The Evolution of Complete Software Systems

by

Mark S. Withall

A Doctoral Thesis

Submitted in partial fulfilment of the

requirements for the award of

Doctor of Philosophy

of

Loughborough University

Friday 13th June 2003

Abstract

This thesis tackles a series of problems related to the evolution of complete software systems both in terms of the underlying Genetic Programming system and the application of that system.

A new representation is presented that addresses some of the issues with other Genetic Program representations while keeping their advantages. This combines the easy reproduction of the linear representation with the inheritable characteristics of the tree representation by using fixed-length blocks of genes representing single program statements. This means that each block of genes will always map to the same statement in the parent and child unless it is mutated, irrespective of changes to the surrounding blocks. This method is compared to the variable length gene blocks used by other representations with a clear improvement in the similarity between parent and child.

Traditionally, fitness functions have either been created as a selection of sample inputs with known outputs or as hand-crafted evaluation functions. A new method of creating fitness evaluation functions is introduced that takes the formal specification of the desired function as its basis. This approach ensures that the fitness function is complete and concise. The fitness functions created from formal specifications are compared to simple input/output pairs and the results show that the functions created from formal specifications perform significantly better.

A set of list evaluation and manipulation functions was evolved as an application of the new Genetic Program components. These functions have the common feature that they all need to be 100% correct to be useful. Traditional Genetic Programming problems have mainly been optimization or approximation problems. The list results are good but do highlight the problem of scalability in that more complex functions lead to a dramatic increase in the required evolution time.

Finally, the evolution of graphical user interfaces is addressed. The representation for the user interfaces is based on the new representation for programs. In this case each gene block represents a component of the user interface. The fitness of the interface is determined by comparing it to a series of constraints, which specify the layout, style and functionality requirements. A selection of web-based and desktop-based user interfaces were evolved.

With these new approaches to Genetic Programming, the evolution of complete software systems is now a realistic goal.

Keywords: Genetic Algorithms, Genetic Programming, Representation, Formal Specification, Graphical User Interfaces, Complete Software Systems

Acknowledgements

This is the obligatory acknowledgements section—more exclusive than the *Queen's Birthday Honours List*—which gives credit to all of the people that made this PhD possible.

First, and foremost, I must thank Dr. Chris Hinde and Dr. Roger Stone for supervising this work, without them life would have been much harder. In addition, I must also thank Dr. Ondrej Sykora for distracting me with a completely different research topic (and a free trip to France).

Merci Beaucoup to André Raspaud and Olivier Baudon for making me feel welcome and making my stay at the *Université Bordeaux 1* in France a pleasant one.

Thanks to Mick O'Doherty and the rest of the guys at Nortel Networks in Maidenhead for supporting me for the first two years of my research.

Thanks to all of the research students in the Computer Science department especially Jason Cooper for the competition which made the work easier and Matthew Newton who distracted me with completely different work and made life harder (but far more varied and interesting).

Thanks to Steve Wright for the loan of his laptop for the trip to France.

A special thanks needs to go to Pam and Steve Cooper who fed me on a regular basis and fixed my monitor when it broke in the last week of work on this thesis.

Thanks to the members of the Soar Valley Canoe Club for giving me something to take my mind off the work (especially during the last few months).

Thanks to all of the people I have shared houses with throughout my time here for making the rent cheaper! A special thanks to Mark Brill for providing an infinite collection of DVDs to provide the entertainment most evenings.

Thanks to everyone who read my thesis and found the errors and suggested improvements.

And finally, I must thank my family for their support throughout my time at university.

Contents

1 Introduction and Literature Survey **1**

1.1 Introduction 1

1.2 The Evolution of Genetic Algorithms 2

1.3 Genetic Algorithms 4

1.3.1 Representation 6

1.3.2 Fitness Testing 6

1.3.3 Parent Selection 7

1.3.4 Genetic Operators 7

1.3.5 Initial Population 9

1.3.6 Termination Conditions 10

1.3.7 Population Size and Mutation Rate 10

1.4 Genetic Programming 11

1.4.1 Cramer 12

1.4.2 Koza 13

1.4.3 Banzhaf 14

1.4.4 Perkis 16
1.4.5 Montana 17
1.4.6 Whigham 18
1.4.7 Paterson & Livesey 19
1.4.8 Ryan, Collins, & O'Neill 20
1.4.9 Summary 22
1.5 The Genetic Algorithm Used Throughout The Thesis 22

2 A New Representation for Evolving Programs 25

2.1 Introduction 25
2.2 Requirements 26
2.3 Representation 30
2.3.1 Genotype 30
2.3.2 Phenotype 31
2.3.3 Mapping the Genotype to the Phenotype 31
2.3.4 Extensions 32
2.3.5 Wrapper 35
2.3.6 Example Individual 35
2.4 Comparison of Padded and Unpadded Representation 38
2.5 Example Problem: Symbolic Regression 42
2.5.1 Fitness Evaluation 44
2.5.2 Results 46
2.6 Summary and Conclusions 48

3 Using Formal Specifications to Create Fitness Functions 51

3.1 Introduction 51

3.2 Formal Specification 52

3.3 The Experiments 56

3.3.1 The Problems 56

3.3.2 The Fitness Functions 57

3.3.3 The Genetic Algorithm 62

3.4 Results 63

3.5 Summary and Conclusions 64

4 Evolving Some Interesting Functions 66

4.1 Introduction 66

4.2 Sumlist 68

4.3 Avelist 72

4.4 Listmax 74

4.5 Listmin 78

4.6 Reverse 79

4.7 Sort 84

4.8 Summary and Conclusions 88

5 Evolving the User Interface 90

5.1 Introduction 90

5.2 Requirements 91

5.3 Representation 92
5.4 Fitness Testing 93
5.5 Example Problems 94
5.5.1 A Text Editor 94
5.5.2 A Personal Details Web Form 98
5.5.3 A Front-end for the List Functions 100
5.6 Extensions 105
5.7 Summary and Conclusions 106

6 Discussion and Conclusions **107**
6.1 Introduction 107
6.2 Evolution of Complete Software Systems 107
6.2.1 Function Evolution 108
6.2.2 Scalability 109
6.2.3 Interactivity 111
6.2.4 Other Uses 111
6.3 Contribution of the Thesis 111

A Publications **124**

B Complete Results for the Formal Specification Test **126**

C Complete Set of List Functions Evolved **131**
C.1 Sumlist 132

C.2 Avelist 133

C.3 Listmax 135

C.4 Listmin 137

C.5 Reverse 140

C.6 Sort 144

D Complete Set of User Interfaces Evolved **149**

List of Figures

1.1 Class hierarchy of biologically-inspired algorithms 3

1.2 The basic flow diagram of a Genetic Algorithm 5

1.3 Crossover and Mutation 8

1.4 Mutation and Crossover for Genetic Programming 15

2.1 Example mapping from individual gene block to program statement 33

2.2 (a)Parent 1 (Padded), (b)Parent 1 (Padded) Mutated 39

2.3 (a)Parent 1 (Unpadded), (b)Parent 1 (Unpadded) Mutated . . 39

2.4 (a)Parent 2 (Padded), (b)Parent 2 (Unpadded) 40

2.5 Crossover Parent1 and Parent2 (Padded) 41

2.6 Crossover Parent1 and Parent2 (Unpadded) 41

3.1 Specification of *SquareRoot* 54

3.2 Specification of *Listmax* 57

3.3 Specification of *Reverse* 57

3.4 Comparison of the location of x in the input and output lists when $n = 3$ 61

4.1 Specification of *sumlist* 68

4.2 Specification of *avelist* 72

4.3 Specification of *listmax* 74

4.4 Specification of *listmin* 78

4.5 Specification of *reverse* 80

4.6 Specification of *sort* 84

5.1 Example gene block 92

5.2 The Widgets and Constraints for the Text Editor 96

5.3 Text Editor GUI - Seed 2 97

5.4 Text Editor GUI (The Menus) - Seed 2 98

5.5 The Widgets and Constraints for the Web Form 99

5.6 Personal Details GUI - Seed 3 101

5.7 The Widgets and Constraints for the List Front-end 103

5.8 Sort GUI - Seed 23 104

D.1 Personal Details Form, Seed 2 150

D.2 Personal Details Form, Seed 3 150

D.3 Personal Details Form, Seed 5 151

D.4 Personal Details Form, Seed 7 151

D.5 Personal Details Form, Seed 11 152

D.6 Personal Details Form, Seed 13 152

D.7 Personal Details Form, Seed 17 153

D.8 Personal Details Form, Seed 19 153
D.9 Personal Details Form, Seed 23 154
D.10 Personal Details Form, Seed 29 154
D.11 Sort GUI, Seed 2 155
D.12 Sort GUI, Seed 3 155
D.13 Sort GUI, Seed 5 156
D.14 Sort GUI, Seed 7 156
D.15 Sort GUI, Seed 11 157
D.16 Sort GUI, Seed 13 157
D.17 Sort GUI, Seed 17 158
D.18 Sort GUI, Seed 19 158
D.19 Sort GUI, Seed 23 159
D.20 Sort GUI, Seed 29 159

List of Tables

1.1 The number of choices for each production rule 20

2.1 Example Genotype 35

2.2 Statement type, type of additional genes, and form of statement 35

2.3 List of variables 36

2.4 List of comparison operators 36

2.5 Conversion of Genotype to Phenotype 37

2.6 List of statements for padding test 38

2.7 List of variables for padding test 38

2.8 List of statements for the symbolic regression 43

2.9 List of variables for the symbolic regression 43

2.10 Test input and expected output for symbolic regression 45

2.11 Results for the symbolic regression with population 500 46

2.12 Results for the symbolic regression with population 7 47

2.13 Results for the symbolic regression with population 7 and minimal language subset 48

3.1 A list of symbols and their meanings 53

3.2 Summary of the results for *Listmax* and *Reverse* (with mean and standard deviation) 63

4.1 List of statements used for *sumlist* 70

4.2 Additional Genes for *sumlist* 70

4.3 The results for *sumlist* 71

4.4 The results for *avelist* 73

4.5 List of statements used for *listmax* 75

4.6 Additional Genes for *listmax* 76

4.7 The results for *listmax* 76

4.8 The results for *listmin* 79

4.9 List of statements used for *reverse* 81

4.10 Additional Genes for *reverse*. All of the list indices are taken modulo the size of the list, however this is not shown for clarity 82

4.11 The results for *reverse* 82

4.12 List of statements used for *sort* 85

4.13 Additional Genes for *sort*. All of the list indices are taken modulo the size of the list, however this code is not shown for clarity 85

4.14 The results for *sort* 86

5.1 Results for the text editor example 97

5.2 Results for the personal details example 100

5.3 Results for the list front-end example 105

6.1 The results from the *sort* experiment with 'Swap' 110

B.1 The results of the *Listmax* experiment with Input/Output pairs 127

B.2 The results of the *Listmax* experiment with Formal Specification based fitness function 128

B.3 The results of the *Reverse* experiment with Input/Output pairs 129

B.4 The results of the *Reverse* experiment with Formal Specification based fitness function 130

Listings

1.1 Example Grammar 20

2.1 The entire phenotype, including header and footer 37

2.2 Example solution found from experiment 2, Seed 0 47

2.3 Example solution found from experiment 3, Seed 0 49

3.1 The fitness function for *Listmax* 60

3.2 The fitness function for *Reverse* 62

4.1 The fitness function for *sumlist* (without header) 68

4.2 Set of test input lists for *sumlist* 69

4.3 Example solution for *sumlist*, Seed 0 72

4.4 The fitness function for *avelist* (without header) 73

4.5 Example solution for *avelist*, Seed 0 74

4.6 The fitness function for *listmax* (without header) 75

4.7 Set of test input lists for *listmax* 75

4.8 Example solution for *listmax*, Seed 0 77

4.9 The fitness function for *listmin* (without header) 78

4.10 Example solution for *listmin*, Seed 0 80

4.11 The fitness function for *reverse* (without header) 80

4.12 Example solution for *reverse*, Seed 0. All of the list indices are taken modulo the size of the list, however this code is not shown for clarity 83

4.13 The fitness function for *sort* (without header) 84

4.14 Example solution for *sort*, Seed 0. All of the list indices are taken modulo the size of the list, however this is not shown in the code for clarity 87

6.1 Example of the *sort* function using 'Swap', Seed 1 110

C.1 Sumlist Seed 0 132

C.2 Sumlist Seed 1 132

C.3 Sumlist Seed 2 132

C.4 Sumlist Seed 3 132

C.5 Sumlist Seed 5 132

C.6 Sumlist Seed 7 132

C.7 Sumlist Seed 11 133

C.8 Sumlist Seed 13 133

C.9 Sumlist Seed 17 133

C.10 Sumlist Seed 19 133

C.11 Avelist Seed 0 133

C.12 Avelist Seed 1 133

C.13 Avelist Seed 2 134

C.14 Avelist Seed 3 134

C.15 Avelist Seed 5 134
C.16 Avelist Seed 7 134
C.17 Avelist Seed 11 134
C.18 Avelist Seed 13 134
C.19 Avelist Seed 17 135
C.20 Avelist Seed 19 135
C.21 Listmax Seed 0 135
C.22 Listmax Seed 1 135
C.23 Listmax Seed 2 136
C.24 Listmax Seed 3 136
C.25 Listmax Seed 5 136
C.26 Listmax Seed 7 136
C.27 Listmax Seed 11 136
C.28 Listmax Seed 13 137
C.29 Listmax Seed 17 137
C.30 Listmax Seed 19 137
C.31 Listmin Seed 0 137
C.32 Listmin Seed 1 138
C.33 Listmin Seed 2 138
C.34 Listmin Seed 3 138
C.35 Listmin Seed 5 138
C.36 Listmin Seed 7 139

C.37 Listmin Seed 11 139
C.38 Listmin Seed 13 139
C.39 Listmin Seed 17 139
C.40 Listmin Seed 19 140
C.41 Reverse Seed 0 140
C.42 Reverse Seed 1 140
C.43 Reverse Seed 2 141
C.44 Reverse Seed 3 141
C.45 Reverse Seed 5 141
C.46 Reverse Seed 7 142
C.47 Reverse Seed 11 142
C.48 Reverse Seed 13 142
C.49 Reverse Seed 17 143
C.50 Reverse Seed 19 143
C.51 Sort Seed 0 144
C.52 Sort Seed 1 144
C.53 Sort Seed 2 145
C.54 Sort Seed 3 145
C.55 Sort Seed 5 145
C.56 Sort Seed 7 146
C.57 Sort Seed 11 146
C.58 Sort Seed 13 147

C.59 Sort Seed 17 147

C.60 Sort Seed 19 148

Chapter 1

Introduction and Literature Survey

1.1 Introduction

This thesis addresses the topic of evolving complete software systems using Genetic Algorithms (GA) and Genetic Programming (GP). In recent years GA and GP have developed a broad following in many fields from plant biology [25] to architecture [18] (see for example [20, 68, 89]), but no one appears to have investigated the development of entire software applications.

This thesis presents a method for GP which addresses some of the main issues with the current approaches to the evolution of computer programs; specifically the representation of a program and the creation of the fitness function. It then looks at the main problems involved in evolving complete software systems: both the basic algorithms and user interfaces.

The thesis is structured as follows:

Chapter 1 introduces the areas of GA and GP and describes some of the current and past methods used for GP, highlighting their strengths and weaknesses.

Chapter 2 proposes a new representation for programs in GP that addresses some of the main weaknesses while retaining their strengths.

Chapter 3 proposes a new method for the creation of fitness functions, based on the formal specification of the problem to be solved.

Chapter 4 applies the methods described in Chapters 2 and 3 to the evolution of some list processing algorithms, such as sorting.

Chapter 5 discusses the problem of evolving graphical user interfaces (GUI) and proposes a method for specifying the requirements and evaluating the performance under the framework described in Chapter 2.

Chapter 6 summarises the work and discusses the future of evolving complete software systems.

1.2 The Evolution of Genetic Algorithms

Using evolution as a problem solving method is not a new idea. Alan Turing suggested it in the 1940s [84]. Evolutionary algorithms are part of a larger set of biologically-inspired algorithms (shown in Figure 1.1). This set includes Artificial Neural Networks based on the way in which the brain is believed to work [75], Simulated Annealing based on the natural cooling process of a system of molecules [53], Particle Swarm Optimisation based on bird flocking or insect swarms [52] and Ant Colony Optimisation based on the movements of ants along pheromone trails [24].

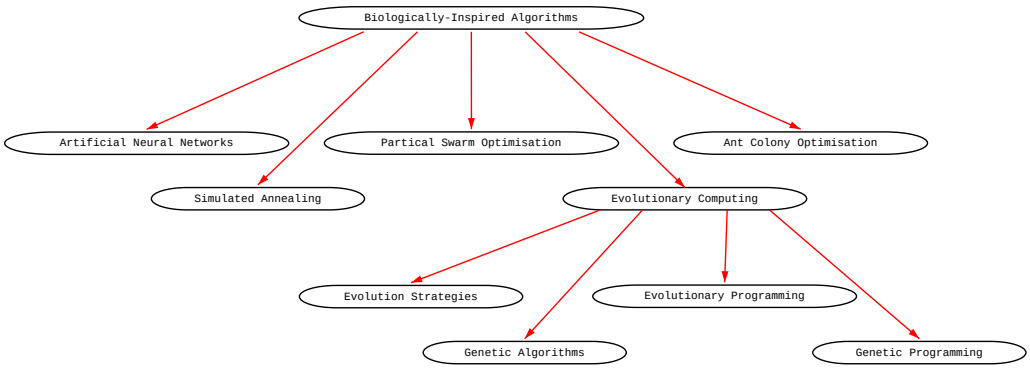


Figure 1.1: Class hierarchy of biologically-inspired algorithms

Within the class of evolutionary computing, there are two distinct groups. The first group contains Evolution Strategies [73] and Evolutionary Programming [27] which work with small populations and generally only use asexual reproduction and mutation as genetic operators. The second group contains Genetic Algorithms (GA) [48] and Genetic Programming (GP) [55]. These algorithms tend to work with larger populations and use more complex genetic operators and sexual reproduction.

Evolutionary algorithms are based on the theory of natural selection. This theory is mainly attributed to Charles Darwin, who put forward the ideas in his book of 1859 "The Origin of Species" [21]. However, Alfred Russel Wallace independently came to the same conclusions at around the same time [61]. Both Darwin and Wallace had similar experiences which inspired their theories. Darwin had his historic voyage on HMS Beagle [22], while Wallace had an expedition to the Amazon [14]. Both are also believed to have been inspired by "An Essay on the Principle of Population" by Thomas Malthus written in 1798 [60]. In his work, Malthus argues that population will always outgrow the available resources of an area, and hence competi-

tion between those occupying the same area will control the population. In essence, Malthus is saying that those in the population that are better able to acquire, or control, the available resources will prosper, whereas those who are less able will die out through lack of those resources. This is an early example of what later became known as "survival of the fittest".

For natural selection to work, it is necessary for the offspring of individuals to inherit the characteristics of their parents. This was first shown to be the case by Gregor Mendel in the mid-19th century [62]. However, his work was not widely known until the early 20th century [44]. Mendel was a monk with a fascination for gardening. His ground-breaking work was on breeding pea plants.

1.3 Genetic Algorithms

John Henry Holland is widely accepted as the father of Genetic Algorithms in their current form, even though people had been using evolution strategies and evolutionary programming for problem solving before Holland. Holland is said to have been inspired by the work of R.A. Fisher [26] on the mathematical modelling of evolution and published his book "Adaption in Natural and Artificial Systems" in 1975 [48], although he had published work on GAs well before that [46, 47].

David Fogel, in his book "The Fossil Record" [29], presents a large selection of early papers from various areas of evolutionary computing and two surveys of the history of evolutionary computing [3, 28]. These surveys tell of researchers, other than Holland, who were working on similar algorithms to simulate genetic systems at the same time and even earlier. These include the work of Fraser [30–34], Bremermann et al. [6–13], and Reed et al. [74].

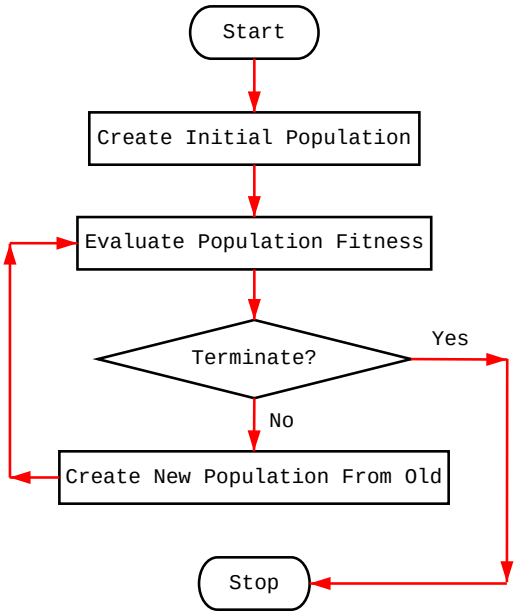


Figure 1.2: The basic flow diagram of a Genetic Algorithm

A genetic algorithm is a problem solving technique used to search a solution space until some termination criteria are met. A GA takes a population of possible solutions to a given problem (individuals), evaluates these individuals based on some criteria (fitness) and then genetically recombines them based on the fitness of the individuals in the population to form a new generation of the population. This process is repeated until some termination criteria are met. Figure 1.2 gives the flow diagram of the basic GA.

Given that a high proportion of fitter individuals are chosen as parents, the tendency is for good attributes to remain present throughout the generations while poor attributes are discarded.

The following sections describe aspects of the GA in more detail.

1.3.1 Representation

The way that individuals in a population are represented (genome) can have a great effect on the performance of a GA. In general, the larger the search space that the representation allows, the longer it will take to find an acceptable solution. Goldberg [41] states that most GAs work with a coding of the parameter set to be optimized, not with the parameters themselves. The encoded form is often referred to as the 'genotype' and the parameter set it represents, the 'phenotype'. This separation of genotype and phenotype means that the genetic operators can be implemented more efficiently without the restrictions that may be imposed on the actual parameters. This separation of genotype and phenotype is one of the criticisms of GAs [66]. This is most likely due to an indirect translation between the genotype and phenotype having the effect that small changes in the genotype cause large disruptions in the phenotype. For this reason alone it is necessary to have a fairly direct mapping between the coding and the actual parameters when such a GA is used.

A more detailed discussion on representing problems is given in [77], which looks at the use of redundancy and many-to-one mappings between the genotype and phenotype.

1.3.2 Fitness Testing

The fitness testing, for a GA, determines how good an individual from a population is at solving a given problem. The fitness is traditionally given as some numerical value where the higher the value the greater the fitness. For problems where the better fitness values are lower scores a normalization function can be used to adjust the scores so a higher score is better. The

fitness values are used to select parents for the reproduction stage of the algorithm. The normalization function can also be used to adjust scores to prevent one or two individuals completely dominating the parent selection. For multi-criteria problems, a weighted sum of all the individual criteria values can be used as the overall fitness value for an individual. It is vital that the fitness accurately assesses an individual's ability to solve the problem as any errors may be exploited by the algorithm to achieve better fitness scores.

1.3.3 Parent Selection

Parent selection is used to determine which individuals from a population will be used to create the next generation of individuals. There are two main methods for selecting parents for reproduction: *Fitness Proportionate Selection* and *Tournament Selection*.

Fitness Proportionate Selection: For this method of parent selection, individuals are selected randomly with their chance of being selected being proportional to their fitness values.

Tournament Selection: For this method, two individuals are selected randomly from the population and the individual with the higher fitness value is used. This process can be repeated to find a parent from a larger number of individuals.

More parent selection methods are given in [41] and [59].

1.3.4 Genetic Operators

Genetic operators are used to manipulate the genes of selected parents to create new individuals for the next generation. The main aim of the repro-

duction stage is to produce a new generation that retains any useful characteristics and discards any poor ones without losing the diversity in the population. This allows the population to increase its average fitness while still having access to a large amount of the search space. There are two main genetic operators: *Crossover* and *Mutation*.

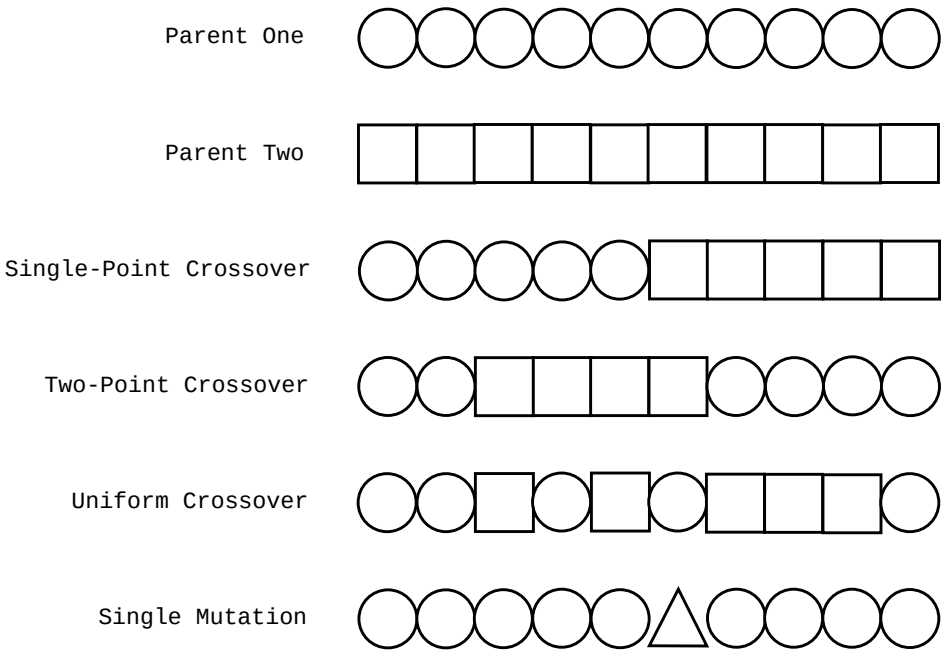


Figure 1.3: Crossover and Mutation

Crossover This consists of combining the genes from two or more parent individuals to create a new individual. This can take the form of picking a random point on the genome and using the first part from one genome and the second part from the other (See single point crossover, Figure 1.3). Multiple points can be taken and used in the same way. At the most extreme, a decision can be made for every gene for which parent to take it from. This assumes that all genomes are of the same fixed length, although this is not essential. A comparison of these methods,

which suggests uniform (every-point) crossover is best, can be found in [83]. For a discussion on different numbers of parents see [87], where the conclusion is that two parents generally produce the best result but the overall solution can take longer to achieve than when more parents are used.

Mutation: This consists of changing one or more genes in a single individual to a new value.

1.3.5 Initial Population

The initial population of a GA is important as it specifies the gene values that the GA has to work with. This gene pool can then be expanded using mutation as the GA runs. There are three main ways of creating the initial population.

