way to the end. Any difficult cases are only moved towards the start of the set by one position. Changing the error limit can have a much bigger impact on the BOGI fitness.

In essence, these parameters control the change in difficulty of the problem in response to the performance of the population in the previous generation. Many experiments have indicated that it is better to minimise the impact of the changes to error limit and set order, especially E-DECREMENT and E-BUBBLES. The population is given time to adapt to the new version of the problem. If it proves too difficult, the problem is made slightly easier. If it proves too easy, the problem is made slightly harder. The ultimate aim is to reduce the error limit to zero, i.e. for the BOGI to make no errors.

Related Work Closely related to LEF is the idea of co-evolving host and parasite populations, [Hillis 90, Rosin & Belew 95], niches, [Goldberg & Richardson 87, Greene & Smith 93], competitive fitness functions, [Angeline & Pollack 93], (These are described more fully in Section 5.3), and training subset selection, (Chapter 6).

7.2 GP Details

The GP setup is kept simple, and Automatically Defined Functions (ADF) are not used. Generational replacement with elitism is used, with panmitic tournament selection of size 4, using population sizes from 100 to 800. The operators (and selection probabilities) are:

- 40% CROSSOVER AT ANY POINT
  - crossover between two parents producing one child
- 20% MUTATE SUBTREE
  - mutate a subtree in a parent to produce a child
- 20% MUTATE BY SUBTREE PROMOTION
  - replace a subtree in a parent by one of its own subtrees to produce a child
- 20% MUTATE ANY NODE
  - replace a random node in a parent with another node of the same arity

The function and terminal sets are described below, in Section 7.3.

An individual's fitness is based upon the number of classification errors it makes (i.e. the fewer the better) and, for LEF, the number of training cases left uncovered after it exceeds the error limit (i.e. also the fewer the better). Parsimony, a penalty for large trees, is added to the fitness score as a factor 0.001 times the number of nodes in the tree. Since the maximum allowed tree size is 999 nodes, the contribution from parsimony never reaches 1.0, and so differentiates only between trees which perform equally well on the training set. A smaller fitness score corresponds to a fitter tree, with a minimum (of less than 1.0) equal to the parsimony factor of a tree which makes no errors on the training set.

The basic GP settings can certainly be improved. In particular, the tournament size seems to be too large. Some studies [Blickle & Thiele 95, Hancock 94] and several discussions with GA practitioners seem to indicate that smaller tournament sizes work better. The choice of operators is also important. They can always be improved, and care should be taken so that they do not impede GP, (see Section 4.1), though the ones used here prove reasonably successful. The main aim of this section is to show the possibly beneficial impact of LEF on a standard GP, not to optimise GP parameters.

7.3 The Even N Parity problem

The Even N Parity problem has been used by Koza as a problem which causes difficulties for GP in [Koza 92]:

> "The parity family of functions is a very difficult family of functions to learn. For example, after trying 20 runs of genetic programming without automatic function definition, no solution was found for the even-5-parity problem using a population size of 4000 and the given function set F (although we did find one solution on our eighth run after we increased the population size to 8000). However, if automatic function definition is used, solutions to both the even-5-parity and the even-6-parity functions can be readily found with a population size of 4000."

The training set consists of all the 2^N possible combinations of N binary inputs (64 for N=6, and 128 for N=7). The correct classification is the parity of the N inputs, i.e. TRUE where an even number of inputs are TRUE, and FALSE where an odd number of the inputs are TRUE. The parity changes with any change in a single input value. The task for GP is to find a tree which correctly classifies all the cases in the training set using the following function and terminal sets:

- Terminal Set - { $b_0, b_1, b_2, ..., b_{N-1}$ },
  N boolean variables
- Function Set F - { AND, OR, NAND, NOR },
  standard logical functions, computationally complete.

As N is increased, the problem becomes exponentially harder for a simple GP and, for N=6 or greater, supposedly impossible (or exceedingly unlikely to be solved) even with a very large population size of 8000. At this point, Koza then demonstrates the power and might of Automatically Defined Functions (ADF) which he uses to successfully solve the Even N Parity problem up to N=6, [Koza 92], and up to N=11, [Koza 94], but using large population sizes of 4000, taking roughly 20 generations for N=6. ADF is a more powerful representation, particularly suited to the structure inherent in the Boolean Even N Parity problem, allowing GP to construct hierarchical function definitions. LEF used in combination with a simple GP, without ADF, successfully reaches the dizzy heights of N=7, with small population sizes ranging from 100 to 800, and has the potential to solve for larger N. Population size has a major impact on the speed of GP, and especially on the run-time memory requirements. It can easily exceed the usable memory generally available in present-day workstations, causing them to run very inefficiently.

7.4 Results

A series of runs (of the order of 50) were carried out with an assortment of population sizes and parameter settings, though no runs used ADF. Since each run took several hours, especially all the runs without LEF, there are not sufficient runs to provide sound performance statistics. However, some clear trends do emerge. The results from runs on the Even N Parity problem are as summarised in Table 7.1.

The results confirm that simple GP is incapable of solving the Even N Parity problem for N=6 or N=7 (or greater) with an assortment of population sizes ranging from 100 to 800, allowed to run for 4000 generations for N=6, and 8000 generations for N=7. Letting GP run on even longer would almost certainly not result in optimal trees being discovered since the runs showed no signs of improvement.

Graphs from two sample runs for N=6, with a population size of 400, are shown in Figures 7.1 to 7.8. The graphs are from a typical, successful run of GP with LEF, showing the changes in BOGI fitness, Figure 7.2, and tree size (or 'bushiness'), Figure 7.4, the population fitness standard deviation, Figure 7.6. These graphs are shown alongside the equivalent graphs from an unsuccessful run of GP without LEF (they all failed to find an optimal tree). The next two graphs show the error limit, Figure 7.7, and the number of tree evaluations per generation, Figure 7.8, for the run with LEF. The runs

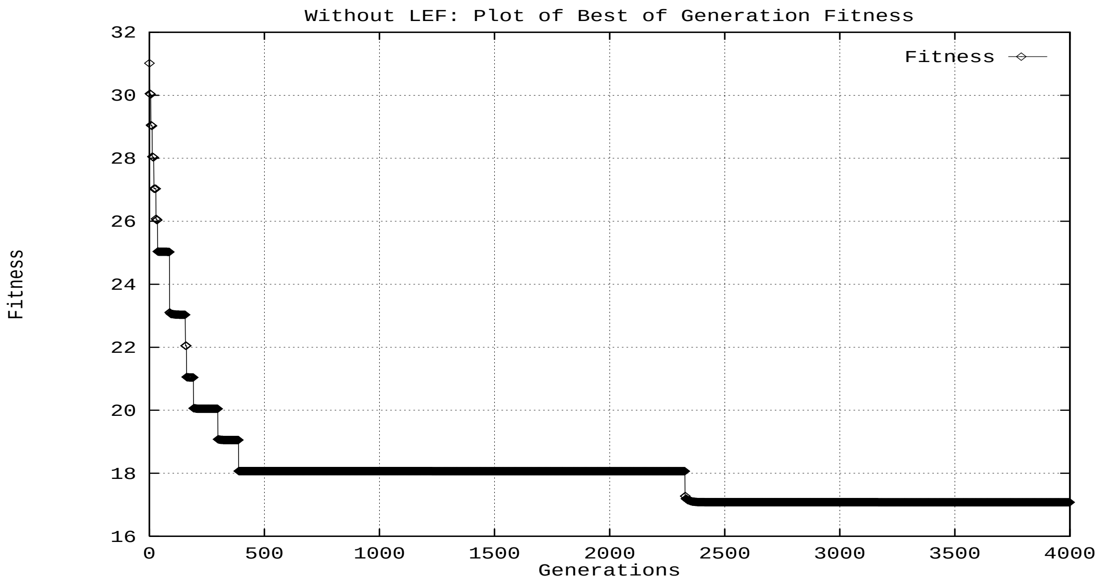


Figure 7.1: Best of Generation Fitness during a typical run of GP without LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

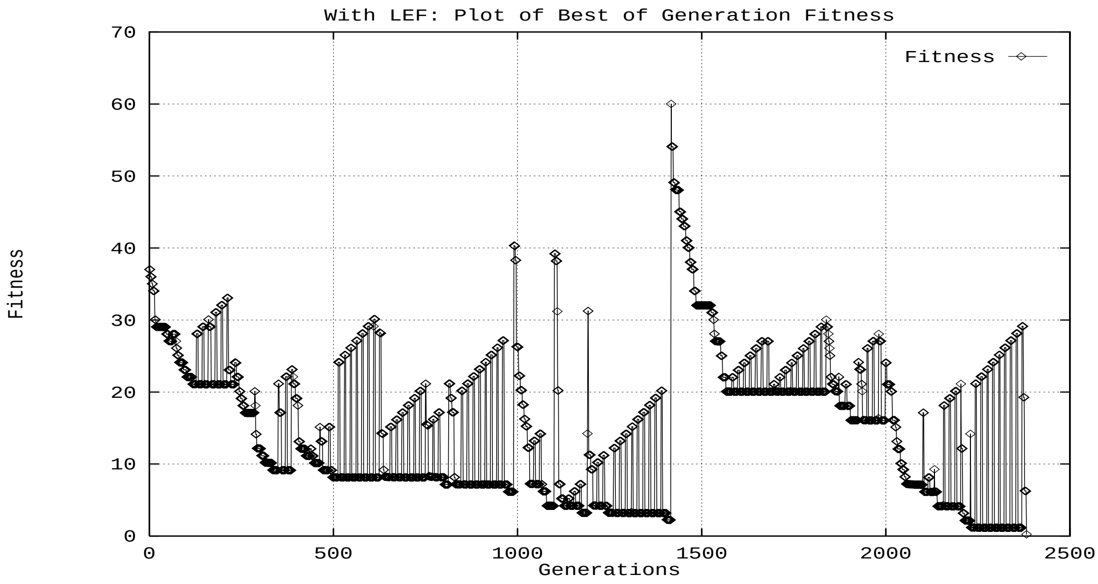


Figure 7.2: Best of Generation Fitness during a typical run of GP with LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

