Canonical Representation Genetic Programming

John R. Woodward
University of Nottingham
199, Taikang East Road, University Park
Ningbo, 315100, People's Republic of China
John.Woodward @Nottingham.edu.cn

Ruibin Bai
University of Nottingham
199, Taikang East Road, University Park
Ningbo, 315100, People's Republic of China
Ruibin.Bai @Nottingham.edu.cn

ABSTRACT

Search spaces sampled by the process of Genetic Programming often consist of programs which can represent a function in many different ways. Thus, when the space is examined it is highly likely that different programs may be tested which represent the same function, which is an undesirable waste of resources. It is argued that, if a search space can be constructed where only unique representations of a function are permitted, then this will be more successful than employing multiple representations. When the search space consists of canonical representations it is called a canonical search space, and when Genetic Programming is applied to this search space, it is called *Canonical Representation Genetic Programming*.

The challenge lies in constructing these search spaces. With some function sets this is a trivial task, and with some function sets this is impossible to achieve. With other function sets it is not clear how the goal can be achieved. In this paper, we specifically examine the search space defined by the function set $\{+, -, *, /\}$ and the terminal set $\{x, 1\}$. Drawing inspiration from the fundamental theorem of arithmetic, and results regarding the fundamental theorem of algebra, we construct a representation where each function that can be constructed with this primitive set has a unique representation.

Categories and Subject Descriptors

I [**Computing Methodologies**]: Artificial Intelligence—*Automatic Programming*

General Terms

Algorithms

Keywords

Genetic Programming

GEC'09, June 12–14, 2009, Shanghai, China.

1. INTRODUCTION

1.1 Genetic Programming

Genetic Programming (GP) is the search of the space of objects or 'computer programs'[1, 2]. This search space may consist of *representations* of integers, rational numbers, bit strings, logical expressions, arithmetic functions, computer programs and other representations specifically designed for the purpose of evolution including Cartesian Genetic Programming[3] and Cramer's seminal work on evolving primitive recursive functions[4]. The set of objects which can potentially be evolved using Evolutionary Computation, is as broad as the objects which can be represented on a computer. In some sense these objects can the thought of purely as numbers (or bit strings), but in a different sense they can be thought of as data structures, depending on how we decide to interpret our objects[5].

The representations being manipulated *directly* by the genetic operators are contained in the search space. These are the genotypes. Objects in the space of genotypes are then interpreted and expressed as phenotypes. For example, the space of genotypes may contain bit strings, which could be used to express integers or programs in the phenotype space. Objects in the phenotype space are in effect manipulated indirectly by the genetic operators (i.e. we cannot manipulate a function directly, we can only directly manipulate a representation of a function).

1.2 Re-sampling

While there is a plethora of representations which could be the focus of evolutionary processes, most of these representations have in common the fact that, when representing functions there is, typically, multiple ways of representing the same thing. Given that our goal is to find a solution to a problem as efficiently as possible, having a search space in which items are represented only once is a better approach than having items represented multiple times. The reason for this is that if we are repeatedly sampling the search space at each generation, while it is possible to re-sample the same object, it is not desirable if different items in the genotype space map to the same object in the phenotype space. Thus there are two reasons why the same function may be re-sampled. Firstly, the same representation of that function may be sampled again (i.e. the same point is visited twice in the search space). Secondly, a different representation of that function may be sampled (i.e. two different points are visited which represent the same function).

Langdon[6] examines the nature of the search spaces ex-

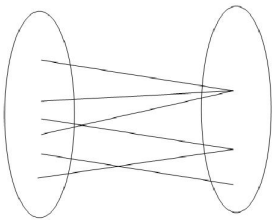

Figure 1: Many search spaces in GP consist of multiple representations of the same object. This is illustrated by many points (trees) in the genotype space mapping to single points (functions) in the phenotype space. There is a many to one mapping between the genotype and phenotype space. The program (genotype) space is the left ellipse and the function (phenotype) space is the right ellipse.

plored by GP by examining the frequency with which functions are represented. He proves in some cases, and argues in others, that the frequency with which functions are represented does not change after the size of the search space is greater than some threshold. What is also evident is that these distributions are far from uniform. Hence we can conclude from this work that some functions are represented far more frequently than other functions. This is regarding the search space, and does not explicitly say anything about the sampling done by GP.

Howard[7] examines the frequency of functions represented by modules in the encapsulation process. Here he shows that some functions are represented more frequently than others as modules.

1.3 The aim and outline of this paper

The aim of this paper is to show how, given a non-trivial function set, a search space can be constructed where only one representation of a function exists. That is, our search space consists only of canonical forms. In other words, the mapping between the phenotype space and genotype space is one to one.