The first, and most common, method is to randomly generate a selection of individuals to fill the population. The major advantage of this method is that it can provide great variation between individuals.

The second method is to seed the initial population with known solutions to try and improve them. This gives the GA a head start towards finding a solution of the required fitness. However, it can also limit the areas of the search space that the GA has initial access to, making it less likely to find a solution in other areas.

Finally, the third method is to create a random population using pre-defined blocks of genes, rather than individual genes. This allows some knowledge of the problem domain to be inserted into the population without ignoring as much of the search space as using whole solutions.

In addition, it would be possible to combine elements of the three approaches, however, it is possible that one approach may dominate the other(s). For example, if a few highly fit individuals were put into a randomly generated population, they would be likely to take over the whole population in a few generations.

1.3.6 Termination Conditions

The two main methods of terminating a GA are to pre-specify a number of generations over which to run the algorithm or to run until a member of the population reaches a specific fitness level. A GA can also be terminated manually or using any method appropriate to the problem being tackled.

1.3.7 Population Size and Mutation Rate

Both population size and the probability of mutation can be instantiated with different values. The values set can greatly affect the performance of a GA. The larger the population size being used the greater the likelihood of good characteristics being present in individuals within the initial generation and therefore the lower the number of generations which need to be run. However, the larger the population the longer it will take to fitness test each generation. Goldberg presents a method for determining the correct population size for a given problem in [42] but generally it is easier to run a few tests and adjust the value manually. Varying the rate of mutation will affect the rate of convergence to a solution. If the mutation rate is too high then the search will in effect be random but if the mutation rate is too low the population will quickly converge on a sub-optimal solution.

1.4 Genetic Programming

One of the earliest known pieces of research on using evolution to create computer programs was that of Friedberg et al. [36, 37]. Although the word 'evolution' does not appear in either paper the intent to simulate evolution was plainly in the minds of the researchers [29]. Friedberg et al. adopted the task of generating a set of machine language instructions that could perform relatively simple calculations (in this case adding the numbers in two data locations). The work of Fogel, Owens and Walsh [27], which evolved finite state machines, and the work of Holland [48] and others on learning classifier systems could also be classed as programming but in a much more restricted sense.

Possibly the first work that explicitly used Genetic Algorithms to generate programs was that of Cramer [17] in 1985. This was closely followed by the work of Fujiki et al. [38, 39], who used the method to solve the prisoner's dilemma, and the work of Hicklin [45]. All of the above used a tree representation for their programs. Banzhaf et al. describe three types of representation; tree, linear, and graph [5].

The work which popularized the area (which became known as 'Genetic Programming') was that of John Koza, initially with his 1989 paper [54] and more so with his three epic works [55–57]. This work also used the tree representation with the target programming language LISP (the same as Fujiki et al.).

Since the work of Koza, a vast amount of research has been done on the area of Genetic Programming. Some of the main systems which have been developed are looked at in more detail in the following sections.

1.4.1 Cramer

One of the first attempts that explicitly used Genetic Algorithms to evolve programs was detailed in Cramer's paper "A Representation for the Adaptive Generation of Simple Sequential Programs" [17]. Cramer demonstrated an adaptive system for generating short sequential computer functions (in the paper two-input, single-output multiplication functions were evolved). The functions were written initially in the simple language called **JB**, and later in **TB**, which was a modified version of **JB** with a tree-like structure. The representation for the programs was a list of integers which were then decoded to produce a well-formed program.

For the **JB** language, the list of integers was first divided into fixed-length groups that are long enough to specify any statement in the language subset (in the case of the paper, three). Any integers remaining at the end of the list were ignored. The first of the statements was then taken to be the main statement and the remaining statements were auxiliary statements. The functions executed the main statements which, typically, called one or more of the auxiliary statements. This method had the advantage that any list (of sufficient length and with the relevant constraint on the size of the integers) could be used to generate a well-formed program. The problems with this method are that infinite loops can be generated by the auxiliary statements and, more seriously, the semantic-positioning of an integer-list element is extremely sensitive to change. This problem is most damaging when changes occur in the main statement.

To address these problem, a modified version of **JB** was created, called **TB**. **TB** was fundamentally the same as **JB** except that auxiliary statements were not used. Instead, when a **TB** statement is generated, any subsidiary

statements which the statement contains are recursively expanded giving a tree-like structure. Again, all lists of integers map to a syntactically correct program. However, to avoid the problems of 'catastrophic minor changes' the mutation and crossover operators have to be constrained. In this case, the mutation operator can only change leaf statements or non-leaf statements that only have arguments that are leaf operators. For the crossover, subtrees are swapped between two parents.

Cramer also pointed to the work of Smith [82], which discussed that a major problem is that of 'hand-crafting' the fitness evaluation function to give partial credit to functions that exhibit behaviour similar to that which is desired, without actually performing the desired task. Cramer proposed four types of behaviour (for his multiplication function problem), with each successive type given more credit.

1. Has the output value changed from its initial value?
2. Is the output value dependent on the input value?
3. Is the input value a factor of the output value?
4. Is the function multiplication?

Functions that were beyond a certain length were also penalized to ensure functions remained short. In addition, the run-time of a function was limited.

1.4.2 Koza

Koza first introduced his method of Genetic Programming in his 1989 paper "Hierarchical Genetic Algorithms Operating on Populations of Computer

Programs" [54]. This work was then expanded on in great detail in three large volumes [55–57].

For Koza's Genetic Programming, the programs are represented as parse trees. The language LISP was used, as a subroutine in LISP (or s-expression) is essentially a parse tree expressed in a linear fashion. For Genetic Programming, the user defines all the functions, variables and constants which can be nodes in the parse tree. Variables, constants and functions which take no arguments are called 'terminals'. Functions which take arguments are called 'non-terminals'. The search space is the set of all parse trees which only use elements from the set of terminals and non-terminals.

Due to the complex nature of the structure of the genomes (LISP s-expressions), the genomes cannot be easily generated randomly for the initial population. The individuals in the initial population must be carefully constructed to preserve syntactic correctness. In addition, the genetic operators used cannot be the standard versions of crossover and mutation. Instead, mutation is accomplished by picking a random node in the tree and replacing the subtree with a randomly generated (but syntactically valid) subtree. The crossover operator is accomplished by swapping subtrees from two parent individuals. The mutation and crossover operators are shown in Figure 1.4.

This method of Genetic Programming is still one of the most widely used methods.

1.4.3 Banzhaf

In 1993, Wolfgang Banzhaf published "Genetic Programming for Pedestrians" [4]. This paper introduced a method of Genetic Programming based on traditional Genetic Algorithms. The method introduced mechanisms like

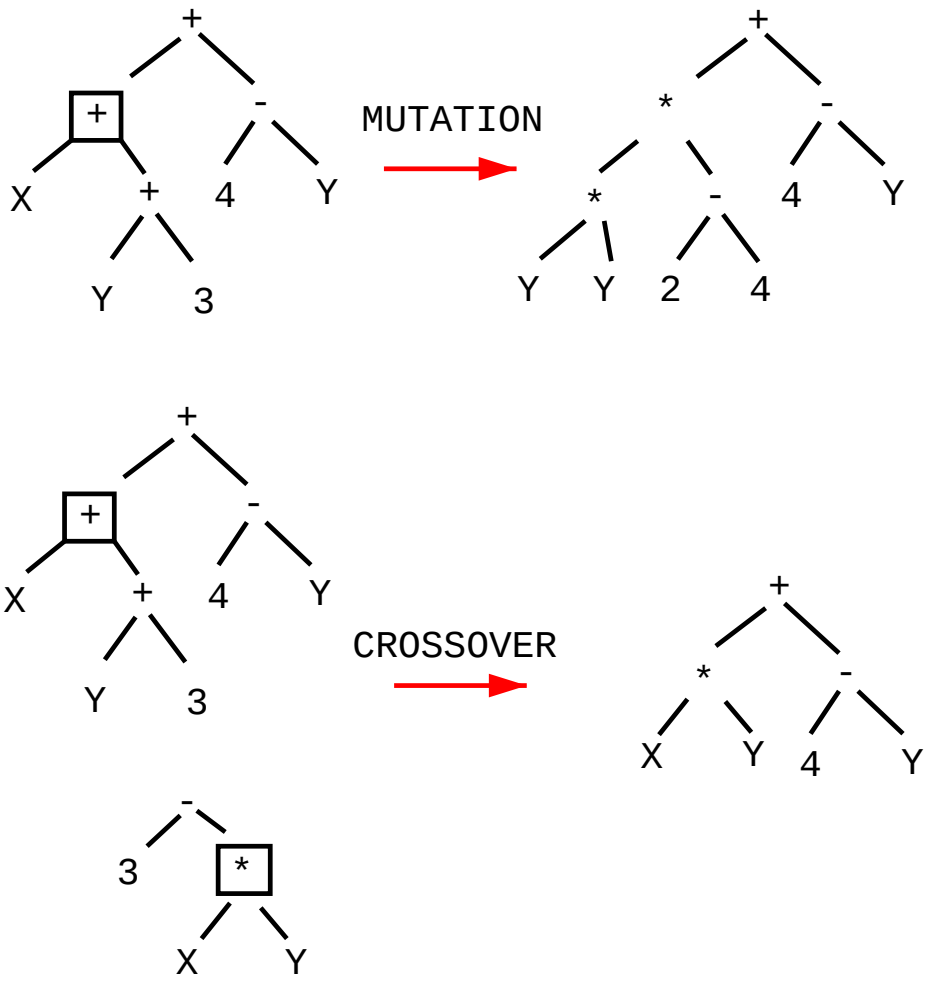


Figure 1.4: Mutation and Crossover for Genetic Programming

transcription, editing and repairing and was applied to the problem of the prediction of sequences of integer numbers. This is one of the first methods to go back to using a linear genome, since Cramer [17] rejected the idea in favour of a tree representation.

Banzhaf starts out with a population of binary strings which are subsequently interpreted as programs. This interpretation is achieved by using a coding or transcription table specifying which binary code of given length corresponds to which element from the set of functions and terminals available. The generated program can not necessarily be guaranteed to be a working program. After the binary strings have been translated into the programming language, the resulting code segments are checked to see if they are syntactically correct and any errors are repaired.

For the example problem (number sequence prediction) the fitness is evaluated as the sum of the square of the error i.e. the square of the difference between the expected and actual output values for the program. One interesting feature of the method is that, even though fixed length genomes were used, the resulting programs could vary in length.

The work is extended by Kellar and Banzhaf in "Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes" [51]. Here they borrow heavily from molecular biology and only use the mutation operator for genetic manipulation. They also show that their method can map to an arbitrary context-free language.

1.4.4 Perkis

Timothy Perkis, in his 1994 paper "Stack-Based Genetic Programming" [71], presented yet another approach to the evolution of programs that does not re-

quire specialist genetic operators. In this approach, the genome is a sequence of functions and terminals. Each element in the sequence is evaluated in turn. If the element is a function it takes the necessary number of arguments off the stack and puts its output back onto the stack. If there are not enough values on the stack then the function is ignored. If the element is a terminal it is pushed onto the stack. As there are no syntactical constraints on the sequences, the sequence can be treated in the same way as a traditional binary string for a Genetic Algorithm.

The method was demonstrated using some symbolic regression problems from Koza [55] and showed some evidence of improvement when the list of functions used were adjusted to be more appropriate for the stack-based method.

One limitation of the method presented is that it has no mechanism for branching. In addition, it would be difficult to generate a program in a specific target language using this method.

1.4.5 Montana

Montana was one of the first people to look at the problem of typing in Genetic Programming (although it had been mentioned by Koza [55]) in a technical report written in 1994 called "Strongly Typed Genetic Programming" [65]. The approach built directly on top of the approach which Koza used. Another approach was presented by Perkis using multiple stacks, one for each type [71]. The use of strong typing in GP becomes essential when the target language is e.g. c or pascal, as type mismatches would cause the programs to fail to compile.

In addition to the method that Koza used (see Section 1.4.2), each func-

tion was given requirements for the type of its arguments and its return type and the generation of the initial population and the genetic operators were changed to account for the new type constraints. Montana also introduced *generic functions* (e.g. to handle matrix functions with different size matrices) and *generic data types* for use with the functions, to ensure correct typing without having to be specific about the type.

The method presented by Montana also introduced handling of runtime errors, whereas the method used by Koza forced all functions to return valid values e.g. the protected divide function returned 1 when dividing by zero, rather than an error.

1.4.6 Whigham

One of the first people to use a context-free grammar as the basis of their representation was Whigham, in his paper "Grammatically-based Genetic Programming" [88]. Whigham still used the tree structure and therefore still had the problem of complex genetic operators. One of the main advantages of using the context-free grammar is that it allows the method to be applied to any programming language. The grammar also allows variable typing to be incorporated easily.

Whigham also used the idea of 'bias' (structuring the grammar in such a way as to improve the chances of creating good programs). This is equivalent to including extra knowledge about the problem e.g. if it is known that the program should start with an `if` statement. Rather than manually adjusting the grammar, Whigham modified the grammar during the evolution based on analysis of fit individuals. Each generation some new individuals were created from the updated grammar and incorporated into the population.

In addition, Whigham used weighted production rules to make selection of good rules more likely. This weighting was calculated as the new production rules were created.

1.4.7 Paterson & Livesey

Paterson and Livesey continue on from the work of Banzhaf, in their paper "Distinguishing genotype and phenotype in genetic programming" [69], by introducing a method that converts a linear genotype (in this case a string of integers) into a program. Unlike Banzhaf, however, there is no need for a repair stage as the list of integers maps directly onto a BNF definition of the language subset by recursively replacing all non-terminals with the production rule that corresponds to the next integer in the genotype. Like Koza and others, Paterson and Livesey initially use LISP as their target language but in later work they use C [70], showing the advantage of the method being language independent. One disadvantage of mapping the list of integers to the BNF is that there is no guarantee that a complete program will be generated. There may be unresolved non-terminals when the string of integers runs out. Other methods must then be used to fill in the missing data. One interesting experiment conducted by Paterson and Livesey was to compare two grammars that represent the same language subset. This highlights the difficulty of specifying the language subset in the most appropriate way for the problem.

Other work on linear and grammar-based representations for Genetic Programming has been done by Freeman [35] and Ross [76].

Listing 1.1: Example Grammar

```
(1) <expr> ::= <expr><op><expr>          (A)
             | ( <expr> <op> <expr> )    (B)
             | <pre-op> ( <expr> )       (C)
             | <var>                     (D)

(2) <op> ::= + (A)
           | - (B)
           | / (C)
           | * (D)

(3) <pre-op> ::= Sin (A)
               | Cos (B)
               | Tan (C)
               | Log (D)

(4) <var> ::= X
```

Table 1.1: The number of choices for each production rule

Rule No.	Choices
1	4
2	4
3	4
4	1

1.4.8 Ryan, Collins, & O'Neill

Ryan, Collins, and O'Neill present a method that is similar to Paterson and Livesey, in their paper "Grammatical Evolution: Evolving Programs for an Arbitrary Language" [78], but with a much simpler mapping between the genotype and phenotype. Ryan, Collins and O'Neill still use the BNF representation, although they use a binary string instead of a list of integers.

To show how the mapping between the genotype and phenotype works, consider the following genome (expressed as integers for clarity):

220	203	17	3	109	215	104	30

These numbers are used to make choices from the grammar in Listing 1.1. The numbers of available choices for each production rule are summarized in Table 1.1. Starting with `<expr>`, there are four options to choose from. To make this choice, the first gene is taken and its value 220, modulo 4 (the number of options), is used to choose the rule. In this case 220 mod 4 = 0, therefore rule 1A is used. This is now:

```
<expr> <op> <expr>
```

Taking the next gene, 203 mod 4 = 3, the choice is 1D.

```
<var> <op> <expr>
```

There is no choice for the `<var>` rule (and so no need to use a gene).

```
X <op> <expr>
```

The mapping continues, reading genes from the genome as necessary, until no unresolved non-terminals remain. This gives the following expression:

```
X + Sin(X)
```

Any unused genes are ignored and if there are unresolved non-terminals when the end of the genome is reached the genes are reused from the beginning until no unresolved non-terminals remain. The major disadvantage of this method is from the point of view of inheritance of characteristics. When a gene is passed on to a child individual there is a very high chance that the gene will not represent the same value. One change early in the genome can

change the entire path through the grammar and hence the child individual will have very little resemblance to its parent.

Despite this drawback, Grammatical Evolution has been applied to a variety of different problems by the original authors [67, 68, 79]. In addition, the method is developing a following around the world [43, 50].

1.4.9 Summary

This section has summarized the most relevant and interesting work on evolving programs, with a specific bias towards the way in which they represent the programs. This work is built on in Chapter 2 when defining the requirements of a good representation and developing a representation that fits those requirements.

1.5 The Genetic Algorithm Used Throughout The Thesis

The following is a description of the simple Genetic Algorithm which is used for all of the experiments in the thesis. In addition to this algorithm, the way in which the genotype is mapped to the phenotype is discussed in Chapter 2 (for general programming) and Chapter 5 (for generating user interfaces), and the creation of the fitness function is discussed in Chapter 3. The problem specific information is given with the experiments. This algorithm is kept simple to keep the focus on the effects of the representation and the fitness functions.

The simple Genetic Algorithm used is given in Algorithm 1. The genomes

Algorithm 1 The simple Genetic Algorithm used for all experiments

```
P = Initialise_Population
F = Test_Fitness(P)
for generations = 1 to MAXGEN do
   P = Reproduce(P, F)
   F = Test_Fitness(P)
end for
```

are represented as fixed length integer strings, as this is the easiest representation to work with in terms of the genotype/phenotype mapping and the genetic manipulation. The representation does not, however, preclude the use of variable length genomes. In the reproduction function, simple fitness proportionate parent selection is used to select two parents, these two parents are combined using uniform crossover and then there is a probability that each gene will be mutated in the resulting individual. The best individual from the previous generation is copied into the newly created population.

In most of this work, a relatively small population is used (usually 7) and a relatively high mutation rate is used (usually 1 gene in 10 or 20). These values are much smaller than are traditionally used, however, they appear to produce high fitness individuals quickly (see Chapter 2, Section 2.5).

In addition to the above GA, a series of scripts were written to improve productivity and ease the development of new GP tests. These include a script to generate a genotype to phenotype mapping function based on a simply defined language subset and the representation presented in Chapter 2. A template for the GP was also constructed, which only required the addition of the main body of the fitness evaluation function. Other scripts include the automatic running of experiments with a fixed set of random number seeds with the storing of the results of the experiments and a script to summarize the results of a series of experiments. This set of scripts

meant that new GPs could be generated and the results of the execution collated very quickly.

Finally, the experiments were run on a PIII 866 PC with 256MB of RAM, running the Debian GNU/Linux 3.0 operating system. The programs were written in Perl v5.6.1.

Chapter 2

A New Representation for Evolving Programs

2.1 Introduction

This chapter presents a new representation for Genetic Programming that has the explicit inheritance of systems like Koza's [55], which can be used with the simple genetic operators of an ordinary genetic algorithm and can be used to evolve programs in any language in the same way as systems like Grammatical Evolution [78].

The representation presented is a linear representation for Genetic Programming, which has a separate genotype and phenotype. This allows a simple string (or list) of integers to be used for the genetic manipulation needed to create new generations of solutions. This string is then mapped onto the programming language used (in this case a subset of the Perl language [86]), for the purpose of evaluating the fitness of the solution to the given problem. The mapping process takes fixed-length blocks of genes from

the genome and converts them into program statements. This type of mapping allows explicit inheritance of characteristics between parent and child individuals. The separation of the genotype and phenotype leaves the system looking more like a traditional Genetic Algorithm, with the interpretation of the genome contained within the fitness function.

Section 2.2 of this chapter sets out a list of requirements for a Genetic Programming representation, based on the analysis of the previous systems described in Chapter 1. This list of requirements is then used as the basis of a representation, which is presented in detail in Section 2.3. Section 2.4 presents an argument for the use of a fixed-length gene blocks to represent program statements in terms of its preservation of characteristics between parent and child. Finally, an example of the representation applied to a symbolic regression problem is given, which is also used to justify the use of small populations and to highlight the need to be specific with the choice of language subset used.

2.2 Requirements

The following is a list of requirements for a Genetic Programming representation. It starts with some requirements for Genetic Algorithm representations in general and then moves on to some more specific Genetic Programming requirements, including the requirements of the mapping between the genotype and phenotype where they are separate. These requirements are largely based on analysis of the representations presented in Chapter 1.

Quick Translation — In the case where the genotype and phenotype are separate, every newly created individual in a population needs to be

translated into an executable form for fitness evaluation. Therefore, the mapping of the genotype to the phenotype needs to be efficient. For example, if there were 500 individuals in a generation and the Genetic Program was run for 50 generations, the translation of the genotype to phenotype would occur 25000 times. This can be a significant proportion of the running time of the Genetic Program. In the work by Koza [55] and others who use languages such as LISP, this is not a problem as the genes are stored directly as program fragments. In the work of Banzhaf [4] there is the additional complication of the repair of badly structured individuals, which can be costly in terms of run time.

Simple Genetic Manipulation — To create a new individual from one or more parent individuals it is necessary to use some form of genetic manipulation. This usually takes the form of combining the genes of two or more parent individuals and/or performing some kind of random mutation on the new individual. As this process occurs for all or most newly created individuals the representation needs to allow it to be simple and efficient. The representation used by Koza [55] does not allow the use of simple mutation operators due to the complex structural requirements of the genome. It is even harder to create genetic operators for the representation used by Montana [65]. This highlights the advantage of using one representation (the genotype) for genetic manipulation and another (the phenotype) for fitness evaluation.

Inheritable Characteristics — One of the main reasons why Genetic Algorithms work is the principal of inheritance. This allows successful characteristics in individuals to be propagated through multiple generations. Therefore, it is important that the individuals being evolved

are represented in such a way that when a set of genes is passed on to the offspring of the individual, characteristics of the parents are preserved. If the genotype and phenotype are to be separated, then there needs to be a fairly direct relationship between the two in order for phenotypic characteristics to be inheritable. Koza [55] has this property because the genotype and phenotype are the same and, therefore, the child phenotypes are constructed directly from parts of the parent phenotypes. For representations such as Grammatical Evolution [78], there is not a direct mapping between the parent and child phenotypes and, therefore, crucial characteristics from the parents can be lost when the child is generated.

Minimal Solution Space — In general, the smaller the solution space, the faster the Genetic Program will be able to find a solution to the given problem. Alternatively, the larger the percentage of all possible genomes that correspond to good solutions the faster the Genetic Program will find one. However, the solution space should not unduly restrict the range of possible solutions to the problem. The size of the solution space, in Genetic Programming, may be controlled by restricting the language subset available to the Genetic Program. This may be part of the use of the representation rather than the representation itself. For example, in Grammatical Evolution [78] the solution space is dependent on the BNF grammar given to the system by the user. An additional factor which can have an effect is the shape of the solution space. If for example there are many local minima (or maxima) then it may be a more difficult space to search for the Genetic Algorithm.

Maintain Syntactic Correctness — The solution space is restricted also

by only allowing syntactically correct programs (phenotypes) to be generated. This rules out a large number of programs that are badly formed. Representations, such as Koza's [55], needed complex genetic operators to maintain syntactic correctness.

Limit Execution Errors — As well as errors in the syntax of the programs, other errors such as illegal array indexing and variable overflow can cause problems during the fitness evaluation. Montana [65] had problems with his system, in that very few of the initially generated population of programs were correctly typed. It therefore took a long time to find initial viable programs before they could be improved to solve the task set. These problems need to be avoided where possible. In addition, problems such as infinite loops can disrupt the fitness evaluation process and are especially difficult to deal with when the programs are being tested in their natural environment rather than with limited runtime or through emulation.

Consistent Genotype to Phenotype Mapping — In the cases where the genotype and phenotype are separate, it is essential that a given genotype always maps to the same phenotype, in order to result in a deterministic and robust fitness evaluation. Paterson and Livesey [69] suggested that one possible approach in their representation, when the genome ran out of genes in the mapping process, was to randomly generate the rest of the phenotype. This approach is not very good from the perspective of inheritable characteristics.

To summarize, a representation is required that has a separate genotype and phenotype, where the genotype is a simple representation for genetic operators, which has no special constraints and the phenotype is a program

in the target language. Every genotype should only map to syntactically correct programs in a concise language subset and where possible the language subset should be restricted to avoid problems such as infinite loops. Most importantly, the mapping between the genotype and phenotype must be simple and fairly direct, so that the characteristics in the child phenotype can be inherited from the parent phenotype during the genetic manipulation.

2.3 Representation

This section describes the representation and is divided into six subsections. Sections 2.3.1 and 2.3.2 describe the genotype and phenotype. Section 2.3.3 describes the mapping between the genotype and phenotype. Section 2.3.4 describes useful extensions to the approach. Section 2.3.5 describes the use of wrappers around the evolved code and finally Section 2.3.6 gives an example individual and the mapping from its genotype to the phenotype.

2.3.1 Genotype

The genotype for the genetic program is stored as a simple string (or list) of integers. The integers used in all the examples in this thesis are 8-bit (ranging between 0 and 255) but any size integer is acceptable as long as it satisfies the requirements of the mapping process. Representing the individuals as a string of integers simplifies the process of genetic manipulation, crossover and mutation.

2.3.2 Phenotype

The phenotype, to which the genotype maps, is a program written in a subset of some language, in the case of the examples in this thesis, Perl [86]. There are various reasons for using the Perl language. Perl is an interpreted language, meaning that it is not necessary to compile the programs that are evolved for fitness evaluation. Perl is also capable of executing program statements that are generated during the running of a program, which means that the evolved programs don't have to be run externally to the GA. One final feature of Perl that is an advantage in GP is that it has good error handling and recovery, so if an evolved program does not work properly it won't affect the rest of the GP.

The subset of the language chosen for a particular problem can easily be designed with certain semantic constraints, such as avoiding infinite loops. For example, only include restricted 'For' loops, where the counter variable can't change within the body of the loop.

2.3.3 Mapping the Genotype to the Phenotype

The mapping between the genotype and phenotype ensures that all genotypes map to a syntactically correct program in the required language.

The mapping starts by dividing the genotype into fixed-length blocks of genes, each of which represents one program statement. The length of the gene blocks is dependent on the statement type that requires the most information. Each block is interpreted independently of the others. So, two identical gene blocks in different places in the sequence will be interpreted the same way. It can easily be seen that if all of the blocks are the same size,

and they are interpreted independently, then when a block is inherited by a child individual it will be in the same place and therefore be interpreted the same way. This ensures the inheritance of phenotypical characteristics even though a separated genotype and phenotype is being used. In addition, this method of translating the genotype to the phenotype ensures that a complete program will be generated without running out of genes as in the work of Paterson and Livesey [69] or Ryan et al. [78].

