Scaling of Program Fitness Spaces

William B. Langdon
Centrum voor Wiskunde en Informatica
Kruislaan 413
NL-1098 SJ, Amsterdam, The Netherlands
W.B.Langdon@cwi.nl

Abstract
We investigate the distribution of fitness of programs concentrating on those represented as parse trees and, particularly, how such distributions scale with respect to changes in the size of the programs. By using a combination of enumeration and Monte Carlo sampling on a large number of problems from three very different areas, we suggest that, in general, once some minimum size threshold has been exceeded, the distribution of performance is approximately independent of program length. We proof this for both linear programs and simple side effect free parse trees. We give the density of solutions to the parity problems in program trees which are composed of XOR building blocks. Limited experiments with programs including side effects and iteration suggest a similar result may also hold for this wider class of programs.

Keywords
Stochastic search, genetic algorithms, genetic programing, tree fitness landscapes, NAND gates, Monte Carlo sampling, symbolic regression, Santa Fe trail, artificial ant.

1 Introduction
The use of genetic algorithms and other stochastic search techniques to automatically generate programs which solve problems has become increasingly popular, yet, we know almost nothing about the distribution of solutions within these vast search spaces. They are neither continuous nor differentiable, so classical search techniques are incapable of solving our problems. Instead heuristic search techniques, principally stochastic search techniques, have been used. The term genetic programming (GP) defines techniques which evolve suitable programs by stochastic search of the space of possible programs. Presently, the theoretical foundations of GP are weak, and little is known about the space which it searches, particularly, how it scales. We suggest that, in general, above some problem dependent threshold, program fitness shows little variation with size. The distribution of fitness levels, particularly the distribution of solutions, directly gives us the performance of random search. We can use this as a benchmark against which to compare GP and other techniques. We can also compare the density of solutions in parse trees with those in linear programs. Our analysis shows there is much more variation in big trees than in long linear programs. Some functions are much more common and some much more rare.

We test our claim using a combination of enumeration and Monte Carlo sampling described in Section 2. Tests are performed on: 66340 of the Boolean problems (Sections 3 and 4), a continuous domain symbolic regression benchmark problem (Section 5), and a...
2 Experimental Method

For very short programs it is feasible to generate and test every program of a given length. The length of a tree program is the size of the tree, the number of internal nodes in the tree plus the number of leaves. However, as Figure 1 shows, the number of possible programs grows very quickly with their size and so we must fall back on randomly sampling programs. We use the random tree method given in Alonso and Schott (1995) to uniformly sample all programs of a specific length. Typically, we sample 10,000,000 programs of each length.¹

The ramped-half-and-half method (Koza, 1992, 93) is commonly used to generate the initial population in genetic programming (GP). Half the random programs generated by it are full, i.e., every leaf is the same distance from the root. Therefore, we also explicitly consider the subspace of full trees. In some cases this subspace is radically different from the whole space.

¹The C++ code used to generate random programs is available online at the following address. ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/rand_tree.cc.
3 Boolean Functions

Boolean functions have often been used as benchmark problems. The program trees we consider are composed of \( n \) terminals (D0, D1, ..., D_{n-1}) (the Boolean inputs to the program) and four sets of the Boolean logic functions: NAND, XOR, \{AND, OR, NAND and NOR\}, and \{AND, OR, NAND, NOR and XOR\}. There are \( 2^n \) Boolean logic functions of \( n \) inputs. NAND alone is sufficient to construct any of them and, therefore, so are the last two sets. XOR by itself can only generate \( 2^n \) of them but adding it to the function set can dramatically affect the whole search space. The fitness of each tree is given by evaluating it as a logical expression for each of the \( 2^n \) possible combinations of \( D_n \) inputs. Its fitness is the number of fitness cases when its output agrees with that of the target Boolean function (Koza, 1992).

There are \( n^{(l+1)/2} F^{(l-1)/2} \) different trees of length \( l \) (Koza, 1992; Alonso and Schott, 1995, 213). \( |F| \) is one, four, or five, depending on which of the four function sets is used. This formula is simple, because each function (internal node) has two arguments. The number of programs rises rapidly, approximately exponentially, with increasing program length \( l \) (see Figure 1). If no bounds are placed on the size or depth of programs, then the number of them is unbounded, i.e., the search space is infinite.

4 Boolean Program Spaces

4.1 NAND Program Spaces

Due to the ease of manufacture of NAND gates in integrated semiconductors, and because any Boolean function can be constructed from a network of NAND gates, they are the principal active component in digital electronics. It is, therefore, an interesting function to study. The following sections give the number of different program trees composed only

Figure 2: Proportion of NAND trees which yield each 2-input logic function.
of NAND functions that are equivalent to each Boolean logic function. We specifically look at the program trees composed only of NAND of various sizes for two, three, and four inputs.

4.1.1 2-input NAND Program Spaces

There are $2^{2^2} = 16$ Boolean functions of two inputs. The proportion of NAND trees which evaluate to each is plotted in Figure 2.

Not surprisingly, there are two peaks at size 1 of height 0.5 which correspond to the functions D0 and D1 and no other functions are possible. There is also a peak of the same height for size 3, which is NAND itself, and for two smaller peaks for ND0 (not D0) and ND1 (not D1). Three functions can be constructed from trees of size 5, i.e., two NAND gates and three inputs. Seven from size 7 and so on. It is not until tree size 15 (7 NAND gates) that all of the 16 possible functions can be constructed from a tree of one length, although all can be constructed from trees of size 13 or less. XOR can be fabricated from 5 NAND gates and 6 terminals in a nearly full tree with 11 nodes and a height of 4.

Table 1: Proportion of Two Bit NAND Programs and their fitness values shown for two problems. Lower table gives proportion by fitness value for the two problems.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Proportion</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Always-On</td>
</tr>
<tr>
<td>0</td>
<td>.00490</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.00415</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>.01689</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>.10710</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>.01696</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>.10745</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>.01430</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.15121</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>.03695</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>.01088</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>.07727</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>.10920</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>.07702</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>.10898</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>.04753</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>.10922</td>
<td>4</td>
</tr>
</tbody>
</table>

From Figure 2 we also see that each of the functions quickly converges towards some limiting proportion of the NAND trees of a given length. Thus we can give the proportion of the search space (formed by NAND, D0, and D1 trees) occupied by each of the sixteen
functions (see Table 1).

Each of the $2^n = 16$ Boolean functions can be regarded as a problem with its own fitness function. As indicated in Section 3, each fitness function has $2^n + 1 = 5$ values. A simple problem is to find a tree which always returns 1 (the Always-on problem). This corresponds to function 15 which has the maximum fitness value of 4. The fitness value of the other functions are given in column 4 of Table 1. Column 5 gives their fitness values for the Odd-2-parity (XOR) problem. The lower part of Table 1 gives the fraction of the search space with a particular fitness value for the two example problems. Note that the relationship between function and fitness on a given problem is fixed. Therefore, since the proportion of the search space occupied by each function does not change with respect to length above the threshold, the distribution of fitness values is independent of length above the same threshold. This is true for all possible 2-input problems.