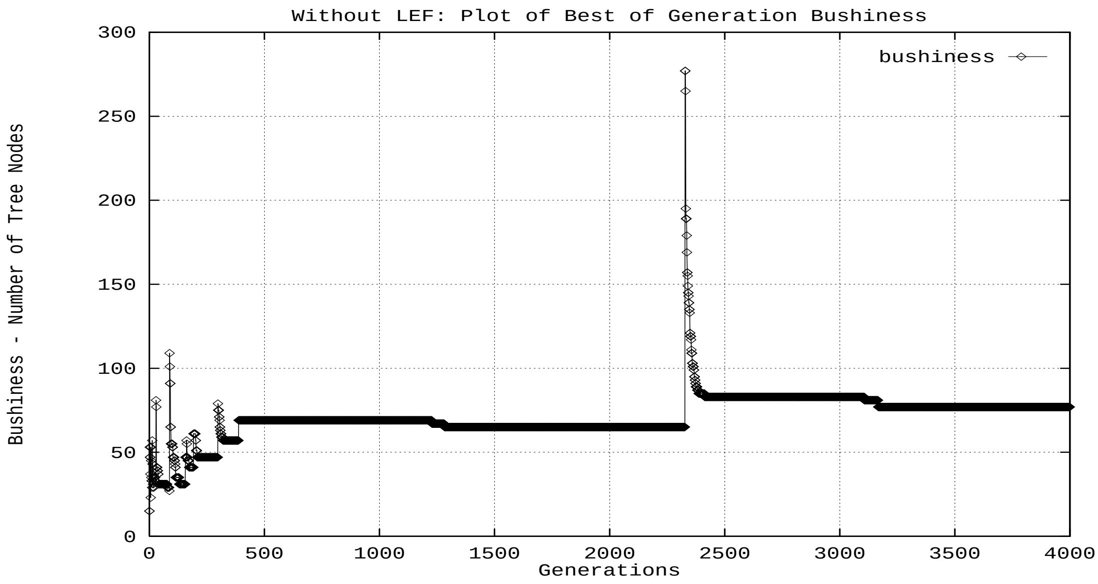


Figure 7.3: Best of Generation Bushiness during a typical run of GP without LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

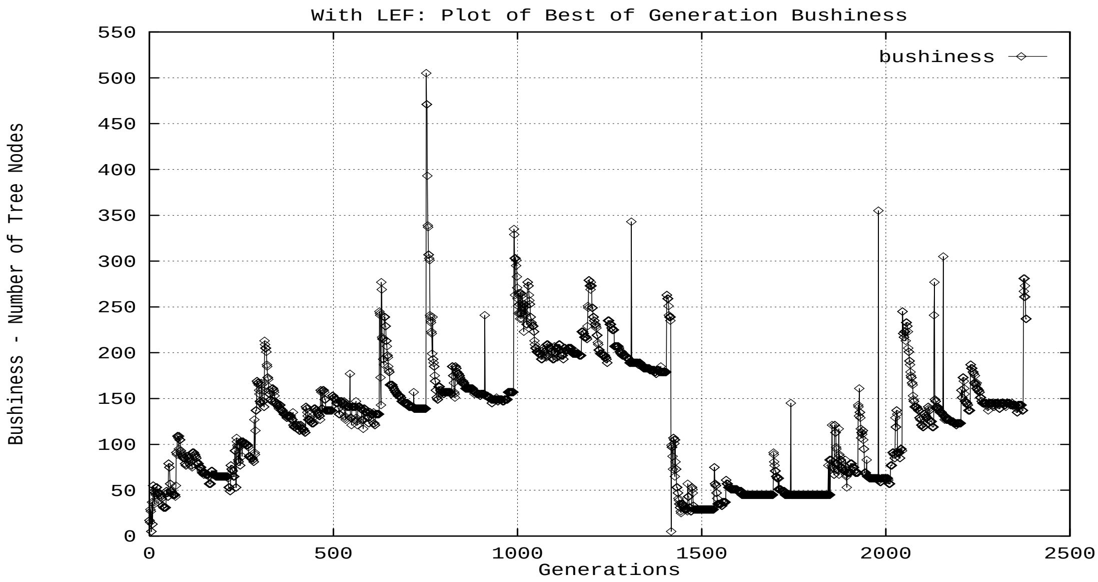


Figure 7.4: Best of Generation Bushiness during a typical run of GP with LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

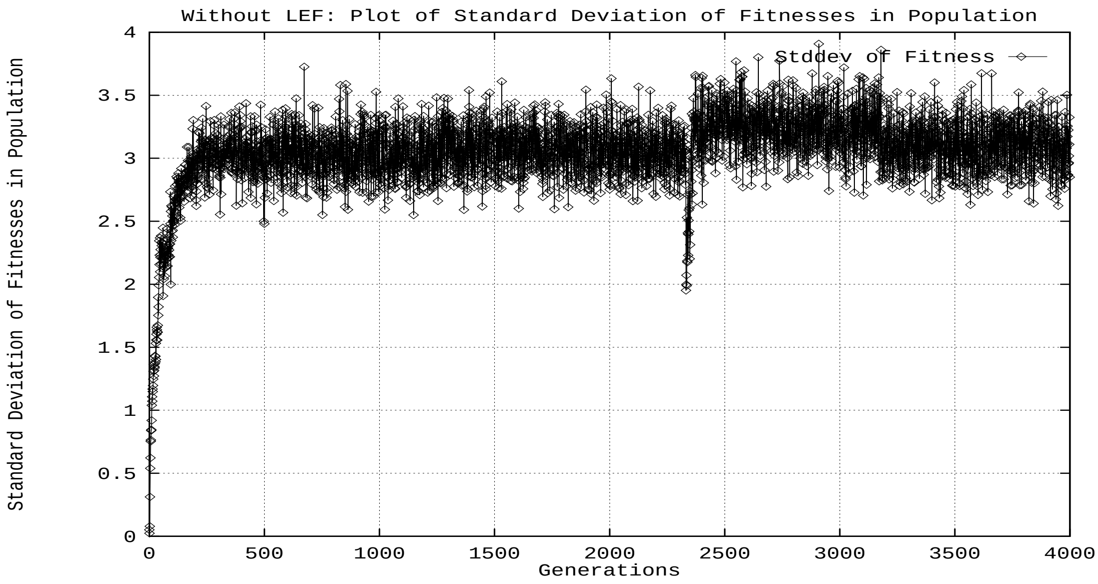


Figure 7.5: Standard Deviation of Fitness during a typical run of GP without LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

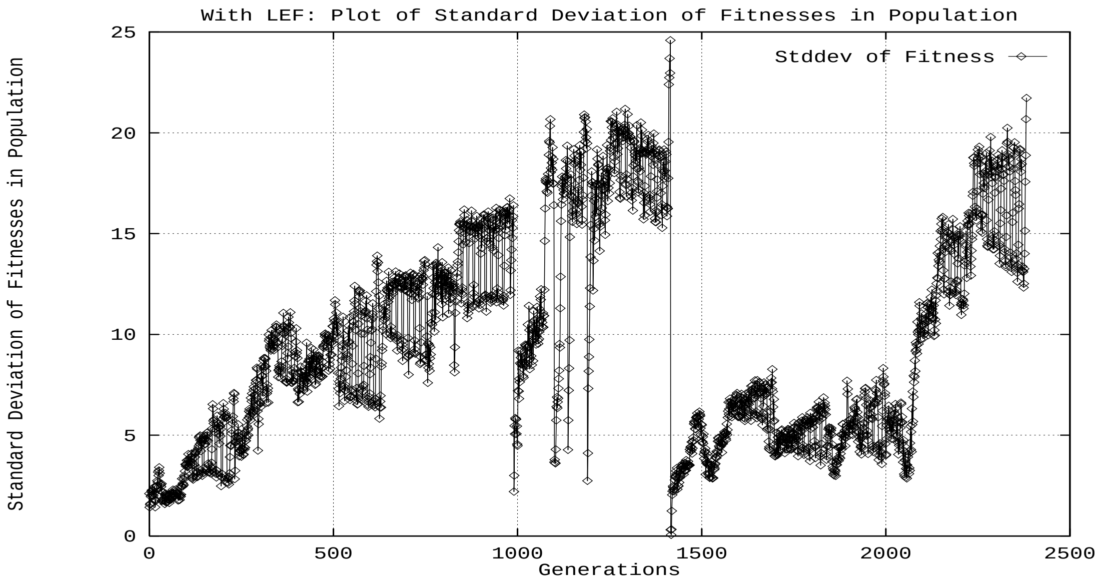


Figure 7.6: Standard Deviation of Fitness during a typical run of GP with LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

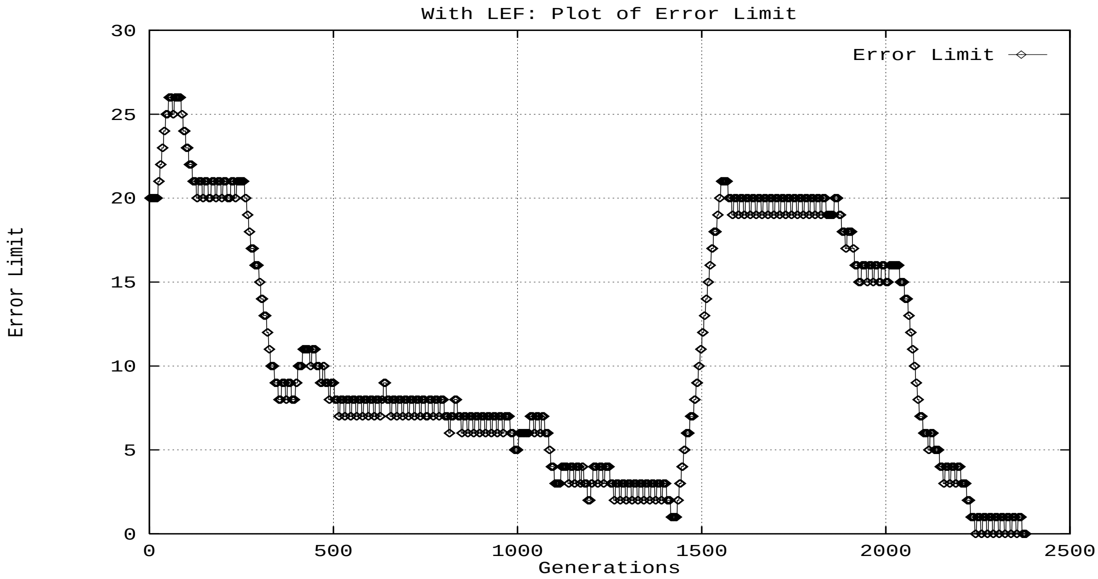


Figure 7.7: Error Limit during a typical run of GP with LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$