The first gene in each block represents the type of statement. The statement type is decided by taking the modulo of the gene value and the number of different program statements. For example, if there were four statement types and the gene value was 23 then 23 mod 4 = 3, so the fourth statement (index number 3) would be chosen.

Each statement type uses the remaining genes in different ways. For example, an 'Addition' statement would require one variable to assign the result to and two variables to add together. Any remaining genes in the block are redundant. Figure 2.1 shows an example of the mapping from a gene block to a program statement and Table 2.2 shows a list of possible statements, what the remaining genes are used for and the form in which the statement is presented in the target language.

2.3.4 Extensions

In addition to the basic mapping, there are a few useful additions to be able to evolve reasonable programs. The first is the need for an ability to provide nested structures, such as looping and branching, without losing the inheritance features of the current mapping. In this work it is achieved by having a statement type e.g. a 'For' statement, which has a corresponding

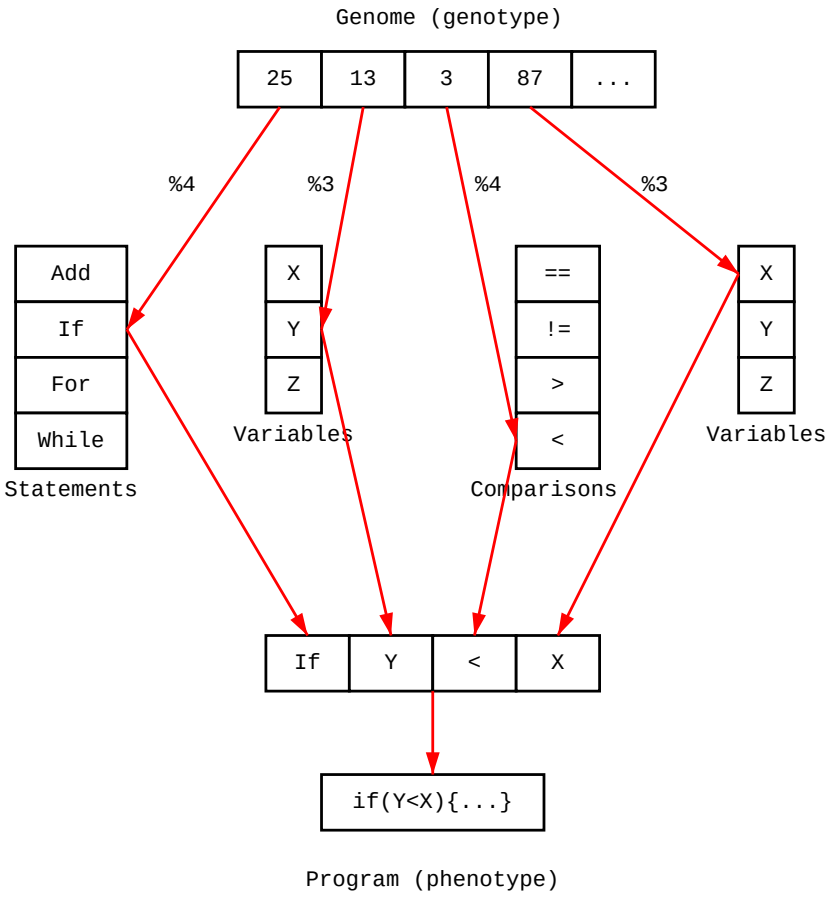


Figure 2.1: Example mapping from individual gene block to program statement

'End' statement. All statements in between these two statements are then nested within the loop or branch. Although a mutation to the statement type, of the loop or branch statement, would change the structure of the subsequent code, the meaning of the individual statements will still be preserved. Any remaining nested structures that haven't been terminated when the end of the genome is reached can then be automatically terminated.

An additional feature that can be used is to distinguish between variables that are read-only and read/write, so that any variables that should not be changed cannot be assigned new values. In practice two sets of variables are stored, one is all of the variables that can be read and the other is all variables that can be assigned new values (therefore read/write variables appear in both sets). For example, this may include loop counters that should only be changed by the loop statement, or variables such as a list of integers that need to be summed. If the list were changed during the execution then the sum might not be accurate for the given list. As well as separating variables by access permissions, it is also possible to separate by type. For example, a list of integers and a list of floats can be kept separate and the statements designed to preserve type correctness.

One final extension that is worth mentioning is the use of a counter to limit the running time of the code. This only needs to be incremented each iteration of a loop and can be used to terminate execution of excessively long programs. For the work in this thesis, the Perl 'eval' function is used to execute the evolved programs and this sets a variable with an error message when there is an unnatural termination of the execution. This can be used to detect errors and also for limiting the execution time of a program.

Table 2.1: Example Genotype

28	34	64	124	127	130	33	83	201	5	41	50	201	9	69	73

Table 2.2: Statement type, type of additional genes, and form of statement

Index	Statement	Additional Genes	Format
0	Assign	variable,variable	`G1 = G2;`
1	Multiply	variable,variable,variable	`G1 = G2 * G3;`
2	If	variable,comparison,variable	`if(G1 G2 G3){`
3	For	variable,variable	`for G1 (0..G2){`
4	End		}

2.3.5 Wrapper

It will usually be convenient to add some header and footer code to the evolved code for the purposes of declaring variables, receiving data passed to the evolved code and returning data after the code has been executed. This could be included as part of the evolution process but would make the problem much harder without any real benefit. For example, see Listing 2.1. This extra code is used in the experiments to allow the use of the Perl 'eval' function to test the evolved programs.

2.3.6 Example Individual

As an example of the mapping from the genotype to the phenotype using the above method, a function to calculate the factorial of a number is presented.

Table 2.1 shows the genotype (the list of integers). This genotype is converted using the statements listed in Table 2.2 and the additional genes are translated using Tables 2.3 and 2.4. As can be seen from Table 2.2, the

Table 2.3: List of variables

Index	Variable
0	`$n`
1	`$fact`
2	`$count`
3	`$zero`

Table 2.4: List of comparison operators

Index	Comparison
0	`==`
1	`!=`
2	`>`
3	`<`

most additional genes required by a statement is three. Therefore, the length of each gene block will be four (to include the choice of statement).

The first block of genes starts with the value 28. This represents the statement type being used. In this case there are five types of statements, so 28 mod 5 = 3 means that the statement is a 'For' (index 3). The 'For' statement requires the use of two more genes to choose from the 'variable' list. The 'variable' list has four elements, therefore, 34 mod 4 = 2 and 64 mod 4 = 0 give the variables `$count` and `$n`. All together this gives the loop header `for $count (0..$n){`. The gene 124 is redundant. The rest of the gene blocks are decoded in the same way (see Table 2.5).

Finally, any missing closing braces are automatically added, along with the wrapper (header and footer) code, to create the complete phenotype. This is shown in Listing 2.1.

Table 2.5: Conversion of Genotype to Phenotype

Genes	Modulo	Statement	Code
28,34,64,124	3,2,0,x	For	`for $count (0..$n){`
127,130,33,83	2,2,1,3	If	`if($count != $zero){`
201,5,41,50	1,1,1,2	Multiply	`$fact = $fact * $count;`
94,231,0,13	4,x,x,x	End	}

Listing 2.1: The entire phenotype, including header and footer

```
# Header
my $n = $ARGV[0];
my $fact = 1;
my $count = 0;
my $zero = 0;

# Evolved Code
for $count (0..$n){
  if($n != $z){
    $fact = $fact * $count;
  }
}

# Footer
return $fact;
```

Table 2.6: List of statements for padding test

Index	Statement	Additional Genes	Format
0	Print	variable	`print G1`
1	For	variable,variable	`for G1 (0..G2){`
2	Add	variable,variable,variable	`G1 = G2 + G3;`

Table 2.7: List of variables for padding test

Index	Variable
0	`$x`
1	`$y`
2	`$z`

2.4 Comparison of Padded and Unpadded Representation

This section investigates the fixed-length gene blocks (padded with redundant genes) in comparison with variable-length gene blocks (unpadded), to examine the preservation of characteristics after mutation and crossover with another individual. The simple set of statements listed in Table 2.6 and the set of variables listed in Table 2.7 are used to map the genotypes to the phenotypes with the method presented in Section 2.3.3.

The first experiment is to compare how the padded version of an individual changes under mutation in comparison with an unpadded individual. Figure 2.2a shows an example individual with fixed-length gene blocks representing the statement and Figure 2.3a shows the same individual without the redundant genes. The 'G' represents an unused gene in the padded gen-

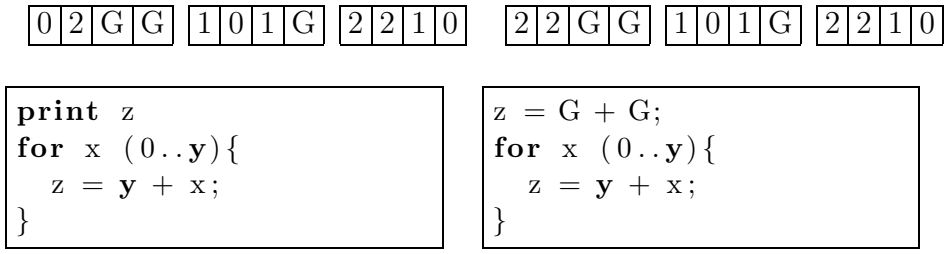


Figure 2.2: (a)Parent 1 (Padded), (b)Parent 1 (Padded) Mutated

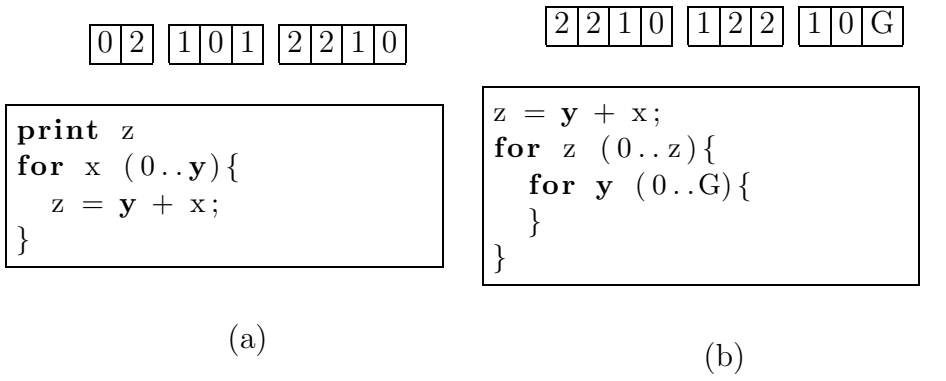
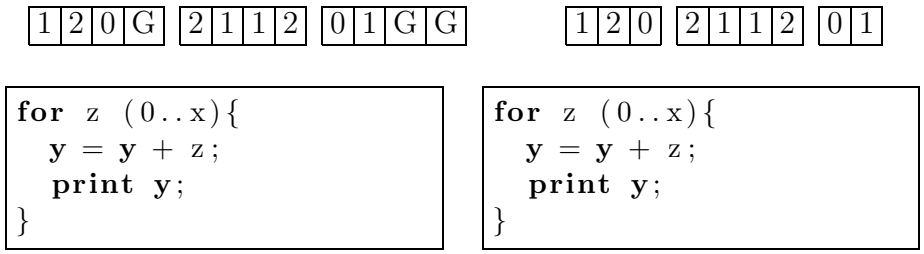


Figure 2.3: (a)Parent 1 (Unpadded), (b)Parent 1 (Unpadded) Mutated



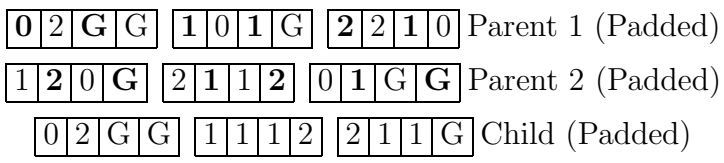
(a) (b)

Figure 2.4: (a)Parent 2 (Padded), (b)Parent 2 (Unpadded)

otype, although these genes may be used after mutation or crossover and are shown in the phenotypes as 'G' when used. The mapping for the unpadded individual's genotype to phenotype just uses the relevant number of genes for each statement and starts the next statement immediately afterwards.

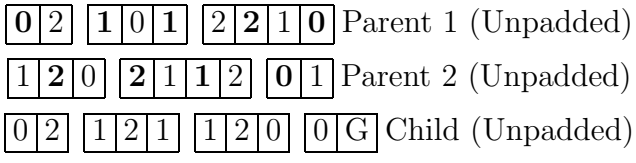
Figure 2.2b shows the first individual (Figure 2.2a) after a mutation of the first gene (from 0 to 2). It can be seen that only the first statement of the phenotype has changed and the rest is identical to the pre-mutation version. However, Figure 2.3b shows the unpadded individual (Figure 2.3a) after the same mutation. The phenotype of the individual is now completely different, very little has been preserved from the original individual. This would not be good from the perspective of the evolution as good characteristics, which caused the individual to be selected for reproduction, are lost whereas with the padded version most of the characteristics are preserved.

This problem would be expected to be even more pronounced when using crossover, as there is much more change when the individuals are combined. Figures 2.4a and 2.4b show the padded and unpadded versions of a second individual, which both map to the same phenotype. When the padded versions of Parent 1 (Figure 2.2a) and Parent 2 (Figure 2.4a) are combined using crossover (taking alternate genes starting with the first individual in this case)




```
print z;
for y (0..y){
  y = y + x;
}
```

Figure 2.5: Crossover Parent1 and Parent2 (Padded)




```
print z;
for z (0..y){
  for z (0..x){
    print G;
  }
}
```

Figure 2.6: Crossover Parent1 and Parent2 (Unpadded)

the individual in Figure 2.5 is created. This individual looks quite similar to the first parent, as the main statement type gene is always taken from this individual (in this example) because the gene-length is even. When the unpadded individuals (Figures 2.3a and 2.4b) are combined in the same way as the padded individuals, Figure 2.6 is produced. Apart from maintaining the statement type of the first individual, it is completely different to either parent. The final gene 'G' in Figure 2.6 represents an extra gene required. The alternative is to not use any gene block with insufficient genes, however, this is not an issue with the padded version.

In conclusion, the small examples shown suggest that inheritable characteristics are much more likely to be preserved when using fixed-length gene blocks to represent individual statements. However, mutation and crossover can still give variation to the child individuals without losing similarity to the parents.

2.5 Example Problem: Symbolic Regression

The following example problem is used to show the representation in action. In addition, it is used to justify the choice of small population sizes used in the experiments in this thesis and to highlight the value of choosing relevant language subsets.

The problem is a symbolic regression, which aims to evolve a program to perform the calculation $x^4 + x^3 + x^2 + x$. This problem has also been used in the work of Koza [55] and Ryan et al. [78]. Unfortunately, they do not give a detailed list of their results.

The Genetic Algorithm used was that described in Chapter 1, Section 1.5.

Table 2.8: List of statements for the symbolic regression

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	variable,variable	`G1 = G2;`
2	Add	variable,variable,variable	`G1 = G2 + G3;`
3	Sub	variable,variable,variable	`G1 = G2 - G3;`
4	Mul	variable,variable,variable	`G1 = G2 * G3;`
5	Div	variable,variable,variable	`G1 = G2 / G3 if(G3 != 0);`
6	Sin	variable,variable	`G1 = Sin(G2);`
7	Cos	variable,variable	`G1 = Cos(G2);`
8	Exp	variable,variable,variable	`G1 = G2 ** G3;`
9	Log	variable,variable	`G1 = Log(G2) if(G2 != 0);`

Table 2.9: List of variables for the symbolic regression

Index	Variable
0	`$x`
1	`$tmp`

The language subset used with the representation is shown in Table 2.8, and is similar to that used by Koza [55] with adjustments to compensate for not using a tree structure (Koza did not use the 'Assign' statement and only used the variable `X`). The genome used was 40 genes long. This value was chosen as it appears to be large enough to cover most solutions. In addition, a 'Null' statement is included to allow for variation in the length of the programs evolved. The Genetic Algorithm was run with three different configurations. Each configuration was run ten times with different random number seeds.

Population 500 — This is the population size used by Koza [55]. The mutation rate used was 1 gene in every 80 and the maximum number of generations was 51.

Population 7 — This experiment was run exactly the same way as the previous one except that the population size was 7 and the mutation rate was 1 gene in 20. The maximum number of generations was 3642, which is equivalent to 51 with a population size of 500.

Population 7 **and Minimal Language Subset** — The last experiment was run exactly the same as the previous one except that only the 'Null', 'Assign', 'Add', and 'Mul' statements were used.

2.5.1 Fitness Evaluation

Table 2.10 shows the list of test inputs with the expected outputs used to evaluate the evolved programs. The fitness of an individual is calculated as the sum of the absolute difference between the expected output and the actual output returned by the evolved program (for each test input). This value is then normalized, so the higher the score the better the individual

Table 2.10: Test input and expected output for symbolic regression

Input	Expected Output
-0.22	-0.17990544
0.85	2.70863125
-0.58	-0.32554704
0.97	3.70886581
0.43	0.72859501
0.17	0.20464821
-0.78	-0.27600144
0.56	1.14756096
0.07	0.07526701
0.73	1.93589941
-0.56	-0.32367104
-0.55	-0.32236875
0.28	0.38649856
-0.82	-0.24684624
0.57	1.18565301
0.9	3.0951
0.51	0.97040301
-0.86	-0.20944784
0.62	1.39049136
-0.2	-0.1664

Table 2.11: Results for the symbolic regression with population 500

Seed	Fitness (%)	Evaluations	Time
0	94.1	25500	4m27s
1	93.7	25500	4m25s
2	94.5	25500	4m25s
3	94.5	25500	4m22s
5	94.1	25500	6m38s
7	94.5	25500	4m53s
11	94.5	25500	4m23s
13	95.4	25500	4m23s
17	94.5	25500	4m21s
19	94.5	25500	4m30s

(this makes parent selection easier). The normalization procedure involved subtracting the raw fitness value from 100. If the normalized fitness value is less than 1 it is rounded up.

2.5.2 Results

Table 2.11 shows the results from the first experiment (with population 500). It can be seen that none of the runs produce a completely correct solution within the short time allowed. All of the solutions are approximately 94% fit.

Table 2.12 shows the results of the second experiment (with population 7). In this experiment (which is limited to run for the same number of fitness evaluations as the first experiment) six out of the ten runs produce a fully fit individual. All four runs that did not produce a fully fit individual, in the time allowed, produced a much better fitness individual than all the failed runs of the first experiment. This suggests that a smaller population size is more appropriate for problems using this representation. Listing 2.2

Table 2.12: Results for the symbolic regression with population 7

Seed	Fitness (%)	Evaluations	Time
0	100	10850	1m35s
1	98.2	25494	3m44s
2	100	2688	23s
3	100	2121	17s
5	100	3934	33s
7	98.6	25494	3m43s
11	100	2450	21s
13	98.1	25494	3m46s
17	100	3402	28s
19	99.8	25494	3m50s

Listing 2.2: Example solution found from experiment 2, Seed 0

```
# Header
my $x = $test[$t];
my $tmp = 0;

# Evolved Code
$tmp = $x;
$tmp = $tmp;        # Redundant
$x = $x;            # Redundant
$tmp = $x * $x;
$x = $x + $tmp;
$tmp = $tmp;        # Redundant
$x = $x;            # Redundant
$tmp = $tmp * $x;
$x = $tmp + $x;
$x = $x;            # Redundant

# Footer
return $x;
```

Table 2.13: Results for the symbolic regression with population 7 and minimal language subset

Seed	Fitness (%)	Evaluations	Time
0	100	1428	12s
1	100	847	6s
2	100	483	4s
3	100	1610	12s
5	100	1197	9s
7	100	938	7s
11	100	3437	26s
13	100	3850	29s
17	100	224	2s
19	100	2772	21s

shows an example of a fully fit individual from the first run (Seed 0) of this experiment.

Table 2.13 shows the results of the final experiment. In this experiment all of the runs produced a fully fit individual in a very short time. This suggests that it is much better to have a more specific language subset where possible. Listing 2.3 shows an example individual from this experiment. It should also be noted that it would be very easy to automatically remove redundant code from the evolved individuals e.g. `x=x;` or `if(x!=x){...}`. In some circumstances the statement `x=y;` can be removed from the sequence `x=y; x=z;` as long as `z` is not dependent on the value of `x`.

2.6 Summary and Conclusions

The following list describes how the new representation fits in with the requirements set out.

Listing 2.3: Example solution found from experiment 3, Seed 0

```
# Header
my $x = $test[$t];
my $tmp = 0;

# Evolved Code
$tmp = $x * $x;
$x = $x + $tmp;
$tmp = $x * $tmp;
$tmp = $x + $tmp;
$x = $x;              # Redundant
$tmp = $tmp;          # Redundant
$x = $tmp * $x;       # Redundant
$x = $tmp * $tmp;     # Redundant
$x = $tmp;

# Footer
return $x;
```

Quick Translation — The representation implements a fairly direct mapping between the genotype and phenotype, which can be performed in linear time.

Simple Genetic Manipulation — As the genotype is linear and there are no special requirements for crossover and mutation, ordinary genetic operators can be used.

Inheritable Characteristics — The representation implements statements as fixed-length gene blocks that are independent of each other. Therefore, each gene block always maps to the same statement in both the parent and child.

Minimal Solution Space — This can be aided by keeping the number of statement types to a minimum. Although, if the programs to be generated are long then the solution space will be large anyway.

Maintain Syntactic Correctness — The mapping from genotype to phenotype ensures that all lists of integers map successfully to a syntactically correct program, as long as the language subset provided is itself syntactically correct.

Limit Execution Errors — This can also be achieved by careful construction of the language subset being used.

Consistent Genotype to Phenotype Mapping — A list of integers always maps to the same program, although multiple genotypes may map to the same phenotype.

To summarize, a list of requirements for a GP representation was put forward and then a new representation that followed the requirements was developed. The choice of fixed-length gene blocks was then justified and a series of experiments were run to justify the use of small population sizes with the representation and to highlight the advantage of keeping the language subset to a minimum.

Chapter 3

Using Formal Specifications to Create Fitness Functions

3.1 Introduction

This chapter focuses on the construction of fitness functions for use with the new representation, presented in the previous chapter.

One way a fitness function for Genetic Programming can be constructed is to use a set of sample inputs for a problem and compare the resultant outputs from the evolved function with the expected outputs. The fitness function can also be some evaluation function which was "hand-crafted" in which case it is difficult to guarantee that all features of the problem have been covered, especially for larger problems. While the first method may be suitable for some simple situations, it is unlikely to generate an accurate fitness score, on larger problems, without a very large number of test inputs. Cramer [17] mentions that a major problem in evolving programs is one of "hand-crafting" the evaluation function to give partial credit to a function that does not work

but exhibits some of the relevant behaviour. In [55], for example, Koza handcrafts his fitness functions based on the natural terminology of the problem but gives little justification of his choices. One possible solution, explored in this chapter, is to work from a formal specification for the problem being tackled and then to manipulate this specification into the form of a fitness evaluation function. By using this process, the fitness evaluation function should be able to perform the minimum number of checks on the individual being tested while still covering all aspects of the problem to be solved.

Section 3.2 introduces the area of formal specification and the notation being used. Section 3.3 presents two experiments to compare the performance of fitness functions created from formal specifications against using input/output pairs to evaluate the fitness. Section 3.4 describes the results of the experiments. Finally, the conclusions are summarized.

3.2 Formal Specification

A formal specification of a function is, in its simplest form, a description of the mapping between the inputs and outputs of the function.

The formal specification of a function is made up from four elements; the name, the type, the pre-condition and the post-condition. The name is just a label which represents the function. The type is a specification of the inputs and outputs in terms of the data they can contain, e.g. $\mathbb{Z}^* \rightarrow \mathbb{Z}$, says that the input of the function is a list of integers and the output is an integer. A function must have an unique output for each input. The pre-condition specifies any constraints on the input to the function, e.g. $\text{pre} - myfunc(in : \mathbb{Z}) \triangleq in > 10$, specifies that an input to the function *myfunc* must be an integer greater than 10. A value lower than 11 would be an invalid

Table 3.1: A list of symbols and their meanings

Symbol	Meaning
$a \wedge b$	a and b
$a \vee b$	a or b
$a \Rightarrow b$	a implies b
$\forall$	for all
$\exists$	there exists
$\triangleq$	is defined as
$\mathbb{Z}$	the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{P}$	the set of positive integers $\{0, 1, 2, \ldots\}$
$\mathbb{R}$	the set of real numbers
$a : T$	a is of type T
$\mathbb{Z}^*$	the set of all lists of integers
$L^\frown M$	list concatenation
$\#L$	size of list
$\langle x \rangle$	the list containing only x

input. When the function is valid for all inputs within the constraints of the input data type, the pre-condition is just defined as *True*. Finally, the post-condition is a boolean function that, given an input that satisfies the pre-condition and an output, returns a True/False value which represents whether the input is validly mapped to the output i.e. whether the input/output pair is valid.

Table 3.1 summarizes some of the symbols that can be used in formal specifications. The creation of specifications is beyond the scope of this section but is covered in [19]. To convert the specification into a fitness function, various manipulations of the specification are required. Cooke, in his book *Constructing Correct Software* [19], describes various techniques for converting the specification into a program which meets the specification. This can be quite hard but in the context of this work it is only required to convert the specification into a program that checks the output of a function

$SquareRoot$
type: $\mathbb{R} \rightarrow \mathbb{R}$
pre-$SquareRoot(in) \triangleq in \geq 0$
post-$SquareRoot(in, out) \triangleq (out^2 = in) \wedge (out \geq 0)$

Figure 3.1: Specification of *SquareRoot*

to see if it meets the requirements of the specification. This is a much simpler problem.

As an example of the conversion of a specification into a fitness function, take the function *SquareRoot* as defined in Figure 3.1. In words, the post-condition of the specification says:

> the output of the function squared is equal to the input and the output is greater than or equal to zero.

This specification is relatively simple to convert into a fitness function but much harder to convert into a function which carries out the specified task.

The first stage of the conversion into the fitness function is to separate the expressions in the post-condition, which are joined by the *logical and* operator ($\wedge$). The first expression

$$out^2 = in$$

can be converted directly into the target programming language (in this case Perl [86]).

```
($out*$out) == $in
```

Some method of keeping track of the fitness is also required. Therefore a variable `$fitness` is introduced to store this information. An *if* statement is added to increment the fitness if the condition is true

```
if(($out*$out) == $in){$fitness++;}
```

The second expression can be converted just as easily.

$$out \geq 0$$

Again this expression can be converted directly into code

```
$out >= 0
```

and then it is surrounded by an *if* statement to keep track of the fitness.

```
if($out >= 0){$fitness++;}
```

There are two ways to arrange the above generated code fragments in the fitness test, either:

```
if($out*$out) == $in){
  $fitness++;
  if($out >= 0){
    $fitness++;
  }
}
```

or:

```
if(($out*$out) == $in){$fitness++;}
if($out >= 0){$fitness++;}
```

In the first nested arrangement, the testing stops after the first failed check (the statements can be nested in different orders) whereas in the second, the total number of checks passed is calculated. The second method is the one used in this thesis. In addition, the fitness of each test could be weighted to give one test more impact on the overall fitness. However, all the fitness increments are identical in this thesis.

