Algorithm Induction, Modularity and Complexity

John R. W. Woodward

A thesis submitted

to The University of Birmingham

for the degree of Doctor of Philosophy

School of Computer Science

The University of Birmingham

Birmingham B15 2TT

United Kingdom

September 2004

Dedication

This thesis is dedicated to my parents, family and friends.

Abstract

We are concerned with the induction of a rule from a set of observations, the goal being to succinctly describe the observed data, but more importantly to place us in a position to make predictions about future data which we have not previously seen. One approach to rule induction is to form a hypothesis space which consists of potential rules or hypotheses, and then search this space for a rule which is consistent with the observed data. One formulation of this approach is Genetic Programming, where the hypothesis spaces consists of computer programs and the search of this space is conducted using biologically inspired search operators and a fitness function.

The well known No Free Lunch theorems are central to search and essentially says all search algorithms perform equally, under a number of assumptions. We examine these assumptions and show that they are invalidated when the hypothesis space contains hypotheses which represent functions with different frequencies, as is the case with Genetic Programming and a number of other Machine Learning paradigms. The Conservation of Generalisation, which is related to the No Free Lunch theorems, implies that generalisation is impossible. This is contrary to Occam's razor. The Conservation of Generalisation theorems and Occam's razor are consistent if we restrict ourselves to representations which do not compress the observed data.

We define the representational complexity of a function to be the minimum size of a given representation which can express the function. Given a primitive set, and the operation of composition, new functions can be constructed, which is the representation standard Genetic Programming uses to express functions. However the complexity of a function under this type of representation will depend on the primitive set. We prove that if a representation is capable of expressing modules (i.e. reuse of component parts of the representation), then the complexity of a function is independent of the primitive set (up to a constant which depends on the primitive set but not on the function being represented).

We then conduct a number of experiments related to the evolution of Turing Complete representations. We argue that, if a representation can address the general case of a variable sized problem then the average number of evaluations to find a solution is independent of the problem size. In Genetic Programming, a fitness function is used to drive the evolutionary process and promote

programs which are more likely to lead to solutions. We experiment with a fitness function which includes information about whether or not a program had to be aborted during it's evaluation and demonstrate that a 10 fold reduction in the number of evaluations can be achieved on an arithmetic problem. Finally, we experiment with a crossover operator which is inspired by the theory of recursive functions and reduced the probability of producing a program which has to be terminated during its evaluation.

Acknowledgments

I wish to acknowledge the following people: Xin Yao, Julian Miller, Jon Rowe, Riccardo Poli, Achim Jung, Mark Ryan, Peter Hancox, Gavin Brown, Mark Roberts, Stefano Cattani James Neil, Tim Kovacs, Stuart Reynolds, Dean Petters, Tebogo Seipone, Rachel Harris, Pete Duell, and Noel Welsh.

Thanks, also, to Alan White for a thought provoking afternoon on the topic of statistical tests.

Contents

Chapter 1

Introduction

1.1 Introduction

A computer is an all purpose machine (universal machine) which requires programming if it is to perform a useful task. This involves developing an effective procedure for the machine to execute, which is normally done by a human programmer. Koza poses the question ([35], page 1, attributed to Arthur Samuel in the 1950's)

> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do want is needed to be done, without being told exactly how to do it.

In other words, how can we induce a computer program?

1.1.1 Induction of rules

The problem of induction is a very general setting for modern science. The aim of induction is to start with a set of observations and attempt to create rules that account for the observations. Induction is the continual process scientists go through, attempting to arrive at better explanations of the physical world as more observations are accumulated. These observations are often not passive, but based on carefully design experiments.

This is a similar process to that a child goes through, albeit unconsciously, when learning a language. Samples of sentences are witnessed and original grammatically correct utterances can be produced by the child after a number of years. The originality and correctness of these sentences implies that the child has learnt the grammar of the language and can apply it to novel situations.

One of the aims of induction is to be able to come up with a succinct explanation or description of

the observations. A different, yet related and perhaps more fundamental aim is the ability to extrapolate from the observed data in order to make general predictions about previously unencountered situations.

1.1.2 Automatic induction of rules

One can ask what is the process of looking for a general rule and can we replace the human by a machine in the inductive process and thus automate the inductive process? This is the concern of Machine Learning: the automation of the learning process. One approach is Genetic Programming (GP). We do not regurgitate the details of GP here, but refer the reader to Banzhaf et. al. [5]. While parts of this thesis are concerned with GP, or more generally Evolutionary Computation, we do not wish to give the reader the impression that this is the only or best approach to algorithm induction and we refer the reader to Mitchell [47] for an overview of the field of Machine Learning. Many of these approaches involve selecting a representation (e.g. a feed forward neural network), and a method of improving the performance according to some measure (e.g. the application of the back propagation rule to the neural network).

1.1.3 Complexity

The complexity of an object can be defined as how difficult it is to describe that object (with reference to some method of description). An object can be taken to mean a concrete physical object (e.g. a car) or an abstract object (e.g. a mathematical function or a set). In this thesis we will talk about abstract objects. A simple object will have a short or small description, a more complicated object will have a longer description. It should be noted that an object does not have a unique description, in general there will be many ways of describing a given object. Complexity is typically defined as the size of the smallest description with respect to some representation (see below for examples), but in general the complexity will depend on the method of representing our description. For instance under one representation, object A could be described simple, and object B would have a more complex description, while under a different representation, object B could be described simply and object A would have a more complex description.

Given a finite sample of data points, there are a number of different representations for describing them. We could describe the sample by giving the coefficients of a polynomial. Alternatively, the sample could be described using a summation of *sine* and *cosine* terms (i.e. as a Fourier transform). These two examples form complete orthogonal bases over a space, and the number of terms to describe the sample will depend on which basis you use. Other representations often using in AI are artificial neural networks. In this situation, a common measure of complexity is the minimum number of weights in a network which describes the underlying function. With Genetic Programming, a

measure of complexity is the minimum number of nodes in a tree which describes the function. Again, the complexity of a function will depend on which method we are using to describe the function, so we should always talk about complexity with respect to a description method.

One important definition of complexity is Kolmogorov complexity [43]. The Kolmogorov complexity of a bit string is defined as the length of the shortest program which runs on a given Universal Turing Machine (UTM) and halts. An important property of this definition is that the complexity of a bit string is invarient against the choice of UTM [43]. One drawback of this definition is that it is incomputable.

In general, the complexity of a function will depend on the type of representation. If we concentrate on the tree based representation used in GP, we find that the complexity of a function will in general depend on the primitive set. However we prove that, if the representation is capable of expressing modules, then the complexity is independent of the primitive set up to an additive constant (see chapter 4).

1.1.4 Modularity

Modularity is the ability of a representation to express its component parts in a form so than they may be reused in a single execution of the whole structure. For example, parts of a computer program maybe executed multiple times during a single execution of the whole program. In many representations used in machine learning this is not the case. For example, nodes in feed forward neural networks are only activated once during the presentation of an input pattern. Similarly nodes in a tree representation used in standard GP are also executed only once when exposed to an example of input data. As a module can be reused, it only needs to be expressed once somewhere in the representation, and therefore this ability has an affect on the complexity of functions when expressed by such a representation.

1.1.5 The relationship between induction, complexity and modularity

The reader may ask, what is the relationship between the nouns that appear in the title of this thesis. The problem of induction is the search for an algorithm which explains the given data. The difficulty to induce a solution is related to the complexity of the solution in the given representation. The more complex a problem, the more difficult it is to induce a solution. Given that the representation we are using syntheses functions from a set of primitives, and that the representation can express modules, then the probability of inducing a solution is independent of the primitive set. Thus when attempting to induce a solution, the complexity and difficulty is independent of the primitive set, if the representation can express modules.

1.1.6 The impact of the type of representation

The type of representation used for induction can have a great impact on the success of the process. Firstly, the representation may not be expressive enough to solve the problem, however this is typically not a problem in Machine Learning. Secondly, the type of representation can affect how easy it is to find a solution expressed in the given representation. We will illustrate these points with a number of different representations and the even parity problem.

Consider three types of representation, a single layered perceptron (SLP), a multi layer feed forward neural network (FFNN), and a recurrent neural network (RNN) (see Mitchell [47]). A SLP can be considered as a single layered FFNN. A RNN is a FFNN which allows feedback in the network. These three representations can be considered as more sophisticated versions of each other. Lets also consider the even-n-parity problem: given n bits, return true if the number of bits is even, otherwise return false.

Lets consider if these types of representation can express solutions to this problem and how easy it is to find a solution. Famously, a SLP cannot express the solution to this problem for any size n (see Mitchell [47] section 4.4.1). A two layer FFNN can express solutions to this problem for a fixed size n (see Mitchell [47] section 4.6.1), but cannot solve it for the general case (as it has no way of addressing an unknown number of variables). As the size of the problem increases (as indicated by n), the size of the smallest solution will grow and thus it will be more difficult to find a solution as n increases. A RNN can solve this problem for the general case, but more importantly, as the size of problem grows the size of a solution needed to solve the problem does not grow. Thus, here we have three types of representation of increasing representational power, and as we move from one representation to another we find that the easy with which a problem can be solved is affected by the representation.

1.2 Motivations for evolving Turing Equivalent systems

There are a number of motivations for studying induction using Turing Equivalent systems. Angeline [2] (in the introduction) states that, *'A poor representation limits what behaviours can be created while a good representation permits the induction of any potentially useful sequences of actions'* (see section 1.1.6). He goes on, *'an obvious candidate representation ... is a standard programming language... having the advantage of being Turing Complete, meaning they can represent any computable behavioural sequence'.*

Teller ([73], page 133) states, *'given that programs written in a Turing Complete language are more difficult to evolve than programs written in less expressive languages....why bother evolving them?'.* He has two answers; Firstly, Turing Complete programs are more expressive than other

reactive functions (which have no memory, which we comment on in section 1.2.3). Secondly, computer programs can express a function in less space than less expressive representations. We also add, there is also related to this second point, a more involved issue; If we change to a more expressive representation, the frequency with which a given function is represented will change, and thus, as we change the expressiveness of our representation, our confidence in our predictions changes (this is related to Koza's lens effect, see section 3.8). Below we describe our motivations for evolving Turing Equivalent systems, which are not listed in any particular order.

1.2.1 Sufficiency and representational power

When supplying a primitive set to GP, we must ensure the sufficiency property is met, which requires that the primitive set is capable of expressing a solution to the problem (Koza [35] page 86). The satisfaction of the sufficiency property guarantees that a solution to the problem *does* exist in the search space. In most cases in GP this is not a problem, however it may not always be clear. For example, Vallejo et. al. [78] are concerned with recognising bio-sequences. A finite state automata can be used, but this is only capable of discovery of repetitious contiguous patterns present in the bio-sequence, and may not be an expressive enough representation for this problem. As a Turing Complete representation is the most expressive representation we have, if a Turing Complete representation cannot meet the property of sufficiency then neither will any other representation.

1.2.2 Toward a theory of Artificial Intelligence

If we are to work toward a general theory of AI, it should certainly be able to make claims concerning induction using Turing Complete representations. Langdon et. al. [40] (page IX) illustrate current AI approaches as separate islands in a sea, and the foundations of AI existing below the sea. As a Turing Complete language is the most expressive representation, we suggest that an understanding of the induction of functions using Turing Equivalent representations must lie deep within such a theory. A theory of AI based on Turing Machines should encompass a theory of induction based on finite state automata or push down automata, as what these automata can express are subsets of what a Turing Machine can express. Vallejo et. al. [78] say, *'a GP based on the evolution of Turing Machines using genetic algorithms provides a convenient framework for understanding the fundamental theoretical capabilities and limitations of GP'*. We believe, not only should a theory of AI include induction using Turing Complete representations, it should be central!

1.2.3 Reactive systems

Much of Artificial Intelligence and Machine Learning has been concerned with learning reactive programs. These are mapping from a set of inputs to an output, where the output does not depend on the previous inputs, only the current inputs. Examples of representations which perform such functions are feed forward neural networks and decision trees. These are suitable for classification type tasks, where classification of the next case does not depend on previous cases. In some situations however we may need to retain state information (Teller [69]). To retain this information we need memory of some sort. There are a number of representations which are capable of storing information. For example Finite State Automata and Pushdown Automata can store information in terms of the current state the machine is in. The ability to store information is also a property of any Turing Complete representation.

1.2.4 Problem difficulty

The difficulty of a problem is related to its complexity (see section 3.2.5). In general the complexity of a problem will depend on the type of representation. The representation determines the problem compleixty. Since a Turing Machine is a universal computational model, using Turing Complete representations allows us to study the inherent problem difficulty.

Why is one problem more difficult to solve than another? We would probably agree that solving an even-n-parity problem is easier than solving a large image recognition problem. However the ease with which a problem can be solved will depend on the tools we have at our disposal. For example if we had a program which readily solves our image recognition problem then this problem is easy (i.e. we already have the solution so the problem is not difficult). Conversely, if we had a program which readily solves the even-n-parity problem, then this would be an easy problem to solve and it would take us time to construct a solution to the image recognition problem. Thus the difficulty of a problem is related to complexity, and just as the difficulty of a problem depends on what we have available to us, the complexity of a problem depends on the representation we have available.

Given that the problem difficulty depends on the representation we have chosen, it would be a desirable goal if we had some measure of the intrinsic difficulty of a problem. If the representation we choose is Turing Complete, then the complexity of a function is independent of which particular Turing Complete representation we use (up to an additive constant). Hence using a Turing Complete representation allows up to study the inherent difficulty of a problem.

1.2.5 Modularity, recursion and memory

It has been recognised in the GP community that modularity, recursion and memory may individually be helpful in providing solutions to problems (see chapter 2). These have been introduced into systems separately, however a system which is Turing Complete contains them all. We do not know in general if the inclusion of any of these features will help to find a better solution. While the addition of these features does not reduce the expressiveness of the representation, these additional features almost certainly make finding a solution more difficult using a Hill Climbing type search algorithm.

1.2.6 Academic interest

Last but not least we simply state that the problem of induction using Turing Complete representations is a worthy field of study simply due to its academic appeal, and the feeling of that *'ah-ha'* moment experienced when you understand something. As the physicist Richard Feynman once put it, it is *'the pleasure of finding things out'* [19].

1.3 Routes to studying the evolution of Turing Equivalent representations

Given that we are interested in the evolution of Turing Equivalent representations, we could 'jump in at the deep end', taking the direct approach and evolve a representation which has been shown to be equivalent to a Turing Machine (see section 2.1.3 for a list of possible models of computation). An alternative route would be to argue that one could create a new representation which is more suitable for the purposes of evolution, for example see [65].

One approach may be to examine the evolution of models associated with the Chomsky hierarchy. For example, one could study the evolution of finite state automata, then push automata, and come to the conclusion that, for example, the use of a graph based representation is better for evolution rather than using a state transition table as a representation for evolution. This may then lead to the insight that a graph based representation is better for the evolution of Turing Machines rather than using a state transition table as a representation for evolution. In studying finite state automata, then push automata, we may be able to learn some lessons which ultimately apply to the evolution of Turing Machines, but would have been more difficult to learn if we had taken the direct approach and attempted to evolve Turing Machines from the start. Another approach would be to study evolution of total recursive functions (see section 2.4) before going onto study the evolution of partially recursive functions.

In section 3.8.2 we introduce a hierarchy of representations which is more relevant to GP; starting with look up tables, then tree based representation often used in GP, then representations which can express modules, and finally Turing Equivalent representations. By studying the evolution of tree based representations, then moving on to study the evolution of representations which can express modules, we may be able to understand the evolution of Turing Equivalent representations better than if we had begun with the study of Turing Equivalent representations in the first place. There are other hierarchies of increasingly sophisticated representations which could be studied in order to ultimately understand the evolution of Turing Equivalent representations. It was the intention in this thesis, that by studying representations which can express modules (which is the concern of chapter 4), that insights into the evolution of Turing Equivalent representations would be gained (which is the concern of chapter 5). This view was taken as the invariants of the complexity of a function under the translation of primitive set is due to the ability of a representation to express modules.

1.4 Contributions contained in this thesis

The main contributions of this thesis are listed below:

- demonstrate that NFL theorems do not apply to situations where there is a many to one non uniform mapping between the representation and the object being represented (section 3.5)
- proof that the complexity of a function is invariant under the translation of primitive set (within a constant bound) (section 4.4).
- demonstrate that as problem size increases, the difficulty of the problem remains constant if a suitable representation is used, and therefore the average number of evaluations needed to find a solution is independent of the size of the problem (section 5.2).
- demonstrate that information about the proper termination of a program can be used in a fitness function to improve efficiency (section 5.3).
- conduct a number of experiments investigating the influence of different genetic operators on the chances of producing programs which terminate properly (section 5.4).

1.5 Publications resulting from this thesis

Below we list the papers published during this period.

- Simple Incremental Testing, Genetic and Evolutionary Computation 2004 Conference June 26-30 2004, Seattle, Washington USA. Late breaking papers.

- Function Set Independent GP, Genetic and Evolutionary Computation 2004 Conference June 26-30 2004, Seattle, Washington USA. Workshop on Modularity, Regularity, and Hierarchy in Evolutionary Computation.

- Evolving Turing Complete Representations, Congress on Evolutionary Computation, Canberra, Australia, 8th - 12th December 2003 (Nominated for Best Paper). (Also appeared in Proceedings of the UK Workshop on Computational Intelligence: UKCI-2003, Bristol, UK: University of Bristol).

- GA or GP, that is not the question, Congress on Evolutionary Computation, Canberra, Australia, 8th - 12th December 2003. (Also appeared in Proceedings of the UK Workshop on Computational Intelligence: UKCI-2003, Bristol, UK: University of Bristol).

- (with James Neil) No Free Lunch, Program Induction and Combinatorial Problems, GP 6th European Conference, EuroGP 2003 Essex, UK, April 2003. (Also appeared in Proceedings of the 2002 U.K. Workshop on Computational Intelligence: UKCI-2002, Birmingham, UK: The University of Birmingham.

- Modularity in Genetic Programming, GP 6th European Conference, EuroGP 2003 Essex, UK, April 2003.

1.6 Outline Of Thesis

In chapter 2, we review the current literature related to the evolution of Turing Equivalent representations, which also includes a short review of modular methods in GP. We comment on fitness functions, the generation of syntactically valid programs, the nature of the landscape produced by computer programs, the disruptive nature of crossover, and the halting problem. In chapter 3, we examine the No Free Lunch theorems, Occam's razor and Koza's lens effect. In chapter 4, we define representational complexity and prove that it is invariant under translation of primitive set if the representation used can express modules. In chapter 5, three separate sets of experiments are conducted. The first experiment shows that, as the size of a problem increases, if a representation is used under which the representation complexity does not increase, then the number of evaluations needed to find a solution remains constant. The second experiment shows that some advantage can be gained by using information in the fitness function about whether a program halted or not during testing. The final experiments investigate the effect of different genetic operators on the chances of

generating programs which halt. In chapter 6, we summarise the content of this thesis and draw conclusions.

Chapter 2

Literature Review

2.1 Introduction

2.1.1 Biological inspiration

Genetic Programming (GP) is the biologically inspired search of the space of computer programs. Survival of the fittest states that fitter individuals are more likely to breed and therefore pass there genetic material onto future generations. The inheritance of genetic material is due to the process of crossover, where genetic material from two parents is taken and combined to produce an offspring. Crossover is not the only action which can affect the frequency of genetic material; mutation may also introduce new genetic material into the population. It is these biological processes which GP takes as a starting point in order to construct a search algorithm to sample the space of computer programs.

The terms genotype and phenotype have been imported from biology to evolutionary computation. In biology the term genotype means the genetic make up of the individual (i.e. the genes which the individual contains). The term phenotype is the observed characteristics (i.e. the manifestation of the trait). In general many genotypes correspond to a single phenotype. In evolutionary computation, the genotype is the structure that is manipulated by the genetic operators. For example, in GP it is an instance of a tree structure, while in GA it is an instance of a bit string. The phenotype corresponds to the interpretation of the genotype. In GP, the phenotype corresponds to the function which is represented by the given tree structure. In GA, the phenotype may correspond to a number, if we are interpreting bit string as numbers.

2.1.2 Standard Genetic Programming

The basic GP algorithm can be described as follows. A set of test cases which define a problem are defined or selected from some set, a program which satisfies these test cases is said to be a solution to the problem. A fitness function is defined which maps a program to a number, which hopefully reflects how good a solution is. A population of computer programs is created from a primitive set (function and terminal set). Groups of programs are selected, based on their fitness (defined by a fitness function). Programs undergo genetic operations: crossover exchanges parts of programs and mutation replaces parts of programs with randomly generated material. New programs are reevaluated by the fitness function. This process is continued until either a solution is found, or some termination condition is met.

There is a large body of work on GP (see Banzhaf et. al. [5] appendices for mailing list, conferences etc). However little of this work has been concerned with evolving Turing Complete representations (i.e. the evolution of representations which have been proved to be Turing Equivalent). While there are potential benefits to be gained from using a Turing Complete representation (e.g. we can express a solution in a compact form) there are also disadvantages (e.g. dealing with the possibility that evolved programs may not halt). For a list of the motivations for studying program induction using a Turing Complete representation see section 1.2.

In this literature review we first examine the use of modules in GP and then go on to examine the evolution of Turing Equivalent representations. If a system is Turing Complete, it is therefore also capable of expressing modules. We regard modular methods to be a relevant step toward the evolution of Turing Complete representations because both of these types of representation allow reuse of components (see section 1.3). The details of GP are not repeated in this chapter and we refer the reader to Banzhaf et. al. [5].

2.1.3 Models of computation

We list a number of models of computation. A number of these have been used as a representation for evolution. We also list a number of 'accidental' models of computation (section 2.10), by which we mean they were intended to study some aspect other than computability but were later found to be Turing Equivalent (e.g. Ray's Tierra [51]).

If one examines the textbooks on computability we find there are many models of computation. For example see Cutland [13] (chapter 3, section 1) who lists 7 different approaches, including general recursive functions, lambda definable functions, Turing Machines and unlimited register machines. These are all models used explicitly to study the field of computability. Post (1943) also proposed 'Post production systems' which have also been evolved (http://www.whatisthought.com/ptech.pdf)

Teller [71] has also shown how the standard GP representation can be extended to become Turing Equivalent by the addition of Indexed Memory (in the form of read and write primitives) along with iteration.

2.1.4 Recursive functions

It is constructive to consider recursive functions as this is the class of computable functions and therefore the class of functions which we can potentially evolve. An understanding of the theory of recursive functions may help us understand the evolution of Turing Equivalent representations better. Recursive functions can be expressed using three basic operations: successor (also known as increment), zero (also known as clear), and projection, and three more sophisticated operations composition, recursion and minimisation, which can be used to construct new functions from previously defined functions. Functions made only from the basic operations of composition and recursion are called total recursive functions (i.e. they are total functions and therefore halt). Yu [93] and Cramer [12] evolve total recursive functions, (see section 2.4).

Basic operations

The basic operations are very simple and consist of the following three operations: increment, zero and projection. The increment operation maps a number n to $n + 1$. The zero operation maps a number n to 0. The projection is slightly different; given a list of n numbers and an integer i, we return the ith number in the list.

Composition

Composition, also known as substitution, is a way of creating new functions. Given functions $f(x)$ and $g(x)$, a new function $h(x)$ can be constructed by composition $h(x) = f(g(x))$. This is the operation standard GP uses to build up more complicated functions from the function and terminal set.

Recursion

Given functions $f(x)$ and $g(x, y, z)$, a new function $h(x, y)$ can be constructed, and is defined as follows;

$h(x, 0) = f(x)$ (base case)

$h(x, y + 1) = g(x, y, h(x, y))$ (recursive case)

We list a number of papers looking at the evolution of recursion in section 2.1.5.

Minimisation

Given a function $f(x, y)$, the function $\mu y(f(x, y) = 0)$ is defined as the least y such that $f(x, z)$ is defined for all $z \leq y$, and $f(x, y) = 0$. Otherwise, if there is no such y it is undefined. It is due to the operation of minimisation that we encounter the partial functions or non terminating programs.

2.1.5 Evolving recursion

We will not dwell on this work but simply acknowledge the fact that work concerning evolution of recursion exists [8, 82, 87].

2.1.6 Outline of chapter

We begin with a brief review of learning with finite automata 2.2 and continue with modular search methods (section 2.3), and then review work regarding the evolution of total recursive functions (section 2.4). We then move onto looking at the evolution of a number of different models of computation: Turing Machines (section 2.5), unlimited register machines (section 2.6), a graph based representation used in PADO (Parallel Algorithm Discovery and Orchestration) (section 2.7), a number of programming languages (section 2.8), Teller's Indexed Memory (section 2.9) and 'accidental' models of computation including Tierra (section 2.10). We then move on to discuss various aspects of the evolution (section 2.11): fitness functions (section 2.11.1), the generation of syntactically valid programs (section 2.11.2), the nature of the landscape produced by computer programs (section 2.11.3), the disruptive nature of crossover (section 2.11.4), the halting problem (section 2.11.5), the generality of solutions produced (section 2.11.7), problems tackled (section 2.11.6), and different program representations (section 2.11.8). We then round off this literature review with a summary and conclusion sections, 2.12 and 2.13.

2.2 Review of learning with finite automata

In this section we briefly review work done concerning learning with automata. Gold [23] showed that deciding if there exists a DFA with n states which is consistent with a set of labelled strings is NP-complete. If all strings of length n or less are given then the problem of finding a minimum sized DFA can be solved in polynomial time in the size of the training set, however the number of the training rises exponentially with the size of the input [50]. Angluin[3] has shown that even if an arbitrarily small fixed fraction of strings is missing, the problem remains NP-complete. If the algorithm is allowed to make membership quires the problem can be solved in polynomial time [4].

2.3 Review of modular search methods

In this section a number of modularisation methods are discussed. For each of these methods a representation is needed along with at least one operation which produces new candidate solutions from old ones.

2.3.1 Automatically defined functions

Automatically Defined Functions (ADFs) [35, 36] are probably the most well known of the modularisation methods used in GP. Along with the main tree, additional branches are allowed to hang down from the root of the tree which define ADFs. ADFs are used just as if they belonged to the primitive set. Before starting the run, the user has to decide on the number of ADFs, the number of parameters each ADF takes, and which ADFs are able to call which other ADFs. These choices all affect what structures are included in the search space.

2.3.2 Architecture altering operations

Architecture Altering Operations [38] are a natural extension to ADFs which allow the structure of a solution to alter thus freeing the user from having to make these choices. ADFs can be created or deleted and new parameters added or removed from the argument lists of ADFs. Thus the number of ADFs and the parameters each one takes can change during the run. As pointed out in Banzhaf et. al. [5] (page 288), this has the benefit that no extra parameters are required in addition to the parameters required for standard GP run without ADFs.

2.3.3 Adaptive representation through learning

Adaptive Representation [54] creates modules on-the-fly during evolution according to population statistics. In a sense this is very similar to Architecture Altering Operations (see section 2.3.2). in that the form of the representation is free to alter during the run. Instead of taking program segments and placing them in a library, the segments are added to the function set, thus adapting its representation as it learns.

2.3.4 Module acquisition

Modules in Module Acquisition are available to all programs in the population [1] unlike ADFs, which are local to each program in the population,. Two additional genetic operators are required to deal with modules: *compression* and *expansion*. Compression randomly selects a sub tree (down to a certain depth), placing a copy of the sub tree in the global library and a reference to the sub tree

in the library in the individual. This has the effect of reducing the size of the individual. Expansion expands out a reference to the library in the individual with the actual sub tree. While a module is in the library it does not undergo evolution. Compression has the effect of protecting modules.

2.3.5 Encapsulation

Encapsulation [35] is very similar to Module Acquisition [1] just described in section 2.3.4. With encapsulation, a non terminal in a tree is chosen and the subtree from that point down to all the terminals in that subtree is 'encapsulated'. This encapsulated subtree is then treated as a new terminal (because it takes no arguments).

2.3.6 Automatically defined macros

Macros are part of most programming languages and are expanded before the compilation stage. A common example is substitution where frequently used pieces of code are replaced by macros. Spector [64] shows how GP can simultaneously evolve a main program along with a set of automatically defined macros (ADMs). There can be side effects which one does not get with ADFs concerning the evaluation of function arguments. However, the semantics of ADFs and ADMs are the same if we are concerned with a functional domain.

2.4 Total recursive functions

In section 2.1.4 we stated that the class of total recursive functions defined by the basic operations along with composition and recursion can only produce halting programs. We identify a number of areas of work which have evolved total recursive functions.

2.4.1 Cramer

Cramer [12] works on the problem of evolving a multiplication function. He takes the PL language (which is Turing Equivalent) which has 4 instructions: `INC`,`ZERO`,`LOOP`, and `GOTO`. Programs consist of lists of these 4 instructions and have an arbitrary number of globally scoped integer variables. He studies a less expressive language (PL-:GOTO) which consists of the instructions of PL without the instruction `GOTO`. Programs written in PL-:GOTO are guaranteed to halt (essentially because we cannot refer back to previous instructions indefinitely) and correspond to the set of primitive recursive functions. He also adds a `SET` instruction and a `BLOCK` operator that accepts two statements as arguments and evaluates them sequentially, which is a grouping operation and calls this the JB language. In the JB language, a program is represented as a list of integers. To interpret a list, it is

split into sections of three integers, the first integer is the instruction and the remaining two integers are the arguments. Each set of three integers therefore corresponds to a program statement. The first statement is called the Main Statement (MS) and the remaining statements are the Auxiliary Statements (AS). He comments that this language is unsuitable for evolution as it is very sensitive to changes because *'a single unfortunate mutation .... would destroy any useful features of the program'* and the language TB is introduced which has a tree like structure. Genetic operators act near the leaves of the tree in order to avoid the *'catastrophic minor change'* problems JB suffered from.

The key to designing an effective evaluation function is one that rewards multiplication-like behaviour (i.e. reflects the fact we are moving nearer the global optima). Cramer [12] used the following scheme for evaluation *'after much experimentation'* (indicating the difficulty of designing a good fitness function). Programs with the following types of behaviour were given successively more credit.

1. the output variables had changed from their initial values
2. simple dependency of the output on the input
3. the value of the input is a factor of the output
4. the output is the product of the two inputs

A limit is placed on the length of time a function is permitted to run, and any function that has not halted in this time is aborted.