Initially, we study how the fundamental theorem of arithmetic can help us construct rational numbers in a canonical form. This can be done because expressing rationals as ratios of integers which are in turn expressed as powers of prime numbers, allows us to spot immediately and trivially whether there are common factors. The fundamental theorem of algebra states that a polynomial of degree n, can be written as the product of n 1-degree polynomials (with certain conditions pertaining). We use this motivation to build ratios of polynomials in a canonical form, analogous to the case with rational numbers.

Let us clarify our aim by stating what we are *not* attempting to do. We are not taking a canonical representation of a function, applying a genetic operator to it, which *may* produce a non-canonical form, and then employing a 'repair' method to return it to a canonical form. We want to design a representation, which is in a canonical form, and the result of the application of a search operator will still be in a canonical form i.e. we only want to work in the space of canonical representations.

The outline of the paper is as follows; In section 2 we examine the nature of the representation of functions and why functions have multiple representations. In section 3 we give some background mathematics. In section 4 we give some examples of canonical representations. In sections 5 and 6 we show how canonical representations of numbers and functions can be constructed. In section 7 we comment on the relevance of the No Free Lunch Theorem. In section 8 Koza's lens effect is interpreted for canonical representations. We end the paper with the customary discussion, further work, summary, and conclusions in sections 9 - 12.

2. THE NATURE OF THE REPRESENTATION OF FUNCTIONS.

In Machine Learning, many types of representation are used to express functions (tree based expressions, artificial neural networks, classifier systems, radial basis functions, binary decision diagrams, support vector machines...)[8]. Given one type of representation, there are many ways to express a given function. In this section, we examine four reasons why a function can be represented in more than one way by a given type of representation.

2.1 Symmetric Functions.

Some functions have a symmetric property in that the result of the function will be independent of the order of the arguments to the function. Hence there will be a number of ways of representing the same function. For example, $a + b = b + a$, $max(a, b) = max(b, a)$, and $a \vee b = b \vee a$. Clearly, if a function set contains symmetric functions, there will be more than one way of representing functions which require this primitive in their construction.

Similarly the following are functionally equivalent;
if (!condition) then (statement2) else (statement1)
if (condition) then (statement1) else (statement2)
where ! is the logical not operation. Thus there is more than one way of expressing a conditional statement. Though strictly `if-then-else` is not a symmetric function, it clearly has symmetric properties. Similarly `switch` statements could be reordered yet represent the same procedure.

Ordered Binary Decision Diagrams avoid this issue in the following way[9]. Each node consists of a Boolean variable, and if the value of the variable is true we descend down the right hand branch, otherwise we decend down the left hand branch. Thus, for each node in the tree, we cannot have the situation above with `if-then-else` statements.

2.2 Null and Inverse Functions

In some GP set ups, primitives have been used which have no affect on the function being calculated. For example, Huelsbergen[10] includes a `nop` instruction ('no operation') in his machine language. This instruction has no effect on the register. The instruction `nop` can be removed from the instruction (function) set without affecting the expressitivity of the representation. A null function can be thought of as an identify function (i.e. $f(x) = x$). In Boolean Logic, the logical NOT is its own inverse function ($!(!a) = a$).

The inverse function, $f^{-1}(x)$, is defined as $f^{-1}(f(x)) = x$ (assuming it exists). In effect, the inverse of a function 'undoes' the function, thus there are many ways a given

function can be constructed if a function and its inverse are present in the function set. The alert reader will notice that an indentity function can be constructed if the inverse is present in the function set (i.e. $f^{-1}(f(x)) = x$).

2.3 Complementary Functions.

Some functions have a corresponding 'complementary' function, for example $+, -$. In the expression $x+a-a$, the first a is added and the second is subtracted, essentially 'undoing' or reversing the addition. In some branches of GP, languages similar to assembly have been used [10]. These languages often contain instructions like *inc* and *dec*, which increment and decrement a memory location respectively. These are clearly complementary instructions. Having complementary functions in a function set will allow combinations of primitives to represent the same function.

2.4 Isomorphic Data Structures.

The type of representation used may allow a function to be expressed in more than one way. These types of representation include Automatically Defined Functions (ADFs) [1], Cartesian Genetic Programming[3], Artificial Neural Networks, Finite State Automata. There may be many ways to express a function due to the nature of the data structure used to represent the function as some instances of data structures can be considered equivalent under isomorphism.

Consider a GP program which consists of a main procedure and a number of ADFs. Two programs are equivalent if the order of their ADFs is altered and the references to the ADFs are altered accordingly. Just as with a computer program, the order of two procedures can be switched, but the program will perform the same operation (as the correct procedure is called by name irrespective of its position in the overall program, however the two programs clearly are different).