Finally, when an individual is referred to as 'fully fit', it still has the traditional GA/GP meaning. That is to say, the individual is correct for all of the test inputs and not necessarily completely correct to the specification for all possible inputs.

3.3 The Experiments

This section will discuss the two example problems, which have been selected to investigate the difference between fitness testing using input/output pairs and fitness functions derived from the formal specification of the problem. Both the input/output pairs and the fitness function created from the formal specification are tested with the same inputs.

3.3.1 The Problems

The problems chosen are *Listmax* and *Reverse*. *Listmax* takes a list of integers as input and returns a single integer value containing the largest element from the list. The specification for *Listmax* is given in Figure 3.2. In words, the post-condition for *Listmax* can be read as:

> the result m must be in the input list L and for any integer z, if z is in the list L then z must be less than or equal to m.

Reverse takes a list of integers as input and returns a list containing the same elements in reverse order. The specification for *Reverse* is given in Figure 3.3. In words, the post-condition for *Reverse* can be read as

> the length of the *out_list* is the same as the length of the *in_list* and for any positive integer n less than the length of the *in_list* there exists an element x and four lists, L_1, L_2, L_3, L_4 such that L_1 concatenated with the list containing x concatenated with L_2 recreates *in_list* and L_3 concatenated with the list containing x concatenated with L_4 recreates *out_list* and the length of L_1 is the same as the length of L_4 and the length of L_1 equals n.

$$
\begin{array}{l}
\textit{Listmax} \\
\text{type: } \mathbb{Z}^* \rightarrow \mathbb{Z} \\
\text{pre-}\textit{Listmax}(L) \triangleq \#L \neq 0 \\
\text{post-}\textit{Listmax}(L, m) \triangleq m \text{ in } L \wedge (\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \leq m) \\
\qquad \text{where } (x \text{ in } L) \triangleq (\exists L_1, L_2 : \mathbb{Z}^*)(L = L_1 \frown \langle x \rangle \frown L_2)
\end{array}
$$

Figure 3.2: Specification of *Listmax*

$$
\begin{array}{l}
\textit{Reverse} \\
\text{type: } \mathbb{Z}^* \rightarrow \mathbb{Z}^* \\
\text{pre-}\textit{reverse}(\textit{in\_list}) \triangleq \text{True} \\
\text{post-}\textit{reverse}(\textit{in\_list}, \textit{out\_list}) \triangleq \#\textit{in\_list} = \#\textit{out\_list} \\
\quad \wedge (\forall n : \mathbb{P})(n < \#\textit{in\_list} \Rightarrow \\
\qquad (\exists x : \mathbb{Z}, L_1, L_2, L_3, L_4 : \mathbb{Z}^*)(\ \textit{in\_list} = L_1 \frown \langle x \rangle \frown L_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \textit{out\_list} = L_3 \frown \langle x \rangle \frown L_4 \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = \#L_4 \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = n))
\end{array}
$$

Figure 3.3: Specification of *Reverse*

These two problems are chosen to represent a fairly simple and a slightly more complex specification.

3.3.2 The Fitness Functions

This section gives the fitness function for input/output pairs and then goes on to show the conversion between the formal specifications given in Figures 3.2 and 3.3 and their corresponding fitness functions.

Input/Output Pairs

The fitness test for the input/output pairs method takes the result of the individual being tested for each input and compares the result with the expected output. If the output is a single value then the fitness is incremented

if it is the same as the expected output and remains the same if it is not. If the output is a list then the fitness is incremented for every element in the individual's output list that matches the corresponding element in the expected output list.

Listmax

The first step in the process of converting the specification, or more accurately the post-condition (as only inputs that satisfy the pre-condition will be used for testing), into a fitness function is to separate out the expressions joined by the $\wedge$ (*logical and*) operator. In more complex specifications it may be necessary to rearrange the post-condition into conjunctive normal form. This does not include $\wedge$ operators in sub-expressions at this stage. Each of these expressions is dealt with as a separate check on the individual being evaluated. The aim is now to translate each expression into the target programming language and add some scoring measure. This is usually a fairly direct mapping for simple problems but on occasion further manipulation is necessary.

The first expression in the *Listmax* post-condition is

$$m \text{ in } L$$

This can be directly put into code as

```
isin($m,@L);
```

To this can be added the scoring measure in the form

```
if(isin($m,@L)){$fitness++;}
```

The function `isin` is defined to check if a given variable is in the given list and return a True/False value. The *if* statement surrounds the boolean

expression to increment the fitness if the statement is true about the output from the current individual being evaluated.

The second expression in the *Listmax* post-condition is slightly more challenging as it requires some manipulation of the expression before putting it into code. The expression

$$(\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \leq m)$$

covers the domain of all integers but only says something about that case where z is in L. When z is not in L the implication expression is always true regardless of the value of m. This is always a problem with implication because it is a restriction on the domain. In general, only the restricted domain needs to be used, eliminating the requirement for the implication. For use in a fitness test, this expression is refined to be

$$(\forall z \text{ in } L)(z \leq m)$$

This can easily be represented in code as

```
foreach my $z (@L){if($z <= $m){$fitness++;}}
```

again with the *if* statement surrounding the boolean sub-expression to count the number of correct occurrences within the check.

The complete fitness function for *Listmax* is given in Listing 3.1

Reverse

Reverse, as specified in Figure 3.3, is now dealt with. Again the expressions separated by the $\wedge$ operator are separated and dealt with individually.

Listing 3.1: The fitness function for *Listmax*

```
sub fitnesstest{
  my ($m, @L) = @_;
  my $fitness = 0;

  if(isin($m,@L)){$fitness++;}
  foreach my $z (@L){if($z <= $m){$fitness++;}}

  return $fitness;
}
```

The first expression in the post-condition for *Reverse* equates the size of two lists

$$\#in\_list = \#out\_list$$

This can easily be translated directly into the code

```
if(scalar(@in_list) == scalar(@out_list)){$fitness++;}
```

The second expression in the *Reverse* post-condition is more problematic. Firstly, taking the surrounding quantification expression

$$(\forall n : \mathbb{P})(n < \#in\_list \Rightarrow \ldots)$$

This can be expressed in code as the loop statement

```
for(my $n=0; $n < scalar(@in_list); $n++){...}
```

Next, taking the subexpression surrounded by the quantification expression

$$\begin{aligned}(\exists x : \mathbb{Z}, L_1, L_2, L_3, L_4 : \mathbb{Z}^*)(\ & in\_list = L_1 \frown \langle x \rangle \frown L_2 \\ & \wedge out\_list = L_3 \frown \langle x \rangle \frown L_4 \\ & \wedge \#L_1 = \#L_4 \\ & \wedge \#L_1 = n)\end{aligned}$$

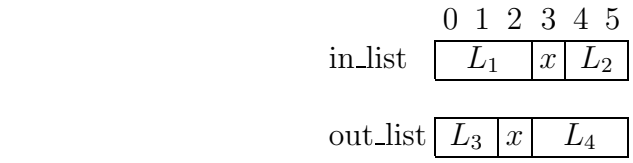


Figure 3.4: Comparison of the location of x in the input and output lists when $n = 3$

The aim is to replace occurrences of x and L_1, L_2, L_3, L_4 by references to *in_list* and n using indexing. Again, the expressions, joined by the $\wedge$ operator, of the sub-expression are separated. First consider

$$in\_list = L_1 \frown \langle x \rangle \frown L_2$$

As Figure 3.4 shows this can be replaced by the expression

$$in\_list[\#L_1] = x$$

where $L[i]$ is the element at index i of the list L. The second expression

$$out\_list = L_3 \frown \langle x \rangle \frown L_4$$

can therefore also be replaced by

$$out\_list[\#L_3] = x$$

as shown in Figure 3.4, and given $(\#in\_list = \#out\_list)$ and $(\#L_1 = \#L_4)$

$$out\_list[\#L_2] = x$$

this can be rearranged as

$$out\_list[\#in\_list - \#L_1 - 1] = x$$

Given $(\#L_1 = n)$ the previous expressions can be changed to

$$in\_list[n] = x \wedge out\_list[\#in\_list - n - 1] = x$$

Listing 3.2: The fitness function for *Reverse*

```
sub fitnesstest{
  my @in_list = @{$_[0]};
  my @out_list = @{$_[1]};
  my $fitness = 0;

  if(scalar(@in_list) == scalar(@out_list)){$fitness++;}
  for(my $n=0; $n<scalar(@in_list); $n++){
    if($in_list[$n] == $out_list[$#in_list-$n]){$fitness
      ++;}
  }

  return $fitness;
}
```

and hence

$$in\_list[n] = out\_list[\#in\_list - n - 1]$$

This can then be converted to the code

```
if($in_list[$n] == $out_list[$#in_list-$n]){$fitness++;}
```

where `$#in_list` refers to the last index of *in_list* rather than its length.

Therefore, putting both parts of the expression together gives

```
for(my $n=0; $n<scalar(@in_list); $n++){
  if($in_list[$n] == $out_list[$#in_list-$n]){$fitness++;}
}
```

Putting all the parts together gives the fitness function for *Reverse* shown in Listing 3.2.

3.3.3 The Genetic Algorithm

The details of the *Listmax* and *Reverse* list experiments can be found in Chapter 4, Sections 4.4 and 4.6. The Input/Output test is run the same way

Table 3.2: Summary of the results for *Listmax* and *Reverse* (with mean and standard deviation)

	ListMax				Reverse			
	Generations		Time		Generations		Time	
Fitness	Mean	S.D.	Mean	S.D.	Mean	S.D.	Mean	S.D.
Input/Output	333.40	219.45	5.90s	5.19	134.67	127.18	6.55s	6.09
Formal	182.14	133.99	4.24s	3.19	105.28	98.74	5.17s	4.87

but replacing the fitness function. Each experiment is run 100 times with a different random seed based on the first 100 prime numbers.

The population size is 7. The individuals are of a fixed length of 40 genes where each gene is an 8-bit integer. Each run of the GP starts from a pseudo-randomly generated population and is terminated after 50000 generations if a solution with the maximum fitness value has not been found.

3.4 Results

Table 3.2 shows a summary of the results from the experiments. The complete set of results can be found in Appendix B. For each test the average number of generations and average time taken to find a solution with the maximum fitness is given.

The results were evaluated using the Microsoft Excel T-Test function and it was found that the formal specification based method produces a fully fit individual in less generations than the Input/Output method with a confidence of 99.9% for the *Listmax* problem and 98.0% confidence for the Reverse problem. In terms of time, it was found that the formal specification based method produced fully fit solutions faster with a confidence of 99.6%

for the *Listmax* problem and 97.2% for the *Reverse* problem.

The results strongly suggest that, for these problems, using the fitness functions created from the formal specification is significantly better than using input/output pairs. The results also suggest that the *Reverse* problem was easier than the *Listmax* problem. This was probably influenced by the choice of language subset, which made the *Reverse* problem easier to solve.

3.5 Summary and Conclusions

In conclusion, the experiments have shown promising initial results by basing the fitness function for Genetic Programming on the formal specification of the problem to be solved. For the experiments shown, a large improvement over using simple input/output pairs was achieved, however, further work is required to establish the performance of using fitness functions created from the formal specification in a wider selection of problems. It is hypothesised that the improved performance using the fitness functions created from the formal specification is due to an increased accuracy in the fitness scoring. For example, whereas the fitness measured by the input/output pairs for *Listmax* would give a fitness of zero for the second highest value in the list, the fitness function created from the formal specification would return quite a high fitness value. In addition, further work is needed on the best way to use the fitness functions created from the formal specification; whether to test an individual completely or until the first non-compliant feature is found. In addition, the proposed method removes the need to write 'ad hoc' functions to test fitness and replaces it with a disciplined alternative that seems to be easy to use.

To summarize, this chapter introduced a new method for the construction

of fitness functions by basing them on the formal specification of the desired function. Two examples were given of the use of fitness functions created from a formal specification compared with the use of simple input/output pairs to assess fitness, with promising results.

Chapter 4

Evolving Some Interesting Functions

4.1 Introduction

This chapter applies the methods previously presented to the evolution of a series of list evaluation and manipulation functions. The particular functions being evolved have some interesting features, which set them apart from traditional GP problems. Firstly, the functions need to achieve a 100% fitness level to be useful. This moves away from the traditional GP optimisation problems, where the idea is to improve upon current results, and just requires search of the solution space for fully fit individuals. Secondly, these functions are commonly used as part of larger programs, so can be used as components to create larger programs after they have been evolved. For example, the *sumlist* function could be incorporated into the statement list for the evolution of the *avelist* function.

The following six sections detail the experiments evolving the following

functions:

Sumlist — find the sum of a list of integers.

Avelist — find the average value of a list of integers.

Listmax — find the largest value in a list of integers.

Listmin — find the smallest value in a list of integers.

Reverse — reverse the ordering of a list of integers.

Sort — sort a list of integers into ascending order.

This set of problems starts with two functions that return numerical results (that do not necessarily appear in the input lists), the second two functions return elements of the input lists and the final two return new lists. This may be thought of as a series of problems of apparent increasing difficulty. The final section summarizes the results.

All of the functions being evolved use the GA presented in Chapter 1, Section 1.5, with a population size of 7 individuals, a mutation rate of 1 gene in 10 and a genome of 40 genes. The GA is run for a maximum of 50000 generations with 10 different random seeds (the first 8 prime numbers, 0 and 1). In addition, the execution time of each individual is limited by keeping count of the number of iterations of loops. If the counter reaches 1000 the program is terminated with an error and receives a minimal fitness value. All of the example evolved programs shown in this chapter are taken from the first run of GP (Seed 0).

$sumlist : \mathbb{Z}^* \rightarrow \mathbb{Z}$
pre-$sumlist(L) \triangleq \#L \neq 0$
post-$sumlist(L, s) \triangleq s = (+x | x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^* | L = L_1 \frown \langle x \rangle \frown L_2)$

Figure 4.1: Specification of *sumlist*

Listing 4.1: The fitness function for *sumlist* (without header)

```
my $t = 0;
foreach my $x (@L){
  $t += $x;
}
$fitness += abs($s-$t);
```

4.2 Sumlist

The first function to be evolved was *sumlist*. This was chosen as a simple starting point and the aim was to evolve a function to find the sum of a list of integers. The specification (shown in Figure 4.1) simply says "the output value s is equal to the sum of the list L". The notation $(+x|x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^*|L = L_1 \frown \langle x \rangle \frown L_2)$, used in the specification, simply means "take each element x in the list L and add it to the other elements in the list". The notation is described in more general terms in [19]. The specification is converted into the function shown in Listing 4.1 using the method presented in Chapter 3. One interesting adjustment to the standard conversion is the use of the absolute difference between the expected and actual results. This replaces the previous method of incrementing the fitness if the values are equal. The adjustment was due to this particular specification only having one test and therefore a very limited hill to climb. However, with the change the hill becomes much bigger and less steep. Initial tests showed that the next function, *avelist*, found great difficulty evolving without the change but when the change was made to the fitness function *avelist* evolved much more

Listing 4.2: Set of test input lists for *sumlist*

```
[4,3,2,1],
[1,2,55,3],
[1,999,2,3],
[71,1,2,3],
[1,2,33],
[100,88,211],
[100,1,2],
[13,7],
[5,55],
[10]
```

easily. The use of absolute differences in values was not used in any of the other functions as there were many more tests. In the only other place among the series of tests that it could have been used (i.e. *reverse*), the performance was good without it. This approach meant that larger values corresponded to worse individuals, therefore the values were normalized by subtracting them from 5000 (with a minimum score of 1) to make parent selection easier.

The list of test inputs used for fitness testing is given in Listing 4.2. The test set includes a variety of different length lists and a variety of orderings of the elements within the lists i.e. the largest and smallest elements are not always in the same location.

Tables 4.1 and 4.2 give the list of statements and the meaning of additional genes. The set of statements used is fairly constrained, although some statements, such as 'Divide', that could be considered unnecessary for the problem were incorporated to see if any interesting use was made of them by the evolution. The statements differentiate between variables that are read only and those that can be assigned new values. For example, the size of the list should not be changed in the evolved function. A read-only variable is used for the counter (G1) in the 'For' statement, so that it would not be

Table 4.1: List of statements used for *sumlist*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	Add	wvars,rvars,rvars	`G1 = G2 + G3;`
3	Subtract	wvars,rvars,rvars	`G1 = G2 - G3;`
4	Multiply	wvars,rvars,rvars	`G1 = G2 * G3;`
5	Divide	wvars,rvars,rvars	`G1 = G2 / G3 if(G3 != 0);`
6	If	rvars,cmp,rvars	`if(G1 G2 G3){`
7	For	rvars,lsize	`for G1 (0..G2){`
8	End		}

Table 4.2: Additional Genes for *sumlist*

Index	wvars	rvars	lsize	cmp
0	`$sum`	`$sum`	`$#list`	`==`
1		`$size`		`!=`
2		`$tmp`		`>`
3		`$list[$tmp%($#list+1)]`		`<`
4				`>=`
5				`<=`

Table 4.3: The results for *sumlist*

Seed	Generations	Time
0	185	9s
1	256	12s
2	21	1s
3	37	2s
5	189	8s
7	166	8s
11	97	4s
13	34	3s
17	15	1s
19	15	1s

changed within the body of the loop. One final interesting feature is the list index (Index 3 of the read-only variables, *rvars* in Table 4.2), which is constrained to only be able to reference elements within the list by taking the modulus of the size of the list. This seemed to be a fairly logical way of constraining the list indexing.

Table 4.3 presents the results for the ten runs of the experiment. All of the runs evolve a fully fit individual within a very small number of generations (and a very short time). Even though all the results are achieved very quickly, there is still some considerable variation in the values. This is the expected result due to the non-deterministic nature of GA and GP.

Listing 4.3 shows an example individual evolved (the complete set of evolved individuals is given in Appendix C). Taking into account that the first two lines of code are redundant, as is the nested 'If' statement and its contents, the resultant program is the same as might be written by a 'real programmer' and will therefore find the sum of any input list.

Listing 4.3: Example solution for *sumlist*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $sum = 0;
my $size = $#list+1;
my $tmp = 0;

# Evolved Code
$sum = $size;
$sum = $tmp;
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp%($#list+1)];
  if($list[$tmp%($#list+1)] < $list[$tmp%($#list+1)]){
    for $tmp (0..$#list){
    }
  }
}

# Footer
return $sum;
```

$$avelist : \mathbb{Z}^* \rightarrow \mathbb{R}$$
$$\text{pre-}avelist(L) \triangleq \#L \neq 0$$
$$\text{post-}avelist(L, s) \triangleq s = (+x | x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^* | L = L_1 \frown \langle x \rangle \frown L_2)/\#L$$

Figure 4.2: Specification of *avelist*

4.3 Avelist

The *avelist* function is a natural progression from *sumlist*. The aim is to find the average (mean) value of a list of integers. The expected result would be the same as that for *sumlist* with a 'Divide' statement on the end. The specification for the problem is given in Figure 4.2 and the fitness function for the problem is given in Listing 4.4. Like the *sumlist* fitness function, the fitness was calculated using the absolute difference between the expected and actual results, as discussed above.

Listing 4.4: The fitness function for *avelist* (without header)

```
my $t = 0;
foreach my $x (@L){
  $t += $x;
}
$fitness += abs($s-($t/($#L+1)));
```

Table 4.4: The results for *avelist*

Seed	Generations	Time
0	141	7s
1	1727	1m17s
2	934	44s
3	165	7s
5	665	29s
7	557	24s
11	42	2s
13	2873	2m12s
17	415	19s
19	460	18s

The experiment was run using the same test data (Listing 4.2) as *sumlist* and also the same set of statements (Table 4.1) and additional gene translations (Table 4.2). The only difference is that the variable `$sum` is replaced by the variable `$ave`.

Table 4.4 gives the list of results for the ten runs of the experiment. The results are slightly slower than those of the *sumlist* function, however, this is to be expected due to the slight increase in problem difficulty (or more accurately, in terms of the GA, a lower proportion of the solution space containing fully fit individuals). All of the runs produced a fully fit individual in a fast time and few generations.

Listing 4.5 shows an example fully fit individual generated. The first

Listing 4.5: Example solution for *avelist*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $ave = 0;
my $size = $#list+1;
my $tmp = 0;

# Evolved Code
$ave = $tmp + $ave;
for $tmp (0..$#list){
}
for $tmp (0..$#list){
  $ave = $list[$tmp%($#list+1)] + $ave;
}
$ave = $ave / $size if($size != 0);

# Footer
return $ave;
```

$$
\begin{aligned}
&\mathit{listmax} : \mathbb{Z}^* \rightarrow \mathbb{Z} \\
&\text{pre-}\mathit{listmax}(L) \triangleq \#L \neq 0 \\
&\text{post-}\mathit{listmax}(L, m) \triangleq m \text{ in } L \wedge (\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \leq m) \\
&\qquad \text{where } (x \text{ in } L) \triangleq (\exists L_1, L_2 : \mathbb{Z}^*)(L = L_1 \frown \langle x \rangle \frown L_2)
\end{aligned}
$$

Figure 4.3: Specification of *listmax*

three lines of code are redundant to the functionality of the program and the remaining code is as would be expected (assuming a valid input to the function) if the function was 'hand-coded'. The complete list of outputs generated is given in Appendix C.

4.4 Listmax

The *listmax* function is a more complex, multipart specification. The aim is to find the largest element of a list of integers. The specification for the function is given in Figure 4.3 and the fitness function derived from that

Listing 4.6: The fitness function for *listmax* (without header)

```
if(isin($m,@L)){$fitness++;}
foreach my $z (@L){
  if($z <= $m){$fitness++;}
}
```

Listing 4.7: Set of test input lists for *listmax*

```
[1,4,2,32,345],
[-42,-34,-12,-235],
[46,0,2,23],
[54,13,1,24,235,35],
[12,245,6]
```

specification is shown in Listing 4.6. The derivation of this fitness function is given in Chapter 3, Section 3.3.2.

The set of test inputs is given in Listing 4.7. The test set includes lists of varying lengths with both positive and negative integers. The number of tests used can affect the performance of the GP from both the perspective of overall time to evaluate fitness and the number of generations required to evolve a fully fit solution. The results, in this case, show that the number of test cases was sufficient.

Tables 4.5 and 4.6 show the list of statements and meaning of additional

Table 4.5: List of statements used for *listmax*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	If	rvars,cmp,rvars	`if(G1 G2 G3){`
3	For	rvars,lsize	`for G1 (0..G2){`
4	End		}

Table 4.6: Additional Genes for *listmax*

Index	wvars	rvars	lsize	cmp
0	`$max`	`$max`	`$#list`	`==`
1		`$tmp1`		`!=`
2		`$tmp2`		`>`
3		`$list[$tmp1%($#list+1)]`		`<`
4		`$list[$tmp2%($#list+1)]`		`>=`
5				`<=`

Table 4.7: The results for *listmax*

Seed	Generations	Time
0	85	3s
1	15	1s
2	192	4s
3	56	1s
5	59	1s
7	236	6s
11	99	3s
13	10	1s
17	157	4s
19	111	2s

genes for the *listmax* problem. As the *listmax* function only returns a value from the list of integers no arithmetic statements are included in the statement list. As with *sumlist* and *avelist* the list indexes are constrained to only index valid list elements.

The results for the ten runs of the *listmax* experiment are given in Table 4.7. All ten runs produce a fully fit individual in a very short time. The performance is similar to that of *sumlist* and faster than *avelist*.

Listing 4.8 gives an example fully fit individual evolved by the GP. This

Listing 4.8: Example solution for *listmax*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $max = 0;
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
for $tmp2 (0..$#list){
  if($list[$tmp1%($#list+1)] <= $list[$tmp2%($#list+1)]){
    $max = $list[$tmp2%($#list+1)];
    if($tmp2 >= $max){
      if($list[$tmp1%($#list+1)] == $max){
        $max = $list[$tmp2%($#list+1)];
        for $tmp1 (0..$#list){
        }
      }
    }
  }
}

# Footer
return $max;
```

$$listmin : \mathbb{Z}^* \rightarrow \mathbb{Z}$$
$$\text{pre-}listmin(L) \triangleq \#L \neq 0$$
$$\text{post-}listmin(L, m) \triangleq m \text{ in } L \wedge (\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \geq m)$$
$$\text{where } (x \text{ in } L) \triangleq (\exists L_1, L_2 : \mathbb{Z}^*)(L = L_1 \frown \langle x \rangle \frown L_2)$$

Figure 4.4: Specification of *listmin*

Listing 4.9: The fitness function for *listmin* (without header)

```
$fitness++ if(isin($m,@L));
foreach my $z (@L){
  $fitness++ if($z >= $m);
}
```

particular example works with the test input set and other similar lists but not in the general case. For example, the list `[-1,3,2,1]` returns the value 1. However, some of the functions evolved do return the correct value in the general case (this was determined by inspection of the functions). Appendix C lists the generated code for each run of the experiment.

4.5 Listmin

The *listmin* function is very similar to the *listmax* function. This function was evolved to see if the slight change in requirements made any significant difference to the evolution process. The aim of the function is to find the smallest element of a list of integers. The specification and fitness function are given in Figure 4.4 and Listing 4.9 and are identical to those of *listmax* except for the change in the comparison operator.

The list of statements and meaning of additional genes are the same as those used for *listmax* as shown in Tables 4.5 and 4.6. The only change is that the variable `$max` is replaced with the variable `$min`. The *listmin*

Table 4.8: The results for *listmin*

Seed	Generations	Time
0	170	5s
1	104	3s
2	160	4s
3	188	5s
5	234	7s
7	59	2s
11	699	19s
13	361	10s
17	270	9s
19	128	3s

function is also evolved using the same test input set (Listing 4.7).

The results for the ten runs of the *listmin* experiment are given in Table 4.8. The number of generations and running times are slightly higher than the *listmax* experiment but not significantly.

An example fully fit program evolved is given in Listing 4.10. As with *listmax*, this example does not completely solve the problem but does work with all of the test inputs. Appendix C has a complete list of evolved programs, which includes programs that do work with any list of integers that conform to the specification precondition.

4.6 Reverse

The *reverse* function is used as a simple example of a function that returns a list rather than a single value. The aim of the function is to reverse the ordering of a list of integers. This function creates the new list in a separate variable whereas the *sort* function in the next section gives an example of a

Listing 4.10: Example solution for *listmin*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $min = 0;
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
for $tmp2 (0..$#list){
  if($tmp2 != $list[$tmp1%($#list+1)]){
    $min = $list[$tmp1%($#list+1)];
    for $tmp2 (0..$#list){
      if($min > $list[$tmp2%($#list+1)]){
        $min = $list[$tmp2%($#list+1)];
      }
    }
  }
}

# Footer
return $min;
```

$$
\begin{aligned}
&\mathit{reverse} : \mathbb{Z}^* \to \mathbb{Z}^* \\
&\text{pre-}\mathit{reverse}(L) \triangleq \mathit{True} \\
&\text{post-}\mathit{reverse}(L, N) \triangleq \#L = \#N \\
&\quad \wedge (\forall n : \mathbb{P})(n < \#L \Rightarrow \\
&\qquad (\exists x : \mathbb{Z}, L_1, L_2, N_1, N_2 : \mathbb{Z}^*)(\; L = L_1 \frown \langle x \rangle \frown L_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge N = N_1 \frown \langle x \rangle \frown N_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = \#N_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = n))
\end{aligned}
$$

Figure 4.5: Specification of *reverse*

Listing 4.11: The fitness function for *reverse* (without header)

```
if(scalar(@L) == scalar(@N)){$fitness++;}
for(my $n=0; $n<scalar(@L); $n++){
  if($L[$n] == $N[$#L-$n]){$fitness++;}
}
```

Table 4.9: List of statements used for *reverse*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	Add	wvars,rvars,rvars	`G1 = G2 + G3;`
3	Subtract	wvars,rvars,rvars	`G1 = G2 - G3;`
4	Multiply	wvars,rvars,rvars	`G1 = G2 * G3;`
5	Divide	wvars,rvars,rvars	`G1 = G2 / G3 if(G3 != 0);`
6	If	rvars,cmp,rvars	`if(G1 G2 G3){`
3	For	rvars,lsize	`for G1 (0..G2){`
8	End		}

function creating the list *in place*. The specification for the *reverse* function is given in Figure 4.5 and the fitness function is given in Listing 4.11. A detailed description of the mapping is given in Chapter 3, Section 3.3.2. The set of test inputs is the same as that used for the *sumlist* function (Listing 4.2).