Functions 10 and 12 (DO and D1) are equivalent to each other in the sense that the other is produced by exchanging inputs. Function 10 and 12 are equally common in the search space. There are three other pairs of equivalent functions, 2 and 4, 3 and 5 (ND1 and ND0), and 11 and 13. It is also apparent that lexical ordering of functions is not the most convenient form of graphical presentation. In other plots they will be presented in order of decreasing frequency and only data for one function of an equivalent set will be plotted.

4.1.2 3-input NAND Program Spaces

There are $2^3 = 256$ Boolean functions of three inputs. However, many of these are equivalent to each other. Taking this into account, these reduce to 80 classes. In Figure 3 we plot the proportion of NAND trees which evaluate to each of these classes using the class ordering given in Koza (1992, Table 9.2).

Comparing Figures 2 and 3 we see that they have several features in common. In
particular, the proportion of each function changes initially with increasing program length, but once some threshold has been exceeded, it changes scarcely at all with size. The variation between each function is far greater than when using only two inputs.

4.1.3 4-input NAND Program Spaces

There are $2^{2^4} = 65536$ Boolean functions of four inputs. Again, many of these are equivalent to each other, so these reduce to 4176 classes. In Figure 4 we plot the proportion of NAND trees which evaluate to each of these classes. The ordering of the classes is given by their measured frequency in trees of size 255.

Comparing Figure 4 with Figures 2 and 3 we see the same common features. While the proportion of each function changes initially, if we look at programs containing 16 or more NAND gates, i.e., size 31 or more, the proportion scarcely changes with size. This appears to be true for all 65536 functions, but as the data is based on Monte Carlo sampling, the data for the rarer functions is correspondingly noisy.

Again, the variation between each function is far greater than when using just two or three inputs. None of the functions in the 842 rarest equivalence classes were discovered in Monte Carlo sampling of 10,000,000 programs of length 255. This includes both the odd and even parity functions with 4 inputs. Neither were discovered in any of the 20 Monte Carlo runs (each sampling 10,000,000 points).
Figure 5: Proportion of functions in each equivalence class \{AND, OR, NAND and NOR\}.

4.2 3-input AND OR NAND NOR (XOR) Boolean Program Spaces

In this section we consider all the Boolean functions for \(n = 3\) when using the larger function sets: \{AND, OR, NAND and NOR\} and \{AND, OR, NAND, NOR and XOR\}. As in Section 4.1.2, there are 256 functions but they can be split into 80 equivalence classes.

Comparing Figure 5 with Figure 3 we see that the bigger search space shares many characteristics with that produced by NAND. In particular, it shows that a certain minimum size is required before the problem can be solved and that the minimum size depends on the difficulty of the problem. Once this threshold size is exceeded, the proportion of programs belonging to the equivalence class grows rapidly to a stable value which appears to be independent of program size. Figure 6 shows that these characteristics are retained if we extend the function set to include XOR. Note that adding the XOR function radically changes the program space. In particular, the two parity functions (equivalence classes 79 and 80) are much more prevalent. The range of frequencies is also reduced. For example, 68 of the 80 equivalence classes have frequencies between 0.1/256 and 10/256 and only 28 with the standard function set.

While Figures 5 and 6 can be used to estimate the fitness space of each 3-input Boolean function across the whole space, there are some interesting areas of these spaces where certain functions are more concentrated. There are far more parity functions among the full trees than there are elsewhere. When XOR is added to the function set, there is a higher proportion of parity functions, but the difference between the full trees and the rest of the search space is less dramatic.

4.3 6-input Boolean Program Spaces

In this section we investigate the distribution of 6-input Boolean functions using the two larger function sets: \{AND, OR, NAND and NOR\} and \{AND, OR, NAND, NOR and XOR\}.
Figure 6: Proportion of functions in each equivalence class {AND, OR, NAND, NOR and XOR}.

Figure 7: Proportion of 6-input Boolean functions {AND, OR, NAND and NOR} by number of ones returned (note linear scale).
Scaling of Program Fitness Spaces

Figure 8: Even-6-parity program space {AND, OR, NAND and NOR}. XOR. It is difficult to analyze all the Boolean functions with more than four inputs. Instead we have concentrated the easiest and hardest Boolean functions of six inputs: the always-on-6 function and the even-6-parity function. Figures 7 and 8 show the proportion of programs of various lengths with each of the possible scores. Figure 9 shows the same when XOR is added to the function set. Always-on-6 and even-6-parity, both with and without XOR, have the same near independence of fitness from length.

The fitness distribution of the even-6-parity problem is much tighter than that of the binomial distribution produced by selecting Boolean functions uniformly at random from the $2^{2n}$ available (centered on $\frac{n}{2}$ with variance of $\frac{n}{4}$ (Rosca, 1997, 62). The measured variance is only 0.12 rather than 1.5. Such a tight fitness distribution and the absence of a high fitness tail suggests that the problem will be hard for adaptive algorithms. When discussing the evolution of evolvability, Altenberg (1994) assumes that high fitness tails exist and can be found by evolutionary search algorithms. We would hope that they would be better than random search at finding and exploiting such tails.

Adding XOR to the function set greatly increases the even-6-parity fitness distribution's width and it retains its near independence of program size (see Figure 9). The standard deviation is now 0.92 However, the more dramatic the effect of the wider distribution, the more feasible it is for our Monte Carlo simulations to find solutions, i.e., programs scoring 64 hits. They occupy about $2 \times 10^{-7}$ of the whole search space.

Figure 7 shows that the distribution the of number of trues returned is a saw-toothed curve. The proportion of programs which have one of the odd scores on the always-on-6 problem is about 0.3%. The proportion having an even score, not divisible by four, is about 1%. Scores divisible by 4 are about 2%, by 8 are 3%, by 16 are 6%, and by 32 are 10%. Note the central peak in the even-6-parity fitness distribution (see Figure 8) is not solely due to a large number of programs which implement always-on-6 or always-off-6. Only 18.6% of programs are of these two types.
Figure 9: Even-6-parity program space (AND, OR, NAND, NOR and XOR).

The distribution of the number of trues returned when XOR is added to the function set is slightly changed but retains its saw-toothed appearance and near independence of program size.

4.4 Even-6-Parity and Always-On-6 Full Trees

Restricting our search to only full trees yields a similar fitness distribution for the even-6-parity problem (see Figure 10). In particular, we have the convergence of fitness distribution once the tree size exceeds a threshold. There is a small variation with size but it does appear to decrease as we consider bigger trees. The distribution of fitness values observed is considerably wider with a range of 25–38 (see Figure 8) and a standard deviation of 0.68. Adding XOR to the function set further widens the distribution and the standard deviation becomes 1.8.