2.4.2 Yu

Yu [93] presents a novel approach using higher order functions to evolve recursive modular programs. Higher order functions are functions which can take functions as arguments. Implicit recursion is used and has the important characteristic that programs always terminate as terminating conditions are incorporated into the higher order functions (page 354 [93]). Yu looks at the general even parity problem, all 40 of the solutions produced are general (page 361 [93]). Interestingly only crossover is applied in this work. A recursion limit of 100 is set. The fitness score works as follows; a programs gets a 1 if it is the right answer, 0 if not and if it a head or tail operation is applied to an empty list its fitness is reduced by half.

2.5 Turing Machines

In this section we review three works ([68, 67, 78]) where the state transition table of a Turing Machine is evolved. Turing Machines are possibly the most well known models of computation, yet

it is interesting that little work has been done using them as a representation for evolution.

Tanomaru et. al. [68] evolves a population of state transition tables which describe Turing Machines. He argues that *'since automata are typically represented by their state transition tables, this seems to be the most natural representation to be adopted'.* The genetic operators do allow creation of automata which contain transitions to non-existent states. His approach allows these machines to exist but assigns them low fitness. Fitness proportional selection is used to select individuals for either crossover or mutation. Crossover exchanges randomly selected contiguous rows in the state transition table, and the children therefore maybe of different length to their parents. The size of the individuals can therefore vary but a maximum size of 20 was imposed. Mutation changes the contents of a percentage of randomly selected cells of the state transition table. A mutation operator which varies the length of the individual was not used.

They tackle two problems of sorting tapes of two symbols and proper subtraction. For sorting, a mixed sequence of a's and b's are sorted. Proper subtraction is applied to numbers represented in unary notation (e.g. 11111 represents the number 5). Only non-negative numbers can result (i.e. if the first operand is smaller than the first, zero is returned - there are no 1's on the tape).

Turing Machines were terminated if:

1. the head advanced beyond the tape's limits, or
2. the machine fails to stop within a maximum number of steps, or
3. the machine stops acting, or
4. the machine refers to a non existent state.

A cost function was used which reflected the error score of a Turing Machine and its complexity and was linearly encoded into fitness values between 1 and 10 (they do not give details).

Solutions evolved were general. Generality is demonstrated by further testing on additional test cases not previously seen by the Turing Machine. The generality of the best evolved solution for the sorting problem is proved in the appendix of the paper [68].

In a second paper Tanomaru [67] introduces an enhanced evolutionary approach. He argues that crossover is not an effective procedure as it can result in meaningless automata (as they refer to non existent states) and thus valuable processing time is wasted. Crossover is entirely dropped as *'it is based on the building block hypothesis, which is very unlikely to hold in the case of automata generation'* (he offers no explanation of this point of view is offered). Three new mutation operators are introduced, the first mutates at the cell level, the second adds or deletes states. In the case of deletion, the least visited state is removed and all references to the deleted state are changed to refer to a different state. In the case of addition, a new state is added and its cells are filled with

random values and one of the cells corresponding to other states is changed to refer to the new state. The final mutation operator replaces the whole automata with an entirely new automata. Three problems were used to assess this improved approach; the recognition of a regular, context free and a context sensitive language. Results indicate that the enhanced approach outperforms the simpler approach detailed in the previous paper [68].

Vallejo et. al. [78] demonstrate that Turing Machines can be evolved with the capabilities of bio-sequence recognition. In order to realise a Turing Machine they introduce two limitations; the size of the tape and the maximum number of computations. A generational genetic algorithm with tournament selection and elitism is used to evolve a population of Turing Machines. The genome consists of a linear concatenation of the rows of the state transition table. They point out that as the representation is not binary, genetic operators are properly designed to avoid generating syntactically invalid structures. The fitness function is the number of correctly classified HIV bio-sequences. Negative training examples are generated randomly with similar nucleotide frequencies. The experiments use a machine with 32 states and a tape alphabet consisting of 8 symbols. As a GA is used the number of states is fixed. It is not stated how many of these states are actually active in the final solution. The GA found a Turing Machine which correctly classified all *training sequences* and accepted *several sequences* not included in the training set.

An alternative to the evolution of the state transition table would be to evolve the input to a universal Turing Machine. Also these three works have a Turing Machine with only one tape. There may be some advantage in having three tapes (an input tape, a working tape and an output tape). For example having a read only input tape would ensure that the input is not mistakenly overwritten and is always available.

We believe applying crossover to a state transition table does not make much sense as contiguous sections of states of a transition table do not relate to each other and a graph based representation is more natural. Thus we suggest a graph based representation may be more suitable for evolution.

2.6 Register machines

2.6.1 Huelsbergen

Huelsbergen presents a series of papers [25, 26, 27] in which he evolves programs for a register machine. His register machine consists of a number of registers which hold integer values, and a flag which can have one of three values (less than, greater than or equal). Programs consist of a series of instructions. A program counter tells the program which instruction is to be executed next and typically has a value between 0 (pointing to the first instruction in the program) and *length of program -1* (pointing to the last instruction in the program). Examples of instructions included

in this language are `compare` and `jump`. The `compare` instruction compares the values of contents of two registers and sets the value of the flag depending on the relative values of the contents of the two registers. There are a collection of conditional `jump` instructions which jump by the value indicated by their argument depending on the value of the flag. A number of other instructions include `increment`, `decrement` and `clear`. In all of these papers [25, 26, 27] an upper limit is placed on the number of instructions that are executed.

In [25] Huelsbergen evolves a multiplication function using an addition instruction. Programs which have not terminated within a predefined number of steps are halted in order to obtain a fitness value. In this system (unlike Cramer's [12]) an explicit looping structure is not used which makes the problem harder. The fitness function is the sum of the magnitude of the error score plus the magnitude of the difference between the program counter and the length of the program. The programs are evolved on a set of 121 test cases (all products of pairs of number between 0 and 10). Fitness proportional selection is employed and two point crossover exchanges sections of contiguous code of the same length between two parents. The positions of an instruction may vary but the overall length of a program does not vary as the two sections have the same length. With a certain probability an instruction in a program is changed to a random instruction. Two experiments were performed, experiment I used two point crossover and experiment II used two point crossover with mutation. Programs had a fixed length of 8 instructions and 4 registers were available. Experiment II produced about double the number of solutions compared to experiment I, and both performed better than random search. The solutions produced are general (checked by hand).

In [26] Huelsbergen evolves programs which generate recursive sequences (squares, cubes, factorial and Fibonacci numbers). In this paper the instruction set has additional instructions (subtraction, multiply and division). In a similar vein to his previous paper, he demonstrates that recursive sequence generating programs can be evolved without using an instruction set which includes an explicit recursion operator. Proportional selection is used. Two point crossover is used (the same as in [25]). A second search operator, exhaustive iterative hill climbing (EIHC) is introduced. For a given program, each instruction is replaced by all possible instructions (i.e. the complete neighbourhood is examined). The neighbour with the largest increase in fitness is returned as the new point in the search space from which we continue the search. If no better neighbour exists, a new program is generated at random. This is a steepest gradient search. Over the four problems crossover, EIHC and a hybrid operator consisting of mixture of the two operators all perform better than random search. The system learns from sequences of length 10, and the solution produced are general.

In [27] Huelsbergen evolves solutions to the parity problem. The test set consists of all bit strings of length 5 (i.e. 2^5 test cases). He claims that no domain specific operators are used, however he

does include a number of logical operators not included in the previous papers [25, 26]. Two point crossover and single point mutation are employed as described above. Headless chicken crossover is also used, where a randomly generated contiguous group of instructions is inserted into a program, rather than being inherited from a parent. Interestingly on this problem macro-mutation significantly outperforms crossover. Solutions obtained were general.

It is constructive to compare these machines described above with the Register machine described in Cutland [13] which uses only four instructions ([13] page 12). For example the addition instruction in [25] is redundant in the sense that it too could be evolved from the remaining primitives. The inclusion of the addition instruction indicates that we have specialist knowledge of the problem, and it would be desirable if this was not necessary.

All of the solutions obtained in these papers [25, 26, 27] could be described as containing a single loop. There is a looping structure which executes a section of code repeatedly however, no nested loops were evolved (but were not needed for these problems). In [27] he gives the corresponding C programs which all contain one loop. As all of the solutions to these problems only contain one loop, there is no obvious building block so it is perhaps not surprising that crossover did not perform well.

2.6.2 Schmidhuber

Schmidhuber et. al. produced a series of papers [30, 58, 57, 59]. These papers use a probability distribution over instructions which are part of programs written in as assembly type language. These papers are not concerned with a population based evolutionary approach but rather learning to learn where an individual program alters its own learning bias as it receives reward from the environment. In these works the agent can potentially use any learning algorithm as the instruction set is Turing Complete (this therefore has the advantage over other self adaptation approaches which can only alter the way they learn in a very restricted sense). There are a number of "normal primitives" which perform standard mathematical operations on the registers. There are also a number of special primitives which can alter the probability distributions of the instructions in the program (and therefore alter how the program alters itself). Another special primitive signals to the program when to assess its own learning.

Schmidhuber et. al. [58] tackle a simple maze navigation task. The solution produced consists of two explicit nested loops [58] (page 9) and [59] (page 116). In a second implementation, a function regression problem is tackled and a single looping program is evolved [58] (page 17) and [59] (page 127).

In principle we believe what Schmidhuber et. al are trying to achieve is interesting as evolved biases may be better than human designed bias. However as the system can only currently solve simple problems, it is unlikely to be evolving complex learning algorithms (this is in a sense part of

the problem).

2.7 Parallel Algorithm Discovery and Orchestration - PADO

In PADO (Parallel Algorithm Discovery and Orchestration) [72], a program is represented as a graph, has 3 parts; a main program, one or more private ADF programs and an Indexed Memory. There is also a library of globally available ADFs. Each program also has an argument stack which all PADO actions pop their inputs from this stack (which is different to other GP systems where the nodes communicate directly) and push their results back onto the stack. The genetic operators (SMART recombination), are a form of meta-learning which are allowed to co-evolve along with the programs in the population. The fitness value of an operator is the relative fitness of the programs they take as input and produce as output. They insist PADO is an "anytime" algorithm meaning that their output can be extracted at anytime.

2.8 Programming Languages

2.8.1 The C programming language

Nordin et al. extend the Compiling GP System (CGPS) [48], which uses instructions of fixed length, to Automatic Induction of Machine Code GP (AIM-GP) [49], which uses instructions of variable length. The original motivation was to directly evolve machine code avoiding the interpretation process high level languages have to go through, achieving an impressive improvement in speed of around 1000 times compared to LISP implementations. In CGPS, crossover points are readily identifiable as instructions are of fixed length. To avoid this issue in AIM-GP, instructions are grouped together in fixed length instruction blocks (the size of the blocks being a globally defined parameter). The block size must be set large enough to accommodate the largest instruction. Crossover exchanges these instruction blocks. Instruction blocks are like an implicit glue (compare this with Cramer's BLOCK instruction in section 2.4.1) and may assist in the protection of building blocks against possible disruption by crossover. As more than one instruction may appear in an instruction block, crossover cannot separate the instructions. Mutation can do one of 3 things, either affect a whole instruction block, an instruction or the operand of an instruction. The halting issue is avoided by having a fixed number of instructions to execute.

Automatically Defined Functions (see section 2.3.1) are incorporated into this system, though are not required to make it Turing Complete. They comment that standard crossover is too brutal as it exchanges sub-tees regardless of their context. Homologous crossover preserves the context in tree based GP by selecting the same crossover point in both parents, thus preserving location. This

is not easy to achieve in a linear system, but one way of achieving it is to use ADFs [49].

They tackle a speech recognition task and compare AIM-GP to a tree based system. Better results for AIM-GP indicate that there are advantages rather than just the speed up due to the skipping of the interpretation phase.

2.8.2 Push programming language

Push is a strongly typed, stack based programing language designed for evolution [65]. While modularity has been introduced into GP in the form of automatically defined functions (see section 2.3), in Push modularity (along with recursion and control structures) 'comes for free'. It should be noted, however, that modularity comes for free in any Turing Complete representation! GP may produce potentially syntactically invalid programs during evolution; there can be either too few or too many arguments on a stack. To avoid this problem, if extra arguments are present they are ignored, and if too few arguments are present the instruction is ignored, thus all programs evolved are syntactically valid. An upper bound is placed on the number of instructions that can be executed. Traditional genetic operators are used which exchange randomly chosen sub expressions with either a new random sub expression or one from another individual. The problem tackled is the following: given a single integer as input, return true if the number is odd, otherwise return nil.

2.9 Indexed Memory

Teller [71] states that standard GP is not capable of evolving Turing Equivalent structures. He adds iteration and memory to make the representation Turing Complete. Some problems require only state information for them to be solved i.e. the current state of the system is enough information to solve the problem. Other problems need more than just state information. For example in Teller's problem the agent has no way of telling which way it is pointing unless it keeps track of that information using Indexed Memory. Teller [69] introduces a new problem to GP. An agent is in a grid world with a number of boxes. The aim is for the agent to push all the boxes to the edges of the environment. He evolves the agent both with and without ADFs and, although ADFs are not required for Turing Completeness, their presence does improve performance. An agent's exploration of the grid is terminated after 80 time steps (a little less than 2 tours of the board). A complex solution is presented in [69] page 211.

Siegel et. al. [62] use GP with Indexed Memory to evolve solutions to the popular arcade game Tetris. An aggregated computation time ceiling is imposed that applies over a series of test cases (rather than a single test case). This encourages faster running programs with minimal damage to performance.

2.9.1 Langdon's work on evolving data structures

In a number of papers Langdon [39, 41] evolves a number of data structures (including stacks, queues and lists) using Teller's Indexed Memory [69].

Langdon [39] generates stack and queue data structures. An individual is five trees, one for each operation of the stack (operations include push, pop, makenull, empty, and top). Only trees representing the same operation are crossed over. The problems only require solutions to implement a stack of 10 integers, however the solutions scale up to stacks of any depth. The fitness function is the number of fitness cases passed.

Langdon [41] again uses Indexed Memory to evolve a list data structure, a generalisation of a stack and a queue which were previously evolved [39]. He evolves ten list operations including insert, delete and locate. A `for-while` loop was used and a limit was placed on the number of iterations (32). This limit was set as low as possible but still allowed loops to span all the available memory. All of the programs which passed the test cases generalised, except one (see section 20.6 [41]).

2.10 Tierra and other 'accidental' models of computation

There are a number of systems which have been created to study areas other than computability. This list, while not exhaustive, contains some well known pieces of work.

Tierra (Spanish for earth) was set up to study the dynamics of the synthesis of life. It is the study of open ended evolution, rather than having a particular termination condition in mind, when a solution to a problem is evolved. Ray [51] starts with organisms which can already replicate (this is given at the starting point), and the aim is to generate increasing diversity similar to the Cambrian explosion. A "slice of time" is handed out to each individual in the community the approximated parallelism (as long as the time slices are small compared to the time to replicate). Ray achieves his aim, and creates an interesting interacting collection of parasites and hyper-parasites (see page 383 [51] for full description). Maley [44] went on to show that this system can implement a Turing Machine.

Haykin [24] (page 748) reports that recurrent artificial neural networks can simulate Turing Machines (Sielgelman et. al. [63]). The game *minesweeper* is Turing Complete
(see *http* : *//web.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm*). So too is John Conway's *game of life* [6]. Wolfram [84] (pages 646-647) shows that a one dimensional universal automata is Turing Complete. Also gene regulatory networks may be Turing Complete though we are not aware of any proof of this.

2.11 Discussion

We first consider how others have generated new computer programs from old computer programs, avoiding the problem of producing syntactically invalid programs. We then consider the landscapes produced by using Turing Equivalent representations. We go on to consider the problems that crossover can cause by its destructive nature and specifically look at the problems caused by global memory, how program flow is controlled and briefly examine the building block hypothesis. The problem of non-terminating programs is highly problematic for GP, and this problem cannot be ignored. We review how others have tackled this problem in section 2.11.5. The final sections looks at the generality of the solutions produced, the problems which have been tackled and which representations have been used.

2.11.1 Fitness functions

Introduction

Designing a good fitness function is critical to the success of an evolutionary run. It is the fitness function that decides (maybe probabilistically) which program in the search space is 'better' than another. 'Better' is often taken to mean 'has lower error', but what we really want is a measure of which programs are more likely to evolve into a solution, which is not the same thing. Ideally the final program will have zero error with reference to some set of test cases (assuming we are dealing with a problem with no noise), but this does not necessarily mean that an error based fitness function will be a good driver for evolution.

Lets consider a two non-evolutionary examples; a student's progress at maths and playing chess. Imagine two pupils doing maths homework. One student does all the correct working out, but makes a mistake at the final stage of the calculation. The other student is lazy and just writes down the answer to the question, and gives the same answer as the first student. A good maths teacher will give marks for working, indicating that the student is 'along the right lines' in the first case, but cannot give any marks in the second case as the teacher does not know how the student arrived at the answer. In other words the final answer given by the student is not really a good indicator as to how well they have done at their studies.

In a game of chess we could use a simple points system for the pieces which remain on the board (e.g. pawns are worth 1 and the queen is worth 9) to indicate which player is in a better position (i.e. who is more likely to win from the given positions) at any point during the match. While this may work for some cases, there is the well known case where one sacrifices their queen (which may give them a temporarily worse score) but in the long term pays off.

If we are working on the even parity problem and we have an individual in the population which

gets all of the fitness cases wrong, this individual will receive zero fitness (assume we give one point for each correct score). However it is very 'close' to the solution in the sense that the program only needs a single mutation to take it to the solution (i.e. NOT the output). The construction of a fitness function is related to the credit assignment problem (i.e. how do we go about assigning credit to a solution which is partially correct)

Crafting a fitness function can be notoriously difficult. As Kinnear [33] (section 1.4.2) states *'any and all boundary conditions in your fitness function will be ruthlessly exploited by the individuals in your population'*. Evolution is like a naughty child, you tell it to do one thing and it does another. For a good discussion about fitness functions see [33] section 1.4.2. Cramer [12] states *'a major problem here is one of 'hand-crafting' the evaluation function to give partial credit to functions that, in some sense, exhibit multiplication-like behaviour, without actually doing multiplication'*. The key here is identifying 'multiplication-like' behaviour.

Examples from the literature

We now examine a number of fitness functions used. We first look at fitness functions used in the evolution of solutions to arithmetic problems. We then look at functions used in classification problems and in particular the even parity problem. A number of arithmetic problems have been tackled. Cramer [12] evolves programs to multiply two integers and gives programs successively more credit if: the output variables have changed from their initial values; the output depends on the input; the value of the input is a factor of the output; and finally, the output is the product of the two inputs. Huelsbergen [25] also evolves a program to multiply two integers. His fitness function consists of the sum of two parts. The first part is the magnitude of the difference between the target value and actual value obtained The first part is the difference between the value of the program counter and the length of the program (a correctly terminating program will have a program counter which have a value equal to the length of the program). Huelsbergen [26] evolves a series of programs which output recursive sequences. The fitness function is the scaled sum of the magnitude of the differences for each position in the sequence. The value of the scale is set in accordance with the maximum value in the target sequence.

A number of classification problems have been tackled. Yu [93] evolves solutions to the even parity problem and gives 1 if the score is correct, 0 otherwise. As her system does list processing there is the possibility that a program may apply the head or tail operation to an empty list, and if this error is encountered during the execution of a program, its fitness is reduced by 0.5. Huelsbergen [27] used a two tier function when evolving solutions for a bit counting problem. The first tier (correctness) is the magnitude between the actual and target number of bit. The second tier (proper termination) is the number of test cases on which the program halted. Two programs with

identical first tier fitness are distinguished by their second tier fitness. He claims that *'by separating the disparate criteria of correctness and termination, we can side step the ad hoc credit assignment problem that often plagues a single fitness function attempting to express multiple objectives.'* He does not state however why correctness is more important than termination. Vallejo et. al. [78] use the number of bio-sequences correctly classified.

Conclusion

While the ideal fitness function is highly problem dependent (i.e. it should define what is a good solution to the problem), there may be some general principles that can be used in the construction of a fitness function as the nature of the search space is universal (i.e. there is problem independent component due to the fixed mapping between the space of individuals in the search space to their functionality and a problem dependent component due to the task we are considering, see figure 3.6). The design of a fitness function for the evolution of a Turing Complete representation suffers from all of the problems we encounter when designing a fitness function for a problem posed to standard GP (i.e. the credit assignment problem), but also some additional problems which are specific to Turing Complete evolution. One general theme that has emerged is how to deal with non halting programs but there is the difficulty of how to treat these programs in the fitness function. While designing a fitness function is problem specific there may be a generally good way to deal with the halting problem independent of the task at hand.

2.11.2 Generating syntactically valid programs

In GP the operations of crossover and mutation take old programs and produce new programs. One relatively minor hurdle to GP is creating syntactically valid programs. In contrast to typical GA representations, all bit strings are syntactically valid, however a little more care is required when we are operating on the space of programs. For example, we may have a representation which uses brackets and therefore the brackets must be balanced at all times.

If we take a standard high level programming language we will run into problems. Angeline [2] states that *'the syntactic sugar of most high-level programming languages pose a difficult problem for any type of intelligent search methods'* and states the reason for this is simple; *'a single missing or misplaced syntactic marker can render an otherwise perfectly correct program uncompilable and hence useless'*. Cramer [12] also expressed that *'it is strongly desirable that the chosen representation be such that all such generated programs stay in the space of syntactically correct programs'*. Tanomaru et. al. [68] allow syntactically invalid automata to be created by crossover (i.e. non existent states are referred to in the state transition table of the evolving Turing Machines). Although these are assigned a low fitness time is wasted by examining these automata which are permitted in the search

space. There are a number of solutions to this problem:

1. do not allow the generation of syntactically invalid programs (by inventing a representation and associated genetic operators which do not produce syntactically invalid programs)

2. find an interpretation of 'syntactically invalid programs' which allows invalid programs to be interpreted. In this way previously invalid programs become valid under the new interpretation.

Not generating invalid programs

We can design genetic operators which do not generate invalid programs from valid programs. For example, in CGPS (Nordin et. al. [48, 49]), instructions are of fixed length so crossover can readily be applied without producing syntactically invalid programs. To avoid this issue in AIM-GP, where instructions are of variable length, instructions are grouped together in fixed length instruction blocks (the size of the blocks being a globally defined parameter). Crossover exchanges these instruction blocks.

Interpretation of invalid programs

An alternative approach is to allow invalid programs, but find an interpretation of the syntax which makes them semantically valid. For example, Huelsbergen's system [25, 26, 27] may produce programs which cause the program counter to point to non existent instructions (i.e. fall out of range of 0 to the program length). This problem is avoided by the following solution. If the program counter becomes negative, it is made equal to zero so that it points to the first instruction in the program. Similarly if the program counter has a value which is greater than the length of the program, then the program is terminated. Thus in Huelsbergen's set up, while syntactically invalid programs are produced, they can still be sensibly interpreted.

Spector's GP system [65] may potentially produce syntactically invalid programs during evolution; there can be either too few or too many arguments on a stack. To avoid this problem, if extra arguments are present they are ignored, and if too few arguments are present the instruction is ignored, thus all programs evolved become syntactically valid.

Conclusion

We cannot think of any benefits of including syntactically invalid programs in the search space as they can only reduce the probability of finding a solution. In addition we cannot imagine a programming language were it is not possible to stay within the space of syntactically valid programs, or find a suitable interpretation of 'invalid programs'.

2.11.3 Landscapes and the genotype phenotype mapping

Landscapes

The metaphor of the landscape has long been used in Evolutionary Computation as an aid to visualising the search process ([40] chapter 2 and [52] chapter 9). In GP, we can consider a landscape as a graph consisting of nodes (programs) which are connected by edges (allowable transitions). If two nodes are connected, this represents the fact that an operator can transform one program into another. If the operators are probabilistic, then the edges are labelled with probabilities. It is nonsense to talk about a landscape in isolation, for example we should not technically talk about the landscapes of Turing Machines. A landscape is defined by the representation, the genetic operators and the fitness function. Changing any one of these three may drastically change the landscape. A landscape which has one local optimum is called unimodal and a landscape which has several optima is called multi-modal. A unimodal landscape is therefore easier to search than a multi-modal landscape due to the number of local optima.

The genotype phenotype mapping

When generating new programs from old ones we hope that the behaviour of the program changes gradually. In moving from language JB to TB Cramer [12] notes that *'because of the structure of JB, semantic positioning of a integer list element is extremely sensitive to change'*. This makes the search space very difficult to search, as *'a useful JB substructure shifted one integer to the right will almost certainly contain none of its previously useful properties'*.

Teller states in his thesis ([73] page 133), *'how 'good' a representation is is only measurable with respect to a particular set of operators'* and goes on *'operators and representations should be designed together so that they coordinate correctly'*. This is because, along with the fitness function, this is what defines the landscape.

Teller [70] states *'because the space of algorithms is so discontinuous, the mutation operator might almost as well erase the old individual and make a new one from scratch'*. Indeed this was one of the operators Tanomaru [67] used. In evolution it is desirable to maintain a behavioural link between a program and its offspring. Teller [70] makes the point that the success of standard GP is due to this implicit assumption when evolving functions (by functions he means representations without iteration and memory), however when the GP system is extended to be able to evolve algorithms, the topology of the space becomes highly discontinuous. A small change in a program can produce huge changes in its behaviour. Brave [8] (section 10.6.2) states something along similar lines; *'small variations in the structure of a recursive program can lead to large variations in functionality and thus fitness'*. In terms of GP terminology the genotype is the program and the phenotype is the

functionality. Ideally genetic operators will generate programs with a similar phenotype to the parent program.

2.11.4 Disruption caused by crossover

Global memory interference

All the Turing Equivalent models reviewed earlier in this chapter have global memory. Let us consider this in the context of the representation used by Huelsbergen [27]. If a section of code is performing some intermediate function, it will in general access the registers. However, if this chunk of code is selected for crossover and is inserted in another program it will perform a different function as there is no guarantee that the rest of the program has left the specific registers in an appropriate state. Register machines suffer from the problem of what we will call *global memory interference*, as what one contiguous section of code would do in one program, when it is crossed over will do something else when inserted in a different program due to global memory. Similarly, the crossed over code may overwrite data in registers the rest of the program was using. It is perhaps not surprising that headless chicken crossover substantially outperforms standard crossover in Huelsbergen [27]. GP with Indexed Memory also suffers from this problem as the memory is a global array accessed by read and write primitives. Nordin et. al. [48] make the note of the problem of global memory interference and overcome it by storing a backup of the registers and restoring them after a function call is completed.

Program flow

Program flow is controlled in some models by either absolute or relative jumps. For example, Turing Machines typically use absolute jumps to states, whereas register machines typically use relative jumps. Relative jumps could be used in Turing Machines, or absolute jumps could be used in register machines. Let us first consider absolute jumps. If a contiguous piece of code is selected for crossover, and transferred to a new location, whatever function it was performing before, it will almost certainly be performing a different function in its new context as the absolute position of the code may have been moved in the crossover process. Relative jumps do solve this problem, as instructions in a contiguous chunk of code will still refer to each other independently of their absolute position in the overall program. However, crossover may separate instructions that were being jumped from and to (i.e. acting together), so in the context of a new program entirely different instructions may be jumped to. Thus crossover can completely destroy the functionality of a contiguous section of code by separating it. In general, if a chunk of code is selected for crossover it will perform a different function after crossover. Let us refer to this as *disruption of modules by*

crossover.

The building blocks hypothesis

There has been discussion within both the GP and genetic algorithm communities about the building block hypothesis. The motivation of crossover is that it will combine good parts of one individual with good parts of another. However, due to the potentially destructive nature of crossover in TC representations, this would appear not to be the case. Tanomaru [67] drops crossover in his second set of experiments as *'the idea of crossover is based on the building blocks hypothesis, which is unlikely to hold in the case of automata generation'*. Huelsbergen [27] raises the question of how beneficial building blocks are given that headless chicken macro mutation substantially outperforms standard crossover. Nordin et al. [49] claim that as compound instructions may appear in one instruction block, crossover cannot separate these so it is easier to protect building blocks against crossover. However, the size of instruction blocks is controlled by the user, and therefore does not evolve, limiting the potential building blocks that may be produced. Cramer [12] introduces a BLOCK instruction to group instructions together.

One potential solution to the disruption caused by crossover is to have explicit module boundaries. Crossover can then only act at this level. This will solve the problem of *disruption of modules by crossover*. If we only allow each module to have local memory this will solve the *global memory interference* problem, thus when a module is transfered to a new location by crossover, it will perform the same function as it did in its original location. Thus crossover will preserve the functionally of modules (see [88] for more details). Explicit modules do not add any expressiveness to the representation but allow crossover to act at a certain level.

2.11.5 The halting problem

We cannot know beforehand if a program will halt or not. Most of the methods discussed in this chapter adopt the simplest solution by putting an upper limit on the computing time of each program on each test case. This limit is removed after evolution, allowing the program to compute on input which may require longer to process than examples included in the training set. The main objection to this method is that if the limit is set too low it will prohibit solutions evolving, and if it is set too high time is wasted waiting for non terminating programs. In addition the optimal time for each test case may not be a constant but will depend on the test case in question. Some researchers also impose a memory limit (e.g. limit the amount of tape for a Turing Machine, or the number of registers of an unlimited register machine), however this is implied by the time limit as reading and writing to memory require time.

Teller [70] describes an alternative method, based on an analogy with cooking popcorn, where

the population is evaluated after a certain proportion of the programs have halted. Possible other solutions include gradually increasing the time limit as evolution progresses, or self adapting the time limit in some way.

Schmidhuber [57] in a sense avoids the halting problem as only a single agent is used. The algorithm determines how long a policy may run for before it is evaluated. Schmidhuber [58] states (page 10) *'since Levin search has a principled way of dealing with non-halting programs and time-limits (unlike e.g. Genetic Programming), Levin search may also be of interest for researchers working in GP and related fields'.*

Teller [72] (section 3.5.4) discusses halting programs. Unlike some systems which reward halting programs by assigning them higher fitness, PADO does not require programs halt (as they use an anytime extraction procedure). However, as an experiment, programs are pressured toward halting by assigning zero fitness to programs which have not self halted by a certain time limit. He observes that the SMART operators were able to reduce the number of programs that do not halt.