Tables 4.9 and 4.10 give the list of statements used and the meaning of additional genes. It is interesting to note the provision of list indexing of the form `L[#L-n]`. This is a logical choice based on the interpretation of the specification.

Table 4.11 gives the results of the ten runs of the experiment. The runs all evolve fully fit individuals very quickly (both in terms of time and generations). This, slightly unexpected, performance is most likely due to the inclusion of the `L[#L-n]` variables, which would appear to make the problem much easier.

Listing 4.12 gives an example individual evolved. When redundant parts are removed the function appears similar to the expected general solution with the inclusion of a few additional statements that do not affect the func-

Table 4.10: Additional Genes for *reverse*. All of the list indices are taken modulo the size of the list, however this is not shown for clarity

Index	wvars	rvars	lsize	cmp
0	`$out[$tmp1]`	`$tmp1`	`$#in`	`==`
1	`$out[$#in - $tmp1]`	`$tmp2`		`!=`
2	`$out[$tmp2]`	`$in[$tmp1]`		`>`
3	`$out[$#in - $tmp2]`	`$in[$#in - $tmp1]`		`<`
4		`$in[$tmp2]`		`>=`
5		`$in[$#in - $tmp2]`		`<=`
6		`$out[$tmp1]`		
7		`$out[$#in - $tmp1]`		
8		`$out[$tmp2]`		
9		`$out[$#in - $tmp2]`		

Table 4.11: The results for *reverse*

Seed	Generations	Time
0	135	9s
1	288	20s
2	149	8s
3	112	6s
5	35	2s
7	70	4s
11	16	1s
13	95	4s
17	74	4s
19	9	1s

Listing 4.12: Example solution for *reverse*, Seed 0. All of the list indices are taken modulo the size of the list, however this code is not shown for clarity

```
# Header
my @in = @{$test[$t]};
my @out = ();
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
$out[($#in - $tmp2)] = $out[($#in - $tmp2)];
for $tmp1 (0..$#in){
  if($in[($#in - $tmp2)] < $in[$tmp1]){
  }
  $out[$tmp1] = $tmp2 + $in[($#in - $tmp1)];
  if($in[$tmp2] <= $in[$tmp2]){
    $out[$tmp1] = $out[($#in - $tmp1)] / $in[$tmp2] if($in[
        $tmp2] != 0);
    $out[$tmp1] = $in[($#in - $tmp1)];
  }
}

# Footer
return @out;
```

$$sort : \mathbb{Z}^* \rightarrow \mathbb{Z}^*$$
$$\text{pre-}sort(L) \triangleq True$$
$$\text{post-}sort(L, N) \triangleq bag\_of(N) = bag\_of(L) \wedge ascending(N)$$
where $bag\_of(\langle\rangle) \blacktriangleleft\blacktriangleright \emptyset$
$bag\_of(\langle x \rangle) \blacktriangleleft\blacktriangleright \{|x|\}$
$bag\_of(L_1 \frown L_2) \blacktriangleleft\blacktriangleright bag\_of(L_1) \uplus bag\_of(L_2)$
$$ascending(N) \triangleq (\forall x, y : \mathbb{Z})(x \text{ before } y \text{ in } N \Rightarrow x \leq y)$$
and
$$x \text{ before } y \text{ in } N \triangleq (\exists N_1, N_2, N_3 : \mathbb{Z}^*)(N = N_1 \frown \langle x \rangle \frown N_2 \frown \langle y \rangle \frown N_3)$$

Figure 4.6: Specification of *sort*

Listing 4.13: The fitness function for *sort* (without header)

```
$fitness++ if(bageq(\@L, \@N));
if($#N > 0){
  for my $x (0..$#N-1){
    $fitness++ if($N[$x] <= $N[($x+1)]);
  }
}
```

tionality. Appendix C gives the complete set of *reverse* functions evolved.

4.7 Sort

The final function evolved was *sort*. Like *reverse*, this function also returns a list rather than an individual value. This is used as a more complex example and is a standard Computer Science problem. The aim of the function is to sort the elements of an integer list into ascending order. The specification for the problem is given in Figure 4.6 and the fitness function is given in Listing 4.13. The fitness function has been simplified to decrease the evaluation time, now only adjacent elements in the list are compared rather than all element pairs.

Tables 4.12 and 4.13 give the list of statements used and the meaning of

Table 4.12: List of statements used for *sort*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	If	rvars,cmp,rvars	`if(G1 G2 G3){`
3	For	counter,lsize	`for G1 (0..G2){`
4	Double	counter,lsize,counter	`for G1 (0..G2){ for G3 (G1+1..G2){`
5	End		}

Table 4.13: Additional Genes for *sort*. All of the list indices are taken modulo the size of the list, however this code is not shown for clarity

Index	wvars	rvars	counter	lsize	cmp
0	`$in[$tmp1]`	`$in[$tmp1]`	`$tmp1`	`$#in`	`==`
1	`$in[$tmp2]`	`$in[$tmp2]`	`$tmp2`		`!=`
2	`$tmp3`	`$tmp1`			`>`
3	`$tmp4`	`$tmp2`			`<`
4		`$tmp3`			`>=`
5		`$tmp4`			`<=`

Table 4.14: The results for *sort*

Seed	Generations	Time
0	47975	1h04m28s
1	14189	19m22s
2	8219	10m22s
3	16312	21m57s
5	31372	43m11s
7	5834	8m00s
11	28944	42m09s
13	18573	25m10s
17	36765	50m40s
19	1840	2m36s

additional genes. The list of statements now has the additional statement type the 'Double' loop (Index 4). This is a standard structure used when comparing elements in a list. In addition, it also demonstrates how larger building blocks can be used to evolve programs (other possible building blocks for this problem could include the 'Swap' function commonly used in sort algorithms). Again, the same test input set is used as *sumlist* (Listing 4.2).

Table 4.14 gives the results of the ten runs of the experiment. It is clear that this problem is much harder for the GP to solve. The times and number of generations required to evolve a fully fit individual are considerably higher than the previous experiments. However, all of the runs still produce a fully fit individual within the number of generations allowed. The increased difficulty is possibly due to there being a small number of *correct* solutions in the set of all possible genomes. In addition, there could be local minima around the optimal solutions, which make it difficult for the GP to produce perfect individuals quickly.

Listing 4.14 shows an example solution from the *sort* experiment. The

Listing 4.14: Example solution for *sort*, Seed 0. All of the list indices are taken modulo the size of the list, however this is not shown in the code for clarity

```
# Header
my @in = @{$test[$t]};
my $tmp1 = 0;
my $tmp2 = 0;
my $tmp3 = 0;
my $tmp4 = 0;

# Evolved Code
for $tmp2 (0..$#in){
  for $tmp1 ($tmp2+1..$#in){
    if($in[$tmp2] > $in[$tmp1]){
      if($in[$tmp1] != $tmp2){
        $tmp3 = $in[$tmp1];
      }
      if($tmp3 <= $in[$tmp2]){
        $in[$tmp1] = $in[$tmp2];
      }
      $in[$tmp2] = $tmp3;
    }
  }
}

# Footer
return @in;
```

code is nearly the same as a 'bubble' sort, however, the inclusion of two additional 'If' statements in the body of the 'Double' loop means that the function will not work in all cases. Some of the runs did evolve solutions that work in the general case. The complete set of evolved programs are listed in Appendix C. All of the solutions are variations on the 'bubble' sort. This is most likely due to the constraints of the language subset used. However, it is interesting to note that not all of the solutions took advantage of the 'Double' statement.

4.8 Summary and Conclusions

One of the biggest problems highlighted by the experiments is that an individual gaining a 100% fitness value isn't always correct in the general case. This is due to the test input set not being exhaustive (if it were the time to fitness test an individual would be impractical). The ideal solution to this problem would be to formally verify the solutions that gained a 100% fitness value, however, this would also be very hard on larger problems. Probably the best compromise is to test any solutions that gain a 100% fitness against a different (perhaps more extensive) set of inputs. Work has been done on generating test data for software testing using Genetic Algorithms [15], however this would probably be impractical for this application as the fitness test for each individual in the Genetic Algorithm would need to evolve a program to test the effectiveness of the test data.

To summarize, a set of list evaluation and manipulation functions were evolved with the interesting feature that they needed to be completely correct to be useful. The experiments showed that it was possible to evolve these functions, with this constraint, in a reasonably short amount of time. The *sort* function, however, showed that as the complexity of the problem rises

the time to solve the problem also rises quite sharply, suggesting that more complex problems would be nearly impossible to evolve as one block of code. There are two approaches to combating this problem. Either the larger functions can be evolved from smaller blocks of code and calls to previously evolved functions or the problem can be broken down into a series of smaller problems that can be evolved, possibly in parallel.

Chapter 5

Evolving the User Interface

5.1 Introduction

In this chapter a method for evolving graphical user interfaces is presented. The method is based on those used for evolving functions presented earlier (see Chapter 2), and is applied to both desktop and web-based user interfaces.

Why evolve Graphic User Interfaces? Firstly, any interactive program requires some form of user interface. For a small program a simple text-based interface may be sufficient, but for larger systems (such as a word processor) a more complex user interface may be required. Secondly, to be able to claim the evolution of a complete software system, the user interface needs to be part of that evolution process. The evolution process allows some element of variety in the design of the interface as it is not a deterministic process. Furthermore, it is claimed that 50% of the project implementation time is spent on the user interface [63].

A number of systems have demonstrated the automatic generation of user interfaces from high level specifications [23, 49, 72, 85]. Most of these systems

are based on a data model specification. Lauridsen extends the approach to work with a more abstract specification [58]. Other approaches include using declarative models [80] and conceptual graphs [40]. No work appears to have been done on using GAs or GP to generate user interfaces.

Section 5.2 covers the required predefined aspects of the problem. Sections 5.3 and 5.4 describe how these are used to construct the representation for the user interface and the fitness evaluation. Section 5.5 presents some example problems to show how the approach works in practice. Finally, a discussion on possible extensions to the method is given along with conclusions drawn from the work.

5.2 Requirements

What information is needed to be able to evolve a graphical user interface?

Firstly, some content to be manipulated is required. There are two types of content required, that associated directly with the interface, such as input widgets (buttons, text boxes, etc.) and text, and that which provides the underlying functionality.

Secondly, some constraints on the use and implementation of this content are required. There are three main areas which can be be constrained: the layout (the position of the widgets in relation to each other), the style (the fonts, colours, etc.), and the functionality (what does each button do when pressed, where do the underlying functions get their input from and where do they put their output).

This information is needed to construct the representation and fitness function of the genetic algorithm. The content is used to create the structure

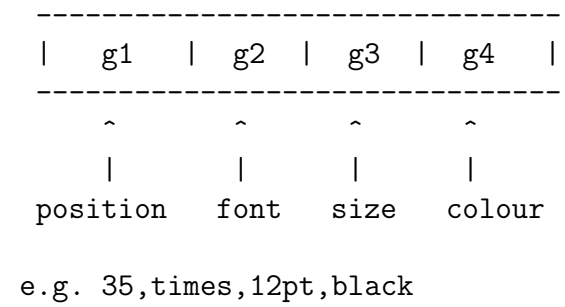


Figure 5.1: Example gene block

of the genome and the constraints are used as the basis of the fitness function.

5.3 Representation

The representation is based on the method for representing programs described earlier (see Chapter 4). Each widget is represented by a fixed-length block of genes in the genome. Each of the genes represents one parameter of the widget (e.g. the font size). The whole genome is made up from all of the blocks of genes, one for each of the widgets. Each widget always appears in the same place in the genome for each individual in the population.

For example, Figure 5.1 shows an example gene block, which gives the position, font, size, and colour of a widget. In a user interface language where there is a one dimensional ordering of the widgets (e.g. HTML), a single position gene can be used. All the widgets are then sorted on this value to give their position within the interface. In the case where two widgets have the same positional value, the ordering is indeterminate.

Finally, a mapping from the genome to the actual program code is required. The genome can map to any language (or even multiple languages), such as HTML or Perl/Tk. From Figure 5.1, the gene block might map to

(where its position in the program is decided as defined above):

```
.
.
<font face="times" size="12pt" color="black">Hello World</font>
.
.
```

In addition, there may be some functionality associated with the widget. Additional genes can be added to the gene blocks to represent the choice of function, where to get the input and where to put the output (see the example in Section 5.5.3).

5.4 Fitness Testing

It would be quite difficult to evaluate the entire user interface by executing the program and seeing how it looks and what it does. One possible solution to this problem is to evaluate all of the constraints individually by comparing the relevant parameters in the genome. The constraint scores are then combined to form the overall fitness of the individual. The combination operator in the examples shown in this chapter is summation. Each constraint can be weighted to give greater importance to certain constraints if necessary, and other combination functions used.

Each constraint can be formally specified (as in Chapter 3). For example, the specification of `i must be before j` might look something like:

$$\text{post-}MustBeBefore(i : \mathbb{W}, j : \mathbb{W}) \triangleq pos_i < pos_j$$

where $\mathbb{W}$ is the set of widgets and pos_i is the position attribute of the widget i. An ANDed series of constraints form the specification, which in turn can then be used to create the fitness function (see Chapter 3).

5.5 Example Problems

To show how Graphical User Interfaces can be evolved, three example problems are presented.

1. The evolution of a simple text editor interface. This demonstrates the layout constraints.

2. The evolution of a simple personal details web form. This adds style constraints.

3. A complete (but simple) application, which is an interface to the previously evolved list evaluation and manipulation functions. This demonstrates the functionality constraints in addition to layout and style.

5.5.1 A Text Editor

The first problem is a simple interface for a text editor. This demonstrates the use of positional constraints. The text editor interface is simply a text input area and a menu to select options (such as load and save). The constraints for this interface are mainly the relative positions of the items in the menu.

Figure 5.2 shows the list of widgets used for the interface and the constraints which need to be met by the evolution process. The informal specification in Figure 5.2 is in two parts. The first part lists the set of widgets to be used to construct the interface, and the second lists the constraints that must be met to achieve a maximum fitness value.

Each widget in the list of widgets has a label, a type, and some text value. In this example, the label also has some meaning attached to it, which determines which items are in which section of the menu. Therefore,

the contents of the menu are completely constrained but the ordering within that hierarchy is free to be evolved.

Some widgets do not require a text label (such as the menubar in the example), in this case the empty string is used.

Each constraint is a relation between two or more widgets or a requirement of one of the attributes of a widget. However, the implementation of the method can be easily extended. One constraint that is worth explaining is `menufile must be before menufilenew` (as well as similar constraints). This constraint is used as the `menufile` widget needs to be declared in the program before any items can be added to it. An alternative approach would be to deal with all of the toplevel menu items first.

In this example, the constraints have the effect of completely constraining the interface, so that only one possible output exists (not including redundant genes). The phenotype is mapped to a Perl/Tk program. Normally, the phenotype would be the program, but here the phenotype is the series of attributes for ease of fitness testing.

Each gene-block in the genome is made up from three genes. The first gene represents the position, and the second two (which are only used for the textarea) represent the way in which the textarea fills the window and expands when the window is resized.

Table 5.1 shows the results of ten example runs and, as the problem is relatively simple, all runs produce a solution that completely satisfies the constraints within a very short time. Figure 5.3 shows a user interface generated, but all fully fit user interfaces are constrained to be identical for this problem. The menus for the user interface are shown in Figure 5.4.

```
# Widgets
title: title "TextEdit"
menu: menubar ""
menufile: menulevel1 "File"
menuedit: menulevel1 "Edit"
menuhelp: menulevel1 "Help"
menufilenew: menulevel2 "New"
menufileopen: menulevel2 "Open"
menufilesave: menulevel2 "Save"
menufileexit: menulevel2 "Exit"
menueditcopy: menulevel2 "Copy"
menueditcut: menulevel2 "Cut"
menueditpaste: menulevel2 "Paste"
menuhelpabout: menulevel2 "About"
textarea: textarea ""

# Constraints
title must be first

menufile must be before menuedit
menuedit must be before menuhelp

menufile must be before menufilenew
menufilenew must be before menufileopen
menufileopen must be before menufilesave
menufilesave must be before menufileexit

menuedit must be before menueditcut
menueditcut must be before menueditcopy
menueditcopy must be before menueditpaste

menuhelp must be before  menuhelpabout

textarea must be last

# Constraints (Style)
textarea must fill x and y
textarea must expand when the window is resized
```

Figure 5.2: The Widgets and Constraints for the Text Editor

Table 5.1: Results for the text editor example

Seed	Generations	Time
2	481	3s
3	60	1s
5	77	1s
7	118	1s
11	387	3s
13	227	2s
17	329	2s
19	189	1s
23	199	2s
29	87	1s

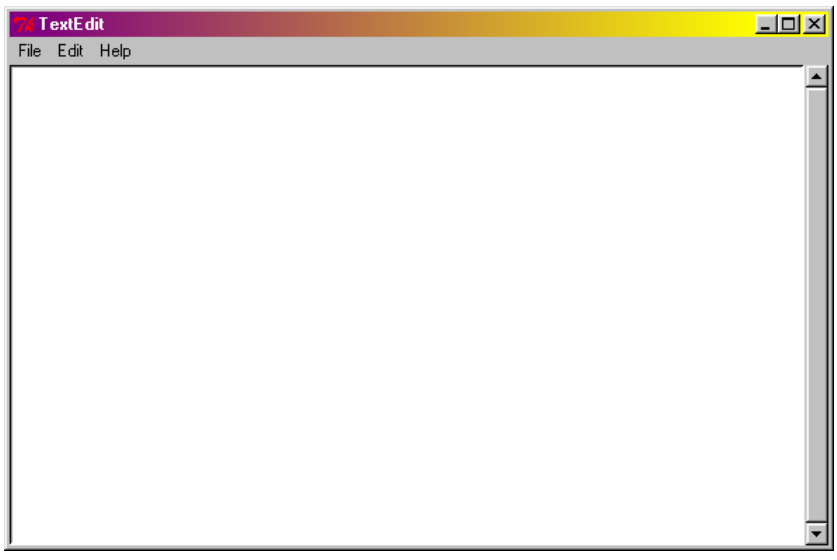


Figure 5.3: Text Editor GUI - Seed 2

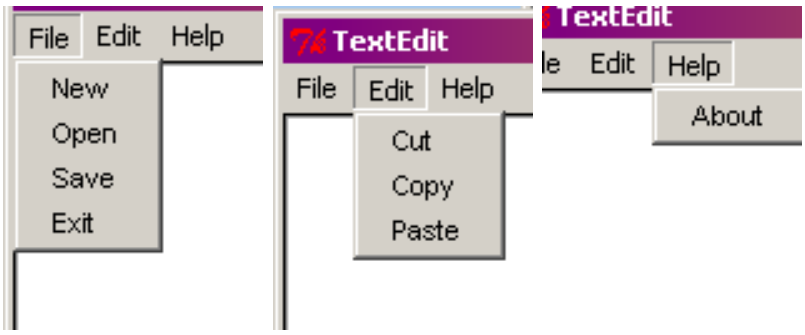


Figure 5.4: Text Editor GUI (The Menus) - Seed 2

5.5.2 A Personal Details Web Form

The second problem is a "Personal Details" web form. This introduces style constraints. The form contains a title and instructions, and a series of input boxes of different types with labels. Again, there is a set of positional constraints (such as labels must be immediately before the corresponding input). There are also style constraints which deal with fonts, colours, etc. For example, the title must be the largest font size and the labels must all be the same style.

Figure 5.5 show the list of widgets and the constraints (both positional and style) which need to be met by the evolution process in the Personal Details web form.

The gene-blocks for each widget are five genes. The first gene represents the position of the widget (with the widgets being sorted on the gene value), and the remaining four genes represent the style attributes: size, colour, font, and alignment respectively. The size is chosen from the list [`12pt`, `14pt`, `16pt`, `18pt`, `24pt`, `32pt`], the colour is chosen from the list [`white`, `lightgray`, `gray`, `darkgray`, `black`], the font is chosen from the list [`serif`, `sans-serif`, `monospace`], and the alignment is chosen from the list

```
# Widgets
title: title "Personal Details"
instructions: p "Enter your personal details in the form provided."
namelabel: p "Name"
nameinput: text ""
addresslabel: p "Address"
addressinput: text ""
townlabel: p "Town"
towninput: text ""
genderlabel: p "Gender"
genderinput: select ("male"/"female")
submitinput: submit ""
resetinput: reset ""

# Constraints (Layout)
title must be first

title must be immediately before instructions

namelabel must be immediately before nameinput
addresslabel must be immediately before addressinput
townlabel must be immediately before towninput
genderlabel must be immediately before genderinput

submitinput must be immediately before resetinput

nameinput must be immediately before addresslabel
addressinput must be immediately before townlabel
towninput must be immediately before genderlabel

reset must be last

# Constraints (Style)
title must have the largest font size
all labels must have the same style
font colours must be much darker than background colour
```

Figure 5.5: The Widgets and Constraints for the Web Form

Table 5.2: Results for the personal details example

Seed	Generations	Time
2	45165	10m04s
3	7521	1m40s
5	8441	1m51s
7	50000	10m45s
11	12277	2m39s
13	5930	1m16s
17	2842	37s
19	2517	33s
23	6964	1m31s
29	7294	1m35s

[`left`, `right`, `center`]. To convert the gene value into the attribute, the modulo of the gene value and the number of elements in the attribute list is taken. The position and attributes are then mapped to an HTML script.

Table 5.2 shows the results of ten example runs. It can be seen from the times that the problem is much harder than the previous example. The run which got to 50000 generations didn't achieve a maximum fitness value within the allowed number of generations. However, the other nine runs did achieve a maximum fitness score. Figure 5.6 shows a user interface generated. The complete set of generated interfaces can be found in Appendix D.

5.5.3 A Front-end for the List Functions

Finally, the third problem is a complete (although simple) application, which is an interface to the list evaluation and manipulation functions that were evolved earlier. This introduces functionality to the interface. In addition, this example also demonstrates a possible approach to the problem of scalab-

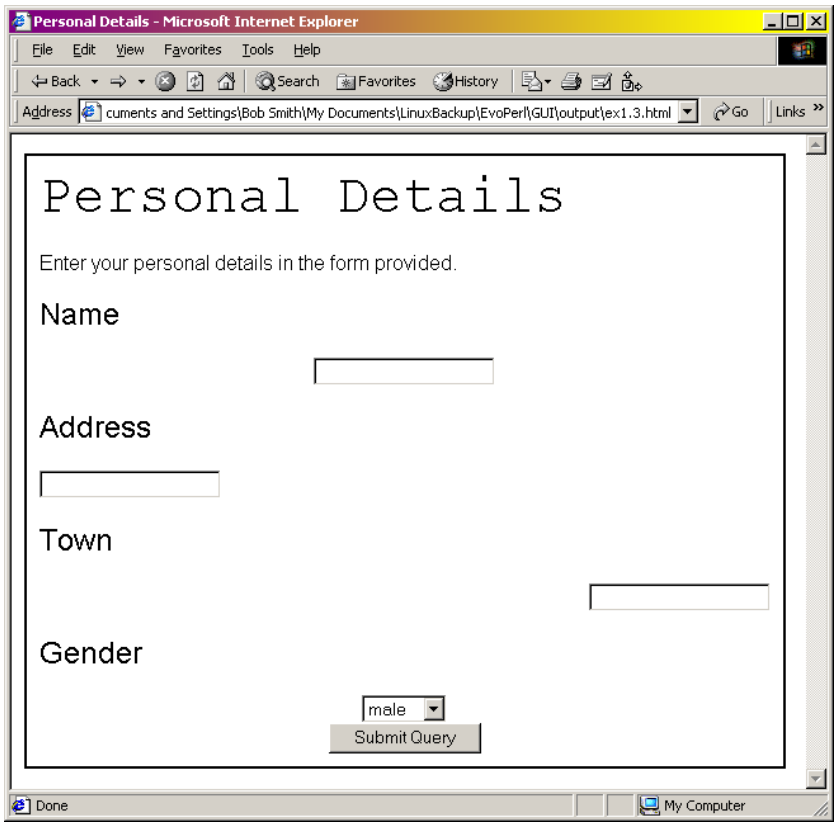


Figure 5.6: Personal Details GUI - Seed 3

ility by evolving component functions and then evolving an interface to connect them. The interface contains an input box, an output box, a list of functions, and a button. The positional and style constraints are similar to the previous problem, but now there are constraints on the functionality, which determine where the function gets its input and puts its output.

There are two choices when dealing with the functionality, either it can be hardcoded into the program or it can be evolved by the GA. The example shows the use of both options:

- Hardcoded functionality is demonstrated with the choice of function being chosen from the listbox (which returns the function name).
- Evolved functionality is demonstrated with the choice of widget the input list for the function is taken from, and the choice of widget the output list for the function is given to. Two extra genes are added to the gene-block from the previous example to allow this functionality to be evolved.