Searching only the full trees yields a similar fitness distribution for the always-on-6 problem as for the whole search space. However, the peaks corresponding to functions returning true multiples of 4, 8, 16, or 32 times are less prominent. Always-on-6 and its complement, always-off-6, now dominate and together represent 35% of all trees compared to 18% when considering asymmetric trees as well. Also, the troughs at odd numbers of hits are less prominent, each representing about 0.5% rather than 0.3% of all programs. Adding XOR to the function set has the effect of further smoothing the distribution. The peaks at either extreme are now 8% with a typical odd value near 32 of 1.4% and an even value of 1.8%. With and without XOR, the distribution of the number of trues returned by full trees shows some dependence on depth of tree. However, as with even-6-parity, this appears to fade away as the programs become bigger.
Symbolic regression is the problem of finding, in a functional form, a model of some dataset. This is a very common requirement and techniques such as linear regression are widely used. Often such models are used to predict values for unknown data points. Where data are complex more complex regression techniques are needed. We investigate a benchmark symbolic regression problem in this section.

We use the sextic polynomial regression problem (Koza, 1994, 110–122). The sextic polynomial is $z^6 - 2x^4 + z^2$, which can be rewritten as the product of three squares, i.e., $x^2(x - 1)^2(x + 1)^2$. The problem for GP is to match it over the range -1.0...1.0 using +, -, x, protected division, the input z, and random constants. We used 250 random constants chosen from the 2001 numbers between -1 and +1 with a granularity of 0.001. No constant was repeated and none of the three special values of -1, 0, or 1 were included. The 50 test points used were chosen uniformly at random from the range -1 and 1. No granularity was imposed. Again, none of the three special values of -1, 0, or 1 where included and no value was repeated. Apart from limiting ourselves to 250 constants, this is as described in Koza (1994, 110–122).

5.1 Sextic Polynomial Fitness Function

The fitness of each program is given by its absolute error over all the test cases (Langdon et al., 1999). This is as described by Koza (1994) except we divide by the number of test cases (50) to yield the average discrepancy between the value it returns and the target value. All calculations were performed in standard floating point representations.
5.2 Sextic Polynomial Fitness Distribution

The distribution of fitness is shown in Figure 11. It is apparent that symbolic regression shares many of the characteristics of the more difficult Boolean problems. The proportion of good programs is very small, but above a small threshold, the proportion of good programs in a given fitness range converges to a value which is independent of their size. Figure 11 shows that the proportion of very bad programs does show variation with respect to length. However, it appears to reach a stable value, but the threshold length is bigger. Each test case where a floating point exception occurs is given a penalty of about 2,000 (Langdon et al., 1999). Figure 11 suggests bigger programs are more likely to cause floating point exceptions, but when programs are big enough this proportion also converges to a limit. In practice, such programs have little effect as they are never selected to be parents of the next generation.

6 Artificial Ant

Our last example is the most complex in some ways. We report the distribution of fitness in a benchmark GP problem which combines both side effects and iteration. The problem chosen is the artificial ant following the Santa Fe trail. The program tree is repeatedly executed during which it controls an artificial ant using the side effects of special leaves. The value returned by the root node is ignored. It also includes functions with more than two arguments. A more complete description, including schema analysis, of this particular problem search space may be found in Langdon and Poli (1998). Here we concentrate upon variation with respect to length.

Figure 12 suggests that, provided programs exceed some small fitness dependent threshold, the distribution of program fitnesses is roughly independent of their size. Much of the fluctuation seen at the higher fitness levels is due to sampling noise inherent in
Monte Carlo measurements.

In Figure 13 (solid line) we present the data for just one fitness value—the solutions. There are no solutions with less than ten nodes. From 11 to 18 nodes, the proportion of solutions in the search space rises rapidly (but not monotonically) to a peak from which it falls. For programs with more than 30 nodes, the concentration of solutions appears to change slowly with program size. It appears to eventually fall to near zero.

In the standard Santa Fe problem, the function set includes both Prog2 and Prog3 functions, however, it is not necessary to have both to solve the problem. The two dashed lines in Figure 13 show the density of solutions when only one of them is included. While the data are subject to sampling noise, it appears that both subspaces formed by excluding one or other of the Prog functions are richer in solutions than the original one. Again, in both subspaces it appears there is only very slow variation in the density of the maximum fitness value with respect to length above some threshold (about 50 or 100 nodes).

7 Long Random Linear Programs

In this section we will proof that, provided certain assumptions hold, each output generated by long random linear programs is equally likely and that this is true regardless of the program's inputs and its length. Consequently, the chance of randomly finding a solution decreases exponentially with the size of the test set.

The state of a computer is determined by the contents of its memory. For our purposes, registers, condition flags, etc., within its CPU and input and output registers, are regarded as part of its memory but we exclude the program counter (PC). If it has $N$ bits of memory it can be in up to $2^N$ states. Program execution starts with all memory initialized. Execution of each program instruction moves the computer from one state to another. Usually the
With the two new function sets, there is little variation with length (note log scale) for longer programs. There may be some variation when using the combined function set (solid line) above 400.

next state will be different but it need not be. For example, an instruction that sets a register to zero will not change the state if the register is already zero. We assume there is at least one such state and instruction.

We assume the designer of the computer (or GP experiment) has ensured that it is possible to reach every state. This is done since 1) inaccessible states correspond to unusable memory, i.e., to inefficient use of hardware, and 2) it makes it is possible to transform any input to any output.

Finally we assume the instruction set is symmetric—if there is one instruction which moves the computer from one state to another there is also an instruction which moves it in the other direction.

Consider a program as a sequence of instructions each of which transforms the computer's state. In particular, consider the case where the program contains \( l \) instructions chosen at random. The computer starts from an initial state given by the program's inputs and terminates \( l \) states later. The program's output is then the state of the computer's outputs. Note: the program itself need not be linear; it can contain branches, loops, function calls, etc. provided it executes state changing instructions at random and terminates after \( l \) of them. We exclude the program counter (PC) from the machine's state so its contents and thus the address of the next instruction need not be random. Program termination could be forced by an external event or by fixing a path \( l \) instructions long for the program counter,
cf. linear GP (Nordin, 1997).

We can represent the computer by a probability vector of length $2^N$. When executing a fixed program, the computer will be in exactly one state at a time, i.e., its probability vector will contain one element of 1.0 and the rest will be zero. We can view each instruction as an $2^N \times 2^N$ matrix which when multiplied by the current probability vector (state) yields the next probability vector (next state). The elements of the matrix are either zero or one and there is exactly one “1” in each row. Therefore it is a stochastic matrix (Feller, 1970, 375).

If we consider all possible programs of length $l$, we can define the average state at a particular time as the mean probability vector $u$ at that time. In a particular program the next state is given by the current state vector multiplied by a particular matrix. When considering all programs, we can say the average next state is given by the mean probability vector multiplied by the average instruction matrix. Note the next state is given by the current state alone. It does not depend upon earlier events. Thus, on average, the state of the computer can be represented by a Markov process. The Markov transition probability matrix $M$ is the mean of all the instruction state matrices.