Siegel et. al. [62] impose an aggregated computation time ceiling that applies over a series of test cases, rather than imposing a limit on a program running on a single test case. This therefore encourages programs to run faster. This is similar to Tierra [51] where the faster you reproduce the more successful you are (as programs are executed in parallel, there is a pressure to increase program speed).

GP takes its inspiration from nature, which is a highly parallel process. The halting problem is not an issue for natural evolution. Any individual 'stuck in a loop' would simply not survive when competing with other individuals, which is similar to the situation described in Tierra [51]. Maybe, as in numerous other cases, GP could take a hint from nature about how to go about solving this problem.

2.11.6 Problems tackled

A number of different types of problem have been tackled including navigation, language recognition problems and arithmetic problems. These are mainly toy problems. The concept of a toy problem is a vague notion, and is often used as a testbed to investigate the utility of a particular search technique before it is tested against real world problems. A good example of a toy problem is the ones max problem (often used with Genetic Algorithms) where the fitness of an individual is defined to be the number of 1's in a bit string. Often we know what a perfect solution looks like and they are typically noise free and the data did not originate from the real world. Maybe a better name for a toy problem would be 'a controlled problem' and we can alter the problem at will as it is artificial.

A number of navigation tasks have been tried. Schmidhuber et. al. [58] [59] tackle a simple maze navigation task. Teller [69] introduces a new problem to GP. where an agent has to push a

number of boxes to the perimeter of a grid world.

A number of language problems have been tackled. Yu [93], Spector [65] and Huelsbergen [27] all evolve solutions to the parity problem. Tanomaru et. al. [67] tackles three problems; the recognition of a regular, a context free and a context sensitive language. The even parity problem is an example of a regular language. Tanomaru [68] sorts a tape of two symbols with a Turing Machine. Huelsbergen [28] produces a register machine which sorts three integers into order.

A number of simple arithmetic problems have also been tackled. Huelsbergen [28] finds a function which returns the maximum of four integers. Tanomaru [68] evolves at Turing Machine which performs proper subtraction. Cramer [12] and Huelsbergen [25] evolve the multiplication function. Schmidhuber also attempts function regression, [58] (page 17) and [59] (page 127).

All of these are toy problems, but real-world problems have been attempted as well; Vallejo et. al. [78] looked at bio-sequence recognition, and Teller (see thesis [73]) looked at image recognition.

2.11.7 Generality of solutions

In some problem domains where we are dealing with toy problems with a finite number of possible inputs, we can test for generalisation by examining the output for all of the possible input combinations. For example with the even-5-parity problem there are 2^5 test cases, so we can test for a general solution by looking at all cases as seeing if the program behaves as required.

With a number of the problems tackled by researchers reviewed in this chapter this approach cannot be taken as the number of cases is infinite. For a number of toy problems we can check by hand that the program does indeed generalise to all input cases. For example, we cannot test a program by purely examining its input output behaviour for the even-n-parity problem. Many of the solutions produced are general. Of course we are never going to have a system which will be guaranteed to produce a general solution every time.

Cramer [12] evolved a general multiplication function. Yu evolves solutions to the even parity problem, all 40 of the solutions produced are general (page 361 [93]). Tanomaru et. al. [68] evolve Turing Machines to do sorting and proper subtraction, in both cases solutions generalised. Tanomaru [67] continues his work and evolves Turing Machines to recognise a regular, context free and a context sensitive language but does not report the generality of the solutions. Vallejo et. al. [78] found a Turing Machine which correctly classified all training sequences and accepted several sequences not included in the training set. In [25, 26, 27] Huelsbergen evolves a multiplication function, programs which generate recursive sequences and solutions to the parity problem. In all cases the solutions produced were general. Spector [65] evolved general solutions to the even-n-parity problem. Langdon evolved a number of data structures (stacks, queues and lists)[39, 41]. General stacks were evolved. Three of the six solutions to the queue problem were general ([39] section 3.7.5) (solutions produced

earlier were general but required indefinite amounts of memory section 3.6 [39]). All but one of the list solutions were general ([41] section 20.6).

Unfortunately we cannot comment on the generality of some solutions as we do not know what the general solution looks like (e.g. Teller [69] and Siegel et. al. [62]). Some of the other problems also consist of real world noisy data and it is also difficult to comment on these solutions.

2.11.8 Program representations

Linear, tree and graph structures have been used as representations for the evolution of programs. Examples linear of structures are; state transition table of Turing Machines ([68, 67, 78]) programs for unlimited register machines [25, 26, 27]. Examples tree of structures are Teller's Indexed Memory [71] and some of Cramer's original work [12]. An example of a graph used for evolution is PADO [72].

Some of the representations used are fixed length, others are variable length. In general we do not know what the length of the program is we are looking for (except in very specific circumstances, or we may have an upper limit), so we will typically need to employ a representation which can alter its length during evolution.

2.12 Summary

We firstly reviewed a number of modular search methods used in GP (section 2.3). We regard this as relevant to the evolution of Turing Complete representations as the ability to express modules is a characteristic associated with any Turing Complete representation. This ability to express modules has some important consequences for the representation of solutions (i.e. the invariants of complexity under the transformation of primitive set, see chapter 4). Of all the modular methods only ADFs have been used in conjunction with Turing Complete representations (e.g. see Teller [69]) where they seem to have a benefit.

A number of researchers have evolved recursive structures and others have worked with total recursive functions (i.e. representations have the ability to express composition and recursion). A number of standard models of computation have been used for evolution including Turing Machines [67, 68, 78], unlimited register machines [25, 26, 27, 30, 57, 58, 59], and the C programming language [48, 49]. New languages have also been created specifically for the purpose of evolution including PADO [72, 75] and Indexed Memory [71]. We also list a number of other accidental models of computation which were invented to study some aspect other than computability (e.g. Tierra [51]) and were later found to be Turing Equivalent.

We then moved on to look at common thread in these works in the discussion section 2.11. The

problem of designing a fitness function to drive the evolution of Turing Complete representations suffers from the same problem as that of standard GP, namely the credit assignment problem: how to assign partial credit to individuals which are more likely to appear on a path to a global optima. The design of a fitness function for the evolution of Turing Complete representations also faces the additional problem of dealing with the halting problem. How should we incorporate information about the proper termination of a program on all or some of the test cases into the fitness function.

Genetic operators produce new programs from old ones and there are essentially two different approaches to generating syntactically valid programs, either do not allow the generation of syntactically invalid programs or find a way of semantically interpreting syntactically invalid programs.

In section 2.11.3 we consider the landscape produced by the representations of computer programs and typical genetic operators. This is a very rugged landscape full of discontinuities. A small change in a computer program can make drastic differences to its input output behaviour and is chaotic in nature. This is largely due to computer programs' ability to express modules and recursion. If a module is altered slightly, it may be called multiple times and therefore can have a great effect on the output. Similarly, if the body of a loop is mutated, if it is iterated over multiple times, the difference, compared to the non-mutated program, can accumulate. While this ability to be able to potentially compress information makes it harder to search this space, it is due to this greater compactness that generalisation is possible.

Crossover, the operation of exchanging genetic material between typically two parents to produce two children, is recognised as having great importance in GP ([5] chapter 6), partly demonstrated by its high use compared with mutation. However crossover can cause huge problems especially if the representation is Turing Complete. If a group of instructions are working together (i.e. in a module), crossover may simply separate them. If a group of instructions is transfered by crossover, then they may perform some entirely different function when inserted into a new program due to the problem of global memory interference (see section 2.11.4). In general, crossover can be a very powerful way to move through a search space as it can potentially combine two partial solutions (i.e. solutions which contain different parts of the answer to the overall problem), making large jumps across the search space, in contrast to mutation which makes alteration on smaller levels (e.g. a row or a cell of a state transition table of a Turing Machine).

The halting problem is a hurdle to any attempt to evolve Turing Equivalent programs. The most commonly used method is simply to place an upper limit on the execution time of a program. If a program has not terminated by the time this limit is reached, its execution is aborted. This method has it's problems If it is set too low, a solution will not be able to be recognised (even if a perfect solution is found it will be terminated before it has had time to compute the output and will therefore fail to be recognised as a solution). If it is set too high, time is potentially wasted

waiting for non halting programs. In addition, setting a constant amount of time for each test case is not ideal as each test case is likely to require a different amount of time. An interesting approach is to set a time limit for a program on *all* test cases, rather than a single test case, encouraging fast completion.

One of the most important aspects of learning is the ability of the solution to generalise on cases not included in the training set. This is one of the fundamental reasons why we are investigating Turing Complete evolution, as we have the ability to produce solutions which are general. Most of the solutions evolved by the researchers covered in this review chapter did evolve solutions which are provably general. It is impossible to have a system which always produces general solutions as there is always a probability that solutions will be produced which satisfy the test cases, but fail to produce the required behaviour on further cases. Huelsbergen [28] has invented a method which can guarantee generality, but this is only for a restricted type of problem where the solution depends on the comparative value of two numbers, not on their arithmetic value.

Most of the problems tackled are typically simple 'toy' problems e.g. simple arithmetic problems and language recognition problems (see section 2.11.6 for a description of a toy problem). If one examines the solutions produced we see that if explicit loops have been used we do see programs containing 2 nested loops but no more [12] [58] (page 9) and [59] (page 116). If explicit loops have not been used only solutions containing a single loop have been evolved [25, 26, 27] and [68]. While we acknowledge the number of loops is not an exact measure of complexity, it gives us a guideline and, maybe this indicates a glass ceiling to the complexity we can evolve using current methods. It is a little more difficult to make statements about the solutions evolved using Indexed Memory, as the solutions are messier and more difficult to interpret, however we feel that they are typically not a lot more complex than the solutions evolved using unlimited register machines or Turing Machines. Only a few real world problems have been tackled notably Teller [73] with image recognition and Vallejo et. al. [78] with bio-sequence recognition. Perhaps indicating that evolving Turing Complete representations is still in its infancy.

2.13 Conclusions

There are a number of basic fundamental problems that need to be overcome if we are successfully going to traverse the search space of algorithms (e.g. how do we deal with the halting issue). There is the question of representation, how to generate new instances of the representation (i.e. what genetic operators to use) and what is a suitable fitness function. The representation and genetic operators much be considered in conjunction (i.e. it does not make any sense to apply subtree crossover the the state transition table of a Turing Machine). It may be possible to study fitness

functions in isolation in this respect as a fitness function could be used with any combination of representation and its associated genetic operator.

The solutions to problems obtained using 'standard models of computation' are typically simple (low complexity, containing only one or two loops). The solutions obtained using GP with Indexed Memory are of similar complexity (we just count nodes as a simple measure of complexity). Maybe the fact that applying evolution to the different models of computation has produced solutions of similar complexity indicates that we have hit a glass ceiling with regard to the complexity that we can tractably evolve with these methods. Perhaps looking at self adaptive methods is the way to go. The fact that only a few researchers have taken on real world problems perhaps indicates that the search for algorithms using evolutionary inspired methods is still in its infancy.

Chapter 3

No Free Lunch Theorems and Occam's Razor

3.1 Introduction

3.1.1 No free lunch theorems and Program Induction

The 'No Free Lunch Theorems' (NFL) [61, 85, 86] concern search algorithms. They essentially state over the set of all functions, all search algorithms perform equally, under certain assumptions which we examine in section 3.4. A number of papers have been published on NFL theorems ([61, 85, 86]) and it is interesting that so much controversy has been caused in the search community (see Droste et. al. [15]) as in one of the original papers [86] (at the end of the introduction) it is explicitly stated:

"We cannot emphasise enough that ***no claims whatsoever*** *are being made in this paper concerning how well various search algorithms work in practise."*

A corollary of the NFL result is the following; we can hack together any old search algorithm, and it will still do as well as the most carefully designed search algorithm, when all functions are considered. In another form; over all search algorithms no function is any more difficult than any other function to search. (this follows from the framework presented in Schumacher [61], which is summarised in section 3.3). We return to this point (see section 4.1.4) as we feel that the difficult of a problem should be an intrinsic property of the problem itself and not of the method to solve it.

In Langdon et. al. [40] (page 9) they state:

'To date no specialised result for GP has been presented'.

One of the main results of this chapter is that NFL theorems do not apply to GP. This is due to the many to one non-uniform mapping between genotype (the representation of a function) and phenotype (the function). This is what Koza calls the lens effect, [36] chapter 26.

3.1.2 Conservation of generalisation

Generalisation, the central goal of learning, is the accuracy with which we predict the outcome of previously unseen test cases which where not included in the training. Schaffer [53, 56] proved that over the set of all problems (i.e. functions), the average generalisation of any algorithm is the same as any other algorithm. This result is called the Conservation of Generalisation (COG). For example, given a set of classification problems with two class labels, we achieve exactly 50 % accuracy over all problems. Essentially, for each problem on which we can correctly predict the class label of an unseen case, there exists the 'contrary' function where we incorrectly predict the class label (i.e. the class labels of all the unseen cases are inverted). This result is also true for other classification problems where there are more than two class to predict, and is also true of function regression.

The COG is related to the NFL result, but is perhaps of more significance. NFL states, if we are searching the domain of a function, *which could be one of all possible functions*, then no 'best' search algorithm exists; in fact they perform the same. COG states, if we are trying to make predictions about unseen values of a function *which could be one of all possible functions*, then no 'best' prediction algorithm exists; in fact they perform the same. Although these two results are related, there are some differences. For example, the COGs result does not depend on most of the assumptions NFL makes (section 3.4). However the central assumption of the COG result, that all functions are equally likely, can be seen as being in contrast to the well known heuristic known as Occam's Razor. In section 3.7 we present a framework and make a formal conjecture about the frequency with which functions are represented.

3.1.3 Occam's Razor

Given some observed data, there are multiple explanations which describe the data. An explanation (or model) is a rule relating input data to output data (i.e. relating independent variables to a dependent variable), and could be expressed using a number of different types of representation (decision trees, artificial neural networks etc). Informally, Occam's Razor states:

'The simplest explanation is best'

(see Cover et. al. [11] page 1). It has been argued that 'shorter' explanations are more likely to give better predictions on unobserved data. Later we will relate the simplicity of an explanation with representational complexity. Occam's Razor is a general principle in science as many scientists are

either trying to find explanations of observed data or make predictions about future, yet, unobserved data. Kearns et. at. [32] (page 32) state that Occam's Razor has become a central doctrine of scientific methodology. Domingos [14] (section 3) and Hutter [29] (page 103, section 3.6.5 Occam's Razor versus No Free Lunch) report that the NFL result negates Occam's Razor.

3.1.4 Contributions of this chapter

Below we list the contributions of this chapter and the sections in which they appear.

1. demonstrate that the NFL theorems do not apply to the majority of representations used in Machine Learning (e.g. GP tree, ANNs, decision trees) due to the fact that some functions are represented more frequently than others with these types of representation.

2. present a formal conjecture concerning Occam's Razor and the frequency with which functions are represented in a hypothesis space.

3. give a qualitative explanation of Koza's lens effect, where each representation in a hierarchy is a subset of the previous representation, and extend it to include Turing Complete representations.

3.1.5 Outline of this chapter

In section 3.2 we give some preliminary definitions. In section 3.3 we present a framework in order to present the NFL theorems In section 3.5 we outline a fundamental difference between certain types of representation. In section 3.4 we examine the assumptions behind the NFL theorems. In section 3.6 we present Occam's Razor and in section 3.7 we revise the Conservation of Generalisation result. In section 3.8 we look at Koza's lens effect using a number of different types of representation. The chapter is brought to a close with a summary and conclusion.

3.2 Preliminaries

In this section we define a number of concepts which will be referred to in the rest of the chapter.

3.2.1 Look up table

Given some observed data, one possible way to represent it is as a look up table (LUT), where the data is simply listed directly (see table 3.1). The size of the LUT is directly in proportion to the amount of data we have (as there is no compression). If we were to use complete LUTs as explanations of the data, we could then make a prediction about the unobserved output, however,

input1	input2	output
0	0	1
1	0	0
0	1	0

Table 3.1: We observe 3 pieces of data, represented as rows, for a 2 arity boolean function. We have not yet observed $\{1, 1\}$ and do not yet know the output.

both tables consistent with the observations have the same size (see table 3.2). If we are using Occam's Razor as a guiding principle (which says pick the shortest description consistent with the data), in this case we have no preference as all LUTs have the same size.

input1	input2	output
0	0	1
1	0	0
0	1	0
1	1	0

input1	input2	output
0	0	1
1	0	0
0	1	0
1	1	1

Table 3.2: Two different accounts of the data given in table 3.1 Using Occam's Razor, as both accounts of the observations are of equal size they are both equally good explanations of the observed data and we cannot prefer the prediction of one over the other.

There are many ways of representing a function; graphically, as a formula e.g. $f(x) = x^2$, or as an algorithm which tells us how to calculate the value of the function. Another way is to represent a function as a look up table. In this type of representation a function is represented essentially as a graph, i.e. a list of ordered pairs of values consisting of the input value and the output value. On the page we can write this in the form of a table, with a number of columns (one for each argument of the function) and a column for the output. This is the form we often collect data in from physical experiments. For example, in one column we record the height from which we drop an apple (the independent variable), and in the other column we record the time it takes the apple to hit the ground (the dependent variable).

Definition 3.1 (Look up table). *A LUT lists explicitly all inputs to a function with its output. A LUT can be thought of as being represented as in tabular form, but may also be represented in other forms (e.g. ANNs). A LUT can only be used to represent functions which are a mapping from a finite set to a finite set.*

The concept of representing a function using a LUT is connected to the idea of incompressibility.

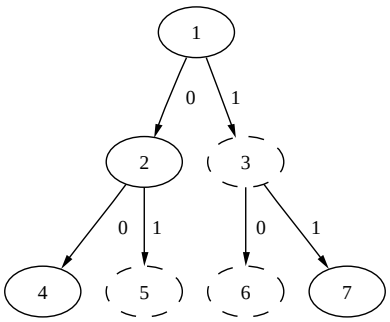

Figure 3.1: A LUT in the form of a finite state automata.

If a function is incompressible then it can only be represented essentially as a look up table. As there is no 'pattern' in the data, it cannot be compressed.

Here we give an example of a function defined using a finite state automata (FSA) in a look up table form. Imagine the following function which we define explicitly in the form $\{bit\ string, accept/reject\}$.

$$\{0, reject\}, \{1, accept\}, \{00, reject\}, \{01, accept\}, \{10, accept\}, \{11, reject\}$$

We can think of this function as the odd-parity function. We can represent this function as a two state machine. However, if we were not aware of the regularity in the function we could express it as a LUT as shown in figure 3.1. Node 1 is the start state, and no two paths thought the automata overlap (i.e. there is a unique path for each input bit string). The accepting states are shown with dashed lines (namely 3, 5 and 6). If more data is observed, it can simply be added to the FSA, e.g. $\{111, accept\}$ would be a new state connected to state 7.

An interesting property of LUTs is the following; Imagine we had made a mistake in the recording of our data, and in fact we should have recorded $\{11, accept\}$. All we would need to do to make our machine consistent with the data would be to relabel state 7 as accept and not reject. This could be done without affecting the performance of the machine on the other bit strings. Similarly if we were working with LUTs, we could alter a row without affecting the other rows in the table. This would not be true if we were working with the 2 state machine, altering if a state is an accepting or rejecting state will in general affect the performance of the machine on other bit strings. As LUTs have this special property where a syntactic change directly affects its behaviour, we can claim that the space of LUTs is unimodal.

Mitchell [47] (see section 4.6.2, Representational Power of Feed forward Networks) describes a scheme for constructing an ANN to represent arbitrary boolean functions in the form of a LUT. For each input vector, a distinct hidden node is added and its weights are set so it is activated if and

only if the specified vector is presented to the network. This produces a network that will have one active hidden node each time a specified input vector is presented to the network. The output layer then implements an OR function, so as long as one hidden node is activated, the whole network will give a positive response.

3.2.2 Hypothesis space

A hypothesis space consists of all the instances of a given type of representation up to a certain size. For example, a hypothesis space could consist of all the finite state machines up to a certain number of states. Another example of a hypothesis space is the space of all programs represented as trees with a given function set; changing the function set would give a different hypothesis space. Our hypothesis space may consist of artificial neural networks (ANNs). In this space, we may fix the architecture of the possible networks (e.g 6 hidden nodes, with fully connected layers), or let the number of nodes and connections evolve [92]. As a definition of hypothesis space, we take that given in Mitchell [47] (chapter 1).

Definition 3.2 (Hypothesis space). *A hypothesis space is the set of all instances of some representation (with some size limit imposed). For example, a set of tree expressions created from a given function and terminal set, up to a depth of size 50.*

A size limit is imposed on the hypothesis space. This could be done explicitly by the Machine Learning practitioner, for example by defining the maximum depth of a tree in the search space. Alternatively, this upper limit could be implicitly imposed on us by the size of the memory of the computer on which we are working.

The hypothesis space, often called the genotype space in EC, is the set of all possible outputs of the search algorithm. In the test-and-generate approach to Artificial Intelligence, the hypothesis space consists of all the candidate hypotheses which can be generated. If a hypothesis exists in the hypothesis space, then it is a potential solution, but if it falls outside the hypothesis space then it will never be generated.

3.2.3 Bias

When learning, we are trying to induce a function which is functionally equivalent to the underlying process which generates the function we observe. We define bias as anything that affects the distribution of functions that is produced by a learning system [46]. An unbiased learning system is one where each function is equally likely to be produced. We state Mitchell's definition of bias [46]

Definition 3.3 (Bias). *Any basis for choosing one generalisation over another, other than strict consistency with the observed training instances.*

Let us take GP as our example. When we construct a GP system, there are a number of choices which have to be made before they system can run ([5] section 5.6). For example, we must choose a function set, a genetic operator, a selection method and a fitness function (this list is not exhaustive). The representation along with function set and terminal set define the hypothesis space (with the addition of a size limit). Both the primitive set and the type of representation (e.g. tree or forest) affect the probability of a given function being generated. We may call this sort of bias, representational bias. The genetic operator, selection method and fitness function affect how we move around the hypothesis space, and therefore affect the order and the probability functions are generated for a given hypothesis space. We may call this sort of bias, search bias. Much of this chapter is concerned with representational bias.

Definition 3.4 (Representational bias). *The bias caused by the representation we are choosing to represent functions.*

An example of an unbiased hypothesis space (i.e. no representational bias) would be a space containing a set of look up tables. All functions are presented with equal probability.

Definition 3.5 (Search bias). *The bias caused by the method used to search the hypothesis space.*

An example of an unbiased search method would be random search. Any bias in a system which is conducted using random search to search the hypothesis space is due entirely to the representational bias.

3.2.4 Compressible functions

The complexity of a bit string is the size (in bits) of the smallest program which prints out the bit string and halt. A bit string is incompressible if it cannot be represented by a program which is shorter than the bit string itself. For example, the bit string

0000000000000000000000000000000000000000000000000000000000000000000000

is highly compressible as it can be represented by a program which is much shorter than the bit string. The following program prints out the bit string above.

```
repeat 70 times {print("0");}.
```

On the other hand, the bit string

0011011011100010111101110011101101011010101010110001101010110101010101

which is a bit string of length 70, is not compressible as there does not exist a short program to represent it. We can represent the bit string with the program

```
print("00110110111000101111011100111011010110101010101100011010101101010101");
```

If some sort of regularity exists in the string, then this pattern can be exploited by the program which prints it, resulting in some compression. If a bit string can be represented by a program which is shorter than the bit string, then the string is compressible. If the bit string cannot be represented by a program which is shorter than the bit string, then it is incompressible.

How does this concept relate to functions? A function could be represented by a look up table, where each input and output is essentially recorded explicitly. If a function can be represented in a smaller space than that taken up as if it were represented by a look up table, then the function is compressible. If a function has to be represented essentially as a look up table, then it is incompressible. Essentially there is no difference in talking about the compressibility of functions or bit strings, as ultimately a function is stored in a computer's memory as a bit string.

Definition 3.6 (Compressible functions). *A function is compressible if it can be represented by a program which is smaller than the program which lists the function explicitly (i.e. expressing the function as a look up table).*

3.2.5 Problem complexity

The complexity of a bit string is defined as the length of the shortest program which prints the bit string and halts (in whatever measure of size we are using e.g. bits or number of nodes in a tree based representation of the program). When we are solving a problem in Machine Learning, we are typically attempting to find the representation of a function which is consistent with the observed data. As a function is stored as a bit string in a computer's memory, we can equate the definition of the complexity of a bit string with the complexity of a function. The complexity of a function can in turn can be equated to the complexity of a problem. Thus the complexity of a problem is the complexity of the underlying function.

Definition 3.7 (Problem complexity). *The complexity of a problem is the complexity of the underlying function.*

3.3 No free lunch theorems

This section repeats a framework presented in Schumacher et. al. [60, 61] that allows a simple analysis of search algorithms. This framework is perhaps simpler than the one presented originally by Wolpert et. al. [85, 86]. We first repeat a number of definitions concerning functions, search operators (vectors), performance measures, population tables and state the NFL result (see Schumacher et. al. [60, 61]).

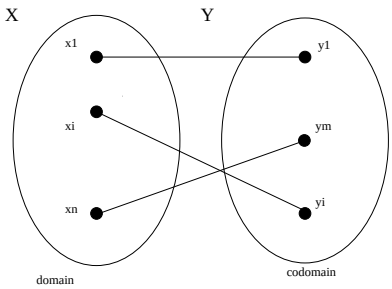

Figure 3.2: A function is a mapping from a finite set X to a finite set Y. Points in the set X map to points in the set Y. In general point x_i maps to point y_i.

3.3.1 A framework for no free lunch theorems

Functions

Let X and Y be finite sets and $f : X \rightarrow Y$ be a function where $y_i \equiv f(x_i)$ (where i is an index running over all points in the set X). The size of X is $|X|$ and the size of Y is $|Y|$. Each value in X maps to a single value in Y. For a given X and Y there are $|Y|^{|X|}$ possible functions. The set X is the domain of the function (i.e. the possible input values). The set Y is the range (or co domain) of the function (i.e. the possible output values) (see figure 3.2).

We can represent a function by simply listing the values in the range corresponding to the order list of points in the domain. Thus if the function was represented as a LUT with two columns (one for the set X and one for the set Y), if the X is always understood to be ordered, then we can just refer to a function in terms of the Y column (see Schumacher's thesis [60] chapter 1.6). For example, the function $f_{<y_2,y_1>}$, lists the two points $< y_2, y_1 >$ mapped to by the points x_1 and x_2, and is pictured in figure 3.3 bottom right.

Search operators and search vectors

A search operator and a search algorithm are taken to be the same and represent any algorithm that produces a search vector. A *search vector,* V is an ordered sequence of points in X, $V \equiv \langle x_a, x_b, \ldots, x_i, \ldots x_n \rangle$. It is assumed that no point is revisited. The ith element in this vector corresponds to the ith point visited in X. A *complete* search vector is any vector that lists all points in X once and only once and therefore has length $|X|$. There are $|X|!$ distinct complete search vectors. For the purposes of NFL theorems we are not concerned with how the search vectors are generated and this is commented on in section 3.4.

A search vector corresponds to a path in X. A given search vector and function will produce a

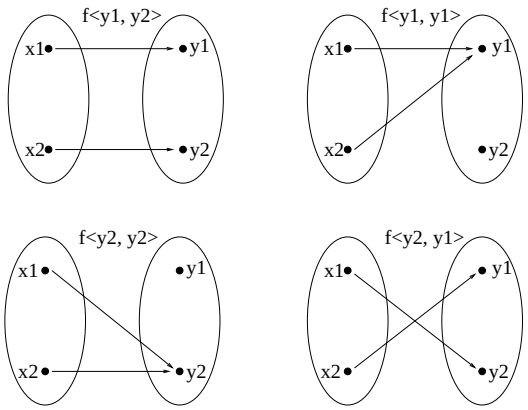

Figure 3.3: All possible functions between two sets which only contain two points are shown. The domain is the set $\{x_1, x_2\}$ and the range is the set $\{y_1, y_2\}$.

corresponding path in Y. Let us call this sequence of points in Y a *performance vector*. A search vector of length l corresponds to a performance vector of length l, given a specific function.

Definition of performance and no free lunch theorems

Define an overall performance measure to be a function that maps the set of performance vectors generated by a given search vector and a set of functions to a real number. A NFL result over a set of functions is defined to exist when two algorithms have the same overall performance measure. In [61] four equivalent statements of NFL theorems are discussed, the third is stated here;

Theorem 3.1 (No Free Lunch Theorem). *Every search algorithm generates precisely the same collection of performance vectors when all functions are considered.*

Population table

A population table is a table where the rows are labelled with the distinct search vectors ($|X|!$ of them), and the columns are labelled by all possible functions($|Y|^{|X|}$ of them) (see sec. 2.2 (Population tables) of [60]). Each element in the table contains the performance vector generated by the corresponding search vector (row) and function (column). For example, the function $f_{<y_2,y_1>}$ (4th column) and search vector $< x_2, x_1 >$ (second row) generated the performance vector $< y_1, y_2 >$ (see table 3.3.1).

	$f_{<y_1,y_2>}$	$f_{<y_1,y_1>}$	$f_{<y_2,y_2>}$	$f_{<y_2,y_1>}$
$< x_1, x_2 >$	$< y_1, y_2 >$	$< y_1, y_1 >$	$< y_2, y_2 >$	$< y_2, y_1 >$
$< x_2, x_1 >$	$< y_2, y_1 >$	$< y_1, y_1 >$	$< y_2, y_2 >$	$< y_1, y_2 >$

Table 3.3: The population table for all functions with a domain $\{x_1, x_2\}$ and range $\{y_1, y_2\}$. The columns are labelled with all possible functions (4 in this case, which are illustrated in figure 3.3). The rows are labelled with all possible search operators (2 in this case). Entries in this table correspond to the performance vector for the given function and search operator. By inspection, we can see that for both search algorithms, the same collection of performance vectors is generated.

3.4 The assumptions of no free lunch theorems

There are a number of simplifying assumptions made in the proof of the NFL theorems [61] which we would normally not ignore when comparing real world search algorithms. We examine them in this section, namely:

1. revisiting points is ignored.
2. all functions are considered.
3. the overhead of generating points is ignored.
4. the evaluation of points takes different amounts of time.