Figure 5.7 shows the list of widgets and the constraints to be met by the evolution process in the list function front-end.

Table 5.3 shows the results of the ten runs. The results show that the problem was quite simple, with all runs producing a result with maximum fitness in a short time. Figure 5.8 shows a user interface generated. The complete set of generated user interfaces can be found in Appendix D.

```
# Widgets
title: title "Sort"
instructions: p "Type in a list of integers, separated by spaces,
  into the Input box, select a function from the listbox and press
  the Run button."
list: listbox ("evolistmax"/"evolistmin"/"evosumlist"/"evoavelist"/
  "evoreverse"/"evosort")
sourcelabel: p "Input"
sourceinput: text ""
targetlabel: p "Output"
targetinput: text ""
runbutton: button "Run!"

# Constraints (Layout)
title must be first

title must be immediately before instructions

sourcelabel must be immediately before sourceinput
targetlabel must be immediately before targetinput
sourcelabel must be before targetlabel

runbutton must be last

# Constraints (Style)
title must have the largest font size
all labels must have the same style
font colours must be much darker than background colour

# Constraints (Functionality)
sourceinput must be input for runbutton
targetinput must be output for runbutton
```

Figure 5.7: The Widgets and Constraints for the List Front-end

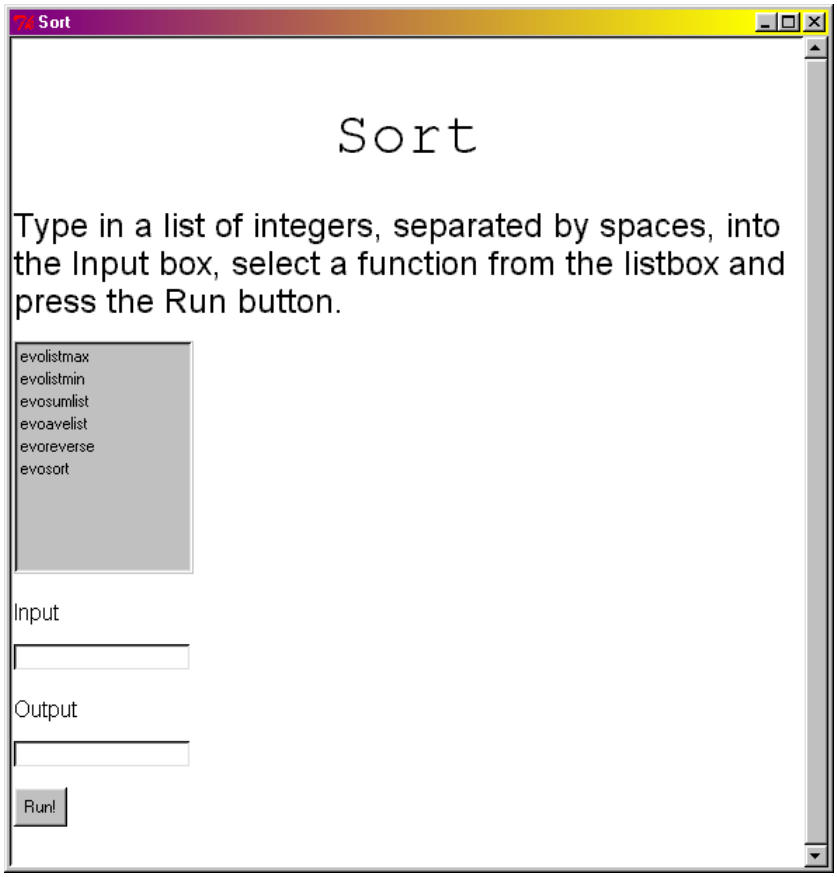


Figure 5.8: Sort GUI - Seed 23

Table 5.3: Results for the list front-end example

Seed	Generations	Time
2	1060	12s
3	310	4s
5	226	2s
7	337	4s
11	1419	16s
13	347	4s
17	1686	18s
19	2627	30s
23	424	5s
29	569	6s

5.6 Extensions

This method can easily be extended to cater for different types of user interface by introducing genes for the required attributes of the interface using the relevant constraints.

In addition, the constraints can be layered, so that there can be some global constraints which all interfaces must follow (such as those published by Apple [2] and Microsoft [64] or more generally following Shneiderman's 'Eight golden rules of dialog design' [81]). Then there can be a layer of constraints for specific groups of people. For example, dyslexic people prefer buttons instead of menus, whereas non-dyslexic people generally prefer the menu [16]. There can even be constraints to the level of the specific user. And finally, there are the problem specific constraints.

The content can also be abstracted, so instead of specifying a list box, a widget that chooses one item from many could be specified (which could be evolved to be a list box or a set of radio buttons, etc.). An intermediate

level of functionality can be evolved, which deals with changes to the user interface (to produce dynamic interfaces). For example, certain menu items might need to be disabled or enabled as a function of the state of the program after a particular action by the user (e.g. disable some menu items if there is no difference between saved and current versions).

The problem of scalability can be addressed by evolving the layout, style and functionality separately for more complex interfaces. In addition, each screen or window can be evolved separately when there are multiple screens in an interface. The above all contribute to improved performance of the evolution process on larger problems. This breakdown follows naturally from the specification of the problem (see Chapter 3).

5.7 Summary and Conclusions

To summarize, a method for evolving Graphical User Interfaces has been presented. This method is based on the previously given methods for evolving programs. This method can be applied to many different problems for many types of interfaces, in any required target language. The front-end for the list functions also addresses the problem of scalability as the interface and each of the functions are all evolved separately to create the whole system. In addition, the method can be used where there are contradictory constraints.

Chapter 6

Discussion and Conclusions

6.1 Introduction

This final chapter discusses the evolution of complete software systems as a whole, including necessary and desirable future work for the area. The contribution to knowledge made by this thesis is also discussed and summarized.

6.2 Evolution of Complete Software Systems

This section discusses the problems and possible solutions of evolving complete software systems. It is firstly necessary to define what is meant by a "complete software system". A "complete software system" in this context is considered to be a computer program that is interactive, multifunctional and usually very big. Unfortunately, all three of these areas have been largely neglected in the area of Genetic Programming. The question is therefore, how do we evolve multifunctional, interactive programs using GAs and GPs?

Before that, it is worth discussing the reasons for using GAs and GPs to create software systems. Firstly, the method is declarative. It is only necessary to express the requirements of the system and not how to actually implement it. Secondly, and this is most apparent in the user interface evolution, the evolution process adds an element of variety to the process. The GP will not produce a fixed, deterministic output. Various different solutions to the problem are likely to be produced from each run of the GP if the specified constraints and requirements allow multiple implementations. Finally, it is possible that the GP will allow a faster implementation time for the system (although this is dependent on the time spent setting up the GP as well as the evolution time).

It must be stressed that the idea is to make the problem to be tackled by the GP as simple as possible. Any knowledge that is already available should be used to simplify the problem. In addition, functions that have already been written, or evolved, should be available to the Genetic Program.

The following three subsections discuss the three main areas of evolving complete software systems: the evolution of functions, the scalability issue and the interactivity.

6.2.1 Function Evolution

The term 'multifunctional' suggests that multiple functions are required to provide the elements of the system's functionality. It would therefore be reasonable to evolve each of these functions separately or even independently in parallel.

There are various other aspects to evolving functions. As well as evolving the body of a function from a fitness evaluation function created from the

post-condition of the formal specification, it may also be desirable to evolve error handling based on the pre-condition to check for valid inputs. This could be included in the wrapper if it is a simple check but may need to be evolved if the pre-condition is complex. In addition to the specification of the function, it may be desirable to add time and/or space complexity constraints to the fitness evaluation.

6.2.2 Scalability

What happens when the functions become larger and more complex? One possible approach is to define the requirements for the function in a hierarchical way, so that the function can be evolved from sub-functions. This is relatively easy to do when working with the formal specification of the function. For example, the post-condition for *avelist* might look like

$$\text{post-}avelist(L, s) \triangleq s = sumlist(L)/\#L$$

and the *sumlist* function would be added to the language subset available. In addition, compound statements can be used such as the 'Double' statement in Chapter 4, Section 4.7.

As an example of using functions and compound statements to improve the performance of the evolution process, the *sort* example from Chapter 4 is extended. The GP was already using the compound 'Double' statement. To the previously used set of statements is added the function 'Swap', which is common to many sort functions. The rest of the language subset and test input is left unchanged (see Tables 4.12, 4.13 and Listing 4.2). The fitness function also remains unchanged (see Listing 4.13).

Table 6.1 shows the results of the ten runs of the experiment. It can be clearly seen that the introduction of the 'Swap' function has made a dramatic

Table 6.1: The results from the *sort* experiment with 'Swap'

Seed	Generations	Time
0	333	30s
1	10	1s
2	247	25s
3	661	1m04s
5	155	14s
7	32	2s
11	147	14s
13	204	19s
17	208	17s
19	354	30s

Listing 6.1: Example of the *sort* function using 'Swap', Seed 1

```
if($inlist[$tmp2] >= $tmp3){
  for $tmp2 (0..$#inlist){
    for $tmp1 ($tmp2+1..$#inlist){
      if($inlist[$tmp2] > $inlist[$tmp1]){
        swap(\$inlist[$tmp1],\$inlist[$tmp2]);
      }
    }
  }
}
```

impact on the performance of the GP. From an average number of generations of 21002 and an average time of 28m47s in Chapter 4, the average number of generations is now just 235 and the average time is just 21s (the fastest being 1s and the slowest being 64s). This is an improvement of roughly two orders of magnitude.

Listing 6.1 shows an example function generated. Apart from the surrounding 'If' statement, the code is the expected 'bubble' sort using the 'Swap' function to exchange elements of the list.

6.2.3 Interactivity

Finally, to connect the functions and provide the interactivity for the system, some form of user interface is required. For large, complex systems this user interface is usually graphical. Depending on the application the user interface can be either web-based or desktop-based, as appropriate. Often a static user interface is sufficient to connect the functions and provide interactivity, however, if a dynamic user interface is required, additional functions may need to be evolved to manipulate the user interface.

6.2.4 Other Uses

In addition to evolving complete software systems it may be desirable to evolve only small sections of code. For example, the GP could be used as a programming assistance tool, to give the programmer a starting point for writing a complex function. The user interface evolution could be used as a stand-alone system for quickly testing user interface ideas and allowing quick reimplementation after design changes.

6.3 Contribution of the Thesis

This thesis has addressed problems relating to the evolution of complete software systems. This includes work on both the underlying Genetic Programming system and the application of that system to relevant problems. This section summarizes the main contributions.

A new representation for programs, along with a mapping between the genotype and phenotype, has been presented. This representation has the

benefits of explicitly inheritable characteristics, easy mapping between the genotype and phenotype, support for arbitrary genetic operators, and the ability to represent programs in any language. This representation was shown to have better inheritance of characteristics between individuals than systems such as Grammatical Evolution [78]. In addition, all genotypes map to a complete program without reusing genes or randomly extending the genome, unlike some systems [69, 78].

A new method for the construction of fitness evaluation functions was presented, which is based on the formal specification of the function to be evolved. The post-condition of the formal specification is converted into the target language with each comparison incrementing the fitness value if true. This method was shown to have a better performance than using simple input/output pairs. In addition, the fitness function is easier to construct accurately than some "hand-crafted" function, with less likelihood of missing important features of the problem.

A series of functions was evolved that was more appropriate for general software evolution than traditional GP problems. The functions had the need for a 100% fitness value over the given set of test inputs to be considered useful, although even when this condition is met it is difficult to guarantee that the function matches the specification for all inputs. However, this problem is not unique to the evolution of functions, when humans write computer programs the same problem exists. Previous researchers (e.g. Koza [55]) have applied GPs to problems where the solutions have a *better/worse* classification (such as 'pole balancing') rather than a *right/wrong* classification. The experiments presented in this thesis have shown that GPs can be applied to a wider selection of programming problems.

A method for evolving graphical user interfaces was presented. This area

appears to have been completely ignored by the Genetic Programming community. This method was based on the representation for evolving programs presented previously. The method uses an intermediate translation of the genotype to test a series of constraints (which can be formally specified) before converting fit individuals into actual programs. Various possible extensions were also suggested, which allowed for dynamic user interfaces and also generation of interfaces with layered constraints. Although the example user interfaces were not perfect, all of the tools required for the evolution of user interfaces were demonstrated and it was shown that the approach is feasible.

A method of dealing with scalability issues was briefly introduced, which evolved larger functions from more abstract code segments such as compound statements and function calls. This showed a dramatic improvement from the previously presented version of the experiment.

Future work should involve the evolution of a 'real' software system, such as a simple word processor or an online shop. Work on the time taken to evolve functions (including setting up the GP) should be carried out and compared to the manual programming time. In addition, it would be interesting to experiment with the evolution of programs within other language paradigms. Prolog, for example, might be interesting as the program structure is quite different to a procedural language and there is less dependence on the order of the 'statements'. The only changes to accommodate the new paradigm would be the use of a different language subset and the appropriate compiler/interpreter.

With these new approaches to Genetic Programming, the evolution of complete software systems is now a realistic goal.

References

[1] Angeline P.J. & Pollack J.B. (1992). *Evolutionary induction of subroutines.* Proc. of the 14th Ann. Conf. of the Cognitive Science Society, Lawrence Erlbaum, Hillsdale, NJ, pp. 236–241.

[2] Apple Computer, Inc. (2002). Aqua User Interface Guidelines. Apple Computer, Inc.

[3] Bäck T., Hammel U., & Schwefel H.-P. (1997). *Evolutionary Computation: Comments on the History and Current State.* IEEE Transactions on Evolutionary Computation.

[4] Banzhaf W. (1993). *Genetic Programming for Pedestrians.* 5th International Conference on Genetic Algorithms ICGA-93.

[5] Banzhaf W., Nordin P., Keller R.E., & Francone F.D. (1998). Genetic Programming — An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann.

[6] Bremermann H.J. (1958). *The evolution of intelligence. The nervous system as a model of its environment.* Technical Report No.1, Contract No.477(17), Dept. of Mathematics, Univ. of Washington, Seattle.

[7] Bremermann H.J. (1962). *Optimization through evolution and recombination.* In, Self-Organising Systems, M.C.Yovits, G.T.Jacobi, & G.D.Goldstine, Eds. Washington, DC: Spartan Books, pp. 93–106.

[8] Bremermann H.J. (1967). *Quantitative aspects of goal-seeking self-organising systems.* In, Progress in Theoretical Biology, vol.1, New York: Academic Press, pp. 59–77.

[9] Bremermann H.J. (1968). *Numerical Optimization Procedures Derived from Biological Evolution Processes.* In, Cybernetic Problems in Bionics, H.L.Oestereicher & D.R.Moore, Eds. New York: Gordon & Breach, pp. 543–562.

[10] Bremermann H.J. (1973). *On the Dynamics and Trajectories of Evolution Processes.* In, Biogenesis, Evolution, Homeostasis, A.Locker, Ed. New York: Springer-Verlag, pp. 29–37.

[11] Bremermann H.J. & Rogson M. (1964). *An Evolution-Type Search Method for Convex Sets.* ONR Technical Report, Contracts 222(85) and 2656(58), UC Berkeley.

[12] Bremermann H.J., Rogson M., & Salaff S. (1965). *Search by Evolution.* In, Biophysics and Cybernetic Systems, M.Maxfield, A.Callahan, & L.J.Fogel, Eds. Washington, DC: Spartan Books, pp. 157–167.

[13] Bremermann H.J., Rogson M., & Salaff S. (1966). *Global Properties of Evolution Processes.* In, Natural Automata and Useful Simulations, H.H.Pattee, E.A.Adlsack, L.Fein, & A.B.Callahan, Eds. Washington, DC: Spartan Books, pp. 3–41.

[14] Brooks J.L. (1984). Just Before the Origin: Alfred Russel Wallace's Theory of Evolution. Columbia University Press.

[15] Burgess C.J. & Thomas M. (2001). *Generating High-Quality Test Data Using Genetic Algorithms.* 9th Annual International Conference Software Quality Management.

[16] Carter M. (2003). Computer based writing support for dyslexic adults using language constraints. PhD Thesis, Loughborough University.

[17] Cramer N.L. (1985). *A Representation for the Adaptive Generation of Simple Sequential Programs.* In Grefenstette J.J., editor, Proceedings of the First International Conference on Genetic Algorithms, pp. 183–187.

[18] Coley D.A. (2002). *Evolving Green Buildings.* In Cantú-Paz E., editor, Late-Breaking Papers, GECCO 2002, pp. 62–68.

[19] Cooke J. (1998). Constructing Correct Software: The Basics. Springer-Verlag.

[20] Cooper J. & Hinde C. (2002). *Comparison Of Evolving Against Peers And Fixed Opponents Using Corewars.* In Proceedings of GECCO 2002, p. 887. Morgan Kaufmann.

[21] Darwin C.R. (1859). The Origin of Species. John Murray.

[22] Darwin C.R. (1909). The Voyage of the Beagle. P.F. Collier & Son.

[23] de Baar D.J.M.J., Foley J.D., & Mullet K.E. (1992). *Coupling Application Design and the User Interface Design.* In CHI '92 Conference Proceedings, ACM Press. pp. 259–266.

[24] Colorni A. Dorigo M. & Maniezzo V. (1992). *Distributed Optimization by Ant Colonies.* In Varela F. & Bourgine P., editors, Proceedings of the First European Conference on Artificial Life, Paris, France, pp. 134–142. Elsevier Publishing.

[25] Dyer J.R. & Bentley P.J. (2002). *PLANTWORLD: Population Dynamics in Contrasting Environments.* In Cantú-Paz E., editor, Late-Breaking Papers, GECCO 2002, pp. 122–129.

[26] Fisher R.A. (1930). The Genetical Theory of Natural Selection.

[27] Fogel L.J. Owens A.J. & Walsh M.J. (1966). Artificial Intelligence Through Simulated Evolution. John Wiley & Sons, Inc.

[28] Fogel D.B. (1994). *An Introduction to Simulated Evolutionary Optimization.* IEEE Transactions on Neural Networks.

[29] Fogel D.B. (Ed.) (1998). The Fossil Record. IEEE Press.

[30] Fraser A.S. (1957). *Simulation of genetic systems by automatic digital computers. I. Introduction.* Australian J. of Biol. Sci., vol. 10, pp. 484–491.

[31] Fraser A.S. (1957). *Simulation of genetic systems by automatic digital computers. II. Effects of linkage on rates of advance under selection.* Australian J. of Biol. Sci., vol. 10, pp. 492–499.

[32] Fraser A.S. (1960). *Simulation of genetic systems by automatic digital computers. IV. Epistasis.* Australian J. of Biol. Sci., vol. 13, pp. 329–346.

[33] Fraser A.S. (1962). *Simulation of genetic systems.* J. of Theor. Biol., vol. 2, pp. 329–346.

[34] Fraser A.S. (1968). *The evolution of purposive behaviour.* In, Purposive Systems, H.von Foerster, J.D.White, L.J.Peterson, & J.K.Russell, Eds., Washington, DC: Spartan Books, pp. 15–23.

[35] Freeman J.J. (1998). *A linear representation for GP using context free grammars.* In Genetic Programming 1998: Proc. Third Annual Conference, pp. 72–77. Morgan Kaufmann.

[36] Friedberg R.M. (1958). *A Learning Machine: Part I.* IBM J. Research and Development, vol.2:1, pp. 2–13.

[37] Friedberg R.M., Dunham B., & North J.H. (1959). *A Learning Machine: Part II.* IBM J. Research and Development, vol.3, pp. 282–287.

[38] Fujiki C. (1986). An evaluation of Holland's genetic operators applied to a program generator. Master's Thesis, University of Idaho, Moscow, ID.

[39] Fujiki C. & Dickinson J. (1987). *Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma.* Genetic Algorithms and Their Applications: Proc. of the 2nd Intern. Conf. on Genetic Algorithms, J.J.Grefenstette (ed.), Lawrence Erlbaum, Hillsdale, NJ, pp. 236–240.

[40] Gerbé O. & Perron M. (1995). *Presentation Definition Language Using Conceptual Graphs.* In Proceedings PEIRCE Workbench. pp. 48–57.

[41] Goldberg D.E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley.

[42] Goldberg D.E. (1989). *Sizing Populations for Serial and Parallel Genetic Algorithms.* In Schaffer. J.D., editor, Proceedings of the Third International Conference on Genetic Algorithms, pp. 70–79. Morgan Kaufmann.

[43] Hemberg M., O'Reilly U.M., & Nordin P. (2001). *GENR8: A Design Tool for Surface Generation.* In Late Breaking paper at GECCO-2001.

[44] Henig R.M. (2000). A Monk and Two Peas. Weidenfeld & Nicholson.

[45] Hicklin J.F. (1986). Application of the genetic algorithm to automatic program generation. Master's Thesis, University of Idaho, Moscow, ID.

[46] Holland J.H. (1969). *Adaptive plans optimal for payoff-only environments.* Proc. of the 2nd Hawaii Int. Conf. on System Sciences, pp. 917–920.

[47] Holland J.H. (1973). *Genetic Algorithms and the Optimal Allocation of Trials.* In SIAM Journal on Computing, 2(2):88–105, June.

[48] Holland J.H. (1975). Adaption in Natural and Artificial Systems. The University of Michigan Press.

[49] Janssen C., Weisbecker A., & Ziegler J. (1993). *Generating User Interfaces from Data Models and Dialogue Net Specifications.* In INTERCHI '93 Conference Proceedings, ACM Press. pp. 418–423.

[50] Keijzer M. & Cattolico M. (2002). *An example of the use of context-sensitive constraints in the ALP system.* In Grammatical Evolution Workshop, GECCO 2002.

[51] Kellar R.E. & Banzhaf W. (1996). *Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes.* In Genetic Programming 1996.

[52] Kennedy J. & Eberhart R.C. (1995). *Particle swarm optimization.* In Proceedings of IEEE International Conference on Neural Networks, vol IV, 1942–1948, Piscataway, NJ.

[53] Kirkpatrick S. Gelatt Jr. C.D. & Vecchi M.P. (1983). *Optimization by Simulated Annealing.* Science, 220, 4598, pp. 671–680.

[54] Koza J.R. (1989). *Hierarchical Genetic Algorithms Operating on Populations of Computer Programs.* Proc. of the 11th Intern. Joint Conf. on Artificial Intelligence, N.S.Sridharan (ed.) Marga Kaufmann, San Mateo, CA, pp. 768–774.

[55] Koza J.R. (1992). Genetic Programming: On the Programming of Computers by means of Natural Selection. MIT Press.

[56] Koza J.R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press.

[57] Koza J.R., Andre D., Bennett F.H., & Keane M. (1999). Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufmann.

[58] Lauridsen O. (1995). *Abstract Specification of User Interfaces.* In CHI '95 Conference Proceedings. ACM Press.

[59] Ma Q. (1995). The Application of Genetic Algorithms to the Adaption of IIR Filters. PhD Thesis, Loughborough University.

[60] Malthus T. (1798). An Essay on the Principle of Population.

[61] McCarthy M. (2001). *A Tale of Natural Selection (with only one survivor).* The Independent, Tuesday Review. 8 May.

[62] Mendel G. (1865). Experiments in Plant Hybridization.

[63] Meyers B.A. & Rosson M.B. (1992). *Survey on User Interface Programming.* Proceedings of the Conference on Human Factors in Computing Systems.

[64] Microsoft Corporation. (1995). The Windows Interface Guidelines for Software Design: An Application Design Guide. Microsoft Press.

[65] Montana D.J. (1994). Strongly Typed Genetic Programming. Technical Report.

[66] Moore J.P. (2000). Exploring and Exploiting Models of the Fitness Landscape: a Case Against Evolutionary Optimization. PhD Thesis, University of Plymouth.

[67] O'Neill M. & Ryan C. (1999). *Evolving Multi-line Compilable C Programs.* EuroGP '99.

[68] O'Neill M., Brabazon T., Ryan C., & Collins J.J. (2001). *Developing a Market Timing System using Grammatical Evolution.* In Proceedings of GECCO 2001.

[69] Paterson N. & Livesey M. (1996). *Distinguishing genotype and phenotype in genetic programming.* Late-breaking Papers, GP-96.

[70] Paterson N. & Livesey M. (1997). *Evolving caching algorithms in C by genetic programming.* Genetic Programming 1997, pp. 262–267. Morgan Kaufmann.

[71] Perkis T. (1994). *Stack-Based Genetic Programming.* IEEE World Congress on Computational Intelligence.

[72] Puerta A.R., Eriksson H., Gennari J.H., & Musen M.A. (1994). *Beyond Data Models for Automated User Interface Generation.* In People and Computers IX, Proceedings of HCI '94, Cambridge University Press. pp. 352–366.

[73] Rechenberg I. (1973). Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution. Frommann-Holzboog.

[74] Reed J., Toombs R., & Barricelli N.A. (1967). *Simulation of biological evolution and machine learning.* Journal of Theoretical Biology, vol.17, pp. 319–342.

[75] Rosenblatt F. (1958). *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, 65:386–407.

[76] Ross B. (1999). Logis-based Genetic Programming with Definite Clause Translation Grammars. Technical Report No. CS-99-02. Brock University.

[77] Rothlauf F. (2002). Representations for Genetic and Evolutionary Algorithms. Springer Verlag.

[78] Ryan C., Collins J.J., & O'Neill M. (1998). *Grammatical Evolution: Evolving Programs for an Arbitrary Language.* EuroGP '98.

[79] Ryan C., Collins J.J., & O'Neill M. (1998). *Grammatical Evolution: Solving Trigonometric Identities.* In Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets, pp. 111–119.

[80] Schlungbaum E. & Elwert T. (1996). *Automatic User Interface Generation from Declarative Models.* CADUI '96.

[81] Shneiderman B. (1987). *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company.

[82] Smith S.F. (1983). *Flexible Learning of Problem Solving Heuristics though Adaptive Search.* Proc. IJCAI '83.

[83] Syswerda G. (1989). *Uniform Crossover in Genetic Algorithms.* In Schaffer J.D., editor, Proceedings of the Third International Conference on Genetic Algorithms, pp. 2–9. Morgan Kaufmann.

[84] Turing A.M. (1992). *Intelligent Machinery.* In Ince D.C., editor, Collected Works of A.M. Turing: Mechanical Intelligence, pp. 107–127. North-Holland.

[85] Vanderdonckt J.M. & Bodart F. (1993). *Encapsulating Knowledge for Intelligent Automatic Interaction Object Selection.* In INTERCHI '93 Conference Proceedings. ACM Press. pp. 424–429.

[86] Wall L., Christiansen, & Schwartz R.L. (1996). Programming Perl. O'Reilly & Associates, Inc., Second Edition.

[87] Ward M. (1999). Virtual Organisms: The Startling World of Artificial Life. Pan Books.