Since $M$ is the mean of stochastic matrices, it will also be stochastic. At least one of the elements on its diagonal will be greater than zero. The period of this state will be 1, i.e., it will be aperiodic (Feller, 1970, 387). Therefore, the greatest common divisor (g.c.d) of all the states is 1. Since any state can be reached, $M$ is irreducible. Thus $M$ corresponds to an irreducible ergodic Markov chain so $u$ will tend to a limit $u_\infty$ independent of the starting state (the program’s inputs) as the length $l$ of the program is increased (Feller, 1970, 393).

Because the instruction set is symmetric, $M$ will be symmetric and both its rows and columns will each sum to 1.0 ($M$ is doubly stochastic). Therefore, in the limit all states are equally probable (Feller, 1970, 399), i.e., there is a limiting probability distribution for the states of the computer and at the limit each state is equally likely. If the instruction set is asymmetric, there is still a limit but the states are no longer equally likely.

### 7.1 The Chance of Finding a Solution

We define a solution to mean that the program passes all the test conditions. As an example, suppose there are $T$ non-overlapping tests. Each test specifies a number of input bits and a target output pattern of $n$ bits. After executing a long random program, each final state is equally likely. In particular, each combination of bits in the output registers is equally likely. Thus the chance of generating exactly the target bit pattern is $2^{-n}$. There are many functions which implement this transformation, and there is equal chance of randomly selecting any of them. In particular, if the test cases don’t overlap, the chance of selecting a function which also passes the second test case is also $2^{-n}$. Therefore, the chance of finding a program which passes both the first and second test case is $2^{-2n}$. Generalizing, the chance of finding a program which passes all $T$ tests (solves the problem) is $2^{-nT}$. This can be viewed as: the chance of finding a solution is given by the information content of the the test set ($nT$ bits).

### 7.2 An Illustrative Example

Consider a system with two Boolean registers $R_0$ and $R_1$. At the start of program execution each is loaded with an input. When the program terminates, its answer is given by $R_0$. There are eight instructions:
There are \(2^2 = 4\) states \((R_0R_1 = 00, 01, 10, 11)\).

If we use each of the instructions with equal probability the Markov transition matrix is the average of all 8:

\[
M = 1/8 \begin{pmatrix}
4 & 2 & 2 & 0 \\
2 & 4 & 0 & 2 \\
2 & 0 & 4 & 2 \\
0 & 2 & 2 & 4 \\
\end{pmatrix}
\]

We can see \(u_\infty = 1/4(1, 1, 1, 1)\) satisfies \(u_\infty M = u_\infty\), so it is the limiting probability distribution. Alternatively, we can proof this by noting \(M\) is symmetric and hence doubly stochastic. It has at least one non-zero diagonal term. Thus, the theorem from Feller (1970, 399) holds, and in the limit all states are equally probable.

The eigenvalues \(\lambda\) and corresponding eigenvectors \(E\) of \(M\) are

\[
\lambda_{00} = 1/2 \begin{pmatrix} 0 & -1 & 1 & 0 \end{pmatrix}, \quad \lambda_{01} = 1/2 \begin{pmatrix} -1 & 0 & 0 & 1 \end{pmatrix}, \quad \lambda_{10} = 1 \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}, \quad \lambda_{11} = 0 \begin{pmatrix} 1 & -1 & -1 & 1 \end{pmatrix}
\]

Note since \(M\) is symmetric the other eigenvalues are also real.

### 7.3 Rate of Convergence and the Threshold

The eigenvectors \(E\) form an orthonormal set, so any vector can be expressed as a linear combination \(w\) of them: \(u = wE (w = uE^{-1})\), where \(E\) is the \(n \times n\) matrix formed by the \(n\) eigenvectors of \(M\) and \(E^{-1}\) is its inverse. \(EM = \lambda E\), where \(\lambda\) is the \(n \times n\) diagonal matrix formed from the eigenvalues of \(M\). Thus, the probability vector of the next state is
Scaling of Program Fitness Spaces

\[ u_1 = uM \]
\[ wEM \]
\[ w\lambda E \]
\[ w_1 E \]

where \( w_1 = w\lambda \). The new probability vector \( u_1 \) can also be re-expressed as a linear combination of the eigenvectors of \( M \). It is actually \( w_1 \). Note that the components have shrunk by a factor given by the corresponding eigenvalue.

The probability distribution of the second state is \( u_2 = u_1 M = w\lambda EM = w\lambda^2 E \) and so that of the \( i \)th state is \( u_i = w\lambda^i E \).

As \( i \) increases only the components with the largest eigenvalues will survive. Other components will quickly vanish exponentially, i.e., the probability distribution will converge to the limit. The slowest terms to be removed that are not part of the limiting distribution are given by the eigenvectors corresponding to the second largest eigenvalue \( \lambda_2 \). The number of steps (the threshold size) is dominated by the magnitude of this eigenvalue.

If we require the largest transient term to be less than some \( \epsilon > 0 \) then

\[
\lambda_2^h < \epsilon \\
h \log(\lambda_2) < \log(\epsilon) \\
h > \log(\epsilon)/\log(\lambda_2) \\
\text{Threshold size} = O(1/\log(\lambda_2))
\]

Returning to our example, suppose both inputs are 0. Then \( u = (1, 0, 0, 0) \). From \( M \) we can calculate \( \lambda = (1/2, 1/2, 1, 0) \), \( E \) and \( E^{-1} \).

\[
E = \begin{pmatrix}
0 & -1 & 1 & 0 \\
-1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & -1 & -1 & 1
\end{pmatrix}
\]

\[
E^{-1} = 1/4 \begin{pmatrix}
0 & -2 & 1 & 1 \\
-2 & 0 & 1 & -1 \\
2 & 0 & 1 & -1 \\
0 & 2 & 1 & 1
\end{pmatrix}
\]

\[
w = uE^{-1} = (0, -1/2, 1/4, 1/4)
\]

\[
w_1 = w\lambda = (0, -1/4, 1/4, 0)
\]

\[
u_1 = w_1 E = (1/2, 1/4, 1/4, 0)
\]

\[
u_2 = w\lambda^2 E = (0, -1/8, 1/4, 0)E
\]

\[ = (3/8, 1/4, 1/4, 1/8) \]

Note all elements of \( u_2 \) are already with 50% of their limit values. In this example the other eigenvalues are not close to 1.0, so convergence is rapid. \(-1/\log(\text{next largest eigenvalue}) = -1/\log(1/2) = 1.442695\).
8 Big Random Tree Programs

In this section we will prove that, provided certain assumptions hold, there is a limiting distribution for functionality implemented by large binary tree programs. Initially we allow functions with any number of inputs but later sub-sections simplify this by only considering binary functions.

8.1 Introduction

We wish to establish a similar result for tree programs that we have already shown for linear programs. We will start by assuming that all the state information is held within the tree (the functions have no side effects) and that the output of the program is returned only via its root. It should be possible to combine the two results to include trees with external memory. We begin with an example tree.