Another example of the way the isomorphic nature of the representation contributes to the number of ways a function can be represented maybe seen in neural networks; the hidden nodes can be permuted and this will be the same function [11]. This is similar to the situation with Cartesian Genetic Programming[3], and this is because the underlying representation used is a Directed Acyclic Graph.

Finally, there are many ways in which a finite state automata can represent a function. However, the minimum sized finite state automata is unique under isomorphism[12], and thus a function has a canonical representation when finite state automata are used.

3. BACKGROUND MATHEMATICS.

In this section we review some background mathematics. We firstly remind ourselves of some classes of numbers. Secondly, we look at how primes can be used to represent integers in a unique fashion. Finally, some results from abstract algebra concerning polynomials are presented.

3.1 Numbers.

We use the standard notation for the different classes of numbers;

$\mathbb{N} = \{0, 1, 2, ...\}$, natural numbers,
$\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$, integers,
$\mathbb{Q}$ are the quotient of two integers, rational numbers
$\mathbb{R}$ are the reals.
$\mathbb{C} = \{a + ib \| a, b \in \mathbb{R}, i = \sqrt{-1}\}$, complex numbers.

The complex conjugate of $c = a + ib$ is $c^* = a - ib$, and has the property $c.c^* = a^2 + b^2$, i.e. $cc* \in \mathbb{R}$; We call c and c^* a conjugate pair. (Note also that the operation of conjugation is its own inverse function and complementary function). Complex numbers have two common representations; cartesian and polar. Note that the cartesian representation is canonical, while the polar is not unless we define the angular coordinate to be between 0 and 2π, and the radial coordinate to be non-negative. Numbers can be ordered, with complex numbers being ordered too, first by ordering on one coordinate and then on the second.

3.2 The Fundamental Theorem of Arithmetic.

In number theory, the fundamental theorem of arithmetic (or the unique factorization theorem) states that every natural number can be written as a unique product of prime numbers [13]. For instance, $6936 = 2^3.3^1.17^2$ and $1200 = 2^4.3^1.5^2$. This canonical representation of integers will allow us to build a canonical representation of rationals.

3.3 The Fundamental Theorem of Algebra.

The fundamental theorem of algebra states that a complex polynomial of degree n has n roots [13]. In other words a polynomial of degree n can be written as

$$\alpha \prod_{i=0}^{i=n} (x - c_i)$$

where c_i are the roots, and α is some constant. It follows from this theorem that every polynomial with real coefficients can be written as a product of polynomials of degree 1 or 2 with real coefficients. As a complex number multiplied by its conjugate is real, it follows that every polynomial with real coefficients can be written as a product of polynomials of degree 1 with real coefficients, or coefficients which are conjugate pairs. In other words, polynomials of degree 2 are replaced by pairs of polynomials of degree 1 whose roots are complex conjugates of each other. If we impose some ordering on the product of these 1 degree polynomials, then we have a canonical representation of polynomials.

4. CANONICAL REPRESENTATIONS

An object is said to be in canonical form if there is only one way of representing it in the given representation. This allows the mathematical or abstract object to be uniquely identified, or encoded. In other words the mapping function between the representation and its interpretation is one to one. In many cases this will mean that it is easy to switch between the representation of an object and the interpretation of the object i.e., it will be easy to produce the interpretation, given the representation, and vica versa. There may be more than one canonical representation of an object.

There is a one to one mapping between the binary encoding and the integer being represented. For example, $1101 = 2^{3.1}.2^{2.1}.2^{1.0}.2^{0.1}$. Given an integer or a bit code, it is easy to switch between the two. An alternative canonical representation is the Gray code, or reflected binary code. An important difference between these two representations is that with the Gray scale, two successive values differ only by a single digit, whereas with the binary representation two successive values may differ by a large number of digits. This gives rise to Hamming Cliffs which have consequences for the efficiency of the search[14].

A Binary Decision Diagram (BDD) is a directed acyclic graph used to represent Boolean functions. If an ordering

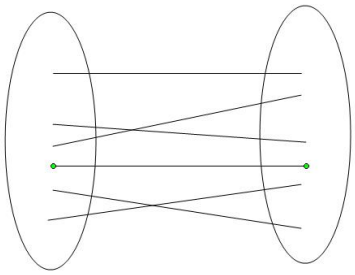

Figure 2: With a canonical search space, each object has only a single representation. This is illustrated by single points (trees) in the genotype space on the left mapping to single points (functions) in the phenotype space on the right. In other words, each object has a unique representation.

is imposed on the Boolean variables, the Binary Decision Diagram is said to be an ordered Binary Decision Diagram, which have a canonical form unlike e.g. conjunctive normal forms[9]

Some objects do not have a canonical form. The representation of a program on a Turing Machine (or its equivalent), does not have a canonical form (by Rice's Theorem) [12].