[88] Whigham P.A. (1995). *Grammatically-based Genetic Programming.* Workshop on Genetic Programming.

[89] Withall M.S., Hinde C.J., Stone R.G., & Cooper J.L. (2003). *Packet Transmission Optimisation using Genetic Algorithms.* In Proceedings of 16th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems. Springer-Verlag.

Appendix A

Publications

- "Evolving Readable Perl" (2002) Mark Withall, Chris Hinde & Roger Stone, GECCO 2002 Poster Paper.
- "Evolving Perl" (2002) Mark Withall, Chris Hinde & Roger Stone, GECCO 2002 Late-Breaking Paper.
- "Genetic Programming: The Evolution of Complete Software Systems" (2002) Mark Withall, Chris Hinde & Roger Stone, Internal Report, Computer Science 1069, November 2002. Loughborough University.
- "Investigation into the effects of varying the parameters of packets travelling across the Internet" (2003) Jason Cooper, Mark Withall, Chris Hinde & Roger Stone, Internal Report, Computer Science 1070, February 2003. Loughborough University.
- "A Parallel Approach to Row-Based VLSI Layout using Stochastic Hill-Climbing" (2003) Matthew Newton, Ondrej Sýkora, Mark Withall & Imrich Vrt'o, IEA2003AIE.

- "Packet Transmission Optimisation using Genetic Algorithms" (2003) Mark Withall, Chris Hinde, Roger Stone & Jason Cooper, IEA2003AIE.

In addition, the following seminars were given:

- "Evolving the User Interface" at Postgraduate Seminar 16th April 2003. Loughborough University.
- "Genetic Programming: The Evolution of Complete Software Systems" at Postgraduate Seminar 6th November 2002. Loughborough University.
- "Automatic Drawing of Gate-Level Circuit Diagrams using Genetic Algorithms" at PARC Mini-Symposium 18th June 2002. Loughborough University.
- "The Evolution of Genetic Algorithms" (with Jason Cooper & Chris Hinde) Department of Computer Science, Internal Seminar 30 March 2001. Loughborough University & Nortel Networks(Maidenhead) Seminar 5 April 2001.

Appendix B

Complete Results for the Formal Specification Test

This appendix contains the complete results of the formal specification based fitness functions compared to to the input/output pair fitness functions (Chapter 3).

- Table B.1 gives the results for input/output pairs for the *Listmax* problem.
- Table B.2 gives the results for formal specification based fitness functions for the *Listmax* problem.
- Table B.3 gives the results for input/output pairs for the *Reverse* problem.
- Table B.4 gives the results for formal specification based fitness functions for the *Reverse* problem.

Table B.1: The results of the *Listmax* experiment with Input/Output pairs

Seed	Gens	Time
2	145	2s
3	544	10s
5	125	3s
7	194	3s
11	258	4s
13	147	2s
17	483	8s
19	135	2s
23	533	9s
29	813	15s
31	241	4s
37	242	4s
41	511	8s
43	45	1s
47	142	2s
53	59	2s
59	407	7s
61	439	8s
67	195	3s
71	353	6s
73	110	3s
79	387	7s
83	396	7s
89	149	3s
97	378	6s
101	88	1s
103	180	3s
107	152	2s
109	424	8s
113	13	1s
127	552	10s
131	85	2s
137	383	7s

Seed	Gens	Time
139	598	10s
149	282	5s
151	396	8s
157	444	7s
163	82	2s
167	30	1s
173	102	2s
179	37	1s
181	302	5s
191	733	12s
193	197	3s
197	215	4s
199	1396	25s
211	386	7s
223	32	1s
227	202	3s
229	426	7s
233	62	1s
239	606	10s
241	86	1s
251	220	3s
257	130	3s
263	1170	19s
269	318	6s
271	514	9s
277	353	7s
281	350	7s
283	65	1s
293	1035	18s
307	1655	29s
311	231	4s
313	834	15s
317	302	5s
331	229	4s

Seed	Gens	Time
337	244	5s
347	574	10s
349	368	7s
353	200	3s
359	78	2s
367	344	7s
373	158	2s
379	386	7s
383	281	5s
389	206	4s
397	177	4s
401	192	3s
409	1045	18s
419	252	5s
421	128	3s
431	345	6s
433	842	15s
439	152	3s
443	414	8s
449	551	11s
457	46	1s
461	418	9s
463	303	5s
467	75	1s
479	123	2s
487	9	1s
491	132	3s
499	445	7s
503	111	2s
509	47	1s
521	169	2s
523	436	8s
541	761	17s

Table B.2: The results of the *Listmax* experiment with Formal Specification based fitness function

Seed	Gens	Time
2	192	4s
3	56	1s
5	59	1s
7	236	6s
11	99	3s
13	10	1s
17	157	4s
19	111	2s
23	327	8s
29	195	4s
31	15	1s
37	179	5s
41	52	1s
43	42	1s
47	123	3s
53	198	4s
59	256	6s
61	148	3s
67	314	7s
71	236	6s
73	273	6s
79	230	5s
83	91	3s
89	43	1s
97	283	7s
101	274	6s
103	501	11s
107	196	5s
109	106	2s
113	40	1s
127	421	8s
131	182	4s
137	162	4s
139	217	5s
149	94	2s
151	245	6s
157	214	4s
163	531	14s
167	170	4s
173	70	2s
179	18	1s
181	236	5s
191	22	1s
193	562	14s
197	99	3s
199	405	9s
211	161	4s
223	382	9s
227	115	3s
229	67	2s
233	427	10s
239	102	3s
241	148	4s
251	94	2s
257	11	1s
263	423	11s
269	197	4s
271	88	2s
277	311	6s
281	123	3s
283	29	1s
293	291	6s
307	55	1s
311	53	1s
313	82	2s
317	112	3s
331	34	1s
337	158	3s
347	192	4s
349	38	1s
353	213	4s
359	65	1s
367	87	2s
373	80	2s
379	566	14s
383	295	7s
389	187	5s
397	381	9s
401	382	9s
409	131	3s
419	192	4s
421	336	7s
431	197	5s
433	145	3s
439	85	2s
443	263	6s
449	71	1s
457	181	4s
461	47	1s
463	36	1s
467	486	12s
479	333	8s
487	34	1s
491	290	7s
499	93	2s
503	195	4s
509	64	2s
521	9	1s
523	128	3s
541	259	6s

Table B.3: The results of the *Reverse* experiment with Input/Output pairs

Seed	Gens	Time
2	420	20s
3	37	2s
5	18	1s
7	114	5s
11	29	1s
13	93	5s
17	215	11s
19	16	1s
23	4	1s
29	69	3s
31	26	1s
37	744	35s
41	78	4s
43	94	4s
47	212	10s
53	54	3s
59	179	9s
61	133	7s
67	104	5s
71	179	9s
73	60	3s
79	189	9s
83	66	4s
89	201	9s
97	80	4s
101	83	4s
103	24	1s
107	221	10s
109	286	14s
113	217	10s
127	399	18s
131	105	5s
137	88	5s
139	36	2s
149	101	5s
151	10	1s
157	146	7s
163	252	12s
167	219	10s
173	135	7s
179	36	2s
181	124	6s
191	153	7s
193	273	13s
197	87	4s
199	91	5s
211	99	5s
223	58	2s
227	250	12s
229	176	8s
233	127	6s
239	12	1s
241	44	2s
251	151	7s
257	30	1s
269	78	4s
271	17	1s
277	497	24s
281	231	12s
283	11	1s
293	51	2s
307	392	19s
311	90	4s
313	341	16s
317	8	1s
331	179	10s
337	48	2s
347	48	3s
349	313	15s
353	21	1s
359	13	1s
367	127	7s
373	116	5s
379	144	6s
383	32	1s
389	319	15s
397	9	1s
401	166	8s
409	13	1s
419	29	2s
421	313	16s
431	66	3s
433	207	11s
439	124	6s
443	283	15s
449	29	1s
457	286	13s
461	74	3s
463	6	1s
467	7	1s
479	392	19s
487	160	8s
491	93	4s
499	43	2s
503	281	13s
509	18	1s
521	4	1s
523	260	13s
541	81	3s

Table B.4: The results of the *Reverse* experiment with Formal Specification based fitness function

Seed	Gens	Time
2	149	8s
3	112	6s
5	35	2s
7	70	4s
11	16	1s
13	95	4s
17	74	4s
19	9	1s
23	41	2s
29	252	12s
31	239	12s
37	105	6s
41	160	7s
43	238	12s
47	226	11s
53	60	3s
59	6	1s
61	227	11s
67	162	8s
71	480	24s
73	35	2s
79	28	1s
83	168	8s
89	102	5s
97	3	1s
101	30	1s
103	20	1s
107	65	4s
109	147	7s
113	40	2s
127	38	2s
131	215	11s
137	170	9s
139	46	2s
149	42	2s
151	81	4s
157	44	2s
163	43	3s
167	96	4s
173	52	3s
179	85	4s
181	131	6s
191	77	4s
193	55	3s
197	59	3s
199	18	1s
211	77	3s
223	13	1s
227	302	15s
229	120	7s
233	36	1s
239	35	1s
241	64	3s
251	14	1s
257	168	8s
263	16	1s
269	176	9s
271	279	14s
277	87	4s
281	54	2s
283	146	8s
293	48	2s
307	87	4s
311	456	22s
313	265	13s
317	17	1s
331	194	9s
337	118	6s
347	31	1s
349	21	1s
353	15	1s
359	15	1s
367	31	2s
373	108	5s
379	304	14s
383	139	8s
389	229	11s
397	16	1s
401	306	14s
409	28	1s
419	375	18s
421	96	5s
431	38	1s
433	43	2s
439	85	4s
443	109	5s
449	35	2s
457	32	2s
461	29	2s
463	55	3s
467	65	3s
479	42	3s
487	141	7s
491	43	2s
499	40	2s
503	25	1s
509	146	7s
521	0	1s
523	239	12s
541	229	11s

Appendix C

Complete Set of List Functions Evolved

This appendix contains the complete list of functions evolved in the experiments in Chapter 4. The list indices in the code examples are presented without stating that they are modulo the size of the list for clarity. The code examples are also presented with most of the obviously redundant code removed.

C.1 Sumlist

Listing C.1: Sumlist Seed 0

```
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp];
}
```

Listing C.2: Sumlist Seed 1

```
for $tmp (0..$#list){
  $sum = $list[$tmp] + $sum;
  if($list[$tmp] == $size){
    $sum = $sum / $sum if($sum != 0);
    $sum = $list[$tmp] * $sum;
  }
}
```

Listing C.3: Sumlist Seed 2

```
if($sum < $list[$tmp){
  for $tmp (0..$#list){
    $sum = $sum + $list[$tmp];
  }
}
```

Listing C.4: Sumlist Seed 3

```
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp];
}
```

Listing C.5: Sumlist Seed 5

```
for $tmp (0..$#list){
  if($tmp < $size){
    $sum = $sum + $list[$tmp];
  }
}
```

Listing C.6: Sumlist Seed 7

```
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp];
}
```

Listing C.7: Sumlist Seed 11

```
for $tmp (0..$#list){
  $sum = $list[$tmp] + $sum;
}
```

Listing C.8: Sumlist Seed 13

```
for $tmp (0..$#list){
  $sum = $list[$tmp] + $sum;
}
```

Listing C.9: Sumlist Seed 17

```
for $tmp (0..$#list){
  $sum = $list[$tmp] + $sum;
}
```

Listing C.10: Sumlist Seed 19

```
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp];
}
```

C.2 Avelist

Listing C.11: Avelist Seed 0

```
for $tmp (0..$#list){
  $ave = $list[$tmp] + $ave;
}
$ave = $ave / $size if($size != 0);
```

Listing C.12: Avelist Seed 1

```
for $tmp (0..$#list){
  $ave = $list[$tmp] + $ave;
}
$ave = $ave / $size if($size != 0);
```

Listing C.13: Avelist Seed 2

```
for $tmp (0..$#list){
  $ave = $ave + $list[$tmp];
}
$ave = $ave / $size if($size != 0);
```

Listing C.14: Avelist Seed 3

```
for $tmp (0..$#list){
  $ave = $ave + $list[$tmp];
}
$ave = $ave / $size if($size != 0);
```

Listing C.15: Avelist Seed 5

```
$ave = $list[$tmp] + $size;
$ave = $tmp / $list[$tmp] if($list[$tmp] != 0);
for $tmp (0..$#list){
  $ave = $list[$tmp] + $ave;
}
$ave = $ave / $size if($size != 0);
```

Listing C.16: Avelist Seed 7

```
for $tmp (0..$#list){
  $ave = $ave + $list[$tmp];
}
$ave = $ave / $size if($size != 0);
```

Listing C.17: Avelist Seed 11

```
for $tmp (0..$#list){
  $ave = $ave + $list[$tmp];
}
$ave = $ave / $size if($size != 0);
```

Listing C.18: Avelist Seed 13

```
for $tmp (0..$#list){
  $ave = $list[$tmp] + $ave;
}
$ave = $ave / $size if($size != 0);
```

Listing C.19: Avelist Seed 17

```
for $tmp (0..$#list){
  $ave = $list[$tmp] + $ave;
}
$ave = $ave / $size if($size != 0);
```

Listing C.20: Avelist Seed 19

```
for $tmp (0..$#list){
  $ave = $ave + $list[$tmp];
}
$ave = $ave / $size if($size != 0);
```

C.3 Listmax

Listing C.21: Listmax Seed 0

```
for $tmp2 (0..$#list){
  if($list[$tmp1)] <= $list[$tmp2]){
    $max = $list[$tmp2];
    if($tmp2 >= $max){
      if($list[$tmp1)] == $max){
        $max = $list[$tmp2];
      }
    }
  }
}
```

Listing C.22: Listmax Seed 1

```
$max = $list[$tmp1];
for $tmp1 (0..$#list){
  if($max <= $list[$tmp1]){
    $max = $list[$tmp1];
  }
}
```

Listing C.23: Listmax Seed 2

```
$max = $list[$tmp2];
for $tmp1 (0..$#list){
  if($max <= $list[$tmp1]){
    $max = $list[$tmp1];
  }
}
```

Listing C.24: Listmax Seed 3

```
for $tmp2 (0..$#list){
  if( $list[$tmp1] == $list[$tmp2]){
    $max = $list[$tmp1];
  }
  if($max < $list[$tmp2]){
    $max = $list[$tmp2];
  }
}
```

Listing C.25: Listmax Seed 5

```
$max = $list[$tmp2];
for $tmp2 (0..$#list){
  if($max <= $list[$tmp2]){
    $max = $list[$tmp2];
  }
}
```

Listing C.26: Listmax Seed 7

```
$max = $list[$tmp1];
for $tmp1 (0..$#list){
  if( $list[$tmp1] > $max){
    $max = $list[$tmp1];
  }
}
```

Listing C.27: Listmax Seed 11

```
$max = $list[$tmp1];
for $tmp1 (0..$#list){
  if( $list[$tmp1] >= $max){
    $max = $list[$tmp1];
  }
}
```

Listing C.28: Listmax Seed 13

```
for $tmp1 (0..$#list){
  $max = $list[$tmp1];
  for $tmp2 (0..$#list){
    if($max < $list[$tmp2]){
      $max = $list[$tmp2];
    }
  }
}
```

Listing C.29: Listmax Seed 17

```
if($max >= $tmp2){
  for $tmp2 (0..$#list){
    if( $list[$tmp2] >= $list[$tmp1]){
      $max = $list[$tmp2];
    }
  }
}
```

Listing C.30: Listmax Seed 19

```
for $tmp1 (0..$#list){
  if( $list[$tmp1] >= $list[$tmp2]){
    $max = $list[$tmp1];
  }
}
```

C.4 Listmin

Listing C.31: Listmin Seed 0

```
for $tmp2 (0..$#list){
  if($tmp2 != $list[$tmp1]){
    $min = $list[$tmp1];
    for $tmp2 (0..$#list){
      if($min > $list[$tmp2]){
        $min = $list[$tmp2];
      }
    }
  }
}
```

Listing C.32: Listmin Seed 1

```
$min = $list[$tmp2];
for $tmp1 (0..$#list){
  for $tmp2 (0..$#list){
    if($list[$tmp1] < $min){
      $min = $list[$tmp1];
    }
  }
}
```

Listing C.33: Listmin Seed 2

```
$min = $list[$tmp2];
for $tmp1 (0..$#list){
  if($list[$tmp1] < $min){
    $min = $list[$tmp1];
  }
}
```

Listing C.34: Listmin Seed 3

```
if($tmp1 <= $tmp2){
  $min = $list[$tmp2];
  for $tmp2 (0..$#list){
    if($min > $list[$tmp2]){
      $min = $list[$tmp2];
    }
  }
}
```

Listing C.35: Listmin Seed 5

```
$min = $list[$tmp1];
if($list[$tmp2] <= $min){
  for $tmp2 (0..$#list){
    for $tmp1 (0..$#list){
      if($list[$tmp1] <= $min){
        $min = $list[$tmp1];
      }
    }
  }
}
```

Listing C.36: Listmin Seed 7

```
$min = $list[$tmp1];
for $tmp1 (0..$#list){
  if($list[$tmp1] <= $min){
    $min = $list[$tmp1];
  }
}
```

Listing C.37: Listmin Seed 11

```
$min = $list[$tmp2];
for $tmp2 (0..$#list){
  if($min >= $list[$tmp2]){
    $min = $list[$tmp2];
  }
}
```

Listing C.38: Listmin Seed 13

```
$min = $list[$tmp1];
if($tmp2 != $list[$tmp1]){
  $min = $list[$tmp1];
  for $tmp1 (0..$#list){
    for $tmp2 (0..$#list){
      if($min >= $list[$tmp2]){
        $min = $list[$tmp2];
      }
    }
  }
}
```

Listing C.39: Listmin Seed 17

```
$min = $list[$tmp1];
for $tmp1 (0..$#list){
  $min = $list[$tmp1];
  for $tmp1 (0..$#list){
    if($tmp1 < $min){
      $min = $list[$tmp1];
    }
  }
}
```

Listing C.40: Listmin Seed 19

```
$min = $list[$tmp2];
for $tmp2 (0..$#list){
  if($min > $list[$tmp2]){
    $min = $list[$tmp2];
    for $min (0..$#list){
      if($list[$tmp1] < $tmp2){
        $min = $min;
        if($tmp1 > $list[$tmp2]){
          $min = $tmp2;
        }
      }
    }
  }
}
```

C.5 Reverse

Listing C.41: Reverse Seed 0

```
for $tmp1 (0..$#inlist){
  if($inlist[$tmp2] <= $inlist[$tmp2]){
    $outlist[$tmp1] = $inlist[($#inlist - $tmp1)];
  }
}
```

Listing C.42: Reverse Seed 1

```
$outlist[($#inlist - $tmp2)] = $tmp1;
$outlist[($#inlist - $tmp1)] = $inlist[($#inlist - $tmp1)
   ] - $inlist[($#inlist - $tmp1)];
$outlist[($#inlist - $tmp1)] = $inlist[$tmp1] / $outlist[
   $tmp2] if($outlist[$tmp2] != 0);
for $tmp1 (0..$#inlist){
  $outlist[($#inlist - $tmp1)] = $inlist[$tmp1];
}
```

Listing C.43: Reverse Seed 2

```
$outlist[($#inlist - $tmp1)] = $outlist[$tmp2] / $tmp2 if(
    $tmp2 != 0);
$outlist[($#inlist - $tmp1)] = $inlist[($#inlist - $tmp2)
    ] / $outlist[($#inlist - $tmp2)] if($outlist[($#inlist
     - $tmp2)] != 0);
if($outlist[$tmp1] < $tmp2){
  $outlist[($#inlist - $tmp1)] = $outlist[($#inlist - $tmp1
      )];
}
for $tmp1 (0..$#inlist){
  $outlist[($#inlist - $tmp1)] = $inlist[$tmp1];
}
```

Listing C.44: Reverse Seed 3

```
$outlist[($#inlist - $tmp1)] = $tmp2;
$outlist[$tmp1] = $inlist[$tmp1] - $outlist[($#inlist -
    $tmp1)];
$outlist[($#inlist - $tmp1)] = $inlist[($#inlist - $tmp1)
    ] + $outlist[($#inlist - $tmp1)];
$outlist[($#inlist - $tmp1)] = $outlist[$tmp1] + $inlist
    [($#inlist - $tmp1)];
for $tmp2 (0..$#inlist){
  $outlist[$tmp2] = $outlist[$tmp2];
  $outlist[($#inlist - $tmp2)] = $inlist[$tmp2];
}
```

Listing C.45: Reverse Seed 5

```
$outlist[$tmp1] = $outlist[($#inlist - $tmp2)] / $tmp2 if(
    $tmp2 != 0);
$outlist[$tmp2] = $outlist[($#inlist - $tmp2)] * $inlist[
    $tmp1];
$outlist[$tmp2] = $outlist[($#inlist - $tmp2)] - $tmp1;
for $tmp2 (0..$#inlist){
  $outlist[($#inlist - $tmp1)] = $tmp2 * $tmp2;
  $outlist[$tmp2] = $inlist[($#inlist - $tmp2)];
}
```

Listing C.46: Reverse Seed 7

```
$outlist[($#inlist - $tmp1)] = $inlist[$tmp2];
$outlist[($#inlist - $tmp2)] = $outlist[$tmp1] / $inlist
    [($#inlist - $tmp1)] if($inlist[($#inlist - $tmp1)
    ] != 0);
for $tmp2 (0..$#inlist){
  $outlist[($#inlist - $tmp1)] = $outlist[$tmp1] / $inlist[
      $tmp2] if($inlist[$tmp2] != 0);
  $outlist[($#inlist - $tmp1)] = $outlist[($#inlist - $tmp1
      )];
  $outlist[$tmp2] = $inlist[($#inlist - $tmp2)];
}
```

Listing C.47: Reverse Seed 11

```
for $tmp1 (0..$#inlist){
  $outlist[($#inlist - $tmp2)] = $tmp2 + $tmp2;
  $outlist[$tmp1] = $inlist[($#inlist - $tmp1)] - $tmp2;
}
```

Listing C.48: Reverse Seed 13

```
$outlist[($#inlist - $tmp1)] = $tmp2 - $outlist[($#inlist
    - $tmp1)];
for $tmp1 (0..$#inlist){
  if($outlist[($#inlist - $tmp1)] == $inlist[($#inlist -
      $tmp1)]){
    $outlist[($#inlist - $tmp2)] = $tmp2;
    $outlist[($#inlist - $tmp2)] = $outlist[$tmp2];
  }
  $outlist[($#inlist - $tmp1)] = $tmp2 + $inlist[$tmp1];
}
```

Listing C.49: Reverse Seed 17

```
$outlist[$tmp2] = $tmp1;
$outlist[$tmp2] = $outlist[($#inlist - $tmp2)] / $inlist
   [($#inlist - $tmp2)] if($inlist[($#inlist - $tmp2)
   ] != 0);
if($inlist[$tmp2] > $outlist[($#inlist - $tmp2)]){
  $outlist[($#inlist - $tmp1)] = $outlist[$tmp1] / $inlist
     [($#inlist - $tmp2)] if($inlist[($#inlist - $tmp2)
     ] != 0);
  for $tmp1 (0..$#inlist){
    $outlist[$tmp2] = $tmp1 - $outlist[$tmp2];
    $outlist[($#inlist - $tmp1)] = $inlist[$tmp1] - $tmp2;
  }
}
```

Listing C.50: Reverse Seed 19

```
$outlist[($#inlist - $tmp2)] = $outlist[$tmp1] - $inlist
   [($#inlist - $tmp1)];
$outlist[$tmp1] = $outlist[($#inlist - $tmp2)] / $inlist[
   $tmp2] if($inlist[$tmp2] != 0);
$outlist[($#inlist - $tmp2)] = $outlist[$tmp2];
$outlist[($#inlist - $tmp2)] = $outlist[$tmp1] * $inlist[
   $tmp2];
for $tmp2 (0..$#inlist){
  $outlist[$tmp2] = $inlist[($#inlist - $tmp2)];
}
```

C.6 Sort

Listing C.51: Sort Seed 0

```
for $tmp2 (0..$#inlist){
  for $tmp1 ($tmp2+1..$#inlist){
    if($inlist[$tmp2] > $inlist[$tmp1]){
      if($inlist[$tmp1] != $tmp2){
        $tmp3 = $inlist[$tmp1];
      }
      if($tmp3 <= $inlist[$tmp2]){
        $inlist[$tmp1] = $inlist[$tmp2];
      }
      $inlist[$tmp2] = $tmp3;
    }
  }
}
```

Listing C.52: Sort Seed 1

```
for $tmp2 (0..$#inlist){
  for $tmp1 ($tmp2+1..$#inlist){
    $tmp4 = $inlist[$tmp1];
    $tmp3 = $inlist[$tmp2];
    if($inlist[$tmp2] > $inlist[$tmp1]){
      $inlist[$tmp1] = $inlist[$tmp2];
      for $tmp1 (0..$#inlist){
        if($tmp4 != $tmp3){
          $inlist[$tmp2] = $tmp4;
        }
      }
    }
  }
}
```

Listing C.53: Sort Seed 2

```
for $tmp2 (0..$#inlist){
  for $tmp1 ($tmp2+1..$#inlist){
    if($inlist[$tmp1] <= $inlist[$tmp2]){
      $tmp3 = $inlist[$tmp1];
      $inlist[$tmp1] = $inlist[$tmp2];
      $inlist[$tmp2] = $tmp3;
    }
  }
}
```

Listing C.54: Sort Seed 3

```
for $tmp2 (0..$#inlist){
  for $tmp1 ($tmp2+1..$#inlist){
    if($inlist[$tmp1] < $inlist[$tmp2]){
      $tmp3 = $inlist[$tmp1];
      $inlist[$tmp1] = $inlist[$tmp2];
      for $tmp1 (0..$#inlist){
        $inlist[$tmp2] = $tmp3;
      }
    }
  }
}
```

Listing C.55: Sort Seed 5

```
for $tmp2 (0..$#inlist){
  for $tmp1 (0..$#inlist){
    if($inlist[$tmp2] <= $inlist[$tmp1]){
      $tmp4 = $inlist[$tmp1];
      $inlist[$tmp1] = $inlist[$tmp2];
      $inlist[$tmp2] = $tmp4;
    }
  }
}
```

Listing C.56: Sort Seed 7

```
for $tmp2 (0..$#inlist){
  for $tmp2 ($tmp2+1..$#inlist){
    for $tmp1 (0..$#inlist){
      $tmp4 = $inlist[$tmp1];
      if($tmp3 <= $inlist[$tmp1]){
        if($tmp4 >= $inlist[$tmp2]){
          $tmp3 = $inlist[$tmp2];
          $inlist[$tmp1] = $tmp3;
          $inlist[$tmp1] = $inlist[$tmp2];
          $inlist[$tmp2] = $tmp4;
        }
      }
    }
  }
}
```

Listing C.57: Sort Seed 11

```
for $tmp1 (0..$#inlist){
  for $tmp1 ($tmp1+1..$#inlist){
    for $tmp1 (0..$#inlist){
      for $tmp2 ($tmp1+1..$#inlist){
        $tmp4 = $inlist[$tmp2];
        if($inlist[$tmp1] >= $inlist[$tmp2]){
          $inlist[$tmp2] = $inlist[$tmp1];
          $inlist[$tmp1] = $tmp4;
        }
      }
    }
  }
}
```

Listing C.58: Sort Seed 13

```
for $tmp2 (0..$#inlist){
  for $tmp2 (0..$#inlist){
    for $tmp1 ($tmp2+1..$#inlist){
      $tmp3 = $inlist[$tmp2];
      if($tmp3 >= $inlist[$tmp1]){
        $tmp4 = $inlist[$tmp1];
        $tmp3 = $tmp4;
        $inlist[$tmp1] = $inlist[$tmp2];
        $inlist[$tmp2] = $tmp3;
      }
    }
  }
}
```

Listing C.59: Sort Seed 17

```
for $tmp2 (0..$#inlist){
  for $tmp1 (0..$#inlist){
    for $tmp2 ($tmp1+1..$#inlist){
      if($inlist[$tmp2] <= $inlist[$tmp1]){
        $tmp3 = $inlist[$tmp1];
        $tmp4 = $inlist[$tmp1];
        $inlist[$tmp1] = $tmp4;
        $inlist[$tmp1] = $inlist[$tmp2];
        $inlist[$tmp2] = $tmp3;
      }
    }
  }
}
```

Listing C.60: Sort Seed 19

```
for $tmp2 (0..$#inlist){
  for $tmp1 ($tmp2+1..$#inlist){
    if($inlist[$tmp2] >= $inlist[$tmp1]){
      $tmp3 = $inlist[$tmp1];
      $inlist[$tmp1] = $inlist[$tmp2];
      if($inlist[$tmp2] <= $inlist[$tmp1]){
        for $tmp1 (0..$#inlist){
          for $tmp1 ($tmp1+1..$#inlist){
            $inlist[$tmp2] = $tmp3;
          }
        }
      }
    }
  }
}
```

Appendix D

Complete Set of User Interfaces Evolved

This appendix contains the complete set of user interfaces generated in Chapter 5, for the 'Personal Details' web page and the list functions front-end.