Consider the expression \((x - y/3) \times (10 + 2)\). It has four functions (\(-, /, \times,\) and \(+\)) and five leaves \((x, y, 3, 10, \) and \(2)\). We represent it as a tree (see Figure 14). It would usually be interpreted depth first by evaluating \(x\), then \(y/3\), then \((x - y/3)\), then \(10 + 2\), and finally multiplying them together. However, as there are no side effects, the expression can be evaluated in a variety of different orders all of which yield the same answer. In particular the tree can be evaluated from the deepest node upwards—evaluate \(y\) first, then \(y/3\). As we reach each new function node in the tree we have to stop and save the value we have calculated until the value of the function's other arguments have also been calculated. When one execution thread is blocked we create a new one from the next unprocessed leaf, in this case, 10 and then 2. We calculate \(10 + 2\), but when this new thread reaches the \(\times\) node, it also has to stop, so we start a new thread from \(x\). When it reaches the \(-\) node, all its arguments are known, and we can restart the deepest path thread. It calculates \(x - y/3\) and moves up the tree. Upon reaching the root (\(\times\)), all its arguments are known and we can perform the final multiplication using the current value and the previously stored result of \(10 + 2\). This is the value of the tree and evaluation halts.

We will now repeat the analysis used for linear programs. The program is a tree, but we will concentrate on the longest path within the tree (shown with thick lines in Figure 14).

The state of the program at each point in this path is determined by the current value (we are excluding additional memory in this paper). Its initial value is given by the starting leaf. Each function along the way to the root will potentially change the value, and the new value will be propagated towards the root. If the function has more than one argument then, in general, before we can determine the transformation the function will make, we will have to evaluate all its other arguments.

In our example, the first function we reach is \(\text{divide}.\) \(\text{Divide}\) has two arguments.
However, once we have calculated the value of its other argument (it is 3 in our example) we can treat \textit{divide} as a function with one argument. Note the transformation at a given step will change each time the program is executed with different inputs.

This framework is similar to the linear case in that, in random trees, we can view each function along the longest path, in conjunction with its other inputs, as causing a random transformation of the current value. However, the process is not a Markov chain, because even for random trees, the transformation matrices change as we get further from the leaf.

Let \( u \) be a vector whose elements correspond to each of the possible values. For example if we are dealing with an integer problem then there are \( 2^{32} \) possible values \((n = 2^{32})\) and \( u \) has \( 2^{32} \) elements. \( u \) is the probability density vector of the current value. For a given program at a given time, one element of \( u \) will be 1.0 and all others will be zero.

For each function of arity \( a \) there is \( n^{a+1} \) hypercube transformation matrix. When all the inputs to the function are known, they are converted to \( a \) probability vectors. By multiplying the transformation matrix by each of them in turn, we get the output probability vector. The output of the function is given by the non-zero element in the output probability vector.

Since we treat the current path separately from the function's other inputs, we split its \( n^{a+1} \) hypercube transformation matrix into \( a \) \( n^{a+1} \) transformation matrices, one for each argument. First, we determine which argument of the function corresponds to the current path. We then choose the corresponding transformation matrix and multiply it by each input in turn, excluding the current path. This yields a \( n \times n \) matrix which, when multiplied by the current probability vector, yields the next probability vector. In other words, each binary function has \( 2 n \times n \) transformation matrices. If the current path is the function's first argument, we use the first matrix, otherwise we use the second matrix.

We now start the analysis of random programs. Starting from a leaf, the non-zero element of \( u \) will correspond to one of the inputs to the program. For simplicity we will treat constant values as being inputs to the program. We define the average value of \( u \) as the mean of its values across all possible programs. Initially, all the elements of \( u \) which correspond to one of the input values will be non-zero and all other elements will be zero. Call this \( u_0 \) (the first value of \( u \))

We then come to a particular arity \( a \) function as its \( i^{th} \) input. The probability distribution \( u_1 \) after the first function is given by \( u_0 M_1 \) where \( M_1 \) is the transformation matrix corresponding to the \( i^{th} \) input of the first function. Since this is the deepest function, the other branches must also be random leaves, so their probability distributions will also be \( u_0 \). Thus, \( M_1 = u_0^{a-1} N_{af_i} \), where \( N_{af_i} \) is the \( n^{a+1} \) hypercube transformation matrix for function \( f \) input \( i \) (\( a \) indicates its arity). On average, the new probability density function will be the mean of all functions of arity \( a \). Also, on average, the path we have chosen is equally likely to reach any of the inputs of \( f \), so we can also average across all values of \( i \). Let \( N_a = 1/f_a \sum_{j=1}^{f_a} 1/a \sum_{k=1}^{a} N_{ajk} \) where \( f_a \) is the number of functions of arity \( a \) in the function set. On average, \( M_1 = u_0^{a-1} N_a \), and \( u_1 = u_0 M_1 \).

### 8.2 Large Binary Trees

To avoid the complexity associated with considering multiple function arities we will assume that all internal nodes are binary. We define \( N = N_2 \), so \( M_1 = u_0 N \).
The probability vector \( u_1 \) is now propagated up the tree to the next function. Its other argument is determined, \( N_{2jk} \) is multiplied by the vector to yield \( M_2 \). Again, we have to average across all programs so we use the mean \( N \) transformation matrix. The functions’ arguments may be either leaves or trees of height one. If they are leaves, their probability vector will be identical to \( u_0 \) since they are also random. If they are functions, the probability vector is the same as in the longest path, because the same functions are equally probable in each branch. So, \( u_1' = u_1 = u_0M_1 \). Let \( p'_{2h} \) be the average number of functions which are children of the function at distance 2 from the leaf along the longest path (excluding those on the longest path). Also, \( p''_{00} = 1 - p'_{2h} \) is the average number of children which are leaves. Thus, \( M_2 = p'_{00}u_0N + p'_{2h}u_1N \). Note \( M_2 \neq M_1 \) and so the process is not Markov.

\[
\begin{align*}
M_1 &= u_0N \\
u_1 &= u_0u_0N \\
M_2 &= p'_{00}u_0N + p'_{2h}u_1N \\
M_2 &= p'_{00}u_0N + p'_{2h}u_0u_0NN
\end{align*}
\]

At the third level,

\[
M_3 = p'_{000}u_0N + p'_{2h0}u^2_0NN + p'_{22h0}(u^2_0N)(u_0N)N + p'_{222}(u^2_0N)(u^2_0N)N,
\]

where \( p'_{ijk} \) refer to the proportion of children of the fourth node along the longest path (but excluding those on the longest path). \( p'_{000} \) is the proportion of programs where there is only one child, so it is a leaf. \( p'_{2h0} \) is the fraction where the child is a function but both its children are leaves. \( p'_{22h0} \) is the fraction where the child has one child which is a function and \( p'_{222} \) is the remainder, i.e., \( p'_{222} \) is the fraction of programs where the other child of the fourth node is a full subtree as deep as the child on the longest path.

