Foundations of Genetic Algorithms FOGA-7, J. Rowe *et. al.*, editors, Torremolinos, 4-6 September 2002, Morgan Kaufmann

How many Good Programs are there? How Long are they?

W. B. Langdon
Computer Science, University College, London,
Gower Street, London, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk http://www.cs.ucl.ac.uk/staff/W.Langdon

Abstract

We model the distribution of functions implemented by non-recursive programs, similar to linear genetic programming (GP). Most functions are constants, the remainder are mostly parsimonious. The effect of ad-hoc rules on GP are described and new heuristics are proposed. Bounds on how long programs need to be before the distribution of their functionality is close to its limiting distribution are provided in general and for average computers. Results for average computers and a model like genetic programming are experimentally tested.

1 Introduction

Fitness landscapes are an appealing metaphor for evolutionary and local search. Attempts to characterise the fitness landscape of genetic programming tend to contain unquantified statements that the genetic programming (GP) search space is big or that it is "rugged". Occasionally some effort has been spent to define and measure ruggedness but these are specific to tiny fraction of small benchmark problem spaces. Our approach is to provide quantified metrics for the general cases and then refine these towards specifics. Experiments are used to confirm theoretical models, rather than provide yet more indigestible data.

In (Langdon and Poli 2002) we proved for a number of systems close to practical tree and linear GP systems that eventually, for big enough programs, their fitness distribution will converge. In (Langdon 2002) we started to answer the question how long do programs have

to be in practice to get reasonably close to this limit. We concentrated on the distribution of answers produced by all linear programs of a given size. For a number of systems, we were able to quantify the minimum size needed for this distribution to be close to the limit. I.e. overall making programs longer than this has little effect on the distribution of their outputs. Here we provide similar results on the distribution of *functions*.

The next section summarises the Markov model of linear GP, while Section 3 describes total variation distance as a convergence metric. Minorization is described in Section 4 and used to provide a weak upper bound for any computer (Section 4.1). Then the application of Minorization to the average computer is discussed in Section 4.2. Section 5 describes detailed models for the distribution of functions implemented by a computer similar to that used in linear genetic programming. These models are compared with measurements. This is followed by a brief discussion (Section 6) and a summary (Section 7).

2 Markov Models of Program Search Spaces

(Langdon and Poli 2002) deals with both tree based and linear genetic programming (GP). For simplicity we will consider only large linear programs, however we anticipate similar bounds can be found for large trees too.

We want to know about every possible program. That is, we wish to know about the whole search space. Our approach is to sample it randomly (both theoretically and in real experiments) a large number of times. As the sample becomes bigger, its properties will approach that of the distribution from which it is drawn. I.e. the sample tells us how the whole search space behaves. Mainly we are interested in seeing how the search space scales with program length. So we generate a random sample of programs all of the same size and measure their properties. Then we do the same again for a new program size, and so on, until we have a picture of the whole search space.

We use the following model of computer programs. Initially the computer's data memory is zeroed. The program's inputs are loaded into the input register (one or more memory cells). The program runs and in the process reads and writes to memory. When it stops, its answer is read from the output register (again one or more memory cells). We limit ourself to programs that run through a given number of instructions and stop. They do not loop and they always terminate. While this excludes iteration and recursion, many non-trivial programs can be written in this way, indeed practical GP systems are often of this type (Banzhaf, Nordin, Keller, and Francone 1998).

A Markov process is a stochastic process in which the probability of making a transition from one state to another depends only on the current state. In particular the chances involved do not change with time or depend on any previous history (earlier than the current time). In our model the relevant state is simply the computer's memory. The outcome of any computer program instruction only depends on the current contents of memory. For example if we add two numbers (held in memory) we expect the answer to depend only on those two numbers. We should always get the same answer, no matter when we add them, no matter what the computer has done before. I.e. a given instruction and a given contents of memory (i.e. state) will always have the same effect. Since in our model the output of an instruction is written to memory, each instruction can be viewed

as a transition from the current contents of memory to the contents of memory at the next time step. A given instruction always produces the same transitions.