5. AN EXAMPLE USING NUMBERS.

In this section, we illustrate the idea of a canonical representations using rational numbers as the objects we wish to represent. We firstly look at numbers in standard form or scientific notation, then an alternative canonical representation using the Fundamental Theorem of Arithmetic.

Rational numbers are typically represented on a computer as a floating point number. They will have a fixed length (e.g 10 places) and therefore fixed accuracy (e.g. 12345.67890, i.e. 10 places of accuracy). An alternative representation is to store a number (between 0.000000000 and 9.99999999, i.e to 10 decimal places) followed by and exponent (power of ten), e.g. 1.234567890E33. In this case, a number has a single representation.

A rational number can also be represented as the ratio of two integers, but for the rational number to be in its canonical representation any common factors must be removed (e.g. 2/4 can be reduced to 1/2). However, it is not immediately clear how these numerators and denominators can be altered by genetic operators to guarantee that they remain in canonical form, without going through the repair process of finding common factors. This is addressed below.

In number theory, the fundamental theorem of arithmetic states that every natural number can be written as a unique product of prime numbers. We can therefore represent an integer as a vector of the powers of primes, the ith element of the vector being the power of the ith prime (Note that we restrict the order of the primes so that the ordering is fixed and unique). Each power is a natural number. A natural number can be written as

$$\prod_{i=1}^{i=n} p_i^{v_i},$$

where p_i is the ith prime, and v_i is the ith element of the vector V and $v_i \in \mathbb{N}$.

The first few primes are (2, 3, 5, 7, 11, 13, 17, ...). Thus, 6936 can be represented by the vector (3, 1, 0, 0, 0, 0, 2) and 1200 by (4, 1, 2). The rational number 6936/1200 can now be represented in our new prime powers representation as ((3, 1, 0, 0, 0, 0, 2), (4, 1, 2)), the first vector being the numerator and the second vector being the denominator i.e. ((numeratorVector), (denominatorVector)). We can see immediately using this representation that there are common factors i.e. $2^3.3^1 = 24$. Thus the canonical representation of 6936/1200 is ((0, 0, 0, 0, 0, 0, 2), (1, 0, 2)). If the ith component of the numerator and denominator vector are both non-zero, then a common factor exists, and the rational number is not in its canonical form. A rational number is therefore in its canonical representation if either (or both) the ith element of the numerator or denominator are zero, and this is true for all i in the pair of vectors. We can go one step further, and combine the numerator vector and denominator vector into a single vector, as powers in the denominator can be interpreted as having negative power. Thus in this case ((0, 0, 0, 0, 0, 0, 2), (1, 0, 2)), can be re-written as (-1, 0, -2, 0, 0, 0, 2). This encoding means $2^{-1}.3^0.5^{-2}.7^0.11^0.13^0.17^2$. In general a rational number can be written uniquely as

$$\prod_{i=1}^{i=n} p_i^{v_i},$$

where p_i is the ith prime, and v_i is the ith element of the vector and $v_i \in \mathbb{Z}$ (i.e negative and positive powers).

Also, at this stage it is worth pointing out that it is easy to design genetic operators which could generate new rational numbers from old rational numbers which are in their canonical form, without the need for further processing (i.e. finding common factors).

6. AN EXAMPLE USING FUNCTIONS.

In this section we look at how a canonical representation can be constructed for different function and terminal sets. We begin with an arithmetic function set and a terminal set consisting only of 1 and no variables. We then look at a number of cases, starting with a trivial function set and show how a canonical representation can be constructed. This primitive set is then supplemented to finally create a set which would be used in a GP system.

6.1 Primitive Set $\{+, -, *, /\} \cup \{1\}$

Given $\{+, -, *, /\} \cup \{1\}$, we can generate rational numbers. Rational numbers, as we have seen, have a number of canonical representations. This terminal set does not contain any variables (e.g. x), and so can only generate trees which compute constants (i.e. rational numbers). We could carry out function optimisation, using GP to generate rational numbers with this primitive set. However this would probably not be as efficient as using a GA approach to optimisation, where each number has a single representation.