The \( n \times n \) sub matrices of \( N \) are stochastic. \( u_0N \) for arbitrary \( u_0 \) will also be stochastic. \( u_0 = \sum w e \) where \( e \) are the eigenvectors of \( u_0N \) with corresponding eigenvalues \( \lambda \). Note \( |\lambda| \leq 1 \). Thus, \( v_1 \), the probability distribution produced by a random full binary tree of depth 1 is:

\[
\begin{align*}
v_1 &= u_0u_0N \\
&= \sum w e u_0N \\
&= \sum w \lambda e
\end{align*}
\]

\( v_1 \) is closer to (or at least no further away from) the limiting distribution of \( u_0N \) than to \( u_0 \) itself. Similarly, \( v_2 \), the distribution produced by full trees of depth 2 is

\[
\begin{align*}
v_2 &= v_1v_1N \\
&= \sum_i w_i \lambda_i e_i v_1N \\
&= \sum_i w_i \lambda_i e_i u_0u_0NN
\end{align*}
\]
Scaling of Program Fitness Spaces

\[
\begin{align*}
&= \sum_i w_i \lambda_i e_i \sum_j w_j \lambda_j u_0 N N \\
&= \sum_i w_i \lambda_i e_i \sum_j w_j \lambda_j N \\
&= \sum_i \sum_j w_i \lambda_i w_j \lambda_j e_i e_j N 
\end{align*}
\]

Since \(|\lambda| \leq 1\) \(v_2\) is also closer to the limiting distribution \(e e N\), where \(e\) is the eigenvector of \(u_0 N\) with the largest eigenvalue. If \(u_0 N\) has a diagonal element greater than zero, there will be just one eigenvalue with a modulus of 1 and the limiting distribution \(v_\infty\) will not cycle. The matrix components corresponding to asymmetric subtrees are similarly bounded. Thus, as we go higher in the tree, the matrix elements of the higher order components of \(M_i\) will be bounded.

As we go higher in the tree, the number of branches we encounter that are the same length as the longest one falls rapidly, i.e., the higher coefficients \(p'\) are not only bounded but the higher order ones vanish. In large random binary trees the chance that the other subtree is empty tends to a constant value of approximately 0.5 (Sedgewick and Flajolet, 1996, 241). \(p'_{0,0}\) tends to 0.5 as we get further from the leaf (as \(i\) increases), so \(M_i\) is dominated by the first few terms. While \(M_i \neq M_1\), \(M_{i+1} = M_i\), so the current value along the longest path will become Markovian. Therefore, the probability distribution of the output of large random trees does not depend upon their size. From the eigenvalues and eigenvectors of \(M_\infty\), we can calculate the probability distribution of the output of large random trees and the size of the threshold. Unlike linear programs, these may depend upon the programs inputs.

8.3 An Illustrative Example

We take the same four Boolean functions as before (AND, NAND, OR, and NOR) and apply them to a Boolean regression problem of \(n\)-bits. Each is a binary function of one bit input and has two \(2 \times 2 \times 2\) transition matrices. Since each function is symmetric, the two matrices are the same in each case.

As for each function, we also include the function with the opposite output (AND and NAND). The mean matrix has the same value at every element. This means \(M_1\) is independent of \(u_0\) with every element being the same—\(M_1\) is irreducible, doubly stochastic, and has non-zero diagonal elements. Thus, the output of the first random function is independent of its inputs and is equally likely to be 0 or 1. Since the coefficients \(p'\) sum to 1, this is true for every \(M_i\). The output of the whole tree is independent of its inputs and is equally likely to be 0 or 1 regardless of the size of the tree.

8.4 The Chance of Finding a Solution

Because random trees have some inputs near their root, they are more likely to implement program functionality which needs few operations on the inputs compared to functions chosen at random. We can repeat the analysis in Sections 8.1–8.2 but replace the current value with the current functionality. Instead of considering the value output at each node in the tree, we use an index number of the function implemented by the subtree rooted at that node. For example, in an \(i\)-bit-input \(m\)-bit-output problem, there are \(2^m 2^i\) possible
W. Langdon

Figure 15: Distribution of 2-input functions produced by random trees comprised of AND, NAND, OR, and NOR. The solid line gives the measured distribution. Dotted lines are numerical predictions based on simplified models of random trees.

functions and each can be given an \( m^2 \) bit index number. The number of possible values \( n = 2^{m^2} \). The \( n^{o+1} \) hypercube transformation matrices \( N_{a+1} \) now operate on function index values rather than actual values, but we can define average behavior as before, and the functionality of large random trees tends to a distribution which is independent of the tree size. The distribution and the threshold size are given by the eigenvalues and eigenvectors of the limit value of \( M \).

8.5 A Second Illustrative Example

Returning to the example in Section 8.3, the transformation matrices now depend upon the order of the problem. In the case of two inputs, there are \( 2^{12^2} = 16 \) possible functions, so each function has two \( 16 \times 16 \times 16 \) transition matrices. The functions are still symmetric, so we need only consider one of each pair.

However, this does not mean that each function is equally likely when we consider that the average function transition matrix is irreducible, stochastic, has non-zero diagonal elements, and is not symmetric. Thus, there is a limiting distribution independent of the inputs, but it is not uniform. It has several non-zero eigenvalues, so while convergence is rapid, it is not instantaneous. \( M_\infty \) depends on the distribution of \( P \) (subtree sizes). In Figure 15, a measured distribution is compared to the function distribution produced assuming a simple model in which functions within the trees have one branch which is a leaf. The distribution for full trees is shown for comparison.
It is clear that the simple mode is approximately correct. A full model would need to consider the distribution of subtree sizes ($y'$) more carefully. In the case of trees containing only XOR, the output of a tree does not depend upon its shape and we can give an exact theoretical result for the limiting distribution.

9 XOR Program Spaces

We give a theoretical analysis in this section which shows that, in the limit, as program size grows, the density of parity solutions when using EQ or XOR is independent of size but falls exponentially with the number of inputs. First we start with the functions that can be created using EQ and XOR and, in particular, the form of the solutions to the parity problems.

9.1 Parity and Always-on/off Program Spaces

Even parity solutions, where $n$ is even, are of the form \( D_0 = D_1 = D_2 = \ldots = D_{n-1} \). However, given the symmetry of the EQ building block, the inputs to the program can occur in any order. \( D_x = D_y \) is true so \( (D_x = D_y = D_z) = D_y \), therefore, any pair of repeated inputs can be removed from the program without changing its output.

Odd parity solutions, where $n$ is odd, are of the form \( D_0 \neq D_1 \neq \ldots \neq D_{n-1} \). Again, given the symmetry of the XOR building block, the inputs to the program can occur in any order. \( D_x \neq D_y \) is false, but \( (D_x \neq \text{false}) = D_y \), so \( (D_y \neq D_x \neq D_y) = D_y \). Again, any pair of repeated inputs can be removed from the program without changing its output.