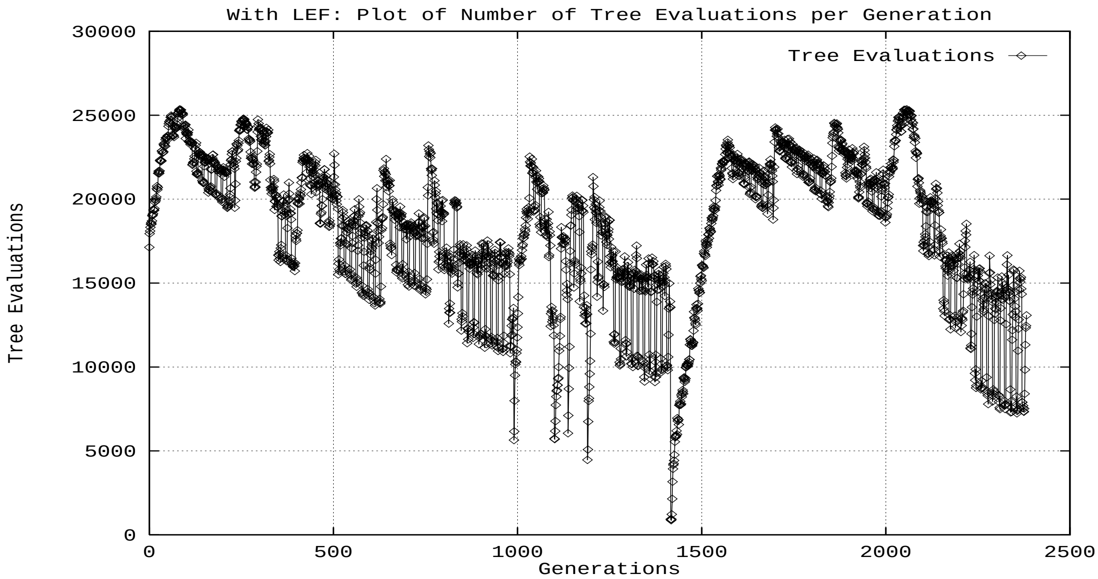


Figure 7.8: Evals per Generation during a typical run of GP with LEF on the Even N Parity Problem, where N=6, and $PopulationSize = 400$. Without LEF, GP would need 25600 evaluations per generation.

Summary of results from runs on the Even N Parity problem			
N	Standard GP	GP with LEF	GP with ADF
$N < 5$	Solvable	Solvable	Solvable
$N = 5$	Koza solves once with Pop Size = 8000	Solved here repeatedly with small Pop Sizes, e.g. 400	Koza solves easily with Pop Size = 4000
$N = 6$	Not Solvable (11 runs)	Solved here repeatedly with small Pop Sizes e.g. 400 and upto 4000 generations	Koza solves easily with Pop Size = 4000
$N = 7$	Not Solvable (7 runs)	Solved here less often with small Pop Sizes e.g. 600 and upto 8000 generations (5 of 10 runs, with Pop=600)	Koza solves easily with Pop Size = 4000
$N = 8$	Not Solvable	Comes close, but needs larger Population and more generations	Koza solves easily
upto $N = 11$	Not Solvable	Not Solved Yet	Koza solves with lots of computing power

Table 7.1: Summary of results from runs on the Even N Parity problem

without LEF do a constant number of evaluations per generation, i.e. in the sample run shown here, $PopulationSize * NumberOfTrainingCases = 400 * 64 = 25600$. The LEF run took 2500 generations before an optimal tree was found. Care should be taken when comparing between graphs with LEF and without LEF, since the scales are not same. However, it is clear that with LEF, the BOGI fitness and bushiness fluctuate a great deal more, and the standard deviation is much higher than without LEF. There is a catastrophic deterioration in the fitness of GP with LEF near the 1400th generation, seen in Figure 7.2, a common feature of runs with LEF. This is followed by a rapid rise in the error limit, seen in Figure 7.7, and the fitness soon falls to reach zero errors less than 1000 generations later. The run of GP with LEF appears to have restarted nearly from scratch, but typically recovers quickly.

In a typical run for GP without LEF on N=6, as shown in Figure 7.1, with 64 cases in the training set, the BOGI would reach a floor of, say, 16 errors over the course of several hundred generations, and would then fail to improve beyond that level. Very occasionally, when allowed to run on for many thousand generations, a fortuitous mutation might bring about a small improvement. The best error total achieved by an individual from GP without LEF for N=6 was 8 errors, and the worst was 20 errors. These results are taken from 11 runs. At no stage, in any run, did the GP without LEF look capable of achieving zero errors. The standard deviations of each population's fitnesses were low, indicating that they consisted mostly of functionally similar variations of copies of the BOGI.

In a typical run for GP without LEF on N=7, with 128 cases in the training set, the BOGI would reach a floor of around 36 errors after 1500 generations and would not improve beyond that level. The best error total achieved was 19 errors, and the worst was 44 errors. These results are taken from 6 runs. The best trees found were all small.

For GP with LEF for both N=6, shown in Figure 7.2, and N=7, the GP populations showed signs of continual change and/or improvement, even after many thousands of generations. In a typical, successful run, the BOGI error total would improve over the course of 1000 generations, between some large fluctuations. If the fluctuation is severe enough, the BOGI starts again from a high error total. Often, as can be seen in Figure 7.2, the recovery from this loss of BOGI fitness is faster than the initial improvement in BOGI fitness before the fluctuation, even when the BOGI tree becomes very small and trivial indeed, as can be seen in Figure 7.4 around generation 1400. Occasionally, the improvement would bottom out somewhere between 1 and 5 errors (for N=6), and 1 and 10 errors (for N=7). However, after several hundred more generations, with a great deal of fluctuation in the BOGI error total, it would improve to reach zero.

The graph of the error limit, in Figure 7.7, shows several occasions when the error limit oscillates, e.g. between generation 200 and 300, and between generation 1600 and 1800. This corresponds to the BOGI matching the error limit, not improving for several generations, the error limit is lowered, and there is still no improvement in the BOGI fitness for several more generations, so the error limit is raised again, and so on. These fluctuations show the population is repeatedly forced by LEF to cope with the difficult cases. The oscillations of the error limit can be seen to coincide with the rising diagonals in the BOGI fitness, in Figure 7.2, just after generation 500, for example. The error limit is lowered, the training set order is bubbled, bringing the difficult cases one position nearer the start of the order, causing the fitness of the BOGI to decrease by one.

The addition of LEF keeps the population in flux, not allowing it to settle down. Often the fluctuations in BOGI fitness are quite extreme, but the population usually recovers to improve upon its previous best. Unsuccessful runs always showed signs of improvement as the generation limit was reached, and looked likely to eventually find an optimal tree if given more time.

The plot of the BOGI bushiness for GP without LEF, in Figure 7.3, shows the 'classic' spike followed by a decay accompanying nearly every major improvement in BOGI fitness when parsimony is being used. It seems to correspond to Crossover producing a much larger, fitter tree by combining two existing fit trees. The large BOGI tree is then gradually reduced in size over the following generations through the actions of the Mutation operators, whilst still maintaining its functionality. In the runs of GP with LEF, the bushiness spikes are much more frequent, but the decay curves are disrupted by the continual fluctuations.

These results were taken from runs with the parameters set as described above in

Section 7.2. Many other runs with different parameter settings produced similar results. Only with LEF was GP able to find solutions for N=6 and N=7. GP with LEF was able to find optimal solutions for N=6 and N=7 with population sizes ranging from 100 to 800 although, perhaps understandably, not very often with a population of size 100 for N=7. But, with a population size of 600 allowed 8000 generations, GP with LEF was successful five times out of ten runs, and had a higher success rate with a population size of 800.

7.5 LEF Discussion

A comparison of the graphs showing GP with and without LEF shows how LEF changes the dynamics of a GP run; the population is never allowed to settle. The standard deviation of fitnesses remains much higher than for GP without LEF for the entire run. As long as the BOGI improves over the generations, the population is left alone. However, once the BOGI stops improving, the fitness function is adjusted by altering the error limit and re-ordering the training set. If the error limit is raised (because the BOGI exceeds the error limit), a wider variety of individuals in the next generation will have similarly good fitness scores, which increases the possibility that a new tree will be produced which can 'break out' of the current sub-optimal situation. If the error limit is lowered or the training set re-ordered (because the BOGI comes under the error limit), fewer individuals in the next generation will have good scores, and these will feature more strongly in the next round of breeding.

The time taken to evaluate each generation increases as the fitness function is made easier, and decreases as the fitness function is made harder. On average, with LEF, every generation involves between a half and two thirds of the number of individual evaluations needed by GP without LEF. It means little unless the overall times to solve the problems are compared but, in the problems described here, only GP with LEF was able to find optimal solutions.

The lowering of the error limit and the re-ordering of the training set (combined with parsimony which favours smaller trees) help nudge the population towards increasing generality. Specialists in the population which have a good score sensitive to the ordering will find their fitnesses reduced as the easy cases are bubbled to the end of the set order, and the harder cases moved nearer the start of the set order. The gradual change of the fitness function provides more opportunities for an individual to be improved in small increments, allowing the mutation operators to work more effectively. With a low error limit, LEF provides a quick, approximate measure of an individual's performance on the entire training set. It discriminates differently between individuals from GP without LEF.

LEF can be seen as a form of fitness 'sharing' [Goldberg & Richardson 87], where the fitnesses of individuals in a 'niche' are reduced according to the number of individuals in the niche. Here, a niche is taken to mean functional equivalence, where two or more individuals correctly classify the same individuals in the training set. As the set is reordered, easy cases (based on the performance of the previous generation) are moved to the end of the set, in effect penalising the currently successful individuals, allowing other, functionally different, individuals to prosper.