6.2 Function set $\{+\}$.

Let us start with the function set $\{+\}$ and the terminal set $\{x\}$, which gives us the primitive set $\{+\} \cup \{x\}$. This defines a search space consisting of trees like $(x + x)$ and

$((x + x) + (x + x))$ and so on. However, it is clear that, there are many ways of constructing the same function. All of these functions are of the form $\{n.x|n \in N\}$. We can easily restrict the space of these trees to contain only a single representation of a given function. We could think of these restricted trees as lists e.g. $(x + x + x + x)$ rather than a tree $((x+x)+(x+x))$, as the brackets are redundant in the case of this primitive set.

6.3 Function set $\{+, -\}$.

Let us now extend the function set to include $\{-\}$, which is the complementary operand of $\{+\}$. Thus the primitive set now consists of $\{+, -\} \cup \{x\}$. Just as in the previous example, there are multiple ways of representing the same function, but now we have the additional interaction of two functions which can counteract one another and in effect cancel each other out. Now we can represent functions of the form $\{z.x|z \in \mathbb{Z}\}$. Any tree in the traditional search space (e.g. (((x+x)-(x-x)....) can be represented in a canonical way. Again in a bracketed representation of a tree, the brackets are redundant and can be removed, and the plusses (+) and minuses (-) can be summed up to give a total of z. Further, the constant 1 can be added to the terminal set to give a new primitive set $\{+, -\} \cup \{x, 1\}$. This primitive set allows us to express functions of the form $\{z_1.x + z_2|z_1, z_2 \in \mathbb{N}\}$.

6.4 Function set $\{+, -, *\}$.

Adding $\{*\}$ to the primitive set gives us $\{+, -, *\} \cup \{x, 1\}$. This allows us to express integer powers of x, and integer coefficients. In other words, we can express polynomials with integer coefficients. We can write this in a closed form;

$$\sum_{i=0}^{i=n} a_i.x^i$$

where $a_i \in \mathbb{Z}$ and n is finite. Note also a_i are all integers (there is no division present in this function set to allow us to create rational numbers, that is the next stage). One simple encoding of a polynomial with integer coefficients would be simply to list the coefficients, and the ith coefficient in the vector is the coefficient of x^i. While this is one closed canonical form, we can also represent a polynomial with integer coefficients as

$$\alpha \prod_{i=0}^{i=N} (x^i - b_i).$$

where $b_i \in \mathbb{Z}$ or pairs of complex conjugates of the form $\mp\sqrt[2]{b}$, and $b_i < b_{i+1}$ (to imposes an order). For example, $x^2 + 2$ can be written as $(x - i\sqrt[2]{2})(x + i\sqrt[2]{2})$. Note that this requires the use of reals, which we can at present not represent exactly on current day computers. However, we can approximate them arbitrarily closely with rationals of increasing precision. Alternatively, rather than factoring a polynomial of degree 2 into two linear functions with conjugate roots, and running the risk of rounding errors, we could keep this in "quadratic form" (i.e. a polynomial of degree 2). Both are canonical forms.

6.5 Function set $\{+, -, *, /\}$.

Let us now add $\{/\}$ to the function set to give us a primitive set $\{+, -, *, /\} \cup \{x, 1\}$. This set is now starting to look like a function set one might actually use in a GP system. In general, functions generated using this primitive set can be written as a quotient of two polynomials with rational coefficients. Rational coefficients can be generated by dividing two natural numbers (1+1+1+...)/(1+1+1+...). One might be tempted to construct a canonical representation as Q/P where Q and P are polynomials, however, one may have difficulty identifying factors as polynomials of degree 5, or above do not have a general solution[13].

We therefore turn our attention to the representation of integer polynomials and rational numbers of the previous sections. An integer can be written uniquely as the product of primes. Similarly a polynomial can be written as the product of polynomials of degree 1. As shown above, ratios of integers (i.e. rationals) can be constructed uniquely as a product and division of primes, where each prime has either a positive or negative power depending on whether it appears in the numerator or denominator. Similarly the quotient of polynomials can be written as the product and division of polynomials of degree 1, where each polynomial of degree 1 has either a positive or negative power depending on whether it appeared in the numerator or denominator. Thus we can write expression of the form

$$\alpha \prod_{i=0}^{i=n} (x - b_i)^{exp_i}.$$

where $b_i \in \mathbb{R}$ or are paired complex conjugates (to ensure real coefficients when multiplied together). Again, some order is imposed, i.e. $b_i < b_{i+1}$, so we have a canonical ordering, and exp_i are integers. Again, real are required if we want to represent a function as the product of integer powers of linear functions. Again we can leave these in their quadratic form.