3.4.1 Revisiting points

NFL theorems only consider unique points visited in X, if a point is visited again it is not counted (i.e. a search vector is a list of unique elements in the function domain). Let us look at the issue of revisiting points. Consider two search algorithms, random search and exhaustive search, and a search space of n points. Random search visits points and also has a chance of revisiting points This is related to the Coupon Collectors problem; how many times must we sample a space in order to have a greater than half chance, say of sampling all the points. (of course we hope our search algorithm is doing better than random search). Exhaustive search moves systematically from point to point visiting each point only once. Exhaustive search is guaranteed to find the target point within n evaluations, while we can only make probabilistic statements about random search. Exhaustive search will do better than random search over all problems due to the fact that random search will revisit points. NFL theorems only considers novel evaluations, so in the above case both algorithms are considered to behave the same (random search can only make n novel evaluations). One might argue that we could supplement the random search algorithm with a memory so it avoids revisiting

points (see [17] page 202, Droste et. al [16] page 2), however this will require a computational overhead (see section 3.4.3). It is interesting to consider the time complexity of adding a memory to a search algorithm. Over all functions, exhaustive search is better than random search if revisiting points is considered.

For many real world search algorithms getting stuck in a local optima is an issue. Most researchers go to some lengths the prevent this (so that given an infinite amount of time the algorithm will converge to the global optimum). NFL ignores this issue.

In GP, some functions are represented more frequently than others in the typical tree based representation used in GP, and as some functions are represented more frequently than others we are effectively revisiting some functionality. It is due to this that NFL theorems is not valid for many to one non-uniform representations (see section 3.5).

3.4.2 All functions

A value in the domain maps to a value in the co-domain under one function, but under another function could map to a different value. If we consider all functions, one value in the domain maps to *every* value in the co-domain. If we only consider one function then the best algorithm visits the optimal point first (see scenario 2 in [15]). This means other points will be visited later. Therefore there exists a function where the target point is visited last in the search. For every function a search algorithm does well on, there is a corresponding function on which it does badly (see figure 3.4).

The NFL theorems do not mean 'all functions', as perhaps a mathematician may think (i.e. including functions we can define with infinite domains). In this framework we mean 'all functions' in a restricted sense (i.e. all mappings from a given finite set to a given finite set). In section **??** we will show that attempting to extend NFL theorems to all functions (i.e. including those with infinite domains) in the mathematical sense fails.

3.4.3 Overheads

The overhead of calculating the next point to visit in the search space is not taken into account. If two search algorithms produce the same search vectors, they appear to behave the same in terms of the points they visit, but one may involved a more expensive computation than the other. In terms of wall clock time they will appear to perform differently. Typically when comparing real world search algorithms, they are run for a certain number of generations, and so the overheads involved in calculating the next point are ignored. As in the case above comparing exhaustive search with random search, the overhead of maintaining a list of points visited is much more computationally expensive than the overhead associated with a typical exhaustive search algorithm.

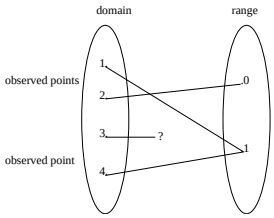

Figure 3.4: We observe 3 points in the domain of the function, shown by lines mapping from the domain to the range. However, knowledge of these three points tells us nothing about a yet unobserved point, indicated by a line starting in the domain at point 3, but leading to a '?'. Point 3 could just as easily map to point 0 or 1 in the range. Also we do not in many real world cases know what the range of a function is beforehand.

3.4.4 Constant evaluation time

NFL theorems assume the same amount of time is taken to evaluate points (a point could represent the evaluation of a function on an argument or the execution of a program on a set of test cases). This is not true for many cases in program induction. Not only do different test cases take different amount of times to evaluate, different programs also take different amounts to time to evaluate. In standard GP, smaller trees will typically be evaluated faster than larger trees. If we are using a Turing Equivalent instruction set some programs may terminate immediately while other programs may enter into an infinite loop and have to be terminated externally, see section 2.11.5.

In addition, in standard GP, the number of test cases is usually fixed, however a number of researchers have realised that time can be saved by altering the number of during a run, and therefore each generation will take a different amount of time to evaluate (see section **??**).

We consider how to count evaluations in section 5.2.2.

3.4.5 An everyday example of the no free lunch theorem

Consider the case where you have locked your bike with a combination lock and forgotten the combination. Say the lock has four dials, each with the digits 0-9. There are 10^4 possible combinations. Imagine that you do not want to break the lock. We can consider the lock as computing one of a class of function (open if the combination is correct, false if the combination is closed). Thus the lock corresponds to one of these permutation of functions. The simplest strategies would mean starting with a combination (say 0000) and incrementing the value each time until the lock is opened. Assuming we do not remember anything about the number which opens the lock, it is easy to see that no one strategy is easier than any other strategy (e.g. starting at 9999 and decrementing the value at each unsuccessful attempt).

3.4.6 Conclusions

In this section we have seen that the NFL theorems make limiting assumptions, which are invalidated when we compare real world search algorithms. Points in the search space are typically revisited by real world algorithms, and it is very expensive to prevent this. For the NFL theorems to be valid, all functions have to be considered, it is due to this reason that NFL theorems do not strictly apply to the representations typically used in GP (see section 3.5.3). Typically each point in the search space does not take the same amount of time to evaluate, and this is especially important when we do not have to evaluate a program on all of the test cases supplied (see section **??**).

3.5 One to one and many to one representations

3.5.1 Representation of numbers and functions

In this section a fundamental difference between the representation of numbers and the representation of functions is examined. There are many ways to represent numbers, however typically there is a one to one mapping between the representation and the number being represented (think of integers or floating point numbers. We do not consider real numbers). With functions there is typically a many to one non-uniform mapping between the program (representation of the function be it classifier systems, artificial neural networks, finite state machines or Turing Equivalent programs) and the function being represented. This difference has consequences when searching the space of numbers compared to searching the space of programs. If the mapping between the representation and the object being represented is one to one, a uniform sampling of the representation will lead to a *uniform* sampling of the objects being represented. If the mapping between the representation and the object being represented is a many to one non-uniform mapping, a uniform sampling of the representation will lead to a *non-uniform* sampling of the objects being represented (i.e. a uniform sampling of the domain will give a non-uniform sampling of the range).

With numbers the representation is essentially 'transparent' (i.e. a number can be represented directly), and a given distribution over the representation will transform into exactly the same distribution over the numbers. With functions, the representation is not 'transparent', and a lensing effect occurs ([36] chapter 26), and a given distribution over the representation will transform into a distorted distribution of the functions (the more flexible the representation, the more this distortion effect, see section 3.8).

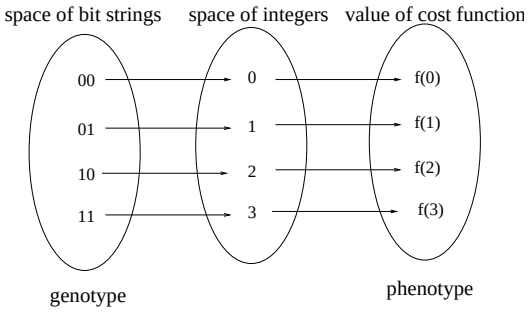

Figure 3.5: There is a problem independent one to one mapping between bit strings (left ellipse) and integers (middle ellipse). There is a mapping between the space of integers (middle ellipse) and the function values (right ellipse) which depends on the problem. This mapping may be one to one or many to one depending on the problem. Over all problems (i.e. all possible mapping between the left ellipse and the right ellipse) NFL theorems are valid.

3.5.2 Numbers: canonical one to one representations.

If our aim is to optimise a function, we choose some way to represent numbers which are input to the function. Common representations are binary and Gray scale among others. These are one to one representations; a number has only one bit string corresponding to it and vice versa. An even sampling of bit strings translates to an even distribution of the numbers which are represented.

Figure 3.3 shows all possible functions between two sets which, in this case only contain two points each. Let us illustrate NFL theorems by simply listing all the performance vectors for all of these functions. There are only two possible search vectors $V_1 \equiv \langle x_1, x_2 \rangle$ and $V_2 \equiv \langle x_2, x_1 \rangle$. Let us consider all of the functions $f_{<y_1,y_2>}, f_{<y_1,y_1>}, f_{<y_2,y_2>}$, and $f_{<y_2,y_1>}$. V_1 produces the performance vectors $\langle y_1, y_2 \rangle$, $\langle y_1, y_1 \rangle$, $\langle y_2, y_2 \rangle$, and $\langle y_2, y_1 \rangle$ respectively. V_2 produces the performance vectors $\langle y_2, y_1 \rangle$, $\langle y_1, y_1 \rangle$, $\langle y_2, y_2 \rangle$, and $\langle y_1, y_2 \rangle$ respectively. By inspection we can see that the same collection of performance vectors is produced by each of the search vectors. This can also be seen be examining the population table, table 3.3.1.

3.5.3 Functions: canonical many to one representations.

If our aim is to learn a function, we need some form of representation in order to express the target function. The representations used here could be of many forms, for example classifier systems (which use conditional rule sets), logical or mathematical expressions, artificial neural networks, finite state automata or computer programs (the list is not exhaustive). An even sampling of programs translates to a non-uniform distribution of the functions which are represented, (see Koza [36] chapter 26, and Langdon [42]). The many to one mapping essentially distorts a uniform sampling

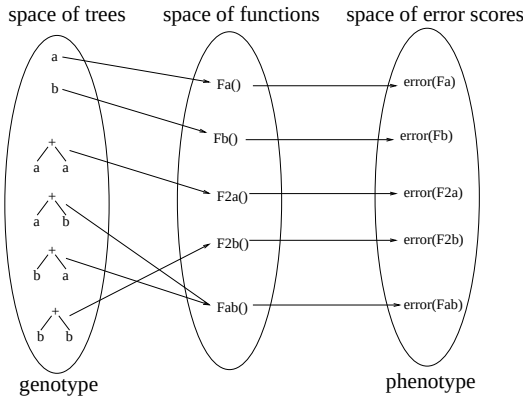

Figure 3.6: There is a problem independent many to one mapping between trees (left ellipse) and the functions they represent (middle ellipse). There is a mapping between the functions they represent (middle ellipse) and the function values (right ellipse) which depends on the problem. Due to the non-uniform nature of the mapping between these spaces (the left ellipse and right ellipse), not all functions exist and the NFL theorem is not valid.

over the space of representations, to a non-uniform distribution over a set of functions, with a bias toward simpler functions. We could use a representation where there is a one to one mapping between the representation of a program and the function it represents, but as we point out later, this offers no generalisation. We do, however use this this type of representation in chapter 4.

Figure 3.7 shows all the mappings that are possible between a set of trees to a set of functions then onto a set of error scores. In this example, tree T_1 has functionality F_1, tree T_2 has functionality F_2 and tree T_3 has functionality F_1. This is a problem independent fixed mapping. All possible mappings between the space of functions, F, and the space of error scores E are shown. There are four possible target functions we are trying to learn, Function 1, Function 2, Function 3, and Function 4 (these could be the four boolean functions of arity one for example). If our GP system generates say tree T_1, then this will always correspond to function F_1 irrespective of the target function we are trying to learn. However, depending on the target function, F_1 will recieve different error scores. If we are dealing with finite boolean problems, then the distribibition of errors is given by the binomial distribution.

Let us illustrate that NFL theorems does not hold in this case by simply listing all the performance vectors for a pair of these cases. There are 6 possible search vectors but we only need to consider two to show that there are differences. Consider $V_1 \equiv \langle T_1, T_2, T_3 \rangle$ and $V_2 \equiv \langle T_1, T_3, T_2 \rangle$. V_1 produces the performance vectors $\langle E_1, E_2, E_1 \rangle$, $\langle E_1, E_1, E_1 \rangle$, $\langle E_2, E_2, E_2 \rangle$, $\langle E_2, E_1, E_2 \rangle$ respectively. V_2 produces the performance vectors $\langle E_1, E_1, E_2 \rangle$, $\langle E_1, E_1, E_1 \rangle$, $\langle E_2, E_2, E_2 \rangle$, $\langle E_2, E_2, E_1 \rangle$ respectively. By inspection we can see that different collections of performance vectors are produced by

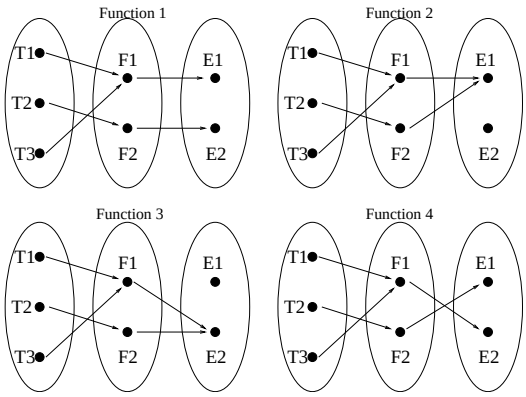

Figure 3.7: The space of trees T maps to the space of functions F, which maps to the space of error scores E (shown as 3 seperate ellipses, and also shown in figure 3.6). The mapping between the space of trees and the space of functions is fixed and is independent of the problem. The mapping between the space of functions and the space of error scores is defined by the problem. For a given target function, each synthesised function may have a different error.

these two search vectors. This difference is due to the non-uniform many to one mapping between the descriptions of functions (i.e. trees) and the functions they represent. In this case search vector V_1 is better than V_2 as the first two trees V_1 visits have novel functionality, whereas the first two trees V_2 visits have the same functionality. In fact V_1 would not have to visit T_3 as it has already visited T_1, which has equivalent functionality.

3.5.4 Non canonical representations

In the above discussion we have considered representing numbers and functions. We could use a many to one representation for numbers and a one to one representation for functions, however neither of these approaches are desirable but we mention them here for the sake of completeness. We do however use a look up table, (which has a one to one mapping with the functions it represents) to a represent module (which forms part of larger programs) in [90].

Numbers: non canonical representations

We could represent numbers using a many to one representation (either with a uniform or non-uniform mapping). But there would be no justification to do this as a many to one representation would mean we are more likely to revisit numbers which have already been represented (see [89] for more detail of these representations). If a many to one non-uniform representation of numbers is used NFL theorems would not be valid.

Functions: non canonical representations

Functions (which are mappings from finite domains to finite ranges) could be represented with a one to one representation (i.e. as a LUT which explicitly lists the output for a given input). While this does avoid having multiple representations of the same function, it gives us no generalisation (i.e. on unseen inputs, the outputs could be anything). If the function is incompressible we essentially have no choice but to represent it as a look up table.

3.6 Occam's Razor

Occam's Razor states prefer the simplest hypothesis that fits the data (see Mitchell [45] page 65), or the most likely hypothesis is the simplest one that is consistent with the observations (see Russell et. al. [55] page 535). This is a central concept in Machine Learning and Statistical Inference. Note that the situation is more involved if there is noise in the observed data and this is not considered in this thesis. Occam did not define what 'simple' meant, and it was not until much later that complexity was strictly defined as Kolmogorov complexity [43]. Occam's Razor, which states shorter consistent hypothesis give better generalization, is in contrast to COG (see section 3.7) which states that we cannot generalise.

3.6.1 Why is a shorter hypothesis more likely to generalise

Occam's Razor states that simpler is better, but why is this? While this may intuitively feel satisfying it is worth further investigation. We should not just accept it without some consideration. We may have two rules that agree with all the observations but we cannot dismiss either as being 'wrong' if they agree with the observed data. But why is one rule a more likely explanation of the observed data than another?

Shorter hypotheses are less likely

Mitchell [45] (page 65), Russell et. al. [55] (page 535) and Cover et. al. [11] (page 161) argue along similar lines. Because there are fewer shorter hypothesis than longer ones it is *less likely* that one will find a short hypothesis that coincidentally fits the data. In contrast, there are often many complex hypotheses that fit the current data but will fail to generalise to subsequent data. While we agree that there are many complex hypotheses which will not fit future data, we argue that there are also many complex hypothesis that *do* fit to subsequent data. Should we not be looking for a hypothesis which is *more likely* to explain the data?

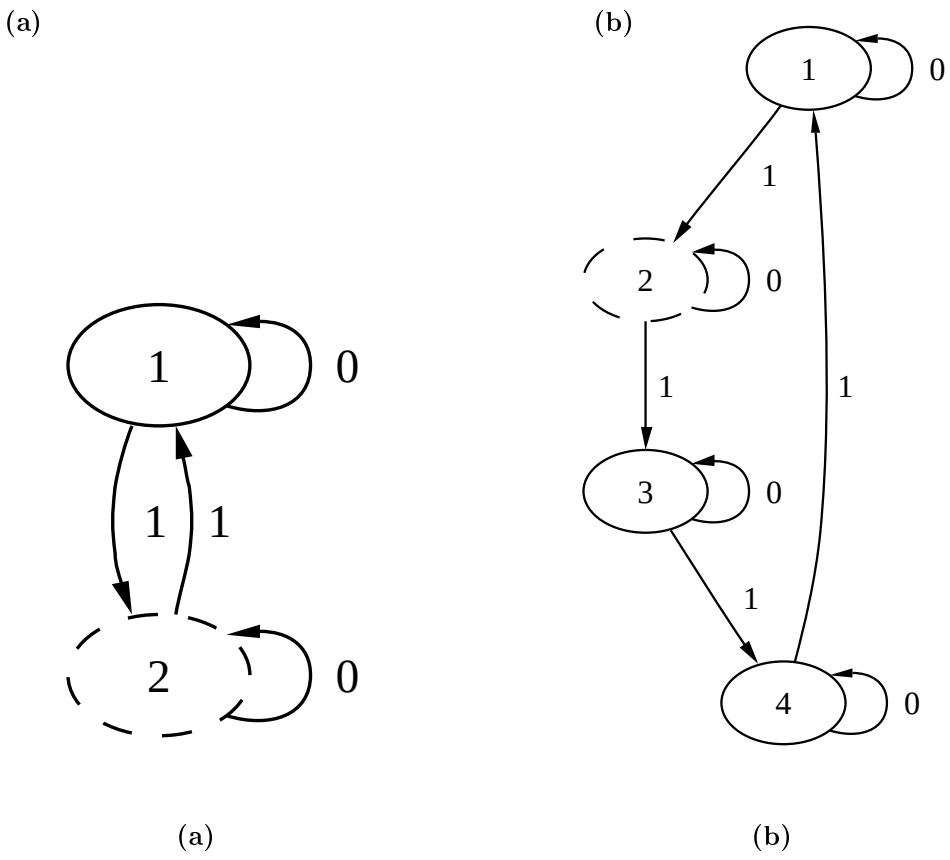

Figure 3.8: Two finite state machines consistent with the observed data.

Possible Hypotheses

Suppose we are given the following data; f(0) = false, f(1) = true, f(00) = false, f(01) = true; f(10) = true; f(11) = false. (Note that this data corresponds to the odd parity problem i.e. if the number of true bits is even, then return true else return false). Consider two finite state automata (FSA) that account for the data (see figure 3.8 a and b). For both machines state 1 is the start state. Accepting states (true) are marked in dots, rejecting states (false) are marked with unbroken lines.

The first FSA (see figure 3.8 a) consists of two states and the second FSA (see figure 3.8 b) consists of four states. Both of these automata agree with the observed data, however on unseen data there will be discrepancies between these two machines. If we choose to pick the smallest FSA consistent with the data, in this case (as we have all the data up to length 2, excluding the empty string) we are constrained to construct the FSA shown as this is the only 2-state FSA consistent with the data. If we allow ourselves to pick FSA with a higher number of states, then there are more choices which are not dictated by the data (i.e. if states can be labelled accepting or rejecting and the labels for the transitions). For example, for the data given above we have the following choices

about the construction of the machine; only states 1, 2, and 3 are visited and state 4 is never reached thus we could label state 4 as reject or accept and FSA will still be consistent with the observed data. Similarly, the transitions from state 4 could be redirected and the resulting machine would again be consistent with the data. There are also 3 state FSA consistent with the data.

3.6.2 How the data was generated?

As we observer more and more data, assuming the function we are observing is consistent with the odd parity function, we will become more confident our 2 state machine is consistent with the data. However, we can never say that a 2 state machine is responsible for the data we observe. It could be any one of the other FSA from the equivalence class of FSA which accept the same language as our 2 state machine. In science we can never prove a hypothesis, we can only falsify hypotheses.

3.6.3 Restatement of Occam's Razor

Occam's Razor states a shorter hypothesis is more likely to generalise. We agree that this *appears* true, however the underlying reason is that the function corresponding to the shortest description consistent with the observed data are represented more frequently. We do not have direct access to the rule behind the data, so it does not make sense to say a short rule is better than a longer one when they are functionally equivalent, we can only judge the validity of a hypothesis on its functionality and its agreement with the observed data. We restate Occam's Razor:

The most probable explanation is the one that is most frequently represented.

There are two points related to restating Occam's Razor. Firstly, this is closer to the induction process. Ultimately we have no access to the underlying rule, so it makes little sense to express our preference for shorter hypotheses. We cannot tell if a short rule is responsible for the data, or a longer rule which has the same functionality. Surely we want to prefer the most probably.

Secondly, if we take Occam's Razor as 'prefer the shorter', this gives us indication of how strong that preference should be. If, on the other hand, we take a frequency based approach we can begin to make some statements about the confidence in our predictions. (If, for example, all our predictions, on a finite set of validation data are correct how can we begin to make calculations about our confidence).

3.7 Conservation of generalisation

Generalisation, the central goal of learning, is the accuracy with which we predict the outcome of previously unseen test cases which where not included in the training. Schaffer [53, 56] proved that

over the set of all problems (i.e. the set of all functions), the average generalisation of any algorithm is the same as any other algorithm. This result is called the Conservation of Generalisation (COG) [56]. For example, given a set of classification problems with two class labels, we can only achieve 50 % accuracy over all problems. Essentially, for each problem on which we can correctly predict the class label of an unseen case, there exists the 'contrary' function where we incorrectly predict the class label (i.e. the class labels of all the unseen cases are inverted).

3.7.1 Framework for generalisation

Wilson et. al. [83] describe the following framework, from which we borrow their notation, in which to talk about some biases being better than others in practise. Let F be the discrete set of functions consistent with some training set. $|F|$ is the number of functions in F. We can define the theoretical average accuracy (i.e. its ability to generalise to unseen data), given by the equation

$$ta(b) = \frac{\sum_{f \in F} g(b, f)}{|F|}$$

where $g(b, f)$ is the average generalisation accuracy of a bias b on a function f. $ta(b)$ is theoretical average accuracy of a bias b over all functions. The COG result can be expressed as $ta(b) = C$ where C is some constant, or alternatively as $ta(b_1) = ta(b_2)$, i.e. the theoretical average accuracy is independent of the bias b_1 or b_2. For functions with boolean outputs, $C = 0.5$ (assuming we assign 0 or 1 to true or false). Alternatively we could assign the two class label the values -1 and +1 and then $C = 0$.

The COG result is in a similar vein to the NFL proofs. Given we are learning with a GP system, NFL does not apply (see section 3.4), however if we are trying to learn all functions the above result holds independently of the learning system we are using. The central assumption behind the COG result is that all functions are equally likely (i.e. we have uniform distribution over all functions). If the distribution is slightly non uniform then one learning algorithm will have a slightly better generalisation performance than another over all functions. If the distribution is highly non uniform then one learning algorithm will have a large increase in generalisation performance than another. We can write down the expected generalisation ability over a general probability distribution $p(f)$ (i.e. the probability, of a function f occurring in practise), which we call the practical average accuracy $pa(b)$, (again this is taken from Wilson et. al. [83])

$$pa(b) = \frac{\sum_{f \in F} p(f) g(b, f)}{\sum_{f \in F} p(f)}$$

This equation reduces to previous one when $p(f) = \frac{1}{|F|}$ (i.e. each function is equally likely). The question is "what is $p(f)$?", or putting it another way, how can we bias our search algorithms to favour functions which are more likely?

3.7.2 Complexity and the distribution of functions

In this section, we examine how we can make Occam's Razor a provable statement and what assumptions are necessary. The assumptions which needed should be made explicit. In the COG proof, the assumption is made that all functions exist with equal probability. If we are attempting to prove Occam's Razor, then we need to make different assumptions. However, Webb states [81]

"Several attempts have been made to provide theoretical support for the principle of Occam's Razor in the machine learning context. However, these amount to no more than proofs that there are few simple hypotheses and that the probability that one of *any* such small selection of hypotheses will fit the data is low."

In other words, the assumptions essentially build in the conclusions and therefore we are gaining very little from such arguments. In this section, we attempt to make headway by not including such assumptions. We take Wilson et. al.'s [83] definition of practical average accuracy and relate it to the complexity of functions. Given a hypothesis space, we can define a probability distribution over this space. Let us assume a uniform probability distribution over the hypothesis space. We repeat the equation for the practical average accuracy from above, but this time define the general probability distribution $p(f)$.

$$pa(b) = \frac{\sum_{f \in F} p(f) g(b, f)}{\sum_{f \in F} p(f)}$$

where $p(f) = \frac{N_{p,f}}{N_p}$, $N_{p,f}$ is the number of hypotheses p which compute the function f, and N_p is the number of hypotheses in search space. $p(f) = \frac{N_{p,f}}{N_p}$ is the fraction of hypotheses in the search space which compute the function f. In words, we take the probability distribution $p(f)$ of a function f to be the frequency with which the functions are represented in the hypothesis space.

We need some way of relating our probability distribution $p(f)$ to the probability distribution of real world problems. Let us assume that each problem corresponds to a hypothesis so we equate a problem with a hypothesis (i.e. the algorithm which computes the problem), and thus the frequency of problems corresponds with the frequency of functions. We attempt to justify this assumption in the discussion below (section 3.7.3).

In the above equation we weight the probability of functions with $p(f)$. We make the assumption that less complex functions are represented more frequently. Thus less complex functions which are consistent with the observed data are more likely to generalise. We do not prove this here, but state it in a form which may be proved or disproved. We believe this is an intuitive statement to make and is consistent with the observations which can be made from the work of Langdon et. al. [40].

3.7.3 Discussion

Why do we equate the set of problems with the set of hypotheses? Behind any set of observations is some sort of 'computation' by a physical system. This could be, for example, dropping an apple from different heights from a tree to 'compute the time to contact' with the ground. We make a set of observations relating height to time. Each problem can therefore be represented by the algorithm which describes the behaviour.

What type of representation shall we use to construct our hypothesis space? The type of representation used will affect the frequency with which functions are represented. A physical system, as far as we can ascertain has finite limits and it is therefore not possible to simulate a Turing Machine. Physical systems may, therefore, due to physical limitations be more like Finite State Machines. Therefore we consider FSM to be a sensible representation to use as a hypothesis space.

There are many biases which can be applied to affect the learning of an algorithm. One important one is cross validation (CV). With CV, part of the test set is not included in the training set and is used for independent testing (see Bishop [7]). This is an important bias as it is exactly what we want our solutions to do: generalise to unseen data. It would be interesting to see what assumptions are necessary to make CV a provably good bias. It has been shown that, under the assumptions of NFL, that CV gives no overall benefit (Zhu et. al. [95]).

3.8 Koza's lens effect and representational bias

3.8.1 Koza's lens effect

Koza [36] (chapter 26) talks about the role of representation and the lens effect. He examines the probability of generating a solution to the even-3-parity problem with three different types of representation; look up tables, tree based representations and modular representations (i.e. ADFs, which we call forest based representations). For a look up table, the chance of generating a correct table is 1 in 256 (i.e. $1/2^8$). There is a uniform distribution of generating any 3 arity boolean function. Given a function set $\{AND, OR, NAND, NOR\}$ he generates 10^7 trees at random, but finds no solutions. However, if ADFs are used (two, two argument ADFs), 35 solutions are generated. Koza calls this difference the lens effect and talks about the problem environment being viewed through the *'lens of a given type of representation'*. It is the aim of this section to explore this effect, and we also include Turing Equivalent representations in this set of representations. Thus in this section, we concentrate on representational bias, rather than how we search the hypothesis space (i.e. search bias).

Cover et. al [11] also consider the effect of representation ([11] section 7.6, page 162), where

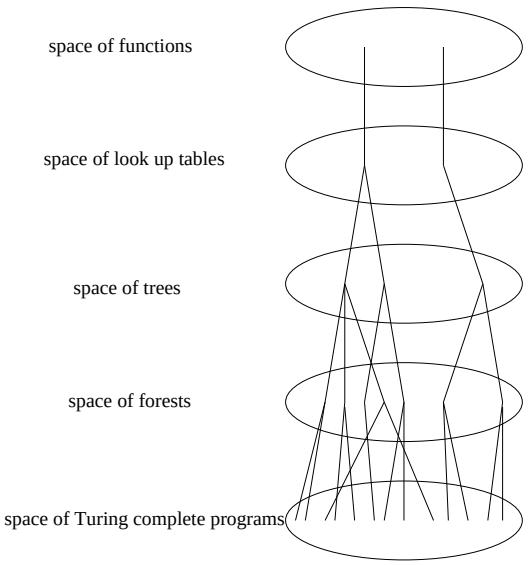

Figure 3.9: For a given amount of computer memory we can represent a set of functions. The set of functions we can represent, and the frequency with which they are represented, depends on the type of representation we choose. If we represent functions with LUTs, the functions are represented with uniform frequency. If we use a tree based representation, the distribution with which functions are are represented is non uniform (i.e. some functions are represented more frequently than others). If we use a forest based representation, we can think of the space of forest based representations mapping onto the space of tree based representations, which maps onto the space of functions (i.e. imagine a forest being unfolded to its equivalent tree representation where each module call is represented explicitly). A uniform sampling of the space of forests corresponds to a non uniform sampling of the space of trees, which corresponds to a non uniform sampling of the space of functions. We can think of looking at the space of functions through a 'tree-lens' or through a 'forest-lens'. Or alternatively, we can think of looking at the space of functions through a compound lens, i.e. we look at the space of trees through a 'forest-lens', then look at the space of functions through a 'tree-lens'. Note that for a given amount of space, the space of forests will be able to represent functions that cannot be represented by trees due to the space restriction. However, in the diagram we have only shown functions which can be represented by LUTs. In the diagram we also include Turing Complete representations.

they consider a monkey typing randomly at a typewriter and at a computer. The probability of producing the works of Shakespeare is exponentially more likely when using a computer, than when using a typewriter. In the terminology of this chapter, the typewriter is analogous to a look up table i.e. the typewriter prints exactly what the monkey types. A computer is analogous to a Turing Machine, i.e the computer executes the program typed by the monkey. There is an increased chance of the works of Shakespeare being printed out as the works of Shakespeare are highly compressible.