With LEF, if the population size is too small or the LEF parameters are too extreme, the disruption caused by lowering the error limit and bubbling the training set order is often catastrophic. It usually occurs after the Exact-Coverage phase of the LEF algorithm, when the error limit is lowered and the training set is bubbled. The BOGI in the next generation has a substantially worse fitness score than in the previous generation. When the error limit is raised again several generations later, after the BOGI fails to improve, the individual(s) which exhibited the previous best fitness has often disappeared from the population, so GP is forced to re-discover something it had already produced. This suggests that with LEF the population acts as a reservoir of previous good solutions, in effect allowing GP to backtrack. The population needs to be large enough to ensure that solutions rendered temporarily worse by a change in the fitness function survive long enough to contribute at a later stage if the fitness function is made easier again.

The explicit preservation of previously fit individuals (perhaps the BOGIs from the last few generations) would seem a sensible modification to GP with LEF, avoiding the catastrophic losses of good individuals, allowing a faster recovery from a reduced and then increased error limit. However, preliminary results using this idea seem to indicate that LEF does not benefit from the preservation of previously fit individuals (with the exception of simple elitism). In fact, LEF actually performed worse in every run where fit individuals from earlier generations were re-introduced into the population. It would appear that the catastrophic disruptions of the population fitnesses might be an essential part of the way LEF works. The changing fitness function penalises previously fit individuals whenever a stumbling-block is reached where the BOGI stops improving, leaving GP to discover new solutions. Re-introducing previously fit solutions would prevent this discovery of new solutions, leaving GP stuck at local optima.

GP runs (without LEF) very often converge prematurely [Ryan 94] and, once they reach such a stasis, are extremely unlikely ever to break out of it and produce fitter individuals. Even when the population contains individuals which can correctly classify most of the training set, it is unlikely to produce individuals in later generations which can correctly classify the entire set. With LEF, on the other hand, the population is continually being moved away from such a stasis and prevented from converging. A similar effect is reported in [Hillis 90] for co-evolving host/parasite populations. With LEF, this often incurs a catastrophic reduction to a low-fitness population; in effect, reinitialising the population. But, every time this occurs, the population rapidly improves to at least match its previous best BOGI in fewer generations than it took to reach it the first time.

A simple GP is not suited to optimisation. With LEF, the few cases not yet consistently classified correctly are continually being brought to the start of the set ordering, forcing the population to deal with the cases it finds 'difficult' by penalising individuals which fail on the 'difficult' cases.

GP runs are frequently limited by computer memory capacity, with an upper limit on population size. Parsimony provides pressure to select for smaller trees. LEF allows much smaller population sizes to be used effectively, albeit requiring many more generations. In fact, GP with LEF seems to benefit more from an increase in population size than does GP without LEF, and could well benefit from a much lower selection pressure than provided by the tournament size used here. It would be worth seeing the effect of LEF on GP with unlimited tree sizes, i.e. no parsimony. Further experiments are needed to clarify this.

LEF is another example of how the training set can be used in different ways to benefit GP and other GA-related algorithms (see Chapter 6). Simply evaluating each individual in each generation on the entire training set ignores the current abilities of the population, which change with each generation, and ignores information gained about the difficulty of cases in the training set. Breaking up the problem into sub-problems, and presenting the sub-problems to be solved in turn, can 'lead' the population to solve the entire problem. Several runs produced optimal trees after only a few hundred generations (as opposed to thousands of generations, on average) with a steadily improving BOGI fitness, suggesting that, in those instances, the route to optimality had somehow been made much easier.

As the easiest case is bubbled to the end of the set order, the hard cases move nearer the start of the order by one place. If the BOGI is only making a few errors, then its fitness is highly dependent on the location of all the cases it finds difficult in the set order. When the error limit is reduced by one, the previous BOGIs get a much worse fitness score, and can actually do worse than individuals which previously had lower fitnesses.

LEF is an automated way of presenting a series of different sub-problems to the population. It is highly dependent on the initial ordering, with a static training set. Some static orderings, combined with the error limit, allow GP to solve the problem much more quickly than with the shuffled orders, but it requires some more in-depth knowledge of the problem (or luck) to produce these particular orderings. Yet other static orderings impeded GP. The initial random order and dynamic 'bubbling' of the training set order seems to offer a reasonable middle-ground of difficulty, based only on the current performance of the GP population, and not on meta-knowledge of the problem.

There are some potential pitfalls when it comes to applying LEF with GP on other, more difficult, problems. If the training set is noisy, GP with LEF might produce individuals which are over-fitted to the training set, and thus perform badly on a test set. However, it should be possible to extract information from the GP runs which highlights the existence of particularly difficult training cases which don't 'fit in', by looking at cases which are frequently misclassified. The parameter values and basic LEF approach used in this section were selected after much experimentation, and there are many other possibilities. Many of them performed badly, impeding GP rather than helping it. Hopefully the values used here, and the reasoning behind them, will transfer satisfactorily to other problems. However, it is not yet obvious how to use LEF sensibly with symbolic regression problems since the error term is not as straightforward as the number of misclassifications. These 'curve-fitting' problems introduce an extra and potentially much larger error term, corresponding to how well the individual fits the data. Constructing a sensible, robust algorithm to incorporate this 'curve-fitting' error into LEF is not easy, and seems to be even more sensitive to bad parameter settings than LEF on a classification problem.

ADF has been shown by Koza to be particularly suited to the Even N Parity problem, solving for large N by simultaneously solving for smaller N. The neat hierarchical nature of its solutions complement the structure inherent in the problem. Other problems do not necessarily have such a helpful structure, so GP with ADF would most likely suffer a similar fate to GP without ADF, or might even hinder GP for problems where ADF was no help. In the case of the Even N Parity problem, ADF, as a modification to GP, performs better than LEF, as a modification to GP. However, LEF could be used in parallel with ADF, and there seems to be no reason why it wouldn't enable GP with ADF to be applied to harder problems with smaller populations.

An early (and unsupported) hypothesis is that, for classification problems such as the Even N Parity problem, if GP with LEF works, it can work more quickly and consistently than GP without LEF, and, if GP with LEF fails to work, then so will GP without LEF. On the Even N Parity problem (and others), even when GP with LEF failed to produce an optimal solution, it always produced a solution making fewer errors than GP without LEF.

Chapter 8

Small Populations, Many Generations

This chapter takes a brief and thought-provoking look at the use of small populations with GP, previously published in [Gathercole & Ross 97a]. The trend in the literature appears to be towards using as large a population as possible. Certainly this is the recommendation put forward in [Koza 92]. The rationale behind using large populations is that the GP is inefficient, i.e. it produces a low ratio of fit children. For a generation of parents to produce fitter children, a large number of children must be generated. For difficult problems, it becomes very unlikely that a particular generation will produce better individuals, thus a larger population size is seen as the main route to increasing this likelihood. Larger populations need more CPU and memory resources. CPU use is made less efficient as the computer's memory use increases. The time taken to process a generation increases.

It is an interesting notion that smaller can mean faster <u>and</u> better. Presented below are the results of a brief comparison of GP, GP with Dynamic Subset Selection (GP+DSS), discussed in Chapter 6, and GP with Limited Error Fitness (GP+LEF), discussed in Chapter 7, on the TicTacToe problem, discussed in Chapter 6, followed by a similar comparison of GP and GP+DSS on the Thyroid problem, discussed in Chapter 6, all using a small population size of 50. For both problems, GP+DSS with the small population size produces a better answer using fewer tree evaluations, than other runs using much larger populations. Even standard GP can be seen to perform well with the much smaller population size.

It is not possible to extrapolate from the Thyroid and TicTacToe problems to the set of all possible problems, and say that a small population size is always best for GP. However, the fact that it is demonstrated here for these two problems, indicates that it is certainly worth an exploratory run or two or three with a small population size before assuming that a large population size is necessary.

Section 8.1 briefly describes the TicTacToe problem, and covers the initial runs needed to find adequate parameter values for GP, LEF, and DSS, before presenting Table 8.4 which summarises the results of using a small population size. Section 8.2 briefly describes the Thyroid problem, before presenting Table 8.5 which summarises the results of using a small population size. Section 8.3 discusses the ramifications of these results.

8.1 Solving the TicTacToe problem with a small population

The task in the TicTacToe problem is to correctly classify a set of 958 cases. The set consists of all legal 3×3 board configurations at the end of TicTacToe games, where player 'x' is assumed to have played first. The target concept is 'win for player x', i.e player 'x' has three 'x's in a row, and indeed player 'x' does win approximately 65.3% of all the positions. Each case has nine fields, corresponding to the nine board positions, and each field can take one of three values {1, -1, 0}, corresponding to {player 'x', player 'o', blank}.

8.1.1 GP parameters

Previous GP runs, with and without DSS, described in Section 6.9, used population sizes of 1000 and 2000, and took several days each. Other experimental runs indicated that a smaller population size might be effective, and several runs carried out with population sizes ranging from 400 down to 50 proved this to be true. These results, in Table 8.1 were produced by a standard GP using generational replacement with elitism, a parsimony factor of 0.001 with an upper limit of 999 nodes in a tree, tournament selection with a tournament size of 4, and different sets of operators and functions from the earlier runs. The operators were

- 40% CROSSOVER -producing one child
- 20% MUTATE SUBTREE
- 20% MUTATE BY SUBTREE PROMOTION
- 20% MUTATE NODE

The function set,

{ AND (*both* > 0), OR (*either* > 0), IFGTZ, IFLTZ, IFEQZ, NEGATE }

is expanded from the flawed one used in Section 6.8 to allow GP to make use of all the possible values in the case fields. The flaw was only discovered during preparations for the work described in this chapter. Initial runs using the small population sizes and the flawed function set produced optimal, showing better performances with small populations than the earlier results for large populations. Initial runs with large populations and the new function set performed similarly as badly as ones with the flawed function set.