Thus a function can be represented as a list of the roots of the polynomial and the number of times that root occurs (i.e. the power). Hence a function can be written as a set or list of points and exponents (i.e. b_i, exp_i). Let us consider representing the list of points b_i on an argand diagram (i.e. in the complex plane). If $b_i \in \mathbb{Q}$, then it will appear on the real line. If $b_i = x + iy$ $(y > 0)$, then it will appear in the upper half of the complex plane. If $x + iy$ is in the list, then its conjugate will also appear i.e. $b_i = x - iy$, and this will be placed symmetrically in the lower half of the plane. With each point b_i we associate exp_i. Thus a function created by this function set can be thought of as a set of points, each labelled with an integer exponent, in the complex plane symmetrically distributed about the real line.

7. THE NO FREE LUNCH THEOREM.

In this section we give an introduction to the No Free Lunch Theorem (NFL)[15, 16] using a simple analogy with a bicycle combination lock. We then state that NFL is not valid for standard GP due to the many to one mapping between the space of genotypes and phenotype [5]. We then state that because each function has only one canonical representation, NFL is valid when GP is applied to a canonical search space.

7.1 Intuitive Example of the No Free Lunch Theorem.

The NFL theorem is often misunderstood, and because of this we wish to give an everyday practical example. Imagine a combination lock often used for bicycles. Typically the lock consists of four rings, each having the digits $0 - 9$.

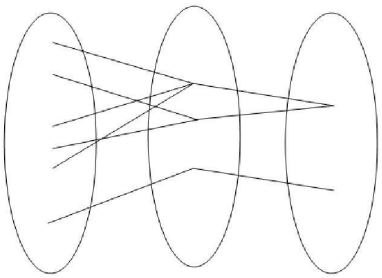

Figure 3: The space of ADFs, can be thought of as mapping onto a space of trees (i.e. each ADF has an equivalent tree representation). The space of trees maps onto the space of functions. Some functions have many different representations as trees. These many to one mapping between these spaces is responsible for the lens effect. In effect, looking at the space of functions using an ADF representation, is like using a compound lens: the first lens corresponding to the mapping between ADFs and trees, and the second lens corresponding to the mapping between trees and functions.

There are 10 possible positions for each ring and therefore 10^4 possible settings for the lock. In an abstract sense the lock can be thought of as the representation of a function. The domain (or inputs) of the function is the set of all possible setting of the lock from 0000 to 9999. The range (or output) of the function is $\{open, closed\}$, i.e. any combination of the lock will either map to open or closed. The lock essentially computes a "needle in a haystack" type function [17], where all the combinations of the lock except one map to closed, and one maps to open. The question is, given that we do not know the combination which opens the lock, what is the best approach to open the lock? There are 10000! possible strategies (i.e. 10000 settings for our first choice, and having eliminated that, 9999 settings for our second choice, 9998 for our 3rd and so on....). Given that we do not know what the combination is, it is purely luck when we find the correct sequence of 4 digits. This is essentially an illustration of the NFL theorem, which says that over all functions (specifically, only all permutations of a function are necessary), then no one search strategy is better than any other. In the case of the lock, and ascending enumeration of the combinations (starting at 0000 and moving through to 9999) is just as fast on average as a descending enumeration (starting at 9999 and moving through to 0000).

7.2 The No Free Lunch Theorem for Canonical Representation Genetic Programming.

The NFL theorems are valid for phenotype genotype mapping, where all functions can be considered. In traditional GP, where there is a many to one mapping between these two spaces, NFL is not valid and it is possible, in principle, to construct a search operator which will do better than average [5]. However, given that we are dealing with a canonical search space, any bias due to representation is lost and the validity of NFL are reinstated.

8. BIAS AND KOZA'S LENS EFFECT.

Bias is necessary if a learning system is to learn. An unbiased system cannot meaningfully learn[18]. Bias is any effect that causes us to select one function over another. In search based systems, bias can arise due to the nature with which functions are represented and the order in which they are sampled (i.e. the effect of the search operators).

Koza[19] (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, trees and ADFs. A look up table is simply a list of all the input-output tuples (i.e. the function is represented explicitly). If we impose an ordering on the inputs, then look up tables are a canonical representation. Using trees to represent functions is more sophisticated and allows some functions to be represented more compactly (depending on the function) compared to representing them as look up tables. ADFs are more sophisticated still, as repeated sub-structures in a tree can be collapsed into a single ADF, saving even more space. These differences in the way functions are represented have an affect on the frequency with which functions are represented with the three types of representation, and in effect distort the space of functions depending on which representational lens we look through. This is refered to as the lens effect. We can of course extend this representational heirarchy to include Turing Complete representations, and this would be a very fundamental lens through which to view the space of functions due to Church's thesis[12].