Given that we are trying to learn a function, we need a way to represent our programs. We consider a set of representations which we will describe in some detail and show how these relate to one another. Possibly the most 'popular' representation used in GP is the tree based structure use in Koza's original work [36]. When a tree based representation is used, a set of primitives (a function set and a terminal set) are specified and tree structures are constructed from these. Another more sophisticated and flexible representation involves modules (i.e. ADFs), which allows reuse of component parts of the system and therefore effectively allowing the representation to change (as modules can effectively define new primitives).

In the 'standard GP' representations described above (tree and modular representations), state information is typically not stored in the representation, and there are no memory and iteration constructs. These types of representation are not Turing Equivalent. Teller [71] shows how we can extend 'standard GP' to become Turing Equivalent by adding indexed memory along with read and write primitives and an iterative construct.

For the sake of completeness we include LUTs as a type of representation. In this representation data is effectively listed directly. NFL theorems are valid for this representation as discussed in section 3.5.2. The other 3 types of representation (trees, modular representations, and Turing Equivalent representations) are all many to one non-uniform and as discussed in section 3.5.3, NFL theorems do not apply to these types of representations.

We need to take into account the fact that we only have a finite amount of computer memory in which we can express our representation, and we consider the nature of this practical limit. We examine this by qualitatively studying the distribution of functions expressed for these 4 different types of representation. We consider these representations in the following order: look up tables, tree based representations, modular representations and Turing Complete representations.

3.8.2 A hierarchy of types of function representation

Look up tables

We can use a LUT to express data. Given some data, it is a straight forward process to generate a LUT, which records the data directly. Observations are encoded directly in the representation, but

this gives us no information about unobserved data. This is a very restricted representation, and cannot even express the independence of the output on a given variable as we are 'forced' to list it in this representation. We will relax this condition in the next representation, in order to allow more flexibility. LUTs *memorise* the data, offering no compression.

While this restricted type of representation is included here as it comes at the bottom of the hierarchy, we would not use such a representation for learning, but this corresponds to the type of representation used in NFL theorems proofs [56]. It has a one to one representation, i.e. a function (set of observations) is represented by a unique LUT (assuming an ordering on the inputs to the LUT). In other words, if we try to induce a function over a space of LUTs, we get the NFL result.

Consider attempting to learn a function using a LUT. Although we could directly assign the outputs to the LUT, imagine moving through a search space as we would when using a learning algorithm. If the fitness function is a straightforward measure of error and we are using a hill-climbing type search algorithm, the landscape would have no local optima (i.e. the landscape is unimodal).

If we relax the ordering on the inputs of the LUT, then any permutation of rows of a given LUT will represent the same function. For example, the LUTs in table 3.2 could be represented in 4! different ways by rearranging the rows in different orders. There will be a uniform many to one mapping between the space of LUTs and the space of functions they represent. The NFL theorems will not be valid as there will exist some search algorithms which are better than others, however such a search space offers no representation bias as all functions are represented with equal frequency. Consider a space of unordered LUTs mapping to the space of ordered LUTs (which is a uniform many to one mapping), and the mapping onto a space of functions (which is a on to one mapping).

Tree based representations

We relax the constrain above that each observation has to be listed directly, and instead use a more widely used tree based expression, which is typically used in GP [5, 35]. This representation may allow some compression of the observed data, and therefore some generalisation. A given tree may have repeated sub tees, but a purely tree based representation cannot express regularity about its own structure (i.e. regularity in the solution itself), hence we introduce the next representation which has the addition of modules.

Modular representations

We may have a system which permits modules (see chapter 4). Any repeated sub trees in a tree structure can be extracted and be replaced by modules calls, the module being defined once elsewhere. Modules have the advantage that they can be reused, so only need to be represented once,

and can be called when needed. Even with this improvement in the 'efficiency' of the representation, we can go further and make the system Turing Equivalent (i.e. included memory and iteration). A tree based representation can be considered as a special case of a modular representation, where each module can only be used once (i.e. there is no reuse in a tree based representation).

Turing Equivalent representations

It does not matter how we extend the previous representation as all Turing Equivalent representations are equivalent and can simulate each other within a constant amount of computer memory. One possible way to supplement the standard tree based representation used in GP is described by Teller [71]. This requires the addition of read and write primitives along with some indexed memory and an iterative construct. Turing Equivalent representations not only have the ability to express modules, but can also express recursion and can store state information about a problem environment in the form of memory. It offers the most compression and is the most expressive representation.

3.8.3 The relationship between these types of representation

Above we have described four different types of representation; look up tables, tree based representations, modular representations and Turing Equivalent representations. In this hierarchy, LUTs are the least flexible and effectively just list the data. Tree structures followed by modular structures are slightly more flexible and offer some compression. Finally any Turing Equivalent representation is found at the top of the hierarchy and offers the most compression and is the most expressive representation. Each representation is a special case of the previous representation, for example, any instance of a tree based representation is also an instance of a modular representation (where there are no modules i.e. a main tree with no ADFs) and any instance of a modular representation is also an instance of a Turing Complete representation (where there is no use of memory or iteration).

The distribution of functions represented will be uniform if we have a uniform distribution over a hypothesis space consisting of LUTs. If a tree based representation is used the distribution of functions will be non-uniform. The distribution of functions represented is distorted by the tree based representation. If we move to a modular representation the distribution of functions will shift again. The differences in the frequency with which functions are represented as we construct hypotheses spaces using different types of representation are what Koza calls the lens effect. We can think of looking at the space of functions through a 'modular lens' and this will make some functions appear more frequently than other functions. Alternatively we can think of looking at the space of trees through a 'modular lens', and then looking at the space of functions through a 'tree lens'. A uniform distribution over a hypothesis space consisting of modular representations corresponds to a non uniform distribution over a hypothesis space consisting of a tree based representation, which

in turn corresponds to a non uniform distribution over the space of functions. This situation is like using a *compound lens*. Similarly as Turing Equivalent representations are a superset of modular representations, we can think of looking at the space of modular representations through 'Turing Equivalent lens'.

LUTs are different from the other types of representation (tree, modular and Turing Complete representations) in the nature of the mapping between the genotype (program) and phenotype (function). LUTs have a one to one mapping between the genotype (program) and phenotype (function), all the other types of representation have a many to one mapping between the genotype and phenotype. With a LUT we can directly manipulate the phenotype (the semantics), a small change in table corresponds to a small change in its behaviour. LUT are 'transparent' with respect to semantics. With the other types of representation we are manipulating the representation (genotype) and small changes in the representation typically correspond to a large changes in behaviour (the phenotype).

3.8.4 Other examples of representational hierarchies

Above, we have taken a basic representation (a LUT) which directly lists the data, we have introduced more sophistication into the representation and migrated to Turing Equivalent representations, which are the best data compressors we have. Alternatively we could have used any other type of representation e.g. artificial neural networks (ANNs) or classifiers systems (CSs) [34] to illustrate this hierarchy. The LUT form of ANNs (for boolean functions) is described in Mitchell [45] (page 105). The LUT form for CS involves leaving the *wildcard* # out (i.e. just listing the observed data directly). Each of these representations could be extended in a similar fashion to the representation we have used in section 3.8.2 as a basis, to end up with a Turing Equivalent representation. We also point out that there are different routes between the basic representation of LUTs and Turing Equivalent representations, for example we could have added iteration (see section 2.1.5) before adding the ability to express modules. Another example of a hierarchy is the Chomsky hierarchy consisting of finite state automata, push down automata and Turing Machines (it is trivial to construct a LUT in this representation).

3.8.5 The hierarchy of functions expressed by different types of representation

For a given amount of computer memory there is a set of functions we can represent with a LUT. For the same amount of computer space, we can express functions using trees. As LUTs are a subset of trees, any function a LUT can represent, a tree can represent. There are also functions that trees

can represent which LUTs cannot represent. For a given instance of a LUT there are many ways to represent that with a tree and this will alter the frequency with with functions are represented using these two representations.

3.8.6 Learning and generalisation

How easy it is to learn a given function using given a type of representation? How well is a given type of representation likely to generalise? This will in general depend on both the target function and the type of representation we are using, however we can make some general statements. With LUTs, all finite boolean functions are easy to learn (see section 3.8.2). All functions (for a given size) are equally easy to learn (consider learning with random search and Koza's lens effect). On the other hand, for Turing Equivalent representations, simple functions are typically much easier to learn than more complex ones (as there are more programs corresponding to simple functions so random search is more likely to find simple functions).

If we are attempting to learn a finite incompressible function (i.e. a function which is a mapping between two finite sets and and cannot be expressed in less space than listing the mapping literally), with say a Turing Machine, it will effectively represent the solution as a LUT. If the problem is compressible, a Turing Machine can exploit any regularity and reduce its size and will therefore be able to generalise (assuming that simpler functions are more frequent than complex ones). Even if a LUT is learning a compressible problem, it is still stuck without being able to generalise as it offers no bias.

Teller [73] (Appendix B) in his thesis examines the hierarchy of languages and their computational expressiveness (the Chomsky hierarchy) and states that *'hill climbing works less well in the more expressive languages'*, which depends on *'the tight coupling of syntactic and semantic changes'*. In section 3.8.2 we have a different hierarchy. As we climb the hierarchy, Hill Climbing gets more difficult, it is trivial for LUT as the landscape has no local optima (due to the tight coupling of syntactic and semantic changes). It is more difficult with representations higher up as the effect of chaos plays a greater role. Mitchell also makes a similar observation when comparing feed forward neural networks and recurrent networks; typically recurrent networks are more difficult to train than networks with no feedback loops.

Teller [73] (section 1.1.2) in his thesis states *'in the space of functions, ..., the density of functions that do something 'interesting' is very low. This is increasingly the case as the expressiveness of the language in which the programs are written moves up the ladder from regular languages to Turing Machines.'* This is also the case with the hierarchy presented here as we have tried to illustrate in figure 3.9. As we move from one type of representation to the next, the frequency with which functions are represented changes, and as we move up the hierarchy, 'interesting' (more complex)

functions are represented less frequently.

Size of different representations

Due to the nature of the way each type of representation expresses data, this will have a connection with the representational complexity of a function expressed in that representation (see chapter 4 for a definition of representational complexity). For example consider the even-n-parity function, where n bits are mapped to true if the number of true bits are even, otherwise false. Consider representing all possible observations (of which there are 2^n) with each of the types of representations. For a LUT the size rises exponentially with n (see Mitchell [45] page 105 for and explanation of how to construct LUTs with ANNs). For a tree based representation the size rises linearly with n. For a forest based representation the size also rises linearly with n. However with a Turing Equivalent representation the size is independent of n (see 5.2). We later argue (chapter 4) that the complexity is related to ease of learning and do experimental work (section 5.2). As the size of a problem increase, the complexity of the solution increases (except in the case for Turing Equivalent representations). Hence the number of evaluations required to find a solution will increase ([36] chapter 10) This is central to chapter 5 where we evolve general solutions to the even parity problem. For a given function, as we climb the hierarchy (from LUTs to Turing Machines), the representational complexity never increases.

3.9 Discussion

3.9.1 Learning and optimisation

While learning and optimisation may appear similar, i.e. we are trying to optimise some performance measure, there are some fundamental differences. An optimiser has to find a good value which has an optimal or near optimal value. Almost all optimisers find good values rapidly. As the majority of functions are incompressible, even random search will produce decent values very soon [18]. In contrast, a learner is evaluated on it predictions (we want to minimise the errors on the predictions), but almost all descriptions of data are incompressible and the learner, therefore, is most likely to make random guesses [18]. As the majority of functions are incompressible, it is very difficult to make predictions about the behaviour on unseen inputs (effectively we can only make random guesses on all but a very few functions which are compressible). A representation needs to be chosen in which to express any regularities that may exist in the data. For representations (other than LUTs), there is a many to one mapping between the representation of a rule and the rule itself (i.e. the functionality). A LUT could be used to represent the data, but this would offer no compression and no generalisation.

Also, with learning we are usually looking for an instance of whatever representation we have chosen, the get a small error score (simultaneously on the training and validation sets). This value must be positive. With optimisation, unless we have specialist knowledge of the function, we do not know what the maximum or minimum value may be.

Also, with learning, we are no simply looking for a solution which has a small error on the training set, but also have a low error on an independent validation set. A high error on the validation set would indicate over fitting and lack of generalisation. The shape of the curve on the validation set is typically 'U' shaped, and when the minimum of this curve is reached, this is usually taken as an indication of when to stop training as over-fitting is beginning to occur. Thus with learning we have some indication of when to terminate our search algorithm. By the nature of the metrics used to measure error, these values are always positive. There is a large amount of literature about stopping criteria, and it is interesting to consider what would happen if we were learning a random function or if we were using a LUT for learning,

With optimisation, the situation is somewhat different. Given a function we are trying to optimise, we have no nature of the value of the optima, which may be positive or negative. Therefore we have no way of telling our algorithm to terminate [16].

There may be further differences between learning and optimisation regarding noise, however we have not considered that possibility here. We do however regard this as important as most real world problems will contain noise of some sort.

3.9.2 Measuring evaluations

One of the things NFL theorems force us to think more carefully about is how we count the number of evaluations. NFL theorems do not allow re-evaluation of points which is not realistic. In Evolutionary Computation, we typically count the number of generations, however in [91] we consider using only a subset of the set of test cases and therefore need a better method of counting (i.e. counting program evaluations on individual test cases). Care should be taken as this may not be a good indication of true performance in terms of wall clock time as some programs take longer to evaluate than others due to their size.

Ultimately we would like to compare wall clock time, as this would give us the measure of comparison we are looking for however, different researchers may implement algorithms in different ways, and we would then be comparing their ability in writing algorithms rather than the efficiency of the algorithms themselves. Also, researchers may be carrying out other processes (for example book keeping for statistical analysis for examining different aspects of the run). Also there will be the issue of the dependency on different machines. Perhaps a fairer way would be to count the number of nodes that are evaluated (this would rely on the assumption that each node takes the

same amount of time to evaluate). This will be the measure we use in chapter 5 and 2.

3.10 Summary

3.10.1 No free lunch

The NFL states that all search algorithms perform equally well over all functions, under some assumptions e.g. revisiting points is ignored. If these assumptions are broken, NFL is not valid. One of these assumptions is that we consider all functions then it is not possible to generalise i.e. after observing some data, all predictions we make are equally likely. This is simply due to the fact that all functions are equally likely hence we cannot prefer one function over another. i.e. all consistent functions are equally likely After making a number of observations, we can discard a subset of all functions. But among the remaining subset, they are all equally likely.

NFL strictly applies to mapping between two finite sets. When we attempt to extend NFL to infinite sets we see that we run into trouble, as there are functions and search algorithms which we cannot express.

When we consider NFL in the context of GP, it is not possible to have all mapping between the set of genotypes (tree based representations of programs) and the phenotypes (the functions the programs represent) due to the many to one nature of the representation. Hence NFL is not applicable to GP, which is a central result of this chapter.

3.10.2 Occam's Razor

Occam's Razor is often taken as meaning 'prefer the simpler hypothesis over more complicated ones, when the hypotheses agree on the observed data' We equate the simplicity of a hypothesis with representational complexity (i.e. the smaller the representational complexity of a hypothesis the simpler it is). If our spaces of possible hypothesis is the space of LUTs, they all have the same representational complexity (as they all directly encode the observed data) and there is no preference of one over another. This situation corresponds to NFL.

LUTs are not a very flexible representation i.e. they offer no compression of data. Computer programs are the most flexible type of representation we know and have the ability to express composition, iteration (recursion) and minimisation. If we have a uniform distribution of programs up to some size, we see that simpler functions are represented more frequently than others.

3.11 Conclusions

3.11.1 Problems with NFL

The NFL theorems are an interesting collection of theorems about search operators. They are subject however to some rather strong assumptions. The NFL theorems have no concept of a landscape which is an important consideration when designing a search operator. This problem is effectively side stepped by ignoring the revisiting of points. There are problems when extending NFL from functions with finite domains to functions infinite domains. There is also problem that we have to assume that we know what the range of the function is, which is often not the case (see figure 3.4).

We could also state NFL as all problems are equally difficult (or easy) i.e. they are easy for some search algorithms but difficult for others but over all search algorithms all problems are just as difficult or easy as each other. This is somewhat counter intuitive as some problem, one would agree are more difficult than others.

3.11.2 NFL and Occam's Razor

There is a conflict between the NFL theorem's and Occam's Razor. A proof is only as strong as the assumptions on which it rests. If we assume that the distribution of problems is uniform, we have to accept NFL theorems as it can be proved on this assumption. Similarly, if we assume a distribution of problems where simpler problems exist with more frequency than complex problems, we have to accept Occam's Razor. (We should note that there is also a problem with Occam's Razor as it does not say anything about the halting problem, but neither does NFL).

3.11.3 Definition of problem classes

One important issue that NFL theorems raise is to qualify what class of problems a search algorithm is suited to. It does not make sense to say a certain algorithm is the best for a given problem (the one shot scenario [15]), but really for a given type of problem. For example, given we want to find the minimum of $y = x^2$, the best algorithm simply prints out 0 and halts. If we are interested in finding the optima of a set of functions e.g. unimodal functions a bisection algorithm would be useful. However, here we have explicit knowledge of the functions. In reality we only have implicit information about a set of functions e.g. it is a set of image recognition problems. Thrun et. al [77] (page 7) state that there is not a best learning algorithm for learning a set of problems (this is the NFL theorem in a different guise). If the size of class of problems is infinite then the NFL theorem does not apply. Burke et. al. [10] in their conclusions that report *'a GA might be better employed*

in searching for a good algorithm rather than searching for a specific solution to a specific problem'. Miller et. al. [31] study an evolutionary approach to the design of digital circuits and argue that *'by studying the evolved design of gradually increasing scale, one might be able to discern new, efficient, and generalisable principles of design'*. This is an example of a situation where a class of problem are defined and we can design a good search algorithm for that class of problem.

Chapter 4

Representational Complexity

4.1 Introduction

4.1.1 Modularity

Modularity is a cornerstone of software engineering and is often introduced in software engineering and programming books early on. Those of us who have programmed computers are familiar with modules perhaps under different names: procedures, subroutines, functions (see Banzhaf et. al. [5] page 283 for a discussion of modules). Modularity allows *'the division of software into manageable units, each of which performs only part of the overall task'* (Brookshear [9] page 275, section 6.3 Modularity). It is this idea of dividing a task into subtasks which is used in the algorithm introduced in [90].

Modules are a potential method to avoid reinventing the wheel: once a sub-problem is solved, it does not have to be resolved. The appropriate module can be called again with the appropriate arguments when needed. Hence the ability to represent modules avoids having to represent solutions multiple times. They can be represented once and called on demand. This idea of reuse cannot be expressed using a tree based representation as during the execution of a tree based program each node can be evaluated at most once.

4.1.2 Modularity in genetic programming

In standard GP, a tree based representation is used (see Koza [35]). Functions are created by the operation of composition, from a given set of primitive functions (we refer to the primitive set rather than a function set and terminal set). We will call this type of representation a *tree based representation* (see definition 4.2). Modules cannot be expressed in a tree based representation. An extension to tree based representation is the ability to express modules (i.e. reuse), (see definition

4.4), we will call this type of representation a *forest based representation* (see definition 4.4), or a *modular representation* to distinguish it from the standard tree based GP representation.

4.1.3 Representational complexity

In this chapter, we define representational complexity as the size of the smallest expression needed to express a given function (see section 4.2). In general this will depend on both the function being expressed and the type of representation being used (i.e. a tree based representation or a modular representation). We prove that if the representation being used is capable of expressing modules, then the representational complexity of the function is independent of the primitive set of the representation being used (within a constant). We call this *representational complexity* to explicitly remind the reader we are concerned with representation, and that the term 'complexity' is not to be confused with time or space complexity associated with the study of the performance of algorithms. The fact that the complexity of a function is independent of the primitive set if modularity can be expressed, is used as inspiration for a representation and search operator which perform independently of the primitive set [90].

4.1.4 Complexity and problem difficulty

Some problems are more difficult to solve than others. One of the reasons for this may be the difference in the underlying complexity of the problem. For example, consider two different problems of different complexity and let enumeration be our search algorithm. Enumeration will solve the simpler problem first (assuming we order the solutions from simple to complex). Similarly, random search will find the solution to the simpler problem first (on average) as the simpler solution is represented more frequently (see section 3.6.3). It is more difficult to make general statements of this kind for other search algorithms.

Consider two problems of the same complexity, one problem is compressible and the other is incompressible. If random search or enumeration are used, then, as they have the same complexity, they will be solved equally easily. If enumeration is used, as the problems have the same complexity the solutions will appear in approximately the same places in the enumeration (i.e. enumeration lists all solutions of a given size before listing all solutions of a larger size, so it will find solutions of the same size at approximately the same time). Similarly, if random search is used, the solutions will be found at around the same time as simpler solutions are represented more frequently. However, if a different learning algorithm is used, which makes use of gradient information in the landscape for example, it is not clear how the performance will depend on the problem.

4.1.5 Representation and complexity

In GP the choice of function set is critical. We will use the term primitive set to mean the union of the function and terminal set. For a given problem, it is often easy to define a primitive set expressive enough to solve the problem, but this choice can greatly affect the performance of a run. We are only aware of one paper which addresses this issue (Wang et. al. [80]), which is perhaps a little ironic as this choice is the first choice listed in 'The Basic GP Algorithm' in Banzhaf et. al. [5] (section 5.6, page 133).

For example, consider learning (or just representing) the even parity problem using tree based GP. If the function XOR is included in the primitive set, the problem is trivial, but it is more difficult if other primitive sets that do not include XOR or its negation are used . This difference could be due to the difference in complexity of the function under the two primitive sets, i.e. the complexity under the two different representations. If we have modularity in the representation the difference in complexity is removed within a constant, and this inspires the representation proposed in [90]. The claim is made that if a tree based GP system is used, there will be a difference in performance when comparing runs using two different primitive sets, however this difference is reduced if a modular GP system is used.

4.1.6 Contributions of this chapter

Below we list the contributions of this chapter:

1. a proof that the complexity of a function is invariant under the translation of primitive set (within a constant bound additive) if the representation is capable of expressing modules.

2. a proof that is bound is the tightest bound we can achieve.

3. a proof that the complexity of a function has a constant lower (within a constant bound subtractive) if the representation is capable of expressing modules and that this is the tightest bound.

4.1.7 Outline of this chapter

In section 4.2 we give some preliminary definitions regarding representation and complexity. In section 4.3 we prove theorems concerning the complexity of functions when tree based representations are used to express a function. In section 4.4 we prove theorems concerning the upper and lower bounds of a function when expressed using representations capable of expressing modules. Finally in section 4.5 we provide a summary of the chapter.

4.2 Preliminary definitions

In this section we provide a number of definitions. A number of these definitions are taken from the GP literature, and the reader may be familiar with them. A number of other definitions are provided concerning different types of representation and what it means to translate from one primitive set to another primitive set.

Definition 4.1 (Primitive set). *A primitive set is an atomic set of functions, which are taken as given. This set is finite in size. The primitive set can be considered to be building blocks for more complicated functions and are the only functions that can appear in a GP solution.*

For example, we could have the primitive set P_1 which consists of the set $\{NOT, AND, a, b, c, d\}$, where NOT and AND are logical functions, and a, b, c and d are boolean problem variables or constants (i.e. functions without input). We also introduce a second primitive set at this point P_2 which consists of the set $\{NAND, a, b, c, d\}$, which will serve us in a number of examples later on.

In the standard GP paradigm we often refer to the primitive set as two separate sets; a function set and a terminal set (see Banzhaf et. al. [5] section 5.1, "Terminals and Functions - The Primitives of Genetic Programs"). Typically the terminal set contains the problem variables and can be thought of as function with zero arity (i.e. do not take an input). The function set is a set of functions which are considered to be useful or necessary to solve the problem at hand. The primitive set is simply the union of the function set and the terminal set. In the first example above (primitive set P_1), the function set is $\{NOT, AND\}$, and the terminal set is $\{a, b, c, d\}$. Typically we are not concerned with how these functions are implemented. In fact they may be implemented as look up tables (this is in fact done in [90]). Given a set of primitives, we can construct more functions by taking the output of one primitive function and feeding it into another primitive function. There is an important distinction between two ways of generating more functions from a set of primitives. In the first case, constructed functions can only be used once (see definition 4.2), whereas in the second case (see definition 4.4) constructed functions can be *reused*.

Definition 4.2 (Tree based representation). *A representation is called* tree based *if it can be expressed using a tree data structure. Each node in the tree can only be executed once when data is presented.*

This is the typical type of representation used in Koza style standard GP (Koza [35]). See figure 4.1. Note that each node is executed at most once during and execution of the whole data structure. A tree may be evaluated bottom up or top down. If it is evaluated top down then in some cases evaluation time can be saved. For example if a node contains the primitive AND, and the left child node evaluates to false, then we do not need to calculate the right child node as we know the whole

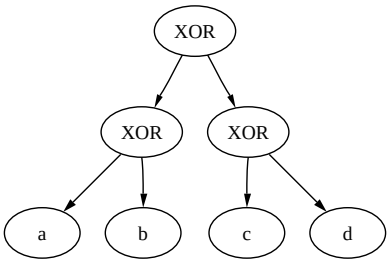

Figure 4.1: An example of a function being expressed using a tree based representation. The primitive set contains the function XOR and the terminal set contains the primitives $\{a, b, c, d\}$.

subtree will evaluate to false. A single node in a tree can never be evaluated twice during a single execution of the whole representation.

We should distance ourselves from the case where Turing Complete programs are represented as tree structures (as Teller does [71]). In this case subtrees in the whole structure may be executed multiple times. We distinguish from this type of interpretation of a tree representation by demanding that nodes may only be executed once in an execution of the program.

Definition 4.3 (Module). *A module is a function that is defined in terms of a primitive set or previously defined modules. New modules are defined by the operation of composition. A module can be used as a primitive and thus can be used, once, multiple times or not at all. See section 2.1.4 for the definition of composition.*

For a review of modular search methods in GP see section 2.3. In all of these works modules are defined in accordance with this definition. Note that the size of a primitive set must be finite, as it is with GP. See section 2.1.4 for a definition of composition.

Definition 4.4 (Modular representation). *A modular representation is any representation capable of expressing modules (i.e. reuse of component parts of the system). This corresponds to the forest data structure (i.e. a set of tree data structures).*

As a single module can be represented as a tree and a modular representation will in general contain multiple modules, we are therefore expressing the function using a forest data structure. Note that each module could be expressed as a tree or a forest.

In order to prove the results of the following sections it will be useful to be able to switch between different primitive set (i.e. to be able to express a function using one primitive set or a different primitive set). This process of switching from one primitive set to another is called *translation of primitive set*, and is achieved using a dictionary which we now define.

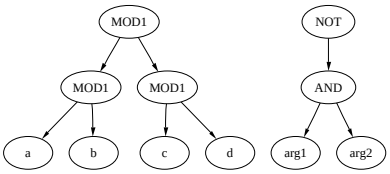

Figure 4.2: Example of a function expressed using a modular representation. The primitive set contains the function set AND and NOT and the terminal set contains the primitives $\{a, b, c, d\}$. On the right hand side is a module which takes two arguments (denoted by arg1 and arg2) and returns the result. On the left hand side is the main program which makes use of the module, calling it 3 times.

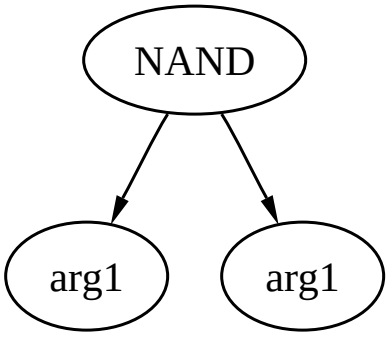

Figure 4.3: A term. This is the primitive NOT (from primitive set P_1) being expressed in terms of primitives from primitive set P_2).

Definition 4.5 (Term in a dictionary). *A term, t, in a dictionary $D_{P_1P_2}$ tells us how to express a primitive in the primitive set P_1 using only the primitives in set P_2. A term will be a tree if a tree based representation is being used, or a forest if a modular based representation is being used.*

A term is a function expressed in terms of the primitives in the current primitive set, which expresses a primitive from a different primitive set (i.e. the function performed by the primitive from one primitive set is expressed in terms of primitives from a different primitive set). The function corresponds to the primitive being translated from, and is expressed entirely of primitives from the set being translated to. See figure 4.3 for an example of a term.

Definition 4.6 (Dictionary). *A dictionary $D_{P_1P_2}$ is a set of terms which express each member of primitive set P_2 in terms of primitive set P_1.*

The existence of $D_{P_1P_2}$ does not imply the existence of $D_{P_2P_1}$. The size of $D_{P_1P_2}$ is denoted by

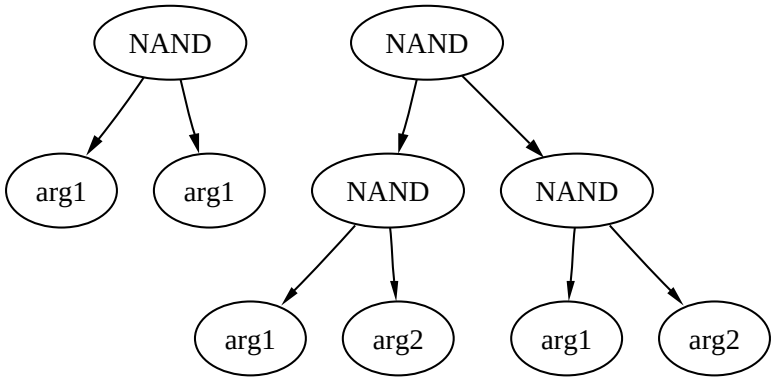

Figure 4.4: A dictionary $D_{P_1P_2}$. The tree on the left expresses NOT in terms of primitive set P_2. The tree on the right expresses AND in terms of primitive set P_2. Thus, given a function expressed using P_1, we can use this dictionary to express the function in terms of P_2. (Note that there are many ways of expressing this dictionary).

$|D_{P_1P_2}|$, and in general, $|D_{P_1P_2}| \neq |D_{P_2P_1}|$ (whenever $|D_{P_2P_1}|$ exists). A dictionary must be of finite size.