The terminal set used is

{ cornerNW, edgeN, cornerNE, edgeW, centre, edgeE, cornerSW, edgeS, cornerSE }

corresponding to the positions on the TicTacToe board

As can be seen in Table 8.1, a population size of 200 appears to be the most likely of the ones tested to result in GP producing an optimal tree. Even so, using a population size of 50 still results in a success rate of 35%, half of the success rate for a population size of 200, but in under a third of the number of tree evaluations (indicating the potential that in the long run there might be more chance of success for the same amount of effort expended using the smaller population size). Thus, for the sake of speed, a population size of 50 could be considered reasonable. The runs with a population size of 400 were very slow, taking several days, and thus only nine runs were completed. With a population size of 200, the ten runs were also quite slow, but had a much higher success rate. It was possible to complete twice as many runs with population sizes of 50 and 100, to provide statistically more sound averages.

Results from looking at the effects of small Population Sizes with standard GP on the TicTacToe problem (tournament size = 4, maximum 4000 generations)					
Pop. Size	Runs	Successes	% Success	Avg Gens	Avg Evals
50	20	7	35%	3013	1.4e+08
100	20	9	45%	2178	2.1e+08
200	10	7	70%	1907	3.7e+08
400	9	4	44%	2556	9.8e+08

Table 8.1: Looking for a Small but still Effective Population Size for the TicTacToe problem

Given a population size of 50, it is not obvious what is a reasonable tournament size. Results in Section 4.1.4 indicate that the 'ideal' tournament size is a fickle thing, changing with each problem and population size. Three tournament sizes are compared, and the results presented in Table 8.2 indicate that, for this problem and population size of 50, a tournament size of 4 is marginally better than tournament sizes of 2 or 6.

Results from looking at the effects of different Tournament Sizes with standard GP on the TicTacToe problem (population size = 50, maximum 4000 generations)					
Tourn. Size	Runs	Successes	% Success	Avg Gens	Avg Evals
2	29	1	3%	2283	1.1e+08
4	30	7	23%	3013	1.4e+08
6	30	5	17%	2988	1.4e+08

Table 8.2: Looking for a good Tournament Size to go with Population size 50 for the TicTacToe problem

Assuming all the other GP parameters have reasonably effective values, this completes the brief search for an adequate setup for GP.

8.1.2 LEF parameters

For LEF, however, there are extra parameters, described in Section 7.1, which need to be set. The LEF algorithm uses the idea of an error limit. During the evaluation stage, when an individual is evaluated on each case in the training set, the evaluations are curtailed if the individual exceeds the error limit. The remaining cases are then treated as misclassifications and included in the individual's fitness rating. The error limit and the order of cases in the training set are modified according to the performance of the best of generation individual (BOGI), and are left unchanged if the BOGI is improving.

If, over the course of several generations without improving, the BOGI exceeds the error limit (OVER-COVERAGE), the BOGI is then raised. If, over the course of several generations without improving, the BOGI matches (EXACT-COVERAGE) or comes under the error limit (UNDER-COVERAGE), the error limit is lowered and the training set order 'bubbled', i.e. in a single pass of a bubble sort, the easiest case (the one misclassified least often) is bubbled away from the start of the set to the end. Experimental runs in Section 7.1 indicate that LEF works better if the changes to the error limit and the training set order are kept small. However, it is not clear what are suitable values for the delays (or PAUSE) between noticing that the BOGI is not improving and altering the error limit and set order.

There are three distinct phases (EXACT, OVER, UNDER) each with its own delay. Twenty runs each for three possible combinations are presented in Table 8.3, in a brief attempt to find adequate values. The best of the three is taken as the settings for LEF used below in the main comparison with GP, GP+LEF, and GP+DSS.

For this chapter, the initial error limit was arbitrarily set to 50, but, with hindsight, there should perhaps have been some more effort put into finding a good initial value.

In [Gathercole & Ross 97b], very similar dynamics are noted for LEF and Hillis's co-evolving host and parasite populations, [Hillis 90]. Other similarities are noted with niches and fitness-sharing, [Rosin & Belew 95], and competitive environments, [Angeline & Pollack 93], using the population as a "reservoir for comparison". These are described more fully in Chapter 5.

Results from looking at the effects of different settings of the PAUSE parameters for LEF (population size = 50, tournament size = 4, maximum 4000 generations)							
PAUSE			Runs	Successes	% Success	Avg Gens	Avg Evals
Exact	Over	Under					
15	5	15	20	2	10%	2928	1.0e+08
5	5	5	20	4	20%	3164	1.1e+08
<u>15</u>	<u>15</u>	<u>5</u>	20	6	<u>30%</u>	2550	8.9e+07

Table 8.3: Looking for good LEF Pause Parameters on TicTacToe problem

8.1.3 DSS parameters

DSS, like LEF, has its own fair share of parameters. The DSS algorithm, described more fully in Section 6.3, involves picking a subset of cases from the full training set for each generation. Each generation of GP is evaluated using only this subset. The cases are selected according to how difficult they are, i.e. how often they were misclassified last time they were selected, and how old they are, i.e. how many generations since the last time they were selected. Older and more difficult cases have a greater likelihood of being selected to be in the next subset. This means that more difficult cases will be re-selected more often, but also ensures that all cases will repeatedly selected, allowing their difficulty rating to be updated according to the changing abilities of later generations. Siegel describes a similar algorithm in [Siegel 94], but does not make use of the age aspect.

The three main parameters are subset size, bias given to difficulty, and bias given to age. The two biases take the form of exponents which increase the size, i.e. importance, of the difficulty and age values of each case. Initial runs indicated that a subset size of 100, half the size of the one used in Section 6.9, worked well with the population size of 50, and a tournament size of 4. The age exponent was kept the same at 3.5. The difficulty exponent, however, was increased from 1.0 to 2.0 to accommodate the smaller population size (and thus lower difficulty values). Unlike the more random subset selection algorithm in described in Section 6.3, a version of roulette-wheel selection has been used to ensure that exactly the specified number of cases are selected to be in the subset.

8.1.4 Comparison of GP, GP+LEF, and GP+DSS, on TicTacToe

Using the GP, LEF, and DSS parameters established above, the results of fifty runs for each of GP, GP+LEF, and GP+DSS, are presented in Table 8.4, alongside some of the results from Section 6.9 using much larger population sizes. The runs involving populations of size 50 all have the same number of permitted tree evaluations, equivalent to GP over 4000 generations. As can be seen, GP+DSS performs very well, discovering an optimal tree in all fifty runs. GP and GP+LEF both have much lower

Results on TicTacToe Problem from comparing GP, GP with DSS (subset size 100), and GP with LEF, (population size = 50, tournament size = 4, maximum 4000 generations, or equivalent 1.9e+08 evaluations)					
Algorithm	Successes /Runs (%)	Average		Best	
		Gens [StdDev]	Evals	Gens	Evals
GP	12/50 (24%)	2727 [0835]	1.3e+08	833	4.0e+07
GP + LEF	16/50 (32%)	3830 [1461]	1.3e+08	1396	5.3e+07
GP + DSS	50/50 (100%)	8577 [5431]	5.1e+07	2326	2.8e+07
Population Size = 1000, DSS subset size = 200					
GP	0/1 (0%)	...failed...		114	1.1e+08
GP + DSS	1/1 (100%)	196 [ ]	3.9e+07	196	3.9e+07
Population Size = 2000, DSS subset size = 200					
GP	0/1 (0%)	...failed...		94	1.8e+08
GP + DSS	1/1 (100%)	131 [ ]	5.2e+07	131	5.2e+07

Table 8.4: Comparison of GP, GP with DSS, and GP with LEF, on TicTacToe Problem.

success rates, with GP+LEF discovering optimal trees slightly more often than GP, and with a similar number of tree evaluations to GP, on average. Both GP and GP+LEF had the potential of discovering an optimal tree in approximately 1000 generations. GP+DSS on the other hand progressed much more slowly in terms of generations, but much faster in terms of tree evaluations, towards discovering an optimal solution. The slowest run by GP+DSS took just over 26000 generations, though this falls well short of 32158 generations which, for GP+DSS, involves an equivalent number of tree evaluations to that of GP over 4000 generations.

Comparing the performance of GP and GP+LEF runs which failed to find an optimal tree, GP on its own often resulted in runs which bottomed out very quickly on a high number of errors, in a few hundred generations, then showed no signs of improvement over the remaining thousands of generations. A few GP runs were still improving even by the end of 4000 generations, very slowly, but had only reached a relatively modest number of errors. Most of the failed GP+LEF runs still showed signs of improvement even after 4000 generations, and had usually achieved fewer errors than GP. Even the GP+LEF runs which had not improved for many hundreds of generations would, if allowed to run longer, have undergone some substantial upheavals as the LEF algorithm altered the training set order sufficiently to render the current best of generation individual unfit. There were, however, very few instances of the catastrophic decrease in the best of generation fitness described in Section 7.5.

The results for populations sizes of 1000 and 2000 are only reported for individual runs, and are not averaged. They are more to indicate the potential of those population sizes. Only GP+DSS was able to find an optimal solution with these large populations sizes, needing similar numbers of evaluations to GP+DSS with a population size of 50. However, the much smaller population size requires much less runtime memory, and consequently the CPU runs more efficiently.

8.2 Solving the Thyroid problem with a small population

The Thyroid problem, described in Section 6.6 is tackled again in this section, but using a much smaller population size of 50, as used in the TicTacToe problem above. The task in the Thyroid problem is to correctly classify a set of 3772 training cases. There is a test set of 3428 cases to help assess the potential performance of the classifier on unseen cases. The Thyroid data comes from patients at a clinic. A small proportion of the patients later developed illnesses related to their thyroid glands. Each case consists of 21 fields, made up of 15 binary values and 6 floating point values, and falls into one of three possible classes. Class 3 signifies a 'normal' thyroid gland and is by far the most common class in both training and test data sets, whilst classes 1 and 2 signify that the patient later experienced a thyroid gland problem. To be useful in practise in identifying potential thyroid problems, a classification scheme would have to correctly classify significantly more than 92% of all cases, since over 92% of all patients have a normal (class 3) thyroid gland. As described in Section 6.6, the Thyroid problem can be split into two sub-problems: distinguishing between Class 3 cases and all others (hard), and distinguishing between Class 1 and Class 2 cases (easy). To come up with a reasonable solution to the hard problem, GP requires a great deal of computing power. The results from GP using large population sizes is presented in Section 6.7.