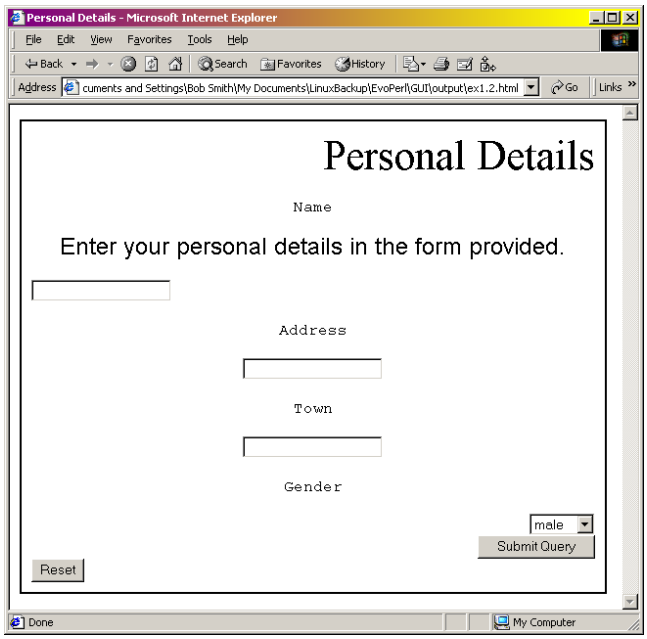


Figure D.1: Personal Details Form, Seed 2

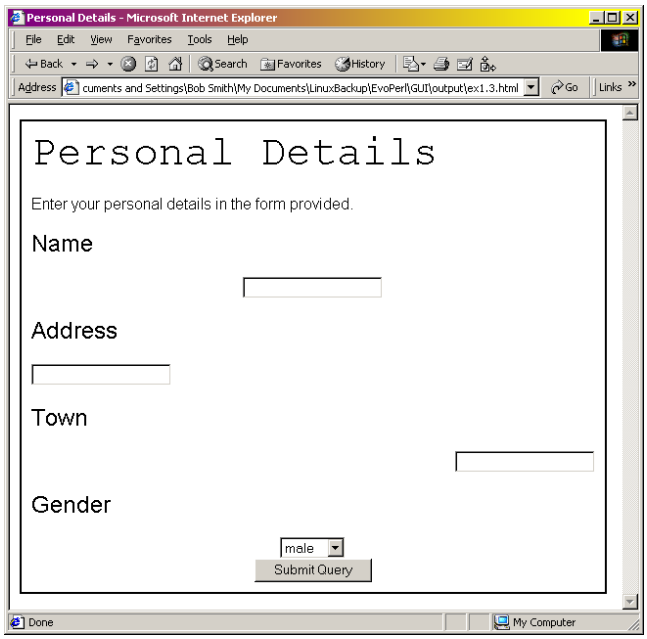


Figure D.2: Personal Details Form, Seed 3

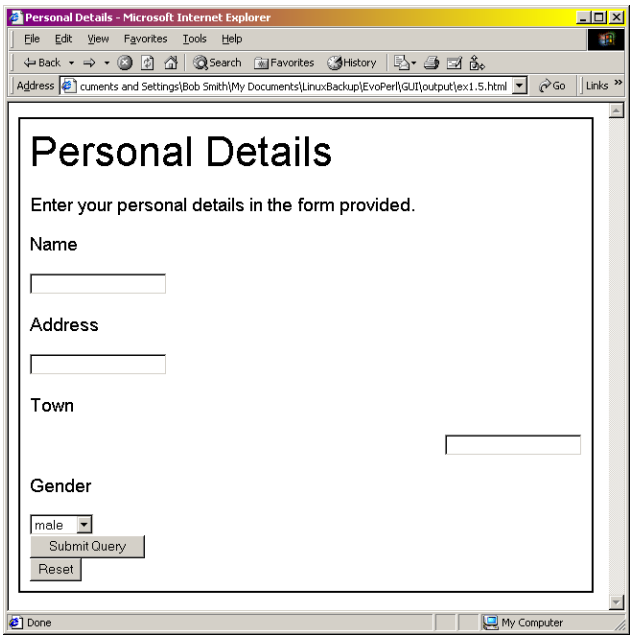


Figure D.3: Personal Details Form, Seed 5

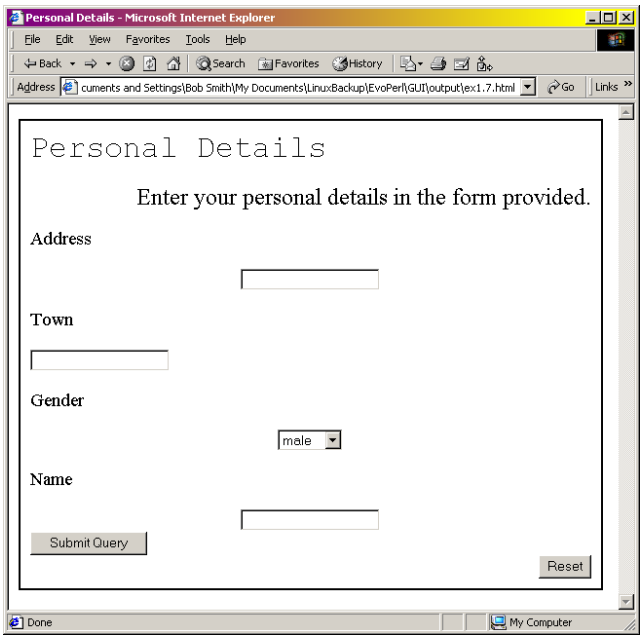


Figure D.4: Personal Details Form, Seed 7

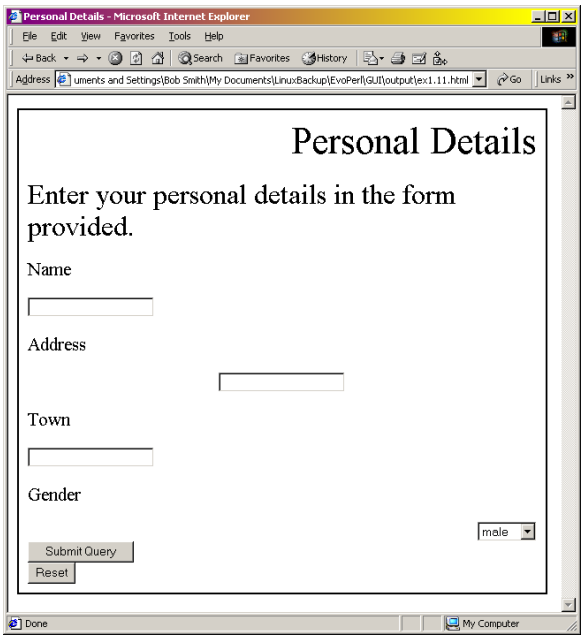


Figure D.5: Personal Details Form, Seed 11

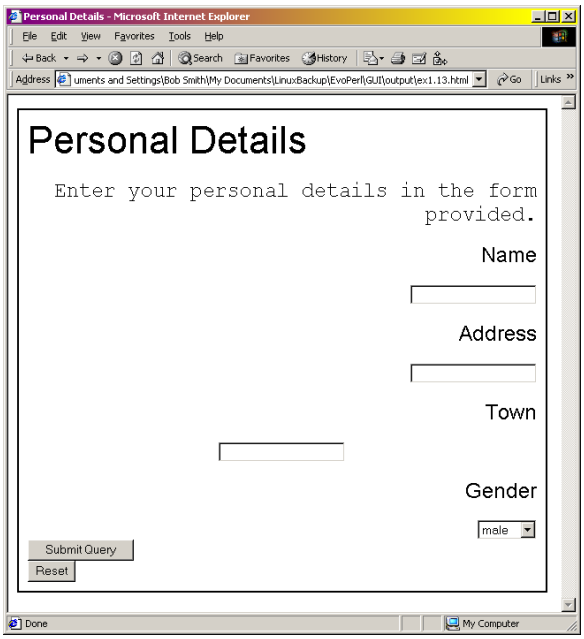


Figure D.6: Personal Details Form, Seed 13

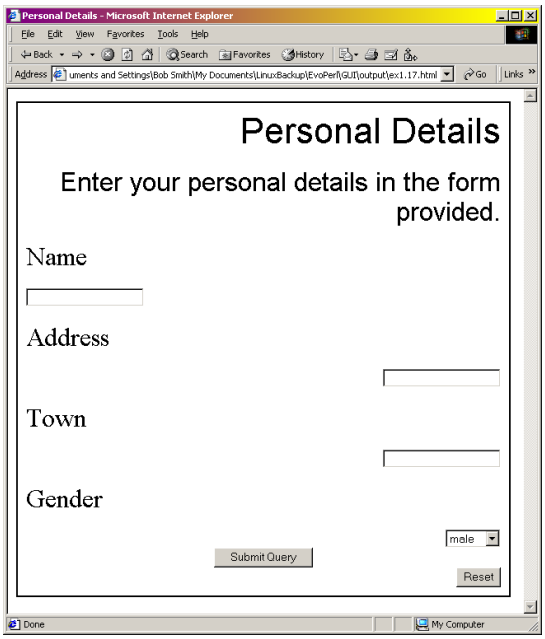


Figure D.7: Personal Details Form, Seed 17

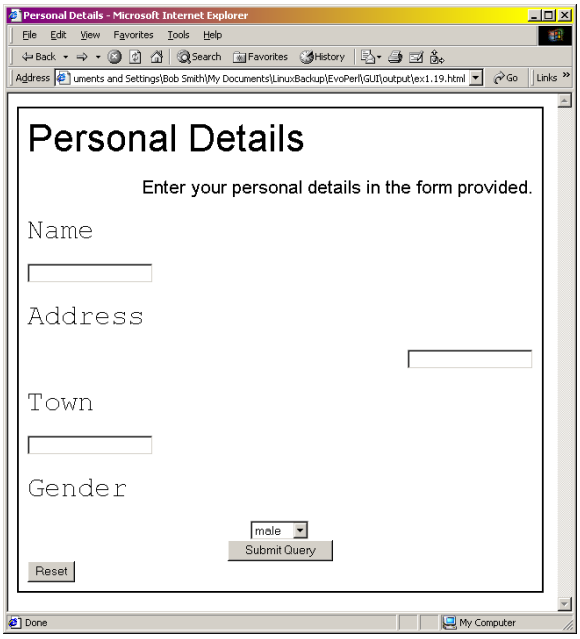


Figure D.8: Personal Details Form, Seed 19

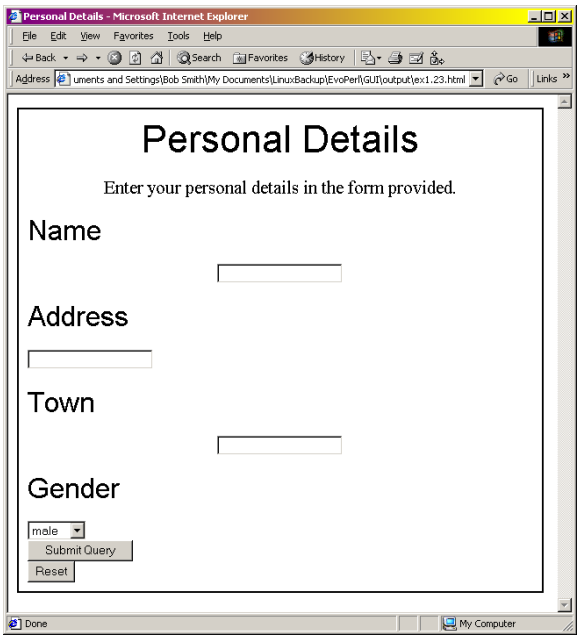


Figure D.9: Personal Details Form, Seed 23

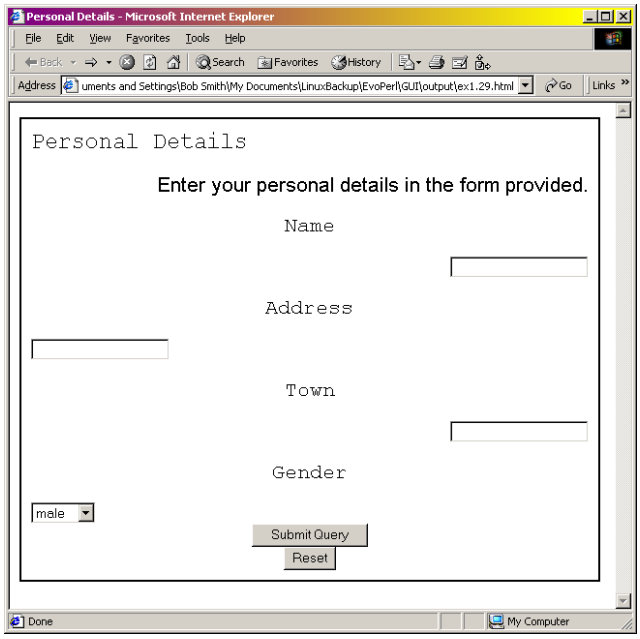


Figure D.10: Personal Details Form, Seed 29

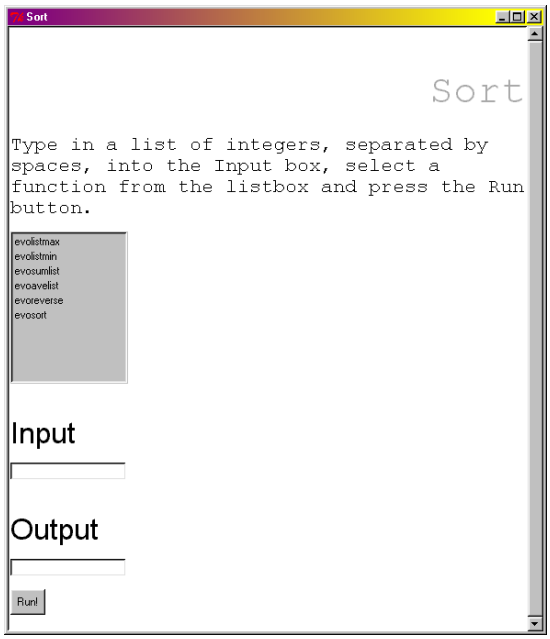


Figure D.11: Sort GUI, Seed 2

Figure D.12: Sort GUI, Seed 3

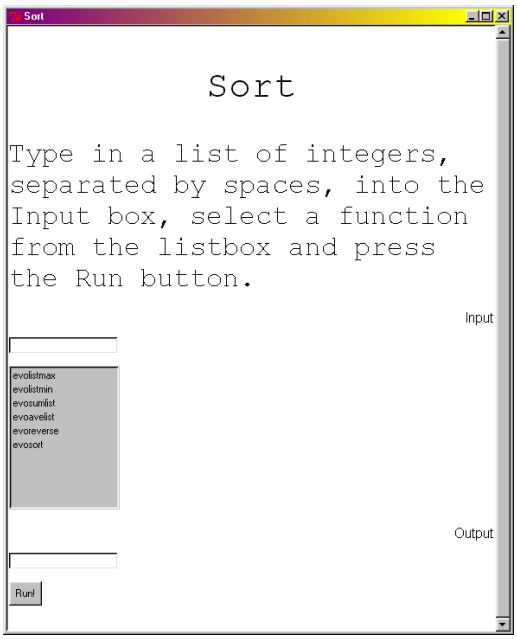


Figure D.13: Sort GUI, Seed 5

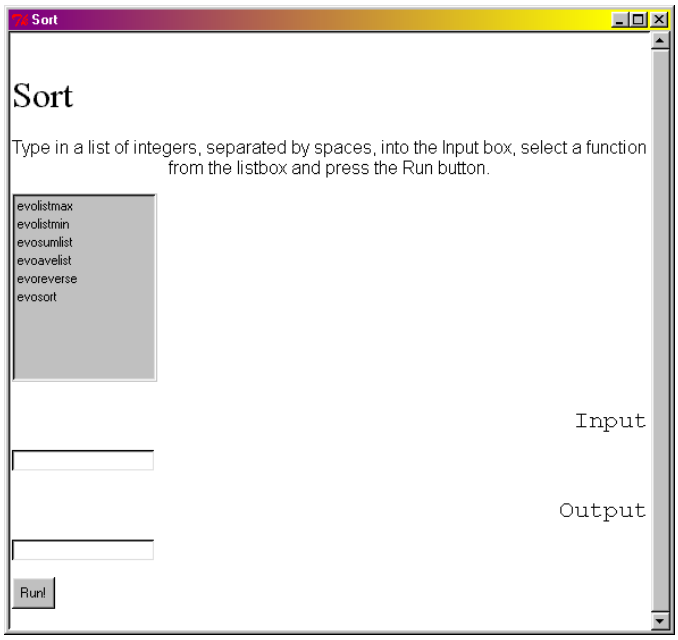


Figure D.14: Sort GUI, Seed 7

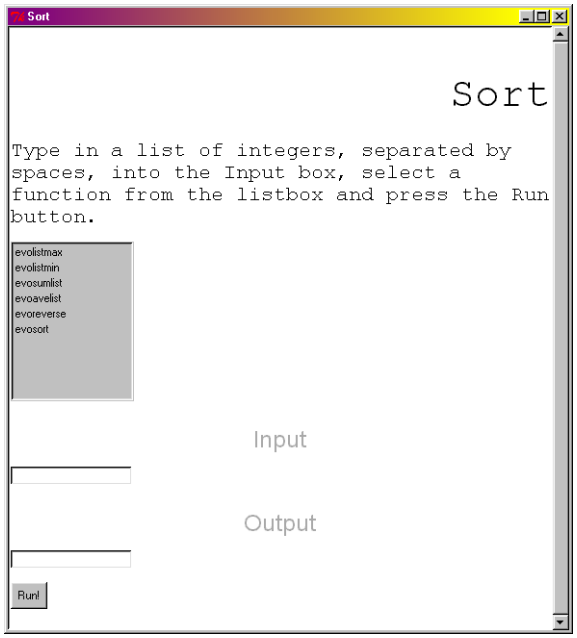


Figure D.15: Sort GUI, Seed 11

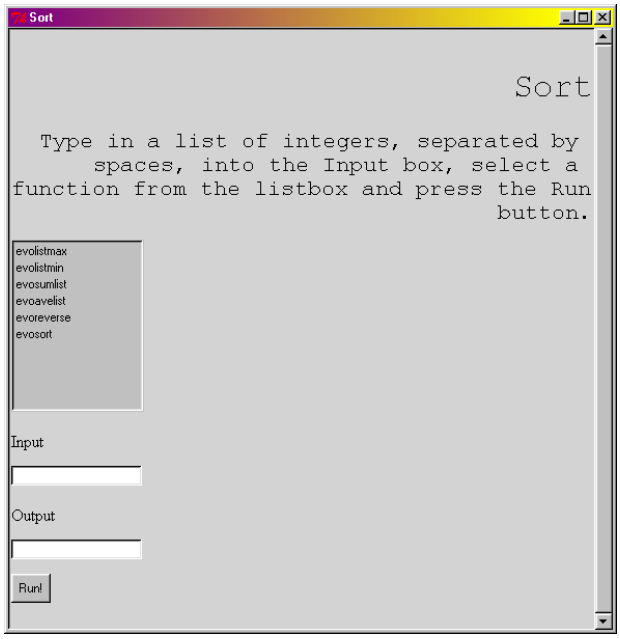


Figure D.16: Sort GUI, Seed 13

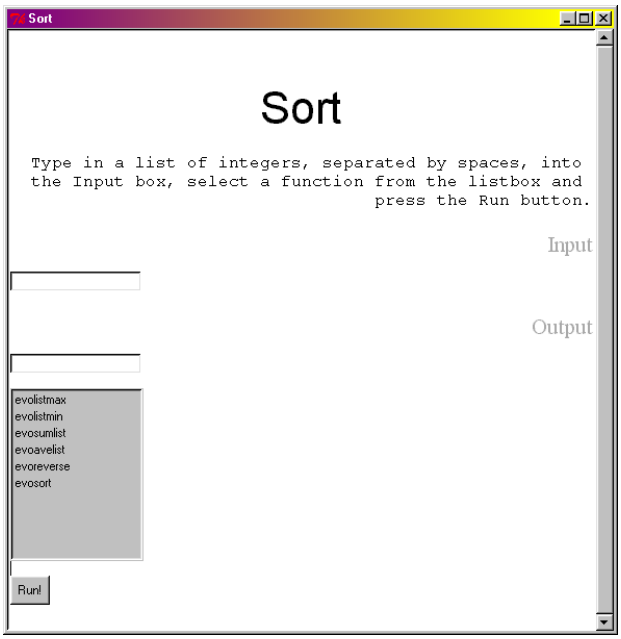


Figure D.17: Sort GUI, Seed 17

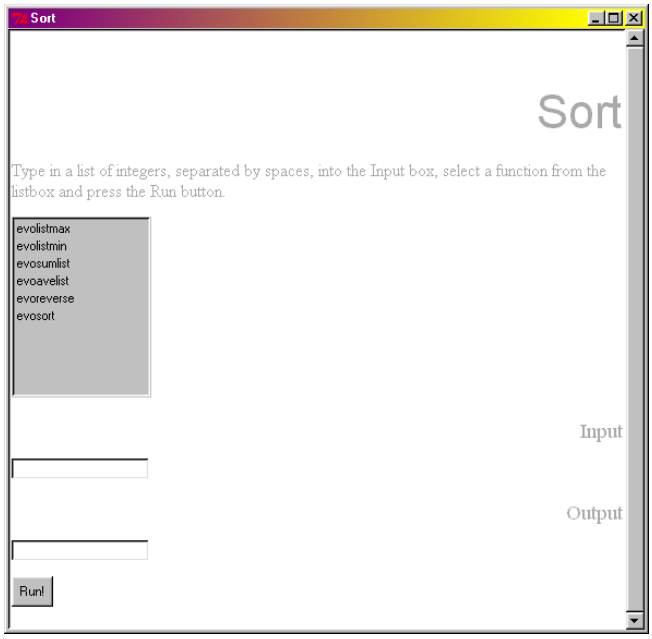


Figure D.18: Sort GUI, Seed 19

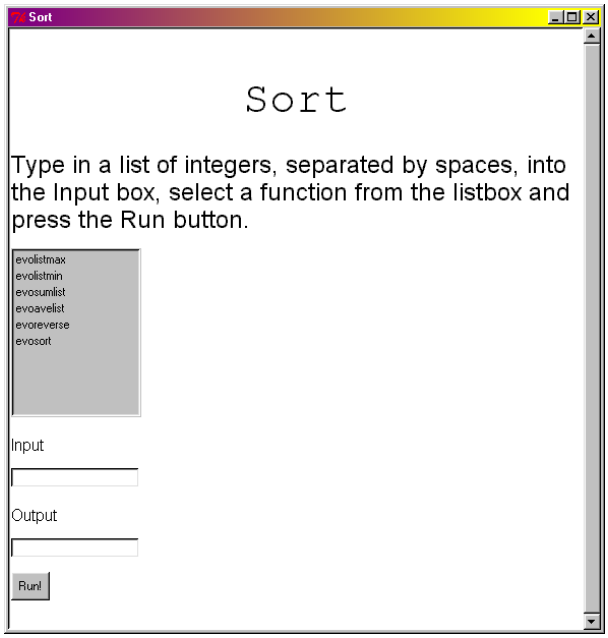


Figure D.19: Sort GUI, Seed 23

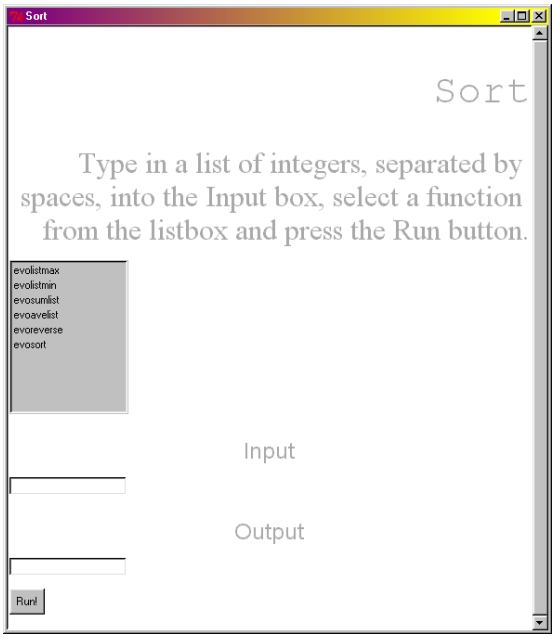


Figure D.20: Sort GUI, Seed 29