The number of terms in a dictionary will be at most the number of primitives in the set (as each term needs to be expressed in the second primitive set). The number of terms in a dictionary could be zero if the new primitive set is a superset of the previous primitive set (i.e. none of the primitives need to be translated as they are all contained in the new primitive set). The primitives which appear in both sets (i.e. $P_1 \cap P_2$) do not need to be translated and therefore do not need to appear in the dictionary.

Definition 4.7 (Equally expressive primitive sets). *Two primitive sets P_1 and P_2 are equally expressive if and only if a pair of dictionaries $D_{P_1P_2}$ and $D_{P_2P_1}$ exist and are of finite size.*

Two equally expressive primitive sets can express the same set of functions, only the number of primitives in the expressions will differ. There are many examples of equally expressive primitive sets. For example, any pair of logically complete sets are equally expressive by definition (logically complete meaning any logical function can be expressed). Similarly, most programming languages are equally expressive as they are Turing Complete (i.e. one can simulate the other and vice versa, and both can simulate a Turing Machine).

Definition 4.8 (Minimal). *A primitive set is minimal if no member of the set can be expressed in terms of the other members of the set.*

Within a primitive set, one or more of the members of the set may be expressed in terms of other

members of the set. This implies that there is some redundancy in the set and these members could be removed without affecting what could be expressed with that set. If a primitive set is minimal it implies there is no redundancy in the set.

An example of redundancy is the `for` loop construct present in most programming languages. It could be removed from the language without affecting what could be expressed in the language, as a `for` loop could be implemented in terms of a `while` loop with an integer counter. A `for` loop construct does not add anything in terms of expressiveness, but it is included in most languages for the sake of convenience.

An example of a set which is logically complete and minimal is $\{NAND\}$, as this is logically complete on its own, and obviously removing $NAND$ from this set, it will no longer be logically complete.

Cutland [13] introduces a Universal Register Machine, which consists of some addressable memory and a set of instructions. Removing any one of these instructions will reduce the expressiveness of the system.

Definition 4.9 (Size). *The size, $S_P(r)$, of a representation r (which could be a tree or modular representation), which expresses a function f, in terms of a primitive set P, is the number of nodes it contains.*

Will illustrate the idea of size by looking at the previous examples, and in doing so give the definitions of size of tree representations, modular representations, terms and dictionaries. The size of the tree in figure 4.1 is 7. The size of the forest in figure 4.2 is 11. The size of the term in figure 4.3 is 3. The size of the dictionary in figure 4.4 is 10.

Other acceptable definitions of size exist (e.g. the number of bits used to express the tree or forest), however we choose the number of nodes as it is one of the most natural and commonly used in GP (Banzhaf et. al. [5] page 70). Other definitions of size could be the depth of the tree or the number of leaves in a tree.

Definition 4.10 (Representational complexity). *The representational complexity, $C_P(f)$, of a function f, is the size of the smallest tree or forest expressing f, in terms of a primitive set P. In general, we must state what the complexity is with respect to the primitive set we are using and the type of representation we are using (i.e tree based or forest based).*

Note that if we use the definition of size as the number of bits to express the function, and the representation is Turing Complete, this definition corresponds to the definition of Kolmogorov complexity [43] (i.e. the length of the shortest program in bits that expresses the function).

Definition 4.11 (Translating primitive set). *If we have a function expressed in one primitive set using a fixed type of representation (be it a tree based representation or a modular representation)*

we can express the same function in a second primitive set by translating primitive set. This process is simply achieved by replacing primitives with terms from the dictionary. Hence a function expressed using one primitive set is now expressed in terms of a second primitive set.

With a tree based representation each node in the tree has to be translated separately. However with a forest based representation, a translation only needs to occur once as it can be reused.

Definition 4.12 (Representational complexity of a dictionary). *The complexity of a dictionary is the minimum number of nodes it takes to express the dictionary.*

The complexity of a dictionary is the size of the smallest set of terms which express one primitive set in terms of the other. We need to distinguish between the case when we are working with trees and working with forests. If we are working with trees, each terms in one primitive set is expressed as a *tree* structure using primitives from the other primitive set. The complexity of a term is simply the size of the smallest tree which performs the desired function. The complexity of a dictionary when working with trees is therefore the sum of the complexities of each of the terms. While there maybe subtrees in different terms which are the same, these cannot be expressed as modules.

The situation is however a little different when working with forests. If we construct the set of terms for converting from one primitive set to another, if there are shared subtrees within the dictionary, then these can be expressed as modules. Therefore the complexity of a dictionary will depend on whether we are working with trees or forests. For a given pair of primitive sets, the complexity of the dictionary when working with forests will always be less than or equal to the complexity of the dictionary when working with trees.

4.3 Tree representations and complexity

In this section we prove theorems concerning the depth and size of a tree based representation when we switch from one primitive set to a different primitive set.

4.3.1 Translating primitive sets with tree based representations

Theorem 4.1. *Given two equally expressive finite primitive sets P_1 and P_2, the depth of the smallest tree which expresses the function in P_1 is within a constant multiplicative factor of the depth of the smallest tree which expresses the function in P_2, provided modularity is not permitted (i.e. only a tree based representation is permitted). The multiplicative constant is the depth of the deepest term t in the dictionary $D_{P_2P_1}$.*

$$Depth_{P_1}(f) \leq Depth_{P_2}(f) * \max_{t \in D_{P_2 P_1}} Depth_p(p) \tag{4.1}$$

Proof. Consider a tree representing a function in primitive set P_2 which has a certain depth. Each primitive in this tree will be replaced with at least one term from the dictionary. Thus as each layer (set of primitives at a certain depth) in the tree is replaced, in the worst case the deepest term is used each time. □

When a primitive in the original tree is replaced by a term from the dictionary, the term may have higher arity than the primitive it is replacing. Thus, while the depth may alter by a multiplicative constant, the final tree will in general have many more branches that the original tree.

Theorem 4.2. *Given two equally expressive finite primitive sets P_1 and P_2, the complexity $C_{P_1}(f)$ of a function f with respect to primitive set P_1, has an upper bound which is within an exponential factor of the complexity $C_{P_2}(f)$, provided modularity is not permitted (i.e. only a tree based representation is permitted).*

$$C_{P_1}(f) \leq \max_{t \in D_{P_2 P_1}} C_{P_2}(f) * (\max_{t \in D_{P_2 P_1}} Arity)^{C_{P_2}(f)} \tag{4.2}$$

Proof. Given a function expressed in terms of a primitive set P_2, it can be re-expressed in terms of the primitive set P_1 by directly substituting in the terms of each of the primitives from the dictionary. Each function may be replaced with a term from the dictionary with higher arity, and therefore any subtrees below this have to be represented multiple times, leading to exponential growth. □

There are essentially two contributions to the complexity using the new primitive set. This can be seen by thinking of the translation as two stages. Firstly, the tree is rewritten in terms of new 'units' which will have arity equal to or higher than the original primitives (the arity cannot be reduced). Secondly, each new 'unit' will contain in the worst case the most complex term in the dictionary.

For example, consider the tree in figure 4.5 expressed entirely in terms of AND. The primitive AND can be expressed in terms of three $NAND$ primitives, as shown as the second term in figure 4.4. While the primitive AND only has two inputs, the term consisting of three $NAND$ primitives has 4 inputs and each of these must be represented even though there are in actual fact only two distinct inputs. As we are working in a tree representation, there is no way around this. Hence in figure 4.6, while a single AND primitive can be represented as three $NAND$ primitives, any inputs to this has to be represented multiple times.

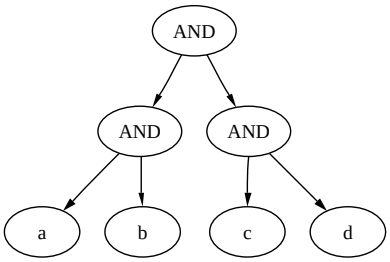

Figure 4.5: A function expressed using a function set containing AND.

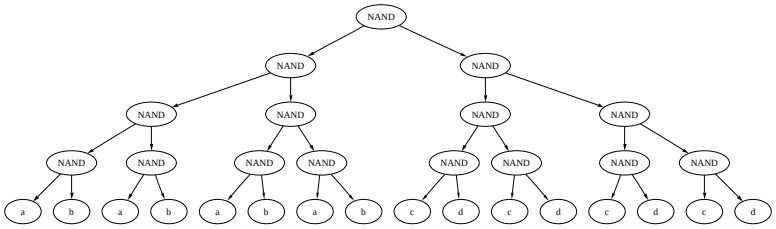

Figure 4.6: The tree (from figure 4.5) expressed in terms a primitive set consisting only of $NAND$. Note that each AND node in the previous tree has to be expressed explicitly in terms of the new primitive set and as the arity of the new primitive is double that of before, each sub-branch has to be represented repeatedly.

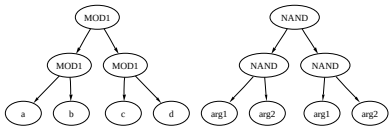

Figure 4.7: This is the representation of the tree from figure 4.5, expressed as a forest in the new primitive set. While AND appeared 3 times originally, it can be represented by referring 3 times to a definition of AND which needs to be represented only once.

4.4 Modular representations and complexity

In this section we prove that the complexity of a function when expressed using a modular representation is bounded by an additive constant. We then go onto prove that this is the smallest bound we can obtain. We also show that a lower bound exists and this is the smallest bound we can obtain.

4.4.1 Translating primitive sets with modular representations

Theorem 4.3. *Given two equally expressive primitive sets P_1 and P_2, the complexity of a function f with respect to primitive set P_1, $C_{P_1}(f)$, has an upper bound which is within an additive constant of the complexity $C_{P_2}(f)$ provided modularity is permitted (i.e. forests are permitted).*

$$C_{P_1}(f) \leq C_{P_2}(f) + K_{P_2P_1} \tag{4.3}$$

Proof. Given a function expressed as a forest in primitive set P_1, it can be expressed as a forest in the primitive set P_2 by adding on the definitions of each of the primitives. Thus in the worst case, all of the primitives need to be rewritten in terms of a forest of size $K_{P_2P_1}$, where $K_{P_2P_1}$ is the complexity of the dictionary $D_{P_2P_1}$. In general $K_{P_1P_2} \neq K_{P_2P_1}$. □

4.4.2 Tightest bound on complexity

Theorem 4.3 tells us that the complexity of a function is bounded by an additive constant. We are now in a position where we can ask ourselves, is there a better constant (i.e. a lower valued constant). We now show that this is the lowest bound we can obtain on the constant.

Theorem 4.4. *Given two equally expressive primitive sets P_1 and P_2, the complexity of a function f with respect to primitive set P_1, $C_{P_1}(f)$, has an upper bound which is within an additive constant of the complexity $C_{P_2}(f)$. This constant is $K_{P_2P_1}$ and is the tightest bound on the complexity of a function when expressed under a different primitive set.*

Proof. Given a function expressed in one primitive set, we wish to translate to a second primitive set. If all of the primitives from the first set are used in expressing the function, then all of the terms in $D_{P_2P_1}$ will need to be used, Thus the whole dictionary must be used. Hence the smallest the additive constant can be is $K_{P_2P_1}$, i.e. the complexity of the dictionary. □

For example, given a primitive set $\{AND, NOT\}$ and a second primitive set $\{NAND\}$, we may have a function expressed in terms of AND and NOT and therefore the whole dictionary is needed in the translation process (i.e. both primitives would need to be translated). If we had a function expressed just in terms of AND only one term in the dictionary would be needed.

4.4.3 A symmetry with the lower bound on complexity

Theorem 4.3 gives an upper bound on the complexity of a function expressed under a different primitive set. By a simple rearrangement of that equation we can arrive a lower bound.

$$C_{P_2}(f) - K_{P_1P_2} \leq C_{P_1}(f)$$

We can then combine these two equations into a single expression.

$$C_{P_2}(f) - K_{P_1P_2} \leq C_{P_1}(f) \leq C_{P_2}(f) + K_{P_2P_1}$$

Note the symmetry in this expression. The size on the upper bound is given by adding the complexity of the dictionary going from primitive set P_2 to primitive set P_1. The size on the lower bound is given by subtracting the complexity of the dictionary going from primitive set P_1 to primitive set P_2. These two bounds are the tightest bounds we can have in general.

Corollary 4.1. *Given two equally expressive primitive sets, the set of functions expressed by forests using either primitive set is approximately the same if the size limit on the forests is large compared to the complexity of the dictionaries ($D_{P_2P_1}$ and $D_{P_1P_2}$).*

This follows as a consequence of Theorem 4.3 as the size of the dictionary can be ignored if the size limit of a forest is much bigger than the complexity of the dictionaries. This additive constant will be ignored from now on and so the set of functions defined by forests (of fixed upper size) is the same.

4.4.4 Discussion of invariants of complexity

Complexity under different primitive sets

Complexity can always be reduced to 1. The complexity of a function can be made equal to one. This can always be the case by introducing a primitive into the set which performs the desired

function. However in reality, in the case of GP, we are not interested in this situation. If we had a primitive powerful enough to express the target function we would not need GP.

The constant is large. Often people state that the constant (i.e. the complexity of the dictionary) could be very large. This is true when we consider programming languages, however this is less so with two primitive sets we may pick for a GP run. If we have a problem specific primitive in both function sets, this will not affect the complexity of the dictionary (i.e. the size of the constant). If a problem specific primitive is included, this would reflect the practitioners knowledge of the problem and not GP.

Different primitive sets. We have some target function we wish to synthesise and we need to choose a primitive set (these are listed as the first two steps of the basic GP algorithm Banzhaf et. al. [5] (page 133), listed separately as defining the terminal and function sets). It is perhaps a little surprising that little research has been done into this given that it is such a fundamental step in the GP process. We must always choose a set which is expressive enough to solve the problem. For example, if we are dealing with a problem that involves logical relations, the primitive set should be logically complete. This is not always the case, for example we use only the primitive XOR to solve the even party problem. In addition we may or may not have domain knowledge we wish to incorporate in the form of extra primitives. For a given problem, two GP practitioners may pick two different primitive sets, but there is unlikely to be a large difference in complexity between the primitive sets in comparison to the complexity of the target function (assuming one does not have 'inside knowledge' about the problem).

Three types of modularity

We can identify three different categories of module hierarchy, each of which can be found in GP systems. Theorem 4.3 is valid for all three types of modularity. In the simplest systems, modules can only be directly constructed from primitives available. This is true of a number of systems (for example see Walker et. al. [79]). In a more sophisticated system we could allow modules to be built from primitives *and* previously defined modules. Finally, we could allow modules to call each other (i.e. calling itself as in recursion and mutual recursion).

Implications for learning

Evolution in GP can be thought of as the movement of a population of points (programs) through a search space. Each point is represented by a tree or forest and corresponds to a function. The aim is to find a point that corresponds to the target function (within some tolerance limit). The terms hypothesis space and search space are used interchangeably and a point in a search space represents a particular hypothesis. A point in the search space is the genotype and the function it

represents is the phenotype. The mapping between these two spaces is many to one, as many trees (or forests) represent the same function (section 3.5). The structure of the search space is defined by the representation, the operators used to move around the space and the fitness function. The structure of the search space is often considered using the landscape metaphor Langdon et. al. [40].

Two different sets of primitives define two different search spaces. These two spaces will map to the same set of functions if the two primitive sets are equally expressive (and the size is unlimited). However, if a maximum depth or size is imposed on a tree, this will effectively cut off part of the space, and therefore limit which functions can be represented. In general, if modularity is not permitted, the set of functions defined by trees in the two spaces will be *different*. If however the representation does allow modularity, and the size of forests in the search space is limited, the set of functions will *the same*. In other words, if we do not have modularity, the functions expressed in two search spaces defined by two different primitive sets will diverge, whereas if the ability to modularise is included the sets of functions will converge as the maximum size of a tree or a forest increases.

Figure 4.8 shows how trees expressed in terms of two different primitive sets map to different parts of the function space. Figure 4.9 shows how forests (i.e. representations with the ability to express modules) expressed in terms of two different primitive sets map to the same part of the space of functions. The left and right rectangles represent the space of all trees (in figure 4.8) and forests (in figure 4.9) without a size limit, expressed in terms of P_1 and P_2 respectively. The central rectangle represents the space of all functions that can be represented. Ellipses in figure 4.8 on the left and right represent trees up to a fixed size. These map to two *different* subsets of functions. Note that the small overlap signifies that some functions can be represented by trees of a fixed size in either primitive set. Ellipses in figure 4.9 on the left and right represent forests up to a fixed size. These map to *the same* subsets of functions which is illustrated as two ellipses which largely overlap.

In summary, given two equally expressive primitive sets with a representation which permits modularity, if the size of the forests is limited then the two search spaces map to the same space of functions. If modularity is not permitted, the two search spaces will map to different spaces of functions.

If we have some probability distribution over the space of programs represented as trees, this will correspond to a probability distribution over the space of functions, which will depend on the primitive set used. If we have some probability distribution over the space of programs represented as forests (i.e. modular representations), this will correspond to a probability distribution over the space of functions, which will *not* depend on the primitive set used.

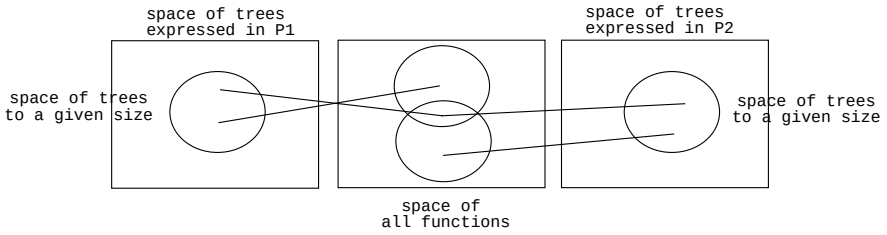

Figure 4.8: In the tree representation, modularity is not permitted and the two spaces of trees of limited size map to two different spaces of functions (which overlap slightly).

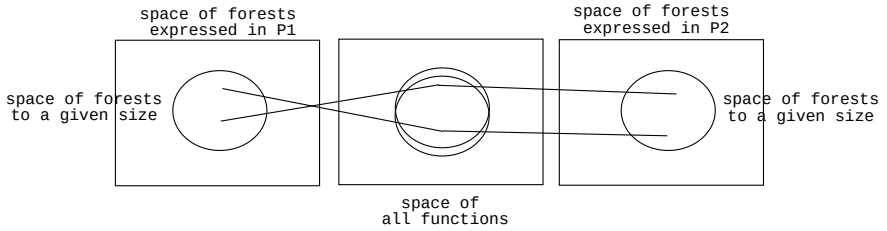

Figure 4.9: In the forest representation, modularity is permitted and the two spaces of forests of a limited size map to almost the same space of functions. The smaller the complexity of the dictionaries involved the more the two spaces of functions overlap.

Tree based landscapes vs. modular landscapes

The section above (section 4.4.4) states that what is included in a search space does depend on the primitive set if modularity is not permitted and does not depend on the primitive set if modularity is permitted. How does the representation affect evolution? We believe that evolution using a modular representation is more difficult than using a tree based representation. GP makes the implicit assumption that fitter individuals will produce fitter offspring. If a subtree is altered (either by crossover or mutation) this will in general lead to an offspring of different fitness, and this dependence will be greater if a modular representation is used, as explained in the following paragraph.

If we are using a modular representation, the genetic operator may affect a subtree contained within a module which is used multiple times (the effect of the operation is therefore potentially magnified, see section 3.8.1 on Koza's lens effect). Imagine the modular representation being 'expanded out' into a single equivalent tree (i.e. each module call being represented explicitly by the contents of the module). The genetic operator acting on modular representation would be equivalent to an operator acting multiple times on a tree based representation. Thus a modular representation will experience larger changes in fitness under the application of a genetic operator, when compared to a tree based representation. It is also the case that we may alter a module contained within the representation which is not used (i.e. a module that is not called by the rest of the program), thus

we also have neutrality in the representation, where the application of a genetic operator makes no difference to the fitness of the individual. Thus, while a modular representation can compress data further than a tree based representation (as it can express regularities about itself), this ability makes the search space more rugged. The situation is even worse if we have a Turing Complete instruction set as we also have iteration to deal with, (see section 3.8.6 for further comments).

4.5 Summary

We began this chapter by stating that modules are useful for representation as they can express reuse. This is in contrast to the standard tree based representation used in GP, where a node in a tree is only ever executed once (on a given test case). There is no reuse permitted in a tree based representation. Modular representations however do permit reuse.

We gave a number of definitions including representational complexity which was defined as the minimum number of nodes needed to express a given function. We then went on to prove that if the representation can support modules, then the complexity of a function is independent of the primitive set (up to a constant) when this type of representation is used. This is a main result of this chapter. A simple rearrangement of this theorem shows that there is also a lower bound on the complexity and we went on to show that this is the tightest bound obtainable.

Chapter 5

Evolving Turing Complete Representations

5.1 Introduction

Typically, as the size of a problem grows, the size of a solution also increases, and therefore so does the number of evaluations required to find a solution. Standard GP has no way of addressing variable sized problems because the terminal set consists of the variables of the problem. If we were to tackle the problem in a more general setting, we would need some way of addressing variables generally. This demands that we change our representation to one that can accept a variable number of problem variables. This can be achieved by Turing Equivalent representations, but also by less expressive representations, e.g. finite state automata. We argue that if the representation complexity of a solution does not increase as the size of the problem increases, then the number of evaluations needed on average to find a solution will remain constant. Thus, even if we are not interested in solving the general case of a problem, by casting a fixed size problem as a general problem we may be able to solve it easier than if we had just considered the large fixed size problem. The trade off point will depend on the problem.

While a Turing Complete representation is the most expressive representation we have, it also raises the halting problem; how do we evaluate programs which may not halt. In some work, a straight forward error based fitness function is used to guide evolution. In this chapter we pose the question; can information about the termination of a program provide useful feedback for the fitness function? We design three fitness functions, an error based function, and a second pair of functions based on the first except that they incorporate information about the termination status of a program.

A second approach is to look at the operators used to create new programs. Different operators may affect the number of programs which are generated which terminate properly. We propose an operator which is more likely to generate programs which will terminate properly. This operator was first proposed in [88]. In a number of experiments we vary the ratio of crossover to mutation and record the number of programs which had to be terminated as they had reached the preset limit.

5.1.1 Contributions of this chapter

Below we list the contributions of this chapter and the sections in which they appear.

1. demonstrate that as problem size increases, the difficulty of the problem remains constant if a suitable representation is used, and thus the average number of evaluations needed to find a solution remains constant.

2. demonstrate that information about the proper termination of a program can be useful in fitness evaluation.

3. demonstrate that the choice of genetic operator can affect the probability of producing programs which terminate.

5.1.2 Outline of this chapter

In section 5.2 we solve the even parity problem for different fixed sizes. In section 5.3 we incorporate information about the termination of a program into the fitness function. In section 5.4 we compare different operators and record the number of non halting programs in a run.

5.2 Scaling of problem size and representational complexity

5.2.1 Scalability of problem size

A number of problems of different size have been studied in the GP literature, for example the even parity problem and the lawn mower problem [36, 37]. Typically as the size of the problem grows, the size of the solution and the time taken to find the solution grows. Another example of a scalable problem includes image recognition problems. For example, we could consider the problem of identifying a certain image on a certain fixed sized image array (e.g. 100 by 100 pixels), but this could be generalised to the problem of identifying the certain image on an n by n array (or more generally to an n by m array). There are also problems which we think of as fixed size problems but can actually be thought of as variable size problems. For example the game of Othello is typically played on an 8 by 8 board but could be played on other boards of 2n by 2n.

Standard GP has difficulties with problems of different size in terms of representation. Teller [71] states that the language used by standard GP is not powerful enough to express many algorithms. In particular, *'There is no mechanism for variable length strings to be shown to the function and no way for the function to iterate an arbitrary number of times'*. A more expressive representation is needed if we are to tackle generalisations of variable size problems. There may be other benefits too by looking at small instances of problems before looking at larger instances (i.e. the small problems may contain enough information to generalise from, without considering larger instances of the problem).

In terms of the framework GP uses, two examples of problems of different sizes are considered as separate problems i.e. we evolve a solution to the even-3-parity problem *or* the even-4-parity problem. This is interesting (or perhaps worrying or disturbing) as most people would consider these to be *instances* of the same problem as the nature of the underlying problem (i.e. is the number of true bits even?) is the same in both cases.

It is constructive to consider the representational complexity of solutions to problems of different size and see how this scales with problem size. To do this lets consider solutions to the even parity problem. If we are using a look up table representation, as there is no compression the size of the table will scale exponentially with the problem. Each instance of input-output has to be represented explicitly in the table. If we used a tree based representation the complexity of a solution will scale linearly. If a modular (forest) representation is used, again the complexity of a solution will scale linearly. The rate a which the complexity climbs for the tree based representation will depend on the primitive set, however will be independent for the modular representation. Modular representations, like tree based representations, have no way to address the different variables of variable size problems. Hence, reference to the problem variables must be made explicit in the solution and therefore the complexity of the solution grows with the size of the problem. If a Turing Complete representation is used, the representational complexity of the solution is independent of the size of the even parity problem. While the rate at which the complexity climbs will vary with the problem, we can be sure that the complexity will never climb faster if we use a more expressive representation.

We do not need a Turing Complete representation to be able to address a variable size problem, other representations are capable of this. For our experiment we use a representation similar to an unlimited register machine. Other suitable representations include finite state automata.

5.2.2 Different methods of counting evaluations

We identify three different ways of counting the number of evaluations a search algorithm makes. Each of the methods is a more accurate way of counting, the last being the most accurate reflection

of wall clock time. Note that the number of program evaluations is always more than the number of test case evaluations which is always more than the number of instructions executed.

Program evaluations

Here we count the number of programs evaluated on a set of test cases. This is often the measure used in GP, which is fine for certain purposes (e.g. comparing two crossover operators on a fixed size problem with a fixed size test set). However this is a poor measure when we are considering variable size problems as, if $n = 3$ (where n is the arity of the boolean problem we are trying to solve), we have a complete set of 8 test cases, however if we have $n = 100$, we have a complete set of 2^{100} test cases. Evaluating a program on 10 test cases will take longer than evaluating it on say 9 test cases. This method of counting evaluations does not take into account the number of test cases, or the amount of time to evaluate each test case. In general evaluating a large tree in GP will take longer to evaluate on a single test case than it does to evaluate a small tree on a single test case.

Test case evaluations

Here we count each program evaluated on each test case. As the size of the problem increases the number of test cases potentially increases exponentially. This is the method of counting evaluations we will use for our comparison purposes. Using this as our evaluation measure, evaluating a program on 3 test cases of even-3-parity problem will be counted as equivalent to evaluating a program on 3 test cases of even-100-parity problem. Of course this does not take into account the fact that evaluating a single test case from an even-100-parity problem will take longer to evaluate than a single test case from an even-3-parity problem. The following measurement will be able to distinguish these two cases.

Instruction evaluations

Here a counter is incremented each time an instruction is executed. This is a good measure as it is the closest reflection of wall clock time. It is typically used in Turing Complete evolution to detect if a program if likely to be non halting (see section 2.11.5). However consider the following case. If we have two even parity problems, even-3-parity and even-100-parity, we may be able to evolve a solution to the even-3-parity before a single program has had enough time to scan the inputs of the even-100-parity problem. Thus this comparison, while it is the most accurate measure of time, is not fair; larger problems will take longer to evaluate.

Why not use wall clock time

We could use wall clock time directly for the comparison purposes of our algorithms however there are some difficulties with this. We may run into problems when comparing algorithms run on different machines. Even if we use the same type machine to compare two algorithms there may be difficulty in the way algorithms are implemented by two researchers (i.e. one may have implemented an efficient method of doing fitness proportional selection, while another may have implemented this inefficiently). If one is doing research into GP, there may also be lots of background processes going on recording information about the statistics of the run and writing information to files (this could be removed for raw comparison purposes). See section 3.4 for the assumptions the No Free Lunch theorems make about evaluations.

5.2.3 System architecture for variable length experiments

The following system was used to evolve general solutions to the even parity problem. A Turing Complete system could have been used but the system described here is expressive enough to solve the problem and support our hypothesis (see section 5.2.6). The system consists of a register, a register pointer, a program (which is a list of instructions), and a program counter (which points to the next instruction to be executed). The register consists of an array of slots, each of which can hold an integer value. For the purposes of this experiment we interpret 1 = true, -1 = false and 0 signals the end of the input sequence. The input data for the given problem is placed on the registers 1 on wards, the final register has a value 0. The zeroth register is initialised with a value of 1. Thus for the even-n-parity problem, the zeroth register has the value 1, the registers 1 through to n have values corresponding to the inputs for the problem, and the n+1 th register has the value 0 indicating that this is the end of the inputs. The output is taken from the zeroth register. Each time an instruction is executed the program counter is incremented by one. If the program does not terminate properly, it is terminated when the counter reaches a predefined limit.

5.2.4 Instruction set

For this experiment we use a very specific instruction set. None of the instructions take any arguments. `isEnd` terminates the program if the register being pointed to contains the value 0. This allows the program to be able to identify the end of the input sequence. `incRegPtr` increments the register pointer by one (which at the start points to register zero). It is this instruction which allows us to address inputs of variable size. `evenParity` is a special instruction which essentially solves the problem. If the value contained in the register being pointed at by the register pointer is positive, then the value in the zeroth register is flipped in sign (i.e the value is multiplied by minus one).

`gotoStart` returns control of the program to the first instruction i.e. the program counter is set to 0. This allows loops to be constructed.

A less specific instruction set could have been used, however this would just have made solutions more difficult to evolve. Other instructions could have been added to this instruction set, e.g. a `decRegPtr` could be introduced to decrement the register pointer, or a general `goto` instruction which takes one argument which is either a relative or absolute jump to another instruction.

5.2.5 Human produced solution

Here we have a human produced solution consisting of four instructions:

[`isEnd`,`incRegPtr`,`evenParity`,`gotoStart`]

This program first checks to see if we are at the end of the input sequence (using the `isEnd` instruction). If we are, then the program is terminated, else the second instruction is executed. The `incRegPtr` instruction increments the register pointer to point at the next register. `evenParity` flips the sign of register zero (i.e. multiplies the contents by -1). Finally, the last instruction, `gotoStart` unconditionally returns control to the first instruction in the program. A walk through this code will confirm to the reader that this is a general solution to the even-n-parity problem.