The chance of generating a given function with a look up 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 $\{AND, OR, NAND, NOR\}$, Koza generates 10^7 trees at random, but finds no solutions. In fact if we look at the work of Langdon[6], we can see distributions for different Boolean function sets and see that they are far from uniform. However, if ADFs are used (two, two argument ADFs), Koza manages to generate 35 solutions. Koza talks about the problem environment (i.e the space of functions) being viewed through the *"lens of a given type of representation"*. We can in fact think of the space of trees being viewed through the space of ADFs, as each program expressed as an ADF corresponds to a tree. Hence we can think of looking through a *compound* lens.

Teller [20] (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. 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.

In terms of a canonical search space, each function is represented once, and therefore with equal frequency. There is no representational distortion of the space of functions when viewing the space with canonical lenses.

9. DISCUSSION

9.1 Genetic Operators.

In this paper we have shown that a search space can be constructed which contains only a single representation of a given function. This is unlike most previous versions of GP. What we have not considered at this stage is how this space can best be searched i.e. what are the suitable operators? While we have achieved the aim of the paper and constructed a canonical search space, which should give us a huge benefit over conventional search spaces, this benifit could be lost by the use of unsuitable search operators. A landscape is defined by both the representation and the operators which dictate how we move around the space. An individual can be represented as a set of points which lie in the complex plane (symmetrically distributed about the real line). We need to investigate how a set of points can be meaningfully manipulated by mutation and crossover.

Here we have a representation, where component parts consist of the product of linear functions $(x-r)$ with integer powers. If r is changed in one or many of the linear functions by a small amount, this will lead to a small change in the function expressed. The smaller the change in r, the smaller the change in the function expressed. Thus, one obvious mutation operator we wish to propose is to change the values of r in one or many of the linear components. A second issue is how to introduce or remove points (i.e. alter the degree of the polynomial).

9.2 Extending to Multiple Variables.

It is not immediately obvious how this approach can be extended to include terminal sets which consist of multiple variables. It may be impossible to construct a canonical representation given a function set consisting of divisions of multivariate polynomials. However, we may be able to construct a space which reduces the number of representations of a function, so that although we do not achieve a canonical representation, we do improve on the chances of re-sampling experienced with traditional search space. Alternatively, we may be able to construct a space which does not contain representations of all possible functions, but is canonical and contains approximations to most of the functions we would practically require.

9.3 Bloat.

One problem with GP is bloat; the uncontrolled explosion in the size of individuals in the population. If a canonical representation is used this will not be a problem and there will be no redundancy in the representation. While this may sound advantageous, some researchers may take a contradictory position, arguing that neutrality is an important property for a representation to have if it is to be used successfully for Evolutionary Search[17].

9.4 Diversity.

Diversity is an issue in GP. Often syntactic diversity is taken into account by defining a diversity measure on items in the phenotype space, however syntactic diversity does not imply semantic diversity. If a canonical representation is used genotypic diversity implies phenotypic diversity.

9.5 A Single Divide Node.

In the function set explored in this paper, a division function exists. In a standard GP tree, it is possible that a division node exists at multiple points in a tree (e.g. $(x/(x-1)/(x-1))$). However, basic algeba shows that these types of expression can be written using a single division operation. We could therefore generate only trees with a single division node at the root and no where else. While this may seem like a restriction, the same set of functions can be expressed. Thus, the search space would consist of trees containing a division node at the root and the left and right branches contain trees build from the primitive set $\{+,-,*\} \cup \{1,x\}$. While this would *not* produce expressions in canonical form, it does drastically reduce the size of the search space (compared to allowing / to exist at any non-leaf nodes).

10. FURTHER WORK.

So far we have shown how to construct a search space for a function set consisting of a number of arithmetic operators. This is one domain, and of course we need to extend this work to other non-trivial domains, for example the logical function set $\{AND, OR, NOTXOR\}$. This function set contains symmetric functions (e.g. AND), and an inverse function (e.g. NOT). Thus within this logical function set, there are lots of opportunities to create the same function in different ways. Again, it is not immediately obvious how a canonical representation of logical functions (for a given function set) can be achieved, and at the same time design genetic operators which will preserve the canonical nature of the representation.

This paper is theoretical in nature. The next stage, now we have created a canonical search space, is to apply it to a real world problem. This will demand the investigation of suitable search operators. In previous work we have tackled a real world combinatorial problem, to which the function set considered in this paper is applicable. It is our intention to compare the search of a canonical search space with the traditional tree based search space.