The results from very few runs are presented here, in Table 8.5. Even with the ad-

Results for GP with and without DSS on Thyroid Problem (population size = 50, tournament size = 4, DSS subset size = 400)						
Algorithm	Runs	Generations (evals)	Errors on Training Set (% success of 3772)		Errors on Test Set (% success of 3428)	
			Best	Avg. [StdDev]	Best	Avg. [StdDev]
Population Size = 50						
GP	3	4000 (7.7e+08)	10 (99.73%)	19.3 [8.1] (99.49%)	32 (99.07%)	61.3 [21.4] (98.29%)
GP + DSS	8	3-11000 (3.3e+08)	3 (99.92%)	7.4 [5.3] (99.80%)	23 (99.32%)	35.25 [11.3] (98.97%)
Population Size = 5000						
GP	1	60 (11.3e+08)	11 (99.70%)		34 (99.00%)	
GP + DSS	1	117 (2.3e+08)	11 (99.70%)		34) (99.00%)	
Population Size = 10000						
GP	0					
GP + DSS	1	69 (2.7e+08)	6 (99.84%)		25 (99.27%)	

Table 8.5: Results for GP with and without DSS on Thyroid Problem. Most of the runs were curtailed early due to time pressures. Only 3 runs with GP completed 4000 generations. 8 runs with GP + DSS completed between 3000 and 11000 generations, where 1 generation by GP is equivalent (in terms of tree evaluations) to 7.06 generations by GP+DSS.

vantages of a small population size of 50, and DSS, runs still take a long time (many days), and all of the GP+DSS runs were curtailed before they reached an equivalent number of tree evaluations to GP over 4000 generations (the equivalent number of generations for GP+DSS is 28239). However, it can be seen that, even with this handicap, GP+DSS produced better results than GP. The average performance by GP+DSS on the training set was better than the best performance by GP. With a population of size 50, GP+DSS has produced a substantially better result than any of the other GP runs, with or without DSS, for the much larger population sizes of 5000 and 10000 shown in Section 6.7, and one of these runs had already beaten the best test set performance by a Neural Network, reported in [Schiffmann *et al.* 92a] and shown in Section 6.7. Compared with the dismal early performance exhibited by GP during the early research done for this thesis, achieving 99.92% on the training set, and 99.32% on the test set is very gratifying.

The average sizes of the best trees found by GP with a population of 50 was 71 nodes, whilst the best trees found by GP+DSS had an average of 301 nodes. Once again this can be seen as a consequence of the DSS fitness function preventing any one tree from remaining as the best of generation individual over several generations. With GP, the one individual is likely to remain as best of generation for many generations, giving the operators more opportunity to prune the tree down to a smaller but functionally equivalent version, responding to the pressure provided by the parsimony penalty. With GP+DSS, there is less opportunity for this effect to take place, since each generation is likely to throw up a different best individual.

8.3 Discussion

This chapter demonstrates that small can be beautiful in the world of GP. A small population, instead of being desirable only from the point of view of computer resource limitations, actually seems to be of benefit in solving the problems investigated here.

A large population clearly evolves in a different way from a small population. It is apparent that a large population running for a small number of generations behaves differently from a small population running for a large number of generations. A large population is more likely than a small population to make bigger improvements in fitness from one generation to the next. A small population is only likely to produce individuals with small increments in fitness from the previous generation. However, in the TicTacToe and Thyroid problems there is a great deal of scope for small improvements in fitness. What is more, the actions of DSS and LEF produce even more possibilities for small increments in fitness. Even the use of parsimony adds the possibility of very small increments in fitness, as superfluous subtrees can be snipped out resulting in slightly smaller and thus slightly fitter trees.

When using tree evaluations as a measure of the computational effort needed by a run, a small population goes through many more generations than a large population.

A population of size 50 can go through nearly 20 times as many generations as a population of size 1000 before matching the number of tree evaluations. Assuming there is at least one optimal tree (as in the TicTacToe problem), a large population has to discover trees with larger increments in fitness between each generation to reach an optimal tree than a small population. The small population can go through many smaller increments to produce trees with the same optimal fitness as the large population. From any generation, small increments in fitness are much more likely than large increments in fitness, especially with GP where the operators very often produce child trees which are less fit than the parents. It seems sensible to suppose that small increments in fitness by a small population are much more likely than large increments by a large population. Thus, a more 'efficient' way to use a GP population is to keep it small. If the likelihood of small increments in fitness by a small population is too low, for example if the problem is too hard or has very few possible increments in fitness values, only then is it worthwhile (or necessary) to use a large population. This could form the basis of a method for characterising problem types, looking at distribution of possible fitness values.

What happens if the population size is taken down to an extreme, i.e. one? This then enters the realms of Hill-Climbing, Simulated Annealing, etc. The characterisation of problems by looking at the range and number of possible fitness values might lead to a way of deciding in advance which is the better approach to use in tackling a problem. It can be argued that if a problem has many possible fitness values, where it is quite possible to improve in small increments of fitness, GP is made more efficient using a smaller population, especially in combination with a more powerful supervised learning algorithm such as Dynamic Subset Selection, which dynamically alters the fitness function in response to each generation's abilities.

With small populations, large increases in fitness are less likely than in large populations. With DSS and LEF, the fitness function is continually changing, allowing super-fit individuals only fleeting moments of glory, and giving the population more opportunity to undergo small increments in fitness, especially with the Thyroid and TicTacToe problems. The combination of these factors leads to the much improved performance of GP.

There is a cost-tradeoff between running large populations a few times and small populations many times. It is very hard to get adequate statistics for runs with large populations, for obvious reasons. However, DSS seems to have made GP with a small population a much more reliable and practical proposition. Experience suggests that GP with DSS is more consistent than GP without DSS.

Part IV

Summary and Conclusion

Chapter 9

Further Work

Anarchically Automatically Defined Functions (AADF) This representation, described in Section 3.3, is exceedingly flexible. Despite its abject failure so far, it does hold great potential in allowing GP to construct arbitrarily complex function definitions within individual trees, including multi-layered hierarchies, recursion, and iteration. As mentioned in Section 3.3:

> "Much care and much more thought is needed to enable GP to take advantage of AADF. One possibility is to allow certain links between nodes to be made inviolate, i.e. the reproduction operators prevented from splitting the trees at those points. Another possibility is to include rewards in the fitness function for making use of the extra features. This might enable GP to retain the extra complications long enough to make use of them."

The MAX problem Described in Section 4.1, the MAX problem highlights several aspects of GP and its operators. It would be worthwhile seeing whether the Mutation operators, without Crossover, would experience the same difficulties as Crossover on its own. Following on from the analysis showing that high level nodes almost never get brought down to the lower layers, it might be worthwhile designing some operators to explicitly overcome this.

Making Use of the Training Set From the fairly brief survey of supervised learning in areas other than GP, described in Chapter 5:

> "Without doubt the GP and Evolutionary Algorithm communities in general are re-discovering a lot of work that the wider Machine Learning community has known about for years."

A more vigorous and wide-ranging survey of the Machine Learning literature is called for to distinguish between genuinely novel and rediscovered techniques.

Dynamic Subset Selection (DSS) Only some aspects of DSS are described or investigated in Chapters 6 and 8. Experience now suggests a useful modification to DSS would be a delay of several generations between changes in the subset. A closer look at the dynamics of the cases in the subset selection process might bring to light some useful information on the performance of DSS.

Historical Subset Selection (HSS) could be used as a first step to prune away useless training cases.

DSS should be tried on many more problems.

Limited Error Fitness (LEF) LEF, described in Chapters 7 and 8, is not a very robust algorithm. More runs on many different problems are needed to identify good choices of parameters. Experience suggests that the error limit should be decreased less often, and perhaps not in combination with a 'bubble'. More effort should be put into choosing a good starting error limit. It would be worth looking more closely at what happens to the population during one of the catastrophic losses in fitness that LEF engenders. Perhaps it is better that they happen.

On several occasions, a particular set ordering allowed GP to solve the Parity problem very quickly. It might be possible to extract some general features of such 'ideal' set orderings and re-orderings, perhaps allowing them to be constructed more deliberately.

Small Populations, Many Generations Further work, begun in Chapter 8, is needed to quantify more precisely when small population sizes might perform more efficiently than larger population sizes. A series of runs with a much wider range of population sizes and different problems might be instructive.

Parsimony Whilst parsimony performed well during this thesis, experience and various statements in the literature suggests that it could usefully be made even less severe. Perhaps it could be applied every few generations instead of every generation, allowing the population more freedom to grow if needed, but still being reined back with an intermittent pressure towards smaller trees.

Overall Perhaps the main area that could or should be pursued is to carry out more runs on many more and different problems to establish better performance statistics for DSS, LEF, and small population sizes in general.

Chapter 10

Summary

Chapter 1 looks at GP in the context of other computer-based search algorithms, showing its similarity to Stochastic Iterated Hill Climbing, and its roots in Darwin's theory of Natural Selection and Genetic Recombination, and describes the mechanics of the basic GP algorithm and Supervised Learning.

Chapter 2 introduces the 'large and messy' Thyroid problem, describing how it has been tackled by several different Machine Learning researchers, and looks at some of the difficulties and lessons learned during the process of using GP to solve the Thyroid problem. This chapter might be useful to anyone using GP for the first time, hopefully enabling them to avoid certain pitfalls.

Chapter 3 discusses the enormous size of the search for GP, and looks at a variety of alternative representations for reducing the size of the search space or increasing the power or speed of GP. The use of linear machine code segments instead of trees leads to a huge increase in speed, with less flexibility than the standard GP tree representation, whilst Automatically Defined Functions and Adaptive Representation increase the expressive power of the standard GP tree representation. The failure of a highly flexible, extremely powerful, but practically useless approach called Anarchically Automatically Defined Functions (AADF) is described. It is possible that AADF could be made to work, but much more research would be needed to establish this.