5.2.6 Experiments

Hypothesis

If we were tackling a boolean problem with 3 inputs (say the even-3-parity problem for example), we could refer to the problem variables explicitly as a, b and c or v_1, v_2 and v_3 and using GP, construct trees representing logical expressions (e.g. a and b). If we had another problem (e.g. even-4-parity problem), which is from the same class of problem (even-n-parity problem), we could again address the variables *explicitly* as a, b, c and d or v_1, v_2, v_3 and v_4.

This method of referring to variables suffers from the difficulty that we cannot refer to the *ith* problem variable (e.g. v_i), so we can never refer to 'the last variable' for example. As each problem variable has to be referred to explicitly in any solution, in general the complexity of a solution will increase. If however the representation has some mechanism of addressing a general instance of a problem variable, then it is possible to reduce the complexity of a solution as we could refer abstractly to problem variables rather than explicitly.

Hypothesis 5.1. *If the representation we are evolving can address the problem variables in a general way (as described above), then the number of evaluations (see section 5.2.2) needed to evolve a solution is independent of the problem size.*

Table 5.1: Parameter settings for variable size experiments.

Maximum number of evaluations	1000
Program length	4
Number of test cases	8
Problem	even-n-parity
Number of programs randomly generated	100
Number of trials averaged over	200

Table 5.2: Results for for variable size experiments.

problem size	attempts	test cases evaluated	instructions evaluated
16	62.755	28888	123078383
25	61.345	29172	120421888
36	62.79	28871	123502202
49	61.81	28808	122051612
64	62.165	29087	122394115
81	61.98	29038	122516925
100	62.98	28680	124702453
121	61.475	29172	122137471
144	62.22	28779	124026370
169	62.445	28870	124906025

Parameter settings

The following parameters were used. In the case of this toy problem, we know the length of program in which we can find a solution (see section 5.2.5). In general we will not know this and a more sophisticated method of varying the length of the program is required. In this experiment the length of the program is set to 4. A set of 8 test cases is generated for each problem size. An upper limit has to be set regarding the number of instructions which can be executed before a program is killed (1000). Again as this is a toy problem we can set this limit to a value which we know our hand coded solution terminates within. Rather than using an evolutionary approach to solve this problem, we simply use random search. This is similar to Koza's approach (see section 3.8.1) where he investigated the nature of the representation rather than the characteristics of a particular search operator.

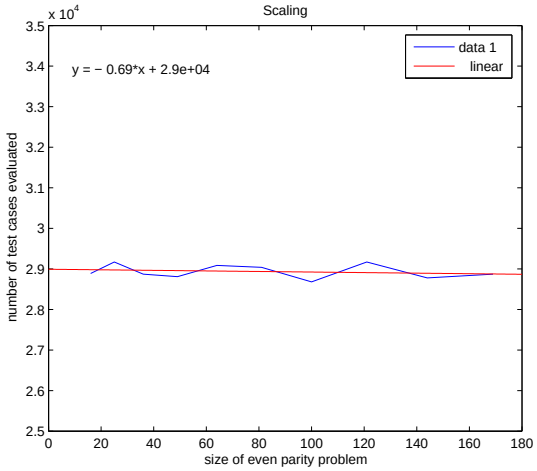

Figure 5.1: Scalability. The horizontal axis shows the size of the even parity problem being tackled. The vertical axis shows the average number of test cases which were evaluated in order to arrive at a solution. As the problem size increases, the number of evaluations is constant (an evaluation is counted as a program evaluated on a single test case).

5.2.7 Results

The results are presented in table 5.2. The first column is the size of the boolean problem. The second column is the average number of attempts needed to produce a solution (averaged over 200 trials). The third column is the number of test cases evaluated on average to find a solution. The fourth column is the average number of instructions evaluated.

Randomly generated solutions

Here we list all the different types of program found by random search:

[`incRegPtr`,`evenParity`,`isEnd`,`gotoStart`]

[`isEnd`,`incRegPtr`,`evenParity`,`gotoStart`]

[`incRegPtr`,`isEnd`,`evenParity`,`gotoStart`]

5.2.8 Discussion of variable size problems

Miller et. al. [31] evolve digital designs of arithmetic functions (addition and multiplication) and argue that *'by studying the evolved design of gradually increasing scale, one might be able to discern new, efficient, and generalisable principles of design'*. They pose the central question in the paper *'Can we by evolving a series of subsystems of increasing size, extract the general principle and hence*

discover new principles?'. While we agree with this approach however, it may be possible to evolve a general algorithm rather than having a human study the evolved designs.

Brave [8] evolves solutions to a planning problem using ADFs with a restricted form of recursion. He shows that the effort required to find solutions with ADFs and recursion remains constant with the problem size. Basic GP scales exponentially and ADFs scale linearly at best, which is consistent with Koza's [37] results for basic GP and GP+ADFs on the lawn mower problem. Our results here are consistent with Brave's [8], if the complexity of the representation does not increase with problem size, then the effort required to find a solution remains constant.

The system presented in this section is capable of evolving solutions to the general even parity problem. Other representation which could achieve this are described in [65, 93]. Another representation which could do this is a finite state automata or any Turing Equivalent representation.

In this section we have evolved a general solution to the even parity problem. In some circumstances we may not need or be interested in a general solution to a problem, we may simply want a solution to a fixed size instance of the problem. For example we may be interested in the solution of the lawn mower problem for a lawn of size 10000 by 10000, or the even-10^6-parity problem. In these cases it would be easier to evolve a general solution to the problems, rather than evolving specific solution. Also tackling smaller instances of the problem make sense i.e. it is the same underlying problem and we can learn the solution to underlying problem before tackling larger instances. Yu [93] (page 361, section 6.2) includes in her test set all cases of even-2-parity and even-3-parity problems (i.e. $2^2 + 2^3 = 12$ test cases). Other example of scalable problems are image recognition problems, where we may be able to evolve solutions to general problems by looking as smaller instances of the problem. Also many games may fall into the category of scalable problems e.g. Othello and Draughts.

5.2.9 Conclusion

In some circumstances we may be interested in general solutions to problems and it makes sense to use a representation which can express general solutions. Not only will solutions be general, but above some certain problem size, it takes less evaluations to evolve a general solution than a solution to a specific sized instance of the problem. For example Yu [93] used only test cases from problems of size 2 and 3 to evolve general solutions. Hence, if we are searching for the solution to some large sized problem, it may be beneficial to examine the general problem.

5.3 Fitness based on halting

When evolving Turing Complete representations we are faced with the problem of what to do with non halting programs, the simplest solution is to abort their execution after some upper limit is reached (see section 2.11.1). This section investigates if information about the halting or non halting of a program can be used in the fitness function.

Some researchers just use a simple error based score in order to assign fitness to an individual program. Other researcher have taken into account in their fitness function whether the program halts itself or not. While we believe this is superficially a good thing to do (i.e. we want our programs to halt naturally), this does not imply that non halting programs are less likely to produce good solution overall. This issue is worth examining further. In this section we investigate if this is a good practise or not. We try 3 different types of fitness function.

We start with a basic error based fitness function (sum squared errors). We call this fitness function 1. From this we create a second fitness function, which returns a maximum error score if the program fails to halt within the time limit on any of the test cases. We call this fitness function 2. This fitness function is trivially extended to form a 3rd fitness function. If the program fails to halt on any of the test cases, then the remaining test cases are not evaluated (as we know the score for the program will be a maximum). These variations on the basic fitness function are listed below:

1. fitness function 1: is an error based (sum squared errors) which does not take into account if the program halted itself or not.

2. fitness function 2: examines the performance of a program on all test cases, and if it fails to halt on any of the test cases, a maximum fitness score is assigned, otherwise it returns the sum squared error.

3. fitness function 3: takes a program and tests it on each test case sequentially. If the program fails to halt on any of the test cases, a maximum error score is assigned. The remaining test cases are not examined (as there is no point). If a program halts on all test cases then it returns the sum squared error.

Fitness functions 1 and 2 are as expensive as each other to evaluate. Fitness functions 2 and 3 are functionally equivalent as they will always return the same value. They give us the same information about the program being tested. However fitness function 2 evaluates a program on all of the test cases regardless of whether a halting program is encountered or not. Fitness function 3 is slightly more sophisticated in that it recognises the fact that when a non halting program is encountered there is nothing to be gained by testing further.

There may be a theoretical proof of why one type of fitness function is better than another however we have not attempted it. A theoretical proof may also give us insight into how much time can be saved and if this if a general result that can be applied in all cases.

5.3.1 Experimental set up

The representation we use is similar to the unlimited register machines used by Huelsbergen [25] but is based on the simpler machine (in the sense that it has less instructions) presented in Cutland [13]. The problem is to evolve a program which adds two non negative integers together. For this problem we use a register machine with 3 registers. The inputs to the problem are placed on registers 0 and 1 and register 2 is initialised to 0. Note that this is similar to Huelsbergen's [25] experiment to evolve multiplication, but he includes an `add` instruction in his instructions set.

The instructions are `inc`, `jmp`, `cpy` and `clr`. The instruction `inc` takes one argument, and increments the value contained in that register. The instruction `jmp` takes three arguments, if the contents of the registers indicated by the first two arguments are equal, the instruction pointer is altered to the value of the third argument and the control of the program 'jumps' to that instruction, otherwise the next instruction in the program is executed. The instruction `cpy` takes two arguments, and copies the contents of the register indicated by the first value to the to the register indicated by the second register. The instruction `clr` takes one argument, and sets the value in the register indicated by that argument to zero. Note that `cpy` is in fact a redundant instruction is the sense that it can be synthesised from the other instructions (see section).

Selection works as follows. A pair of programs are selected at random from the population, and a mutated copy of the better one replaces the worse one. The mutation operator replaces a instruction and its arguments with a new randomly generated instruction and arguments. Arguments that refer to registers are generated uniformly over the number of registers and arguments referring to jumps are generated uniformly over the length of the program. There is also a 1 percent chance that the mutation mutates all instructions in the program, to prevent the evolution getting stuck in any local optima in the landscape. No crossover is used.

5.3.2 Parameter settings

We use a population size of 1000 and fix the program size to four. A single program can execute a maximum of 1000 instructions before it is terminated externally. We allow a maximum number of 10^9 evaluations in a single run (an evaluation is counted as the execution of an instruction). We have ten test cases which are all the pairs of non negative integers whose sum is less that four. A run is conducted 5 times. These parameter settings are listed in table 5.3.

Table 5.3: Parameter settings for experiments with three different fitness functions based on halting information.

number in population	1000
program size	4
maximum number of evaluations on a run	10^9
maximum number of steps on a single test case	1000
number of test cases	10

Table 5.4: The average number of evaluations to produce a solution, for 3 different fitness functions based on halting information.

fitness function 1	1.37753e+07	1.79974e+07	1.25521e+07	1.64369e+07	1.65236e+07
fitness function 2	2.46646e+06	2.77976e+06	4.34217e+06	1.45543e+06	9.89866e+06
fitness function 3	1.44789e+06	1.66443e+06	1.90664e+06	1.64322e+06	1.68866e+06

Hypothesis 5.2. *By using a suitable fitness function based on the termination status of programs, we can affect the success of a run.*

5.3.3 Results

The results are presented in table 5.4. Each row related to fitness function 1, 2 and 3. With fitness function 1 we require approximately 10^7 instructions to be evaluated to find a solution. With fitness function 3 we require approximately 10^6 instructions to be evaluated to find a solution. We set the maximum number of evaluations on a single run to 10^9, but all runs found solutions before this limit was reached. We use the Mann Whitney (Rank Sum) test to test the significance of this result. The two sets of data show clear separation corresponding to a rank sum of 15. We can be confident at the 0.005 level that these two distributions are different.

One of the reasons for this improvement in efficiency maybe that not all test cases need to be examined (as soon as the first non-terminating program is encountered, testing on further test cases is discontinued). However we can examine the results for fitness function 2. If we compare the results for fitness functions 2 and 3 we can be sure at the 0.10 level of confidence that they are from different distributions. From this we can conclude that there is a benefit from simple assigning individuals which do not halt a high fitness value (fitness functions 2) and that the improvement seen with fitness function 3 is not purely due to the fact that we can discontinue testing when we

encounter a non halting example of a program.

5.3.4 Conclusion

For this problem (evolving addition) with a basic error based fitness function, taking into account if a program halts in the fitness function is beneficial and produces about a 10 times speed up. It is certainly worth investigating other problems and other fitness functions (e.g. sum of magnitudes of errors rather than the sum of squares).

It would be interesting to see if this method could be combined with other methods of early termination of testing [20, 21, 22, 66, 74, 94]. The test cases could also be ordered as some may be more likely to produce halting programs. In general the time saved by this method will depend on the problem, the number of test cases, and how high the termination limit is set.

Note that fitness functions 2 and 3 are the reverse of that used by Huelsbergen [27] who used a two tier fitness function where two programs are first compared on their correctness (i.e. error) and, if there errors are the same, compared using the number of test cases on which they halted.

5.4 Conventional crossover vs modular crossover

5.4.1 Introduction

When evolving Turing Complete representations we have to deal with programs which do not halt. The simplest and most often used method is to impose an upper limit on the run time of a program and if it has not halted in this time then its execution is aborted. Much time can be wasted waiting for programs which ultimately do not halt. It may be beneficial to have a search operator which reduces the probability of producing programs which do not halt, therefore saving time. In [88] we proposed a representation and crossover operator based on results from recursive function theory. Teller [72] has also done work with a self adaptive operator which can reduce the number of programs which have to be aborted during a run.

In this section we make observations about the number of non halting programs encountered, and the number of programs processed during a run. If a program does not halt within a time limit (set to 1000 in these experiments), it is considered to be a non halting program. A run is terminated after 10^9 instructions have been evaluated, and thus on different runs a different number of programs may be processed.

5.4.2 Experimental set up

We look at two different types of program representation. One type of representation used is a single program. This is the same type of representation used in section 5.3.1.

The second type of representation is a program of 2 explicitly defined modules (each modules consisting of 8 instructions, the overall program therefore consisting of 16 instructions). Each module can only read and write to registers local to that module. The first module consists of 8 instructions which can only read and write to a local register and is composed of instructions `inc`, `jmp`, `cpy` and `clr` (describe in section 5.3.1). The second module consists of 8 instructions which can read and write to a local register (the problem inputs are placed on this register, positions 1 and 2 and the output is taken from the register at position 0). The instructions which are in the second module can be one of the primitive instructions (`inc`, `jmp`, `cpy` or `clr`) or a call to the first module.

A module to has three arguments, The first argument indicates which register the output of the module will be written. The second and third arguments which registers are used as input for the module.

Genetic operators

New programs are generated using two different types of operator, crossover and mutation, which are selected probabilistically according to the parameter 'mutation percentage'. If the mutation percentage is 0 then crossover is always performed, if it is 100 then mutation is always performed. Mutation takes a single instruction in a program and randomly replaces it with a newly created instruction and arguments. There is also a 1 percent chance that the mutation operator will take the whole module and replace it with an entirely new module (in the case of the first type of representation all instructions of the program are replaced, in the case of the second type of representation all instructions of one of the modules are replaced).

There are a number of different crossover operators used. These act differently according to the representation which they are applied to. Module crossover can only be applied to programs consisting of more than one module. It exchanges complete modules preserving location. One point crossover selects a point somewhere along the length of instructions and exchanges all the instructions from that point on. Two point crossover exchanges all instructions between two randomly selected points. When these crossover operators are applied to programs consisting of more than one module, crossover occurs between like-modules (i.e modules in the same position).

Selection

Selection works as follows. If we are using the mutation operator, a pair of programs in the population are selected and a mutation of the better program replaces the poorer program. If crossover is used,

four programs are selected from the population. Clones of the better pair are crossed over and their children replace the worst pair of the four.

Hypothesis

Hypothesis 5.3. *By using a suitable genetic operator reduce the number of non halting programs and therefore process more programs during a run.*

5.4.3 Parameter settings

The population size is 1000. After 10^9 instructions have been evaluated the run is terminated. If a program does not terminate after 1000 instructions have been executed the program is terminated (and therefore receives the maximum fitness). The problem is to learn to multiply two non negative integers together (which are placed on the first and second registers). The test set consists of the products of all pairs of integers, which sum to less than 6. The results are averaged over 10 runs. Different types of fitness function were used depending on the experiment.

5.4.4 Results

The results are presented in tables 5.6, 5.7, 5.8, 5.9, 5.10 The columns of each table are as follows: The first column is the mutation percentage. The second column is the number of programs which had to be terminated during a run. The third column is the number of program processed during a run. The last column is the fitness of the best program in the population. All of these results are averaged over 10 runs. The results are divided into two sections.

Affect of crossover

These three of runs all consist of programs with the same representation, and use the same fitness function. These runs differ in the type of crossover operator which are used. Table 5.6 contains the results for one point crossover applied to programs consisting of 2 modules each of 8 instructions. Table 5.7, contains the results for modular crossover (crossover which exchanges complete modules) applied to programs consisting of 2 modules each of 8 instructions. Table 5.8. contains the results for two point crossover applied to programs consisting of 2 modules each of 8 instructions. The fitness function for these sets of runs examine each test case in turn, but testing is discontinued if a program fails to halt on any test case and maximum fitness is returned, otherwise sum squared error is returned.

Table 5.5: Parameter settings for crossover experiments.

number in population	1000
maximum number of evaluations on a run	10^9
maximum number of steps on a single test case	1000
number of test cases	21
problem	multiplication
number of runs	10

Conventional two point crossover vs modular crossover

These two sets of runs involve different program representations and different fitness functions. Table 5.9, contains the results for two point crossover applied to programs consisting of 1 module each of 16 instructions. The fitness function used here is the sum squared error and does not give any information about the termination of the program. Table 5.10 contains the results for modular crossover applied to programs consisting of 2 modules each of 8 instructions. The fitness function examines all test cases, but returns maximum fitness if the program fails to halt on any test case.

5.4.5 Discussion

Affect of crossover

In the first three sets of runs (tables 5.6, 5.7, 5.8) we use a fitness function which examines each test case in turn, and if the program does not terminate properly on a test case, then testing is discontinued and a maximum fitness is returned, other wise the sum square error is returned (this is the same as fitness function 3 in section 5.3 except the error based part is multiplication) In these three sets of runs the program representation consists of two modules, each module consists of 8 instructions which can only write to memory local to that module. In the first set of runs (tables 5.6, one point crossover exchanges all instructions in a module from a randomly selected point onward. In the second set of runs (table 5.7), modular crossover exchanges complete modules, and does not separate instructions. In the third set of runs (table 5.8),), two point crossover exchanges a subsection of instructions between two randomly selected points in a module.

We can see patterns in the graphs obtained from these results. In all cases, as we apply mutation with a higher probability than whichever crossover operator is being used, we see an increase in the number of non halting programs encountered during a run. Similarly, we see a fall in the number of programs which are processed during a run. Finally, we see that, as the probability of mutation over crossover increases, the fitness obtained is better. Although we can reduce the number of non

Table 5.6: Varying mutation percentages from 0 percent to 100 percent in steps of 10. Programs consist of two modules each of 8 instructions (i.e. a program is 16 instructions in total). One point crossover exchanges all instructions in a module from a random point on wards. The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness. These results are plotted in figure 5.2

mutation percentage	non halting programs	processed programs	fitness of best
0	726.3	8.22726e+06	22
10	40509.9	8.83972e+06	19.4
20	108067	7349061	17
30	178124	6.34243e+06	15.7
40	226848	6.10666e+06	15.3
50	233582	6.7139e+06	14.9
60	281936	5838019	16.3
70	339571	5.59879e+06	13.8
80	367045	5.0235e+06	11.2
90	357592	5.71471e+06	13.7
100	390508	5.50404e+06	14.5

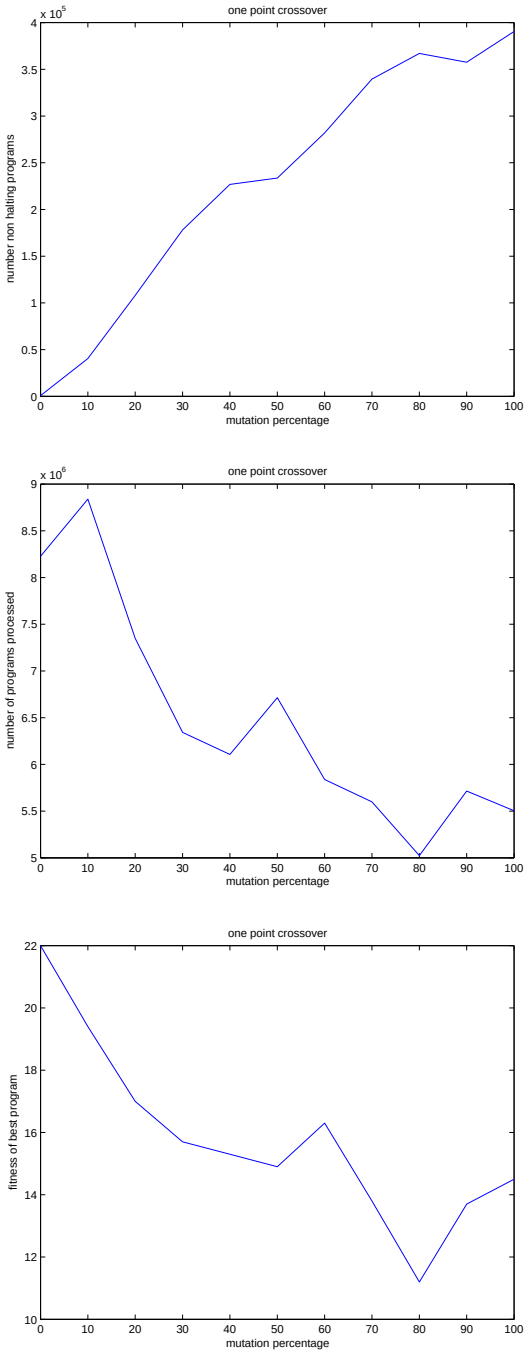

Figure 5.2: Number of non halting programs encountered, number of programs processed, and fitness of best program for different mutation percentages. Programs consist of two modules each of 8 instructions (i.e. a program is 16 instructions in total). The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness. The figures show column 1 plotted against columns 2, 3 and 4 from table 5.6 respectively.

Table 5.7: Varying mutation percentages from 0 percent to 100 percent in steps of 10 for modular point crossover. Programs consist of two modules each of 8 instructions (i.e. a program is 16 instructions in total). Modular crossover exchanges complete modules, not individual instructions The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness. These results are plotted in figure 5.3

mutation percentage	non halting programs	processed programs	fitness of best
0	678.2	8.2796e+06	43.4
10	50593.4	7.8664e+06	16.7
20	114987	5.90123e+06	16.4
30	144592	5938816	16.3
40	201125	6.81597e+06	14.3
50	262347	6.64969e+06	15.4
60	268458	6.28254e+06	16.6
70	287990	6.00599e+06	14.2
80	315915	6.46472e+06	15.5
90	417240	5272836	13
100	426788	5.22518e+06	14.4

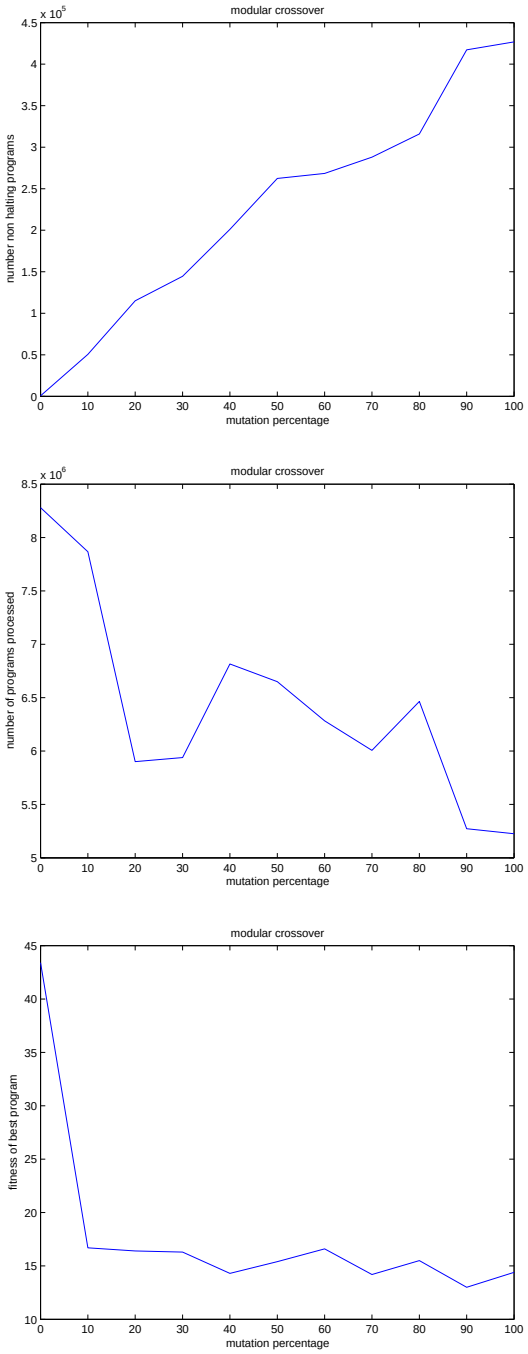

Figure 5.3: Number of non halting programs encountered, Number of programs processed, Fitness of best program for different mutation percentages. Programs consist of two modules each of 8 instructions (i.e. a program is 16 instructions in total). The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness. The figures show column 1 plotted against columns 2, 3 and 4 from table 5.7 respectively.

Table 5.8: Varying mutation percentages from 0 percent to 100 percent in steps of 10 for two point crossover. The program representation is 2 modules of 8 instructions each (i.e a program is 16 instructions in total). Two point crossover takes a subsection of a module and exchanges it with a subsection from the corresponding module of the other program. The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness (i.e. infinity). These results are plotted in figure 5.4

mutation percentage	non halting programs	processed programs	fitness of best
0	833.1	7.75579e+06	26.6
10	52450.5	7.51847e+06	19.6
20	86850.4	8.46966e+06	16.5
30	127840	7.98317e+06	17.6
40	192924	6.49463e+06	15.1
50	255182	6.8089e+06	14.9
60	253842	6.57813e+06	16.5
70	288815	6.4474e+06	15.5
80	376760	5.40409e+06	11.1
90	416109	4.70379e+06	12.8
100	352402	5790598	15.3

halting programs and therefore increase the number of programs we can process during a run (which on the face of it is a positive result), this has a detrimental affect on the fitness.

Table 5.9: Varying mutation percentages from 0 percent to 100 percent in steps of 10 for two point crossover. The program representation is one module of 16 instructions. Two point crossover takes a subsection of a module and exchanges it with a subsection from the corresponding module of the other program. The fitness function examines all test cases and returns the sum squared error and does not take into account if the program had to be terminated. These results are plotted in figure 5.5

mutation percentage	non halting programs	processed programs	fitness of best
0	78553.6	2.51586e+06	9.5
10	167665	388842	4.7
20	144113	215095	5.4
30	135032	277771	4.9
40	139626	202197	5.1
50	111200	151175	4.3
60	138530	243584	3.6
70	133239	196266	3.2
80	118211	167436	3.9
90	114343	153804	3.5
100	99466.1	141215	4.3

Conventional two point crossover vs modular crossover

In table 5.9 and figure 5.5 the program representation is one module of 16 instructions. The fitness function is different from those used in previous experiments (section5.4.5), and examines all the test cases returning the sum squared error (it does not give any information about the termination of the program). Thus in table 5.9 and figure 5.5, we have what could be described as a conventional approach; a single program has two point crossover applied anywhere to it and halting is not taken into account in the fitness function.

In table 5.10 and figure 5.10 the program representation is 2 modules of 8 instructions each (i.e a program is 16 instructions in total). The fitness function examines all test cases but returns the maximum fitness if the program fails to halt on any of the test cases, otherwise it returns the sum squared error. In table 5.10 and figure 5.10 programs consist of two modules and crossover only acts on modules. Due to the fitness function, programs which do not halt properly on all test cases are less likely to breed, and therefore non-halting modules are less likely to be inherited by future generations.

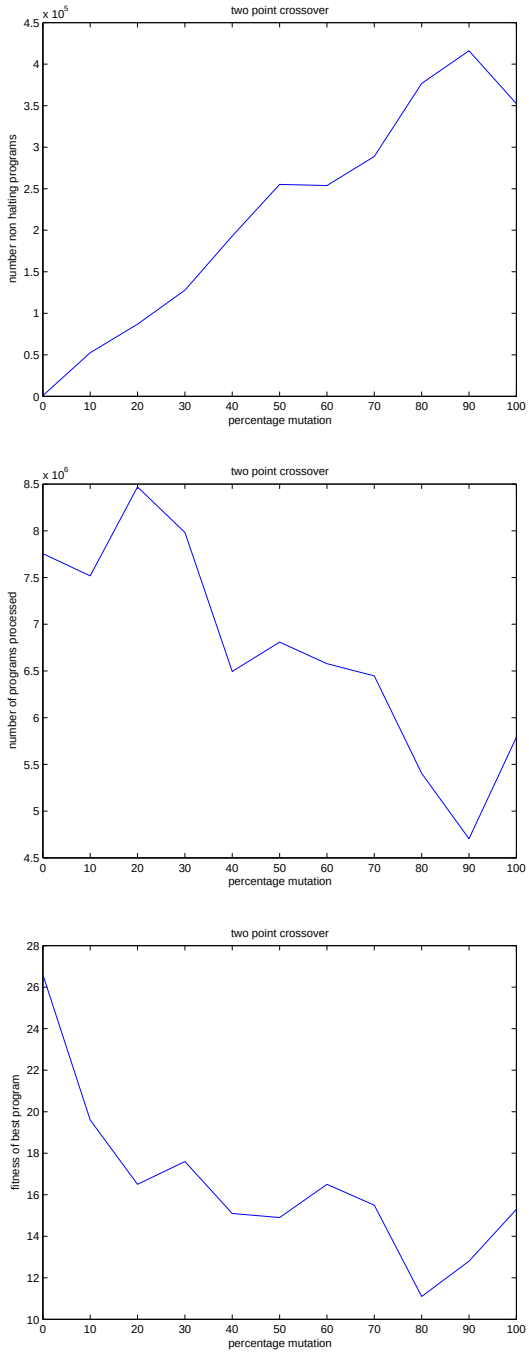

Figure 5.4: Number of non halting programs encountered, number of programs processed, fitness of best program for different mutation percentages. The program representation is 2 module of 8 instructions each (i.e a program is 16 instructions in total). The fitness function returns sum squared error, but if the program has to be terminated, testing is discontinued and a maximum value is assigned to the fitness. The figures show column 1 plotted against columns 2, 3 and 4 from table 5.8 respectively.