Any program, therefore, will be a solution to the parity problem provided it contains an odd number of all $n$ terminals. All solutions contain \( t = n + 2i \) terminals, where \( i = 0, 1, 2, \ldots \). Both EQ and XOR are binary functions, so programs are odd length \( l = 2t - 1 \) and solutions are length \( l = 2t - 1 = 2n + 4i - 1 \).

Any program with an even number of one or more inputs effectively ignores those inputs. Given the nature of the parity function, such a program will pass exactly half the fitness cases.

While XOR (or EQ) may create solutions to the parity problems more readily, they are considerably more limited than NAND and can only generate \( 2^n \) of the possible \( 2^{2^n} \) functions. NAND can generate them all, thus, the results of Section 7 do not directly apply. Unlike NAND, they show long range periodicity generating \( 2^{n-1} \) functions in trees of length \( l = 2n + 4i - 1 \) (for large $i$) and the other \( 2^{n-1} \) functions in trees of length \( l = 2n + 4i + 1 \).

9.2 Proportion of Solutions

If a program's length does not obey \( l = 2n + 4i - 1 \), then it cannot be a solution to the order $n$-parity problem and will score exactly half marks on the parity problems. If \( l = 2n + 4i - 1 \) is true, there is a chance that a randomly generated program will contain an odd number of each input and be a solution. When calculating the fraction of programs of a given length that are solutions, we can ignore the number of different tree shapes. This is because the output of an XOR tree is determined only by how we label its leaves and not by its shape.

To apply our Markov analysis, we consider the current state to be given by the oddness
or evenness of the number of each of \( n - 1 \) inputs. Given these, and that we are only considering programs which obey \( l = 2n + 4i - 1 \), the oddness or evenness of the remaining input is fixed. Therefore, we need only consider \( n - 1 \) rather than \( n \) inputs. The distribution of solutions to the parity problem is given by the chance of selecting a solution at random, i.e., of all \( n - 1 \) inputs having the right parity. Suppose we create long random programs by adding two randomly chosen inputs \( i, j \) at a time. The number of inputs of types \( i \) and \( j \) will both increase by one and so will change from an odd to an even number or vice versa. If both inputs are of the same type \( (i = j) \), then it will swap back and there is no change of state. The chance of moving from one state to another does not depend upon how we got to that state. The process is Markovian and can be described by a stochastic state transition matrix. The chance of moving from one state to another is equal to the chance of moving in the opposite direction, i.e., the state transition matrix is symmetric. There is a \( 1/(n - 1) > 0 \) chance of remaining in the same state after selecting the next pair of inputs. Thus, there is an acyclic limiting distribution and within it each of the states is equally likely (Feller, 1970, 399). There are \( 2^{n-1} \) states, one of which corresponds to a solution to the parity problem. In the limit of large programs, the chance of finding a solution to the parity problem of order \( n \) in a parse tree composed of XOR (or EQ) is \( 1/(2^{n-1}) \), provided the tree is the right size (and zero otherwise). In fact, at the large program limit this is true for each of the possible functions.

Figure 16 shows the fraction of 100,000 random programs which are solutions to the Even-6-parity or always-on-6 problems for a variety of lengths. Figure 16 shows measurement approaches the theoretical large program limit for \( t \geq 16 \) (i.e., \( l \geq 31 \)). This is to be expected as the second largest eigenvalue of the Markov transition matrix is 0.36
Figure 17: Fraction of programs that solve parity problems given appropriate building block. Length of programs increases with number of inputs to ensure leaves $\geq 2^\frac{3}{4}$ inputs. Measurement and theory agree within 2 standard errors (error bars show standard error).

which is far from the first. This means transient terms decrease by about $e^{-1}$ every time the program length increases by 2 leaves.

Figure 17 shows the fraction of random programs which are solutions to the Even or Odd parity problems (100,000 or a million random trials). The lengths of the programs were chosen so that $l = 2n + 4i - 1$ is true and the number of leaves exceeds $6n/16$. $6n/16$ was chosen as a linear variation of threshold where number of inputs was assumed and $2.33n$ was sufficient for $n = 6$. While the second eigenvalue changes only slowly with the number of inputs in the problem, the threshold appears to rise linearly with it. The threshold length $\approx n - 2$. Figure 17 shows agreement between measurement and the theoretical large program limit.

As the Markov analysis predicts, we have observed that (for $n = 4$) the proportion of all possible functions generated by XOR and EQ rapidly converges to the same value with increasing tree size.

10 Discussion

In the previous sections we have shown 66344 examples from diverse problems where our claim appears to be justified and we have proven it for both linear programs and standard GP. In the case of XOR trees, we have given the limiting distribution for each fitness level. In the more difficult problems at the extremes of the fitness range (solutions to the parity and the sextic polynomial problems), even sampling huge numbers of random trees, we
have not been able to amass enough examples to demonstrate our claim. However, our proof covers such cases. Next we consider extending our claim.

10.1 Automatically Defined Function

We can extend our argument to cover programs evolved using Automatically Defined Functions (ADFs) (Koza, 1994). Each ADF can be viewed as a tree in its own right and so, if the ADF exceeds the threshold size, the distribution of possible functions the ADF can implement will also converge to a limiting distribution as the ADF gets bigger. For each function, its value is determined by its input(s). The value of each input is given by a subtree in the calling ADF or main program. When the subtree exceeds the threshold size: the distribution of values used when calling the ADF will converge, the distribution of values returned by the ADFs will also converge, and, finally, the distribution of values returned by the program as a whole.

10.2 Memory

It should be possible to produce a combined proof based on Sections 7 and 8 which covers tree programs which have additional state (memory) outside the tree. We have not measured the distribution of programs with external memory. It may be that memory radically changes the threshold size. However, we suggest that our claim will hold for non-recursive programs which include memory and subroutines.

10.3 Turing Complete Programs

To consider all programs, we need to consider Turing completeness. This requires the addition of either recursion or iteration. The proof advanced in Section 7 for long linear programs can be extended to Turing complete programs provided they halt. First, we note Section 7 requires the programs complete \( l \) instructions. If \( l \) is big enough, programs longer than \( l \) have the same distribution, so exactly when the program halts is not crucial. Second, it is assumed that each instruction is independent of the previous one. In program loops, the instructions are executed in an ordered sequence. However, if each loop is small compared to the program it is reasonable to treat a repeated sequence of random instructions as if it was just a random sequence.

It seems reasonable that a similar result will also apply to big trees including iteration or recursion and memory. In both cases we anticipate that the greater complexity available may radically affect the threshold length, perhaps to such an extent that it is not obvious that our claim will hold to Turing complete programs.

10.4 Parity Problems Landscapes and Building Blocks

Section 9 describes the program space of the parity functions when given the appropriate functional building block (XOR or EQ). For all but the simplest problems at all program lengths the fitness space is dominated by a central spike indicating almost all programs score exactly half marks. A small fraction of programs do solve the parity problems. We have derived a simple analytical expression for the case of large programs which shows that the proportion of solutions falls exponentially with increasing number of inputs. However, for modest numbers of inputs, the proportion is not so small as to be infeasible to find using modern computers.
To derive a fitness landscape, we would also need to consider how genetic or other operators move between points in the search space, as well as the fitness of those points. This is unnecessary in this case, because either we have found a solution or the point in the search space scores half marks. That is, the program space contains no gradient information. Search techniques such as GP, which rely on gradient information will be unable to out-perform random search on such a landscape. We might anticipate population based search without mutation performing worse than random search since genetic drift in a small population means the population may lose one or more primitives. If this happens, then it becomes impossible to construct a solution.