Chapter 4 looks at GP operators, concentrating on the impact of restricted tree size (both depth and number of nodes) on the performance of Crossover, showing (with a large number of runs) how the restriction can leave GP unable to improve upon sub-optimal solutions even with the use of Mutation operators. The work in Section 4.1 looking at the interaction between GP Crossover and restricted tree size involves a problem constructed specifically for this investigation. There aren't many substantial solutions presented for the general difficulties raised, instead the section simply warns of the existence of these difficulties. In the chapter is also a study of the effects of tournament selection on the GP population, and the use of Directed Acyclic Graphs to store the GP population and cache fitness evaluations is described.

Chapter 5 highlights the fitness evaluation bottleneck, and describes what can be done with the training set in Supervised Learning, looking at several approaches for Neural Networks, Decision Trees, Evolutionary Algorithms, and GP.

Chapter 6 describes Dynamic Subset Selection (DSS), where a small subset of the training set is used to evaluate each generation in GP, applying it to the large and messy Thyroid classification problem, and the smaller and neater TicTacToe problem. DSS is a robust algorithm which enables GP to produce better and more consistent results, outperforming a benchmark result reported in a study of Neural Networks on the Thyroid problem.

Chapter 7 describes Limited Error Fitness (LEF), where a fitness evaluation is curtailed when the individual exceeds an error limit, applying it the Boolean Even N Parity problem. LEF allows GP to solve for N=6 and N=7 with a small population without the use of Automatically Defined Functions (ADF), a much more powerful representation, especially suited to the hierarchical structure of the Parity problem. Koza managed to get standard GP to solve up to N=5, and only solved for larger N using ADF.

Chapter 8 looks at the use of small populations over many generations, showing how DSS and LEF can enable smaller populations in GP to work very well, outperforming larger populations with less computational effort.

Chapter 9 details areas for further work.

Chapter 11 brings the thesis to a conclusion.

Glossary gives definitions of many of the terms and acronyms used in this thesis, including a page reference to where each is first mentioned or defined in the thesis.

Chapter 11

Conclusion

The original aim of this thesis was

> "... [to look] at ways of improving standard GP and the use of GP in the particular area of Supervised Learning classification problems. GP has many obvious but circumventable weaknesses, most notable among these are its rapacious demands for computer memory and CPU time. There is lots of scope for improvement."

This thesis can be said to have contributed to GP in the following areas:

Using the Training Set – Highlighting the ease with which GP can be improved if more use is made of the training set.

Dynamic Subset Selection (DSS) , developed within this thesis, is a powerful modification of the standard supervised learning approach. DSS is robust, working well with a variety of parameter settings. It enabled GP to solve the 'large and messy' Thyroid classification problem to a very high level of accuracy, with less computational effort, and with a much smaller population size. DSS enabled GP to solve the TicTacToe problem with a much higher consistency. DSS looks as though it will scale up well to much larger problems.

Historical Subset Selection (HSS) is a simple and obvious approach to quickly find a useful and reasonably effective subset of a large training set with which to train GP. Although not a new technique, it is shown here that it can work well.

Random Subset Selection (RSS) is a less focussed version of DSS which works surprisingly well despite its simplicity. As with HSS, RSS is not a new technique, but can work well.

Limited Error Fitness (LEF) , developed within this thesis, is quite different from DSS, more sensitive to its parameters, but allows GP to solve the very hard Even N Parity problem for larger N than previously achieved without Automatically Defined Functions.

The MAX problem , developed within this thesis, is a simple test of GP which highlights some unfortunate interactions between Crossover, one of the most common and standard GP operators, and a limit on tree size, another common feature of many GP implementations.

Small populations are shown to be rather effective in GP, especially in combination with DSS. Small populations make more efficient use of the way GP works, allowing the GP population to evolve in a series of small (and thus more likely) increments in fitness.

Various weaknesses of this thesis can be summarised as follows:

Need more runs and a wider variety of test problems to establish more reliable performance statistics for DSS, LEF, and the use of small populations.

Some obvious followups not followed up.

Some Re-Inventing of the Wheel – A more vigorous and wide-ranging survey of the Machine Learning literature is called for, since a great many researchers have trod the supervised learning path before.

Still many areas of GP which need improving such as the performance of the standard GP operators, finding effective and robust parameter values, and GP's boundless craving for computer resources.

Whilst GP is a fairly obvious extension of a linear GA, the tree-structured solutions bring with them more and richer dynamics. The tree representation allows GP to be applied to some very interesting problem domains, but unfortunately has a performance penalty, needing a great deal of computer memory and CPU time. This thesis shows several ways of circumventing these limitations in supervised training classification problems, most especially the use of Dynamic Subset Selection in combination with a small population size.

Glossary

`AADF`	Anarchically Automatically Defined Functions	*(pp.54)*
`ADF`	Automatically Defined Functions	*(pp.53)*
`AR`	Adaptive Representation	*(pp.57)*
`BOGI`	Best of Generation Individual	*(pp.141)*
`Bushiness`	Tree Size, i.e. number of nodes	*(pp.147)*
`CSS`	Combined Stepwise Selection	*(pp.112)*
`DAG`	Directed Acyclic Graph	*(pp.102)*
`DFF`	Dynamic Fitness Function	*(pp.134)*
`DSS`	Dynamic Subset Selection	*(pp.117)*
`EA`	Evolutionary Algorithm	*(pp.9)*
`EP`	Evolutionary Programming	*(pp.9)*
`ES`	Evolution Strategie	*(pp.9)*
`GA`	Genetic Algorithm	*(pp.9)*
`GP`	Genetic Programming	*(pp.10)*
`HC`	Hill Climbing	*(pp.7)*
`HSS`	Historical Subset Selection	*(pp.117)*
`IFEQZ`	If Equal To Zero	*(pp.162)*
`IFGTZ`	If Greater Than Zero	*(pp.162)*
`IFLTE`	If Less Than Or Equal to	*(pp.45)*
`IFLTZ`	If Less Than Zero	*(pp.162)*
`LEF`	Limited Error Fitness	*(pp.139)*
`ML`	Machine Learning	*(pp.110)*
`NN`	Neural Network	*(pp.126)*
`Parsimony`	A fitness penalty proportional to tree size	*(pp.63)*
`RSS`	Random Subset Selection	*(pp.117)*
`SA`	Simulated Annealing	*(pp.7)*
`SGPC`	Simple Genetic Programming in C	*(pp.25)*
`SHC`	Stochastic Hill Climbing	*(pp.8)*
`SIHC`	Stochastic Iterated Hill Climbing	*(pp.8)*

`SL`	Supervised Learning	*(pp.18)*
`STGP`	Strongly-Typed Genetic Programming	*(pp.52)*

Bibliography

[Aha 93] David W. Aha. Tic-tac-toe endgame database. The AI CD-ROM, Revision 2, Network Cybernetics Corporation, 1993.

[Andre & Koza 96] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.

[Angeline & Kinnear, Jr. 96] Peter J. Angeline and Kenneth E. Kinnear, Jr., editors. *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA, 1996.

[Angeline & Pollack 93] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270. Morgan Kaufmann, 1993.

[Angeline 96a] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Angeline 96b] Peter J. Angeline. Two self-adaptive crossover operators for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[Angeline 97] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA, 13–16 July 1997.

[Atkin & Cohen 94] Marc S. Atkin and Paul R. Cohen. Learning monitoring strategies: A difficult genetic programming application. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 328–332a. IEEE Press, 1994.

[Bäck *et al.* 91] Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In Lashon B. Belew, Richard K.; Booker, editor, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9, San Diego, CA, July 1991. Morgan Kaufmann.

[Blickle & Thiele 94] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38. Max-Planck-Institut für Informatik (MPI-I-94-241), 1994.

[Blickle & Thiele 95] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, December 1995.

[Blickle 96] Tobias Blickle. Evolving compact solutions in genetic programming: A case study. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 564–573, Berlin, Germany, 22-26 September 1996. Springer-Verlag.

[Brave 96] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[Breiman *et al.* 84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[Cohn *et al.* 94] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.

[Cramer 85] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In

John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, 1985.

[Crosbie & Spafford 96] Mark Crosbie and Eugene H. Spafford. Evolving event driven programs. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 273–278, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Darwin 59] C. Darwin. *On the Origin of Species.* John Murray, London, 1859.

[D'haeseleer 94] 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. IEEE Press, 1994.

[Ehrenburg 96] Herman Ehrenburg. Improved direct acyclic graph handling and the combine operator in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 285–291, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Fisher 36] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[Fogel & Fogel 96] D. B. Fogel and L. J. Fogel. An introduction to evolutionary programming. *Lecture Notes in Computer Science*, 1063:21–??, 1996.

[Fogel 92] D. B. Fogel. An analysis of evolutionary programming. In D. B. Fogel and W. Atmar, editors, *Proceedings of the First Annual Conference on Evolutionary Programming*, pages 43–51, La Jolla, California, 1992.

[Fogel 93] D. B. Fogel. On the philosophical differences between evolutionary algorithms and genetic algorithms. In D. B. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 23–29, La Jolla, California, 1993.

[Fogel 95] D. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE, 1995.

[Francone *et al.* 96] Frank D. Francone, Peter Nordin, and Wolfgang Banzhaf. Benchmarking the generalization capabilities of a compiling genetic programming system using sparse data sets. In John R. Koza, David E. Goldberg, David B.

Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 72–80, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Garces-Perez *et al.* 96] Jaime Garces-Perez, Dale A. Schoenefeld, and Roger L. Wainwright. Solving facility layout problems using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 182–190, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Gathercole & Ross 94a] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 312–321, Berlin, 9-14 October 1994. Springer-Verlag.

[Gathercole & Ross 94b] Chris Gathercole and Peter Ross. Some training subset selection methods for supervised learning in genetic programming. Presented at ECAI Workshop on Applied Genetic and other Evolutionary Algorithms, organised by Gusz Eiben, Bernard Manderick, Zsofi Ruttkay, 1994.

[Gathercole & Ross 96] Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Gathercole & Ross 97a] Chris Gathercole and Peter Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, San Francisco, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[Gathercole & Ross 97b] Chris Gathercole and Peter Ross. Tackling the boolean even n parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, San Francisco, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[Goldberg & Deb 91] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlings, editor, *Foundations of genetic algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, 1991.

[Goldberg & Richardson 87] David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Lawrence Erlbaum, 1987.