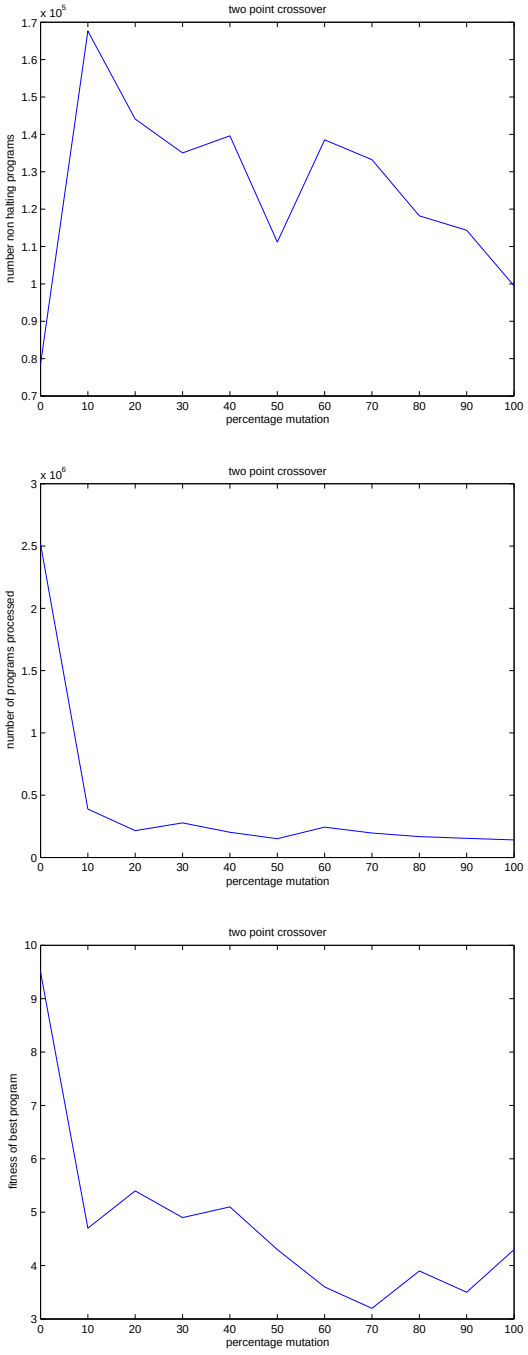

Figure 5.5: Number of non halting programs encountered, number of programs processed, fitness of best program for different mutation percentages. The program representation is 1 module of 16 instructions. The fitness function examines all test cases and returns the sum squared error and does not take into account if the program had to be terminated. Two point crossover is used The figures show column 1 plotted against columns 2, 3 and 4 from table 5.9 respectively.

Table 5.10: Varying mutation percentages from 0 percent to 100 percent in steps of 10 for two point crossover. The program representation is 2 modules of 8 instructions each (i.e a program is 16 instructions in total). Two point crossover takes a subsection of a module and exchanges it with a subsection from the corresponding module of the other program. The fitness function examines all test cases but returns the maximum fitness if the program fails to halt on any of the test cases, otherwise it returns the sum squared error. These results are plotted in figure 5.6

mutation percentage	non halting programs	processed programs	fitness of best
0	566.5	7.2479e+06	37.6
10	44894	4.97036e+06	17.6
20	65653.1	4.31454e+06	17.2
30	71995.9	4.04713e+06	18.6
40	75378.8	3.22733e+06	18.1
50	91897.6	2509778	14.8
60	95248.7	2.558e+06	18
70	98573.9	2.06328e+06	17.7
80	91737	2243598	18.2
90	103505	1586363	14.1
100	96234.7	1.45545e+06	16

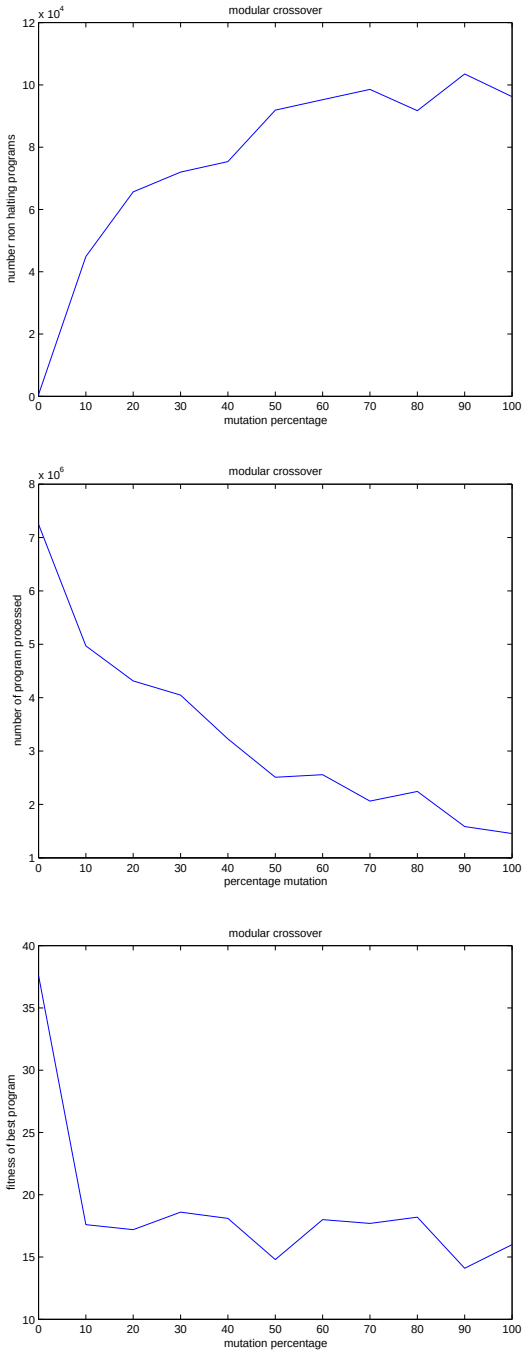

Figure 5.6: Number of non halting programs encountered, number of programs processed, fitness of best program for different mutation percentages. The program representation is 2 module of 8 instructions each (i.e a program is 16 instructions in total). Crossover exchanges complete modules. The fitness function examines all test cases but returns the maximum fitness if the program fails to halt on any of the test cases, otherwise it returns the sum squared error. The figures show column 1 plotted against columns 2, 3 and 4 from table 5.10 respectively.

For the conventional situation, (table 5.9 and figure 5.5) the number of non halting program gradually falls (excepts for a very low initial start) but remains high. The number of programs process is essentially constant with mutation percentage (excepts for a very high initial start).

For the situation where crossover only acts on modules and non halting programs are less likely to breed, (table 5.10 and figure 5.10) and the number of programs processed falls as mutation percentage increases. However, although the number program processed by the second approach is higher than the number processed by the conventional approach, the fitness of the best program is not as good.

5.4.6 Conclusion

In this section, we have shown that we can, through the choice of suitable genetic operators, fitness function and representation, substantially reduce the number of non-halting programs encountered during a run. This therefore leads to more programs being processed during a run. This does not however lead to a better fitness of the best program at the end of the run

5.5 Conclusion

The first experiment in this chapter (section 5.2) showed that we can evolve solutions to large scale instances of a problem, by looking at the general case of the problem, more efficiently that just looking at the large scale instance. This was demonstrated on the even parity problem. We conclude that this method could be applied to a wide range of problems, where we can scale along dimensions other than size. Other example are image recognition problems and maze navigation problems.

We went on to look at the issue of the halting problem. We firstly considered how information about halting could be used in the fitness function. We introduced a simple extension to an error based fitness function.

Finally. we showed that the choice of fitness function, representation and genetic operator can have an affect on the probability of generating programs which do not halt and this leads to more programs being inspected during a run. This superficially positive result does not lead to an increase in the fittest of the best individual in the population. However we still regard this as an interesting research avenue.

Chapter 6

Summary and Conclusions

6.1 Summary

We began this thesis by stating our interest in algorithm induction. This is a fundamental process in science. Our aim is to automate the inductive process and we identify GP as one potential approach. The type of representation GP uses has an affect on its performance. While we are interested in using GP in conjunction with a Turing Complete representation, an interesting stepping stone would be an understanding of the evolution of modular representations (i.e. those representations which are not Turing Complete but can represent modules). This is important as the ability to express modules is an important feature of Turing Complete representations, and is also the reason why complexity is invariant under different primitive sets. Thus, an understanding of the induction of functions using modular representations would be a desirable milestone to reach on the path to a better understanding of function induction using Turing Complete representations. Other approaches (i.e. firstly studying the evolution of finite state automata, then push down automata) were also outlined.

In chapter 2 we reviewed the induction of functions using the evolution of Turing Complete representations. Many different types of representation have been evolved including the state transition tables of Turing Machines, unlimited register machines, especially designed languages and 'accidental' models of computation. In general, we argued that the evolution of Turing Equivalent representations is more difficult than the evolution of less expressive representations. One explanation is that the landscapes produced by Turing Complete representations are more rugged than the landscapes produced by less expressive representations. It is due to the ability of of Turing Equivalent representations to compress data more than less expressive representations which make Turing Complete representations more likely to generalise. There is also the problem of dealing non-halting programs, and how to sensibly assign a fitness value to them. The typical solution is

to predefine a fixed maximum run time for a program, but this has the problem of what exactly to set this bound at, which is highly problem dependent. One of the motivations for evolving a Turing Complete representation is that solutions to general cases of a problem can be represented (this is different to the point above about generalisation and data compression).

A general, but not the only, approach to the problem of induction is the construction and exploration of a search space to find a representation of a function which is consistent with the observed data. Many Machine Learning paradigms can be thought of in this framework.

The NFL theorems are often taken to mean "all search algorithms perform equally", and in related papers it is proved "generalisation is a zero sum game" (i.e. generalisation is impossible). News of this set of theorems comes as a bit of a blow for someone interested in automatic induction and it is therefore worth examining the assumptions of these theorems rather than taking them at face value.

Occam's Razor, which is inconsistent with the COG theorems, has found itself in the Machine Learning literature, and has influenced many learning algorithms (e.g. in the form of pruning algorithms). While a number of authors argue that Occam's Razor is justified as short hypothesis are less likely to coincidentally fit the observations, we find this explanation unsatisfactory: there are also many longer hypothesis that also agree with the data. We can never be certain (by making this type of observation) that the rule we generate is actually responsible for the observations we make. We can only judge the correctness of a rule on a functional level. Therefore it does not make sense to "prefer the shortest", without a little more justification. We can make the conjecture that the shortest rule consistent with some given data, is also the most frequently represented, and we therefore restated Occam's Razor as "prefer the most frequently represented hypothesis". This sort of approach also allows us to think about the probability we are likely to generalise, we cannot do this if we think about Occam's Razor as "prefer the shortest". While the two statements of Occam's Razor are consistent, we find the frequency based statement more satisfying.

Most search algorithms do not meet the assumptions of the NFL theorems. Of particular importance is the consideration of all functions between two finite sets in GP. A genotype (instance of a tree representation) maps to a phenotype (function), however some functions are more frequently mapped to than other functions. Thus some functions are represented more frequently, assuming a uniform distribution over the space of genotypes (up to some fixed size). It is this mapping that breaks the assumptions of NFL and allows us to generalise (i.e. prefer some function over another because it is represented more frequently in the genotype space). This difference in relative frequency that a pair of functions are represented changes the more expressive the representation of the genotype is (i.e. the representation in which the induction is taking place). This is related to Koza's lens effect, the frequency with which functions are represented depends on which "represen-

tational spectacles" we are using to look at the space of functions. Thus as Turing Machines are the most expressive representations we have, and therefore the best data compressors, following this line of argument, they are also the best generalisers as they offer the greatest way of differentiating between the frequencies with which two functions are represented.

The situation presented to us by the NFL framework corresponds to the situation of function induction using look up tables as the underlying representation. Look up tables offer no compression of data and, as they are all the same size, we do not prefer one over another and all functions are represented equally likely.

We introduced the concept of modularity as the ability to reuse component parts of a representation during a single execution of the overall system. In GP terminology, a module can be constructed by composition from the primitive set (function and terminal set) and be used in the construction of further modules as if it were a primitive.

We defined representational complexity of a function to be the minimum size of an instance of a representation that can express the given function. In general, this will depend on the representation. Consider the tree based representation associated with standard GP and the extension to include modules (the most well known being Koza's Automatically Defined Functions). If a tree based representation is used, the complexity of a function will depend on the primitive set. If a modular representation is used, the complexity of a function is independent of the primitive set, and the complexity is symmetrically sandwiched between two constants which depend on the pair of primitive sets but not on the function being expressed.

In chapter 5 we conducted a number of experiments related to induction using Turing Complete representations. We investigated the scaling of problem size, the use of information about the halting of a program being evaluated in the fitness function and the effect of representation on the chance of producing a non halting program during a run.

As problems scale to larger and larger sizes (i.e. the number of variables increase), typically the complexity of the solution will increase and therefore so will the average time taken to find a solution. We argued that, if we use a representation which can address the general case of the problem (which may be Turing Complete but does not have to be) then the complexity will not increase with problem size and neither will the number of evaluations (i.e. programs evaluated on a test case) taken to find a solution. We illustrate this with the even parity problem and a representation capable of addressing a problem with a variable number of arguments. We show that as the size of the problem increases, the average number of evaluations needed to find a solution remains constant. We count a program executed on a single test case as an evaluation, not the wall clock time which will in general increase with problem size.

The undecidability of the halting problem is a central issue which needs to be addressed when

evolving Turing Complete representations. We take the simplest approach here by imposing an upper bound on the run time of a program, which is the approach most other researchers take. What we do investigate is the utility of incorporating information about the proper termination of a program in the fitness function. We evolve a function which adds two non negative integers together using an unlimited register machine. This is evolved using a select and replace scheme and a simple mutation operator. We compare 3 different fitness functions. Fitness function 1 returns the sum squared error regardless of whether the program had to be halted or not. Fitness function 2 tests the program on all test cases, but if the program fails to halt on any of the test cases, the maximum fitness is returned, otherwise the sum squared error is returned. Fitness function 3 is similar to the fitness function 2 except, as soon as we encounter the first test case where the program has to be aborted, testing is discontinued and the maximum fitness is returned, otherwise the sum squared error is returned. Fitness function 3 is functionally equivalent to fitness function 2, but is more efficient i.e. does not take more time to establish the same information. Fitness function 2 takes the same amount of time to execute compared to fitness function 1, but returns different information.

On this problem, with the settings described, runs with fitness function 3 performs better than runs with fitness function 1 by a factor of around 10. It may be argued that fitness function 3 is more efficient because it may not have to process all test cases. Thus we use fitness function 2, which is as expensive as fitness function 1, but functionally equivalent to fitness function 3 in order to determine the reason for this. We find that fitness function 2 still outperforms fitness function 1, and the success of fitness function 3 can therefore partly be attributed to its efficient evaluation, and partly that the information about the halting status of a program helps evolution on this problem.

Finally, we examined the effect of a number of different operators on the number of non terminating programs produced in a run. We use a program representation similar to an unlimited register machine, except it contains explicitly defined modules which have local memory. The fitness function used is fitness function 3, described above, and the problem is to multiply two non-negative integers together. We use 3 different crossover operators: two point crossover, one point crossover and modular crossover which exchanges complete modules. In these cases, as mutation is used more heavily over crossover, the number of non-halting programs encountered during a run increases and the number of programs processed during a run decreases. However, despite this, the fitness of the best individual in the population is worse.

We then compare a conventional two point crossover on programs which consist of a single module of 16 instructions, against a program representation which consists of two modules each of 8 instructions each (i.e. the same overall size). The crossover operator used on this representation operates only at module boundaries, and if a program does not halt, then modules from it are less likely to take part in reproduction (as they may not halt properly). This is a theoretically inspired

method of generating new programs and was originally described in [88]. We find this method does reduce the number of non halting programs, however, it does not lead to an improvement in fitness.

6.2 Conclusions

The ability to evolve Turing Complete representations is essential if we are to ultimately develop a theory of AI. While we do have some theoretical foundations, for example, the undecidability of the halting problem, we do not have a theory of how to evolve Turing Complete representations. We have no understanding as to why one representation may be more evolvable than another representation, if indeed this is the case. We also have little theoretical guidance as to why one search operator may be more suited to the exploration of the space of computer programs than another operator. The picture is similar when we consider the design of an appropriate fitness function. This lack of theoretical knowledge is reflected in the fact that typically simple toy problems have been tackled, and the solutions are typically of low complexity.

The assumptions of the NFL theorems are too restrictive to make it a useful framework in which to compare real-world search algorithms. We could attempt to avoid the revisiting issue, by say, introducing a library to our current search algorithm, but this would be a very costly affair. The COG theorems, which prove that you cannot generalise, conflict with Occam's Razor, which states that shorter hypothesis are more likely to generalise. One's opinion as to whether or not we can generalise really depends on your stance regarding initial assumptions. You can make one set of assumptions and prove the COG theorems, or alternatively you can make a different set of assumptions and prove Occam's Razor.

One important issue that the NFL theorems do raise is to qualify what class of problems a search algorithm is suited too. It does not really make sense to say a certain algorithm is the best for a given problem (the one shot scenario [15]), but really for a given class of problem. Thrun et. al [77] (page 7) state that there is not a best learning algorithm for learning a set of problems. If the size of class of problems is infinite then the NFL theorem does not apply. Burke et. al. [10] in their conclusions report that *'a GA might be better employed in searching for a good algorithm rather than searching for a specific solution to a specific problem'*. Miller et. al. [31] study an evolutionary approach to the design of digital circuits and argue that *'by studying the evolved design of gradually increasing scale, one might be able to discern new, efficient, and generalisable principles of design'*. This is an example of a situation where a class of problem are defined and we can design a good search algorithm for that class of problem, rather than a single instance of the problem. Another example of a type of problem often used is learning to navigate around a maze. In these problems, typically an agent learns a route around a single fixed maze. A more general and powerful approach

would be to learn how to navigate around mazes. This would involve an agent learning to explore the maze, creating some sort of representation of the maze, and learning how to navigate efficiently i.e. not to go over old ground repeatedly. This is in contrast to learning a single fixed route through a maze. We predict that, as we showed for the even parity task, that large instances of the task can be solved more easily by considering the general case. Ultimately the complexity of the task of navigating around a maze has fixed complexity, whereas the complexity to navigate around a fixed maze will have larger and larger complexity. For other examples of groups of problem see [76]

There are different ways of measuring the number of evaluations a search algorithm makes, and which you choose will depend on what you are attempting to illustrate with the experiment. However, ultimately it is wall clock time which is the ultimate measure so we should really use the closest measure to this we can sensible reach.

We believe an understanding of complexity is essential to the construction of a system which induces functions. In this thesis we proved a result regarding representation and complexity. We believe an understanding of the relationship between complexity and test cases is worthy of further investigation, and work has begun on this (see [91]). Similarly, it is desirable to have and understanding of the relationship between genetic operators and the complexity of the program before and after the application of a genetic operator. Finally, there is the question of how an understanding of complexity can help us design better fitness functions rather than the examples we have now based on hamming distance or a simple error metric. A fitness function drives evolution and should reflect 'how far' programs are from the global optima, rather than how good they are at a certain task.

While we have proved that the complexity of a problem is independent of the primitive set, we would like to be able to develop a search process which is both provable optimal and provable independent of the primitive set.

While in many cases we may not be interested in solving the general case of a problem, we may be interested in solving a large fixed size instance of the problem. In this situation it may be advantageous to cast the problem in the general case and generate general solutions using examples of cases drawn from small instances of the problem. The final result, while being general, could be treated as a specific solution the large instance of the problem. We illustrated this with the even parity problem. An example of a problem which could be treated like this is image recognition. While size is one dimension along which we can construct a generalised form of the problem, there are also other similar such dimensions along which we could form generalisations of problems.

We introduced a representation and genetic operator which reduced the number of non halting programs produced during a run. Unfortunately, poor results were obtained with this method regarding fitness. This may be due to the fact that, when a modular crossover operator is applied to two programs which each consist of two modules, there is only one way to combine the modules of

these programs (i.e. swap the corresponding modules). Thus, this leads to a very poor exploration of the search space. Mutation, on the other hand, has much more potential as far as its possible offspring are concerned. This idea of applying a genetic operator to individuals which halt, could also be applied to explicit recursive representations as the recursion of total functions will only produce total functions.

The halting problem is a central concern when evolving Turing Complete representations. We regard a method which reduces the number of non halting programs generated during a run as an important line of research. A fitness function which promotes the production of halting programs is also of interest, as non halting programs maybe more likely to produce non halting offspring, and time is not wasted waiting for these programs to be terminated.

Bibliography

[1] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[2] Peter J. Angeline. A historical perspective on the evolution of executable structures. *Fundamenta Informaticae*, 35(1–4):179–195, August 1998.

[3] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.

[6] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays*, volume 2. Academic Press, New York, 1983.

[7] Christopher M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, Oxford, UK, UK, 1996.

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

[9] J. Glenn Brookshear. *Computer Science, An Overview, 7th ed.* Assison Wesley, 2003.

[10] Kendall G. Newall J. Ross P. Burke E., Hart E. and Schulenburg S. *Handbook of Meta-Heuristics*, chapter Hyper-Heuristics: An Emerging Direction in Modern Search Technology. Kluwer, 2003.

[11] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. John Wiley and Sons, 1991.

[12] N. L. Cramer. A representation for the adaptive generation of simple programs. In *International Conference on Genetic Algorithms and Their Applications.*, pages 183–187, July 1985.

[13] N. J. Cutland. *Computability, An introduction to recursive function theory*. Cambridge University Press, 1997.

[14] Pedro Domingos. The role of occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425, 1999.

[15] Stefan Droste, Thomas Jansen, and Ingo Wegener. Perhaps not a free lunch but at least a free appetizer. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 833–839, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[16] Stefan Droste, Thomas Jansen, and Ingo Wegener. Optimization with randomized search heuristics: the (a)nfl theorem, realistic scenarios, and difficult functions. *Theor. Comput. Sci.*, 287(1):131–144, 2002.

[17] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[18] Thomas English. Optimiztaion is easy and learning is hard in the typical function. In *Congress on Evolutionary Computation*, pages 924–931, La Jolla, CA, USA, July 2000.

[19] Richard P. Feynman. *The pleasure of finding things out*. Penguing Press, 1999.

[20] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866, pages 312–321, Jerusalem, 9-14 1994. Springer-Verlag.

[21] Chris Gathercole and Peter Ross. Some training subset selection methods for supervised learning in genetic programming. Presented at ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms, 1994.

[22] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming*

1997: Proceedings of the Second Annual Conference, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[23] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[24] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1999.

[25] Lorenz Huelsbergen. Toward simulated evolution of machine language iteration. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford University, CA, USA, 28–31 1996. MIT Press.

[26] Lorenz Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 186–194, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[27] Lorenz Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166, San Francisco, CA, USA, 1998. Morgan Kaufmann.

[28] Lorenz Huelsbergen. Abstract program evaluation and its application to sorter evolution. In *Proceedings of the Congress on Evolutionary Computation*, pages 1407–1414, July 2000.

[29] Marcus Hutter. A gentle introduction to the universal algorithmic agent AIXI. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*, number IDSIA-01-03, page 70. 2003. To appear.

[30] Fakultat Fur Informatik and Jurgen Schmidhuber. On learning how to learn learning strategies, February 01 1995.

[31] D. Job J. F. Miller and V. K. Vassilev. Principles in the evolutionary design of digital circuits – part i. *Journal of Genetic Programming and Evolvable Machines*, 1(1), 2000.

[32] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, 1994.

[33] Kenneth E. Kinnear, Jr. A perspective on the work in this book. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

[34] Tim Kovacs. *A Comparison and Strength and Accuracy-based Fitness in Learning Classifier Systems*. PhD thesis, University of Birmingham, 2002.

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

[36] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[37] John R. Koza. Scalable learning in genetic programming using automatic function definition. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 5, pages 99–117. MIT Press, Cambridge, MA, USA, 1994.

[38] John R. Koza. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In John Robert McDonnell, Robert G. Reynolds, and David B. Fogel, editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, San Diego, CA, USA, 1-3 1995. MIT Press.

[39] W. B. Langdon. Evolving data structures with genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[40] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[41] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[42] William B. Langdon. Scaling of program fitness spaces. *Evolutionary Computation*, 7(4):399–428, 1999.

[43] Ming Li and Paul M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Berlin, 1993.

[44] Carlo C. Maley. *Artificial Life III*, chapter The Computational Completeness of Ray's Tierran Assembly Language. Addison Wesley Longman, 1994.

[45] T. M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[46] Tom M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, New Brunswick, New Jersey, 1980.

[47] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[48] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[49] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[50] Sara Porat and Jerome A. Feldman. Learning automata from ordered examples. In *COLT '88: Proceedings of the first annual workshop on Computational learning theory*, pages 386–396, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[51] Thomas S. Ray. An approach to the synthesis of life. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (ALIFE '90)*, volume 5 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 371–408, Redwood City, CA, USA, February 1992. Addison-Wesley.

[52] Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms - Principles and Perspectives A Guide to GA Theory*. Kluwer, 2002.

[53] Peter Hart Richard Duda and David Stork. *Pattern Classification*. John Wiley, 2001. 2nd edition, Chapter 4, ISBN: 0-471-05669-3.

[54] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[55] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[56] C. Schaffer. A conservation law for generalization performance. In *Proceedings of the 11th International Conference on Machine Learning*, pages 259–265, 1994.

[57] Juergen Schmidhuber. A general method for incremental self-improvement and multi-agent learning in unrestricted environments. In X. Yao, editor, *Evolutionary Computation: Theory and Applications*. Scientific Publishing Company, 1996.

[58] Juergen Schmidhuber, Jieyu Zhao, and Marco Wiering. Simple principles of metalearning. Technical Report IDSIA-69-96, Corso Elvezia 36, CH-6900, Switzerland, 27 1996.

[59] Jurgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Mach. Learn.*, 28(1):105–130, 1997.

[60] C. Schumacher. *Black Box Search, Framework and Methods*. PhD thesis, The University of Tennessee, Knoxville, 2000.

[61] C. Schumacher, M. D. Vose, and L. D. Whitley. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[62] Eric V. Siegel and Alexander D. Chaffee. Genetically optimizing the speed of programs evolved to play tetris. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 14, pages 279–298. MIT Press, Cambridge, MA, USA, 1996.

[63] H.T. Sielgelman and E.D. Sontag. Turing computability with neural nets. *Applied Mathematics*, 1991.

[64] Lee Spector. Simultaneous evolution of programs and their control structures. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[65] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[66] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[67] Julio Tanomaru. Evolving turing machines from examples. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume 1363 of *LNCS*, Nimes, France, October 1993. Springer-Verlag.

[68] Julio Tanomaru and Akio Azuma. Automatic generation of turing machines by a genetic approach. In Daniel Borrajo and Pedro Isasi, editors, *The First International Workshop on Machine Learning, Forecasting, and Optimization (MALFO96)*, pages 173–184, Gatafe, Spain, 10–12 1996.

[69] Astro Teller. The evolution of mental models. In *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[70] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

[71] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 27-29 1994. IEEE Press.

[72] Astro Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, Cambridge, MA, USA, 1996.

[73] Astro Teller. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 5 December 1998.

[74] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.

[75] Astro Teller and Manuela Veloso. PADO: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[76] Sebastian Thrun and Lorien Pratt. *Learning to Learn*. Kluwer Academic Publishers, 1999.

[77] Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In Sebastian Thrun and Lorien Pratt, editors, *Learning to Learn*, chapter 1, pages 3–17. Kluwer Academic Publishers, 1999.

[78] Edgar E. Vallejo and Fernando Ramos. Evolving turing machines for biosequence recognition and analysis. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming: 4th European conference*, volume 2038 of *LNCS*, pages 192–203, Berlin, 18-20 April 2001. Springer.

[79] James Alfred Walker and Julian Francis Miller. Evolution and acquisition of modules in cartesian genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP*

2004, Proceedings, volume 3003 of *LNCS*, pages 187–197, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[80] Soule T. Wang G. How to choose appropriate function sets for genetic programming. In *Genetic Programming, Proceedings of EuroGP 2004*, Coimbra, Portugal, 2004. Springer-Verlag.

[81] G. I. Webb. Generality is more significant then complexity: Toward an alternative to occams razor. In *Seventh Australian Joint Conference on Artificial Intelligence (AI'94) - Artificial Intelligence: Sowing the Seeds for the Future*, pages 60–67, Singapore, 1994. World Scientific.

[82] P. A. Whigham and R. I. McKay. Genetic approaches to learning recursive relations. In X. Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 17–27. Springer-Verlag, 1995.

[83] D. Randall Wilson and Tony R. Martinez. Bias and the probability of generalization. In *Proceedings of the 1997 International Conference on Intelligent Information Systems (IIS'97)*, pages 108–114, 1997.

[84] Stephen Wolfram. *A new kind of science*. Wolfram Media Inc., 2002.

[85] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, 1995.

[86] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[87] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[88] John Woodward. Evolving turing complete representations. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 830–837, Canberra, 8-12 December 2003. IEEE Press.

[89] John Woodward. GA or GP? that is not the question. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1056–1063, Canberra, 8-12 December 2003. IEEE Press.

[90] John R. Woodward. Function set independent genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), Workshop on Modularity, Regularity, and Hierarchy in Evolutionary Computation*, Seattle, USA, 26-30 June 2004. Morgan Kaufmann.

[91] John R. Woodward. Simple incremental testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), Late Breaking Papers*, Seattle, USA, 26-30 June 2004. Morgan Kaufmann.

[92] X. Yao. Evolving artificial neural networks. *IEEE: Proceedings of the IEEE*, 87:1423–1447, 1999.

[93] Tina Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, December 2001.

[94] Byoung-Tak Zhang and Dong-Yeon Cho. Genetic programming with active data selection. *Lecture Notes in Computer Science*, 1585:146–153, 1999.

[95] Huaiyu Zhu and Richard Rohwer. No free lunch for cross-validation. *Neural Computation*, 8(7):1421–1426, 1996.