This suggests that techniques which seek to solve the parity problems by evolving the appropriate building blocks are unlikely to find minimal solutions directly. Such evolutionary techniques will probably find programs with fitness well above half marks and will reject any partial solution composed only of the discovered building blocks, because these will score less than the partial solutions already discovered. It is possible that impure solutions to the problem may be found which subsequent evolution, perhaps under the influence of parsimony or beauty pressures, and may evolve into a solution comprised only of building blocks.

It is worth noting that we have only considered the case where all components of the programs are of the same type. So there are no restrictions on how functions work with each other. Performance gains for GP have been reported by using strong typing (Montana, 1995) or grammars (Whigham, 1996) which control program components' interactions. In the case of the Boolean problems, examples include Janikow (1996) who gainfully employed typed inputs, while Yu and Clack (1998) structure their inputs as a list. Work is continuing to understand the role of different function sets and search operators on the Boolean problems (Page et al., 1999; Poli et al., 1999).

10.5 “Random Trees”

On average, half the random trees sampled using the ramped-half-and-half method (Koza, 1992, 93) are full. Therefore, particularly if the depth parameter is increased beyond the usual 6 (equivalent to maximum size of 63), the chances of finding at random both the even-3 and the odd-3-parity functions are considerably higher using it than using uniform search. In contrast, ramped-half-and-half is less likely to find solutions to the Santa Fe ant trail problem than uniform search (see Langdon and Poli (1998, Table 3)). This suggests that the best method to use to create the initial random population is problem dependent.

In large binary random trees about half the functions have one or more terminals as their arguments (Sedgewick and Flajolet, 1996, 241). They are relatively sparse— their average height is $2\sqrt{\pi(l-1)/2} + O(1/\sqrt{l})$ (Sedgewick and Flajolet, 1996, 256), which is greater than the height of full binary trees $\log_2 l + 1$. The subtrees within them are also sparse— the whole search space contains a lower proportion of full or nearly full subtrees than do full trees. Since nearly full subtrees can be used to form XOR from NAND gates, this may be a partial explanation for why the parity functions are so rare in the whole search space but are comparatively more frequent in full trees.

Our studies of the “bloating” phenomena (Langdon et al., 1999) in GP and other search techniques indicate that, in the absence of parsimony, size, or depth restrictions, GP populations tend to evolve towards these large relatively sparse trees which may have few full subtrees within them. This suggests that problems may arise if the problem (and
function set used) needs full or nearly full subtrees to solve it. The use of a depth limit rather than size limit on the evolution of the program trees may encourage the formation of nearly full trees of the maximum permitted depth. These will contain more full subtrees. A depth limit may ease the solution of problems in which full trees contain a higher proportion of solutions. A size limit will discourage the formation of full trees and may help in problems where the density of solutions is lower in full trees.

10.6 GP and Random Search

We have discussed the number of programs which implement each function as a fraction of the total number of programs. Particularly the proportion of solutions. This corresponds directly to the difficulty of the problem for random search and establishes a benchmark with which to compare GP and other techniques. In Koza (1992, Chapter 9), GP performance is shown not to be the same as random search. Indeed, in all but a few of the simplest problems which both GP and random search easily solve, GP performance is shown to be superior to random search. It is commonly assumed that problems that are harder for random search will also be harder for any search technique. There is little evidence to support this. For example, Koza (1992, Figure 9.2) shows a strong correlation in the 3-input Boolean problems between difficulty for random search and difficulty for GP. Thus, the distribution of solutions in the search space can give an indication of problem difficulty for GP.

10.7 Searching Long Programs

Since the density of solutions changes little with program size, there is no intrinsic advantage in searching programs longer than the threshold. It may be that some search techniques perform better with longer programs, perhaps because together they encourage the formation of smoother, more correlated, or easier to search fitness landscapes (Poli and Langdon, 1998). However, in practice, searching at longer lengths is liable to be more expensive both in terms of memory and also time (commonly the CPU time to perform each fitness evaluation rises in proportion to program size).

At present, threshold location is not known in advance. A line of research would be to devise a means of predicting it. This could be of practical value by replacing existing ad-hoc measures to preset the upper bound on the size of programs with a more principled approach.

11 Conclusions

In Section 7 we proved that the distribution of functions implemented by linear programs converges in the limit of long programs, and, at that limit, each is equally likely. While convergence to the limit is exponentially fast with the size of the program, the number of functions is exponential in the sizes of both input and output registers. The chance of finding a long solution at random falls exponentially with the problem size.

Section 8 gives the corresponding proof for binary parse trees. Most parse trees are asymmetric and even if the function set is symmetric, the limiting distribution is asymmetric (i.e., there is a much higher number of some functions than others). Functions that can be produced by small trees are frequent not only in short programs but also in very big ones.

In Section 9 we showed that trees composed of only XOR or EQ functions can be
Scaling of Program Fitness Spaces

treated as special cases of linear programs. This yields a limiting proportion of solutions to
the $n$-input-parity problems of $1/2^{n-1}$. This was confirmed by experiment. An empirical
measure for the rate of convergence is also given. Together with the fitness function, these
give the complete fitness landscape.

In three very different classes of problems: Boolean, symbolic regression, and evolving
agent, we have shown that the fitness space is, in a gross manner, independent of program
length. In general, the number of programs of a given length grows approximately expo-
nentially with that length. Thus, the number of programs with a particular fitness score or
level of performance also grows exponentially, in particular, the number of solutions also
grows exponentially.

Acknowledgments

I would like to thank Trevor Fenner for suggesting the Markov analysis of XOR parity
problems, Tom Westerdale for help and inspiration, and Paul Vitanyi. I would also like to
thank the anonymous referees for particularly helpful comments and suggestions.

References

Massachusetts.


and Sons, Somerset, New Jersey.


Koza, J. R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press,
Cambridge Massachusetts.

Langdon, W. B. and Poli, R. (1998). Why ants are hard. In Koza, J. R., Banzhaf, W., Chellapilla, K.,
Morgan Kaufmann, San Francisco, California.

Spector, L., Langdon, W. B., O’Reilly, U.-M., and Angeline, P. J., editors, Advances in Genetic


thesis, der Universität Dissertau Am Fachbereich Informatik, Germany.

in genetic programming: A preliminary study. In Poli, R., Nordin, P., Langdon, W. B., and
Fogarty, T. C., editors, Genetic Programming, Proceedings of EuroGP’99, volume 1598 of LNCS,
pages 39–49, Springer-Verlag, Berlin, Germany.