[Goldberg 89a] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

[Goldberg 89b] D. E. Goldberg. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[Goldberg *et al.* 93] David E. Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 56–64, San Mateo, CA, USA, 1993. Morgan Kaufmann.

[Greene & Smith 93] David Perry Greene and Stephen F. Smith. Competition-based induction of decision models from examples. *Machine Learning*, 13:229, 1993.

[Gruau *et al.* 96] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Hancock 94] Peter J. B. Hancock. An empirical comparison of selection methods in evolutionary algorithms. In Terence C. Fogarty, editor, *Evolutionary Computing: AISB Workshop*, pages 80–94. Springer-Verlag, Lecture Notes in Computer Science 865, 1994.

[Handley 94] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159. IEEE Press, 1994.

[Haynes *et al.* 96] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[Hillis 90] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. Technical Report TR-1 AE91-1, Thinking Machines Corporation, 1990. (Appeared in Physica D42 [1990]: pp. 228-234).

[Hoffmeister & Back 91] F. Hoffmeister and T. Back. Genetic algorithms and evolution strategies — similarities and differences. *Lecture Notes in Computer Science*, 496:455–??, 1991.

[Holland 75] John Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[Howley 96] Brian Howley. Genetic programming of near-minimum-time spacecraft attitude maneuvers. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 98–106, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Hsu *et al.* 95] F.-H. Hsu, M. S. Campbell, Hoane, Jr., and A. J. Deep blue system overview. In *Proceedings of the Nineth International Conference on Supercomputing*, pages 240–244, New York, 1995. ACM Press.

[Implementations 97] Implementations. Illuminati online's anonymous ftp site for genetic programming, 1997. ftp://ftp.io.com/pub/genetic-programming/code.

[J.D.Schaffer *et al.* 91] J.D.Schaffer, L.J. Eshelman, and D. Offuttt. Spurious correlations and premature convergence in genetic algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 102–112. Morgan Kauffman, 1991.

[Jolliffe 86] J. Jolliffe. *Principal Component Analysis.* Springer-Verlag, 2nd edition, 1986.

[Jones 95] Terry Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* Unpublished PhD thesis, University of New Mexico, Department of Computer Science,University of New Mexico, Albuquerque NM 87131, 1995.

[Keijzer 96] Maarten Keijzer. Efficiently representing populations in genetic programming. In Peter J. Angeline and K. E.

Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA, 1996.

[Keith & Martin 94] Mike J. Keith and Martin C. Martin. Genetic programming in C++: Implementation issues. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13. MIT Press, 1994.

[Kinnear 94] K. Kinnear. *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 1994.

[Kinnear Jr. 93] Kenneth E. Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294. Morgan Kaufmann, 1993.

[Kinnear, Jr. 94] Kenneth E. Kinnear, Jr., editor. *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 1994.

[Koza 92] J.R. Koza. *Genetic Programming: on the Programming of Computers by means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[Koza 94] J.R. Koza. *Genetic Programming II:.* MIT Press, Cambridge, MA, 1994.

[Koza *et al.* 96] John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Koza *et al.* 97] John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors. *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Lang 95] Kevin J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis of Koza's. In *Proceedings of Machine Learning*, 1995. Held in Tahoe City, California, USA. July.

[Langdon & Koza 95] W. Langdon and J. Koza. Bibliography on Genetic Programming: a comprehensive index of GP related items, 1995. Can be accessed at the Advanced Search of the Collection of Computer Science Bibliographies at http://liinwww.ira.uka.de/bibliography/waisbib.html.

[Langdon & Poli 97] W. B. Langdon and R. Poli. An analysis of the max problem in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, San Francisco, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[Langdon 95] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, San Francisco, CA., USA, July 1995. Morgan Kaufmann.

[Langdon 96] W. B. Langdon. *Data Structures and Genetic Programming*. Unpublished PhD thesis, University College, London, 27 September 1996.

[Montana 95] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Nordin & Banzhaf 95] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, San Francisco, CA., USA, July 1995. Morgan Kaufmann.

[Nordin 94] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14. MIT Press, 1994.

[O'Reilly & Oppacher 92] Una-May O'Reilly and Franz Oppacher. An experimental perspective on genetic programming. In R Manner and B Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340, Brussels, Belgium, 1992. Elsevier Science.

[O'Reilly & Oppacher 94a] Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 397–406. Springer-Verlag, 9-14 October 1994.

[O'Reilly & Oppacher 94b] Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. Technical Report 94-04-021, Santa Fe Institute, 1994.

[O'Reilly & Oppacher 95a] Una-May O'Reilly and Franz Oppacher. Hybridized crossover-based search techniques for program discovery. Technical Report 95-02-007, Santa Fe Institute, 1995.

[O'Reilly & Oppacher 95b] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In *Foundations of Genetic Algorithms III*, 1995. To be published.

[O'Reilly & Oppacher 96] Una-May O'Reilly and Franz Oppacher. A comparative analysis of GP. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, Cambridge, MA, USA, 1996.

[Perkis 94] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 148–153. IEEE Press, 1994.

[Press *et al.* 92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.).* Cambridge University Press, Cambridge, 1992. ISBN 0-521-43108-5.

[Quinlan 86] J R Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Quinlan 87] J. R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987.

[Ratford 96] M. Ratford. The Single Chromosome's Guide To Dating. Unpublished M.Sc. thesis, Department of Artificial Intelligence, University of Edinburg h, 1996.

[Ray 91] T. S. Ray. *Evolution and optimization of digital organisms.* School of Life and Health Sciences, University of Delaware, Newark, Delaware 19716, USA., 1991.

[Raymer *et al.* 97] Michael L. Raymer, William F. Punch, Eric D. Goodman, Paul C. Sanschagrin, and Leslie A. Kuhn. Simultaneous feature scaling and selection using a genetic algorithm. In Thomas Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 561–567, San Francisco, July19–23 1997. Morgan Kaufmann.

[Rosca & Ballard 94] Justinian P. Rosca and Dana H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, 1994.

[Rosca 95] Justinian P. Rosca. An analysis of hierarchical genetic programming. Technical Report 566, University of Rochester, 1995.

[Rosca 96] Justinian Rosca. Generality versus size in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Rosin & Belew 95] Christopher D. Rosin and Richard K. Belew. Finding opponents worth beating: Methods for competitive coevolution. In *Proceedings of the 6th International Conference on Genetic Algorithms*, 1995.

[Ross & Corne 95] P. Ross and D. Corne. Comparing genetic algorithms, simulated annealing and stochastic hillclimbing on several real timetabling problems. In *Proceedings of the AISB Workshop on Evolutionary Computing, Sheffield, England*, 1995.

[Ryan 94] Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11. MIT Press, 1994.

[Salzburg 93] Stephen Salzburg. Improving classification methods via feature selection. Technical Report JHU-92/12, Department of Computer Science, John Hopkins University. Baltimore, June 1992; revised April 1993.

[Schiffmann *et al.* 92a] W. Schiffmann, M. Joost, and R. Werner. Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical Report 15, University of Koblenz, Institute of Physics, 1992.

[Schiffmann *et al.* 92b] W. Schiffmann, M. Joost, and R. Werner. Synthesis and performance analysis of multilayer neural network architectures. Technical Report 16, University of Koblenz, Institute of Physics, 1992.

[Siegel 94] Eric V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19. MIT Press, 1994.

[Soule *et al.* 96] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Swayne *et al.* 91] Deborah Swayne, Dianne Cook, and Andreas Buja. User's manual for XGobi, a dynamic graphics program for data analysis implemented in the X Window System (version 2). Bellcore Technical Memorandum TM ARH-020368, 1991.

[Tackett & Carmi 93] Walter Alden Tackett and Aviram Carmi. SGPC: Simple Genetic Programming in C. Available via FTP at ftp://cs.ucl.ac.uk/genetic/ftp.io.com/code/, 1993. Original source code for the GP implementation used in this thesis.

[Tackett 93] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann, 1993.

[Tan 95] C. J. Tan. Deep blue: Computer chess and massively parallel systems. In *Proceedings of the Nineth International Conference on Supercomputing*, pages 237–239, New York, 1995. ACM Press.

[Teller 94] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9. MIT Press, 1994.

[Thearling & Ray 94] Kurt Thearling and Thomas S. Ray. Evolving multicellular artificial life. In Rodney A. Brooks and Pattie Maes, editors, *Proceedings of the 4th International Workshop on the Synthesis and Simulation of Living Systems $ArtificialLifeIV$*, pages 283–288, Cambridge, MA, USA, 1994. MIT Press.

[Tichy 85] Walter F. Tichy. RCS-a system for version control. *Software-Practice and Experience*, 15(7):637–654, July 1985.

[Turney 95] Peter D. Turney. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligence Research*, 1995.

[Tuson & Ross 96a] A. Tuson and P. Ross. Co-evolution of operator settings in genetic algorithms. *Lecture Notes in Computer Science*, 1143:286–??, 1996.

[Tuson & Ross 96b] A. Tuson and P. Ross. Cost based operator rate adaption: An investigation. *Lecture Notes in Computer Science*, 1141:461–??, 1996.

[UCI 97] UCI. Machine Learning Repository. Datasets held at http://www.ics.uci.edu/~mlearn/MLRepository.html, 1997.

[Vere 95] Steven Vere. Genetic classification trees. In *Evolutionary Algorithms in Management Applications*, pages 277–289. Springer, 1995.

[Walsh & Ryan 96] Paul Walsh and Conor Ryan. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 406–409, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Webb 96] Geoff Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, June 1996.

[Weiss & Kapouleas 89] Sholom M. Weiss and Ioannis Kapouleas. An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 781–787, Detroit, MI, USA, August 1989. Morgan Kaufmann.

[Werner 92] Randolf Werner. Thyroid training and test data sets. Obtained from Daimler-Benz. Available via http://www.ics.uci.edu/˜mlearn/MLRepository.html, and from ftp://ftp.ics.uci.edu/pub/machine-learning-databases/thyroid-disease/ann-Readme, 1992.

[Zhang & Muehlenbein 95] Byoung-Tak Zhang and Heinz Muehlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 1995.

[Zhang 94] Byoung-Tak Zhang. Accelerated learning by active example selection. *International Journal of Neural Systems*, 5:67–75, March 1994.