11. SUMMARY.

A canonical representation is a representation where an object (function) has a unique representation. GP search spaces typically do not contain canonical representations (a given function has multiple representations), and therefore the chance of re-sampling a function (phenotype) is high as the function is represented by many different tree data structures (genotypes). We have shown, for a given function set containing arithmetic operators, a search space can be constructed, where each function which can be expressed is expressed only once. It is suggested that a search space of this type makes the induction of target functions easier than using conventional search spaces.

The main result of this paper is the following. Functions, which can be expressed with the primitive set $\{+,-,*,/\} \cup \{x,1\}$, can be expressed uniquely as

$$\alpha \prod_{i=0}^{i=n} (x-b_i)^{exp_i}.$$

where $b_i \in \mathbb{R}$ or are pair of complex conjugates (to ensure real coefficients), $exp_i \in \mathbb{Z}$ and $\alpha \in \mathbb{Q}$.

This work has fundamental implications for NFL, a central theorem in search. While NFL is not valid for traditional GP, where functions have multiple representations, it

is valid for GP when applied to a canonical search space as the genotype phenotype mapping is one to one.

Suggestions were made for suitable genetic operators for the new canonical representation. It was brought to the attention of the reader that this will result in different landscapes, which are probably smoother than those encountered in traditional GP. A slight change in the genotype leads to slight changes in the phenotype i.e. a slight perturbation in one of the values used in the representation will cause a slight change in the function represented. This is unlike standard GP, where the operation of a mutation operator many have huge consequences for the function expressed.

12. CONCLUSIONS

This paper is concerned with the nature of function representation and the construction of search spaces. Typically, the GP search space consists of many representations of the same function and this is likely to be detrimental to the search process due to re-sampling. This is because when sampling syntactically different GP trees, there is a large likelihood that the same semantic function is being sampled. It is undesirable to resample functions repeatedly. One approach would be to construct search operators which sample trees which represent different functions. The approach taken is this paper is to construct a search space which only consists of unique representations of a particular function. There is of course nothing stopping re-sampling due to the revisiting of the same tree.

Given one type of object we wish to represent, there may be many different ways of representing canonical representations. This choice, along with the genetic operators, define the landscape, and it is the interplay of these factors which ultimately determine the success of the search process.

We believe that there is worth in investing time to create these canonical search spaces. They will be able to be reused by the GP community and by researchers in the broader field of Machine Learning.

13. REFERENCES

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

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

[3] Miller, J.F., Thomson, P.: Cartesian genetic programming. In Poli, R., Banzhaf, W., Langdon, W.B., Miller, J.F., Nordin, P., Fogarty, T.C., eds.: Genetic Programming, Proceedings of EuroGP'2000. Volume 1802., Edinburgh, Springer-Verlag (2000) 121–132

[4] Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In Grefenstette, J.J., ed.: Proceedings of an International Conference on Genetic Algorithms and the Applications, Carnegie-Mellon University, Pittsburgh, PA, USA (1985) 183–187

[5] Woodward, J.: GA or GP? that is not the question. In Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T., eds.: Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, Canberra, IEEE Press (2003) 1056–1063

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

[7] Howard, D.: Modularization by multi-run frequency driven subtree encapsulation. In Riolo, R.L., Worzel, B., eds.: Genetic Programming Theory and Practise. Kluwer (2003) 155–172

[8] Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)

[9] Huth, M.R.A., Ryan, M.: Logic in computer science: modelling and reasoning about systems. Cambridge University Press, New York, NY, USA (2000)

[10] Huelsbergen, L.: Toward simulated evolution of machine language iteration. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 315–320

[11] Yao: Evolving artificial neural networks. PIEEE: Proceedings of the IEEE **87** (1999)

[12] Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, Massachusetts (1979)

[13] Fraleigh, J.B.: (A First Course in Abstract Algebra)

[14] Whitley, D.: A free lunch proof for gray versus binary encodings. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 1., Orlando, Florida, USA, Morgan Kaufmann (1999) 726–733

[15] Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM (1995)

[16] Schumacher, C., Vose, M.D., Whitley, L.D.: The no free lunch and problem description length. In Spector, L., Goodman, E.D., Wu, A., Langdon, W.B., Voigt, H.M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, California, USA, Morgan Kaufmann (2001) 565–570

[17] Yu, T., Miller, J.F.: Finding needles in haystacks is not hard with neutrality. In Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B., eds.: Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002. Volume 2278 of LNCS., Kinsale, Ireland, Springer-Verlag (2002) 13–25

[18] Mitchell, T.M.: The need for biases in learning generalizations. Technical Report CBM-TR-117, New Brunswick, New Jersey (1980)

[19] Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge, MA, USA (1994)

[20] Teller, A.: Algorithm Evolution with Internal Reinforcement for Signal Understanding. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA (1998)