When we consider random programs, we mean every possible instruction within the program is equally likely. Note this means the probability of each instruction is fixed. Therefore we can view running a random program as a Markov process in which the computer's state (i.e. the contents of its memory, N bits) undergoes a sequence of random changes. (Note the probability of every state transition is fixed.) While there are many states (actually 2^N) we have to deal with, this approach allows us to use the nice theoretical results about the behaviour of Markov processes after many random steps and how many random steps are enough for these results to apply.

Let us assume, (1) the designer of our computer has ensured that it is possible to set the memory to any value (i.e. every state of the Markov process is accessible from every other). (2) that there is at least one state and corresponding instruction which leaves the memory unchanged. (E.g. clearing a register which already contains zero.) Given (1) and (2) the Markov process is ergodic and irreducible. Which ensures that after a large number of random updates the probability of each memory pattern will settle into a limiting distribution which does not depend upon the programs' inputs.

We can go further and consider not just the answer produced by a program when given an input but also the complete mapping it provides from inputs (i.e. initial memory contents) to final memory contents (which includes the output register and so includes the program's outputs). That is, treat a program as a *function* from input to output. Starting a given random program with different input, means we start the Markov process from a different start point. Each time the program is run, the computer's memory is updated randomly, but the runs (of the same program) are not independent of each other.

In some of what follows we will consider a given program as implementing a table. The table has one row for each different input to the program and each row gives the contents of the computer's memory when the program stops. (Table 1 contains a simple 3 input ($n = 3$) example for a program with one instruction.) Usually we will assume the program is to be run for every one of the 2^n possible input test values. Note if we look individually at each row during the execution of a randomly chosen program, then the row is a random Markov process. On the other hand, while each row is not independent, the whole table contains all the relevant state information, and so when running a randomly chosen program, the whole table can also be treated as a Markov process.

Again we need to impose the same properties on the computer to ensure the new Markov process is ergodic and irreducible. I.e. there must be a program instruction which does not change the table. A trivial way of achieving this is to clear the input register, so the whole table is empty. Then any operation which writes zero to memory will have the desired effect of not changing the table. If we ensure our computer can implement any Boolean function of the inputs and write this to any part of memory the table can eventually be set to any value. (A minor complication is to ensure the inputs are over written last and in such away that their part of the table can be reset arbitrarily.) This will ensure that, given enough random instructions, the probability of each possible inputs $\times$ memory table settles into a limiting distribution π.

Table 1 Memory contents after running a single instruction, r4=AND(0,2), on each of 8 inputs. The 3 bit input register, r0–r2, overlaps the two bit output register, r0–r1. Note r4=AND(0,2) only over writes r4 and so does not change either output bit.

Input			Memory							
									Out	
			r7	r6	r5	r4	r3	r2	r1	r0
0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	0	1	1
1	0	0	0	0	0	0	0	1	0	0
1	0	1	0	0	0	1	0	1	0	1
1	1	0	0	0	0	0	0	1	1	0
1	1	1	0	0	0	1	0	1	1	1

The truth table of the function that a program implements, is a subset of the whole table. It is just the m columns corresponding to the output register of the whole table.

3 Convergence Metric

In some of the following sections we shall use the total variation distance $||\cdot||$ between two probability distributions to indicate how close they are. The total variation distance between probability distributions a and b is the largest value (supremum, sup) of the absolute difference in the probabilities where the difference is taken over *all subsets*, i.e. every possible grouping of states x, not just single points (see Figure 1). In mathematical terminology $||a - b|| = \sup_{x \subseteq \chi} |a(x) - b(x)|$ (Rosenthal 1995; Diaconis 1988). (If $||\cdot||$ were just the largest difference, it would be small as long as a and b were both small, even if the distributions a and b were not similar.)

For two examples in (Langdon 2002) we were able to show that the distribution of functions converged at the same rate as the distribution of outputs. However in neither example (cyclic and bit-flip) is the computer able to do general computation. In Section 5 we consider more useful models.

4 Markov Minorization

When considering Markov processes it is common to use a square matrix whose i,j^{th} element P_{ij} holds the probability of the process moving from state i to state j. Naturally each probability and hence each element is fixed. So for a given process the whole matrix is constant. In our model the matrix is $2^N \times 2^N$ (when considering outputs) or $2^{N2^n} \times 2^{N2^n}$ (when considering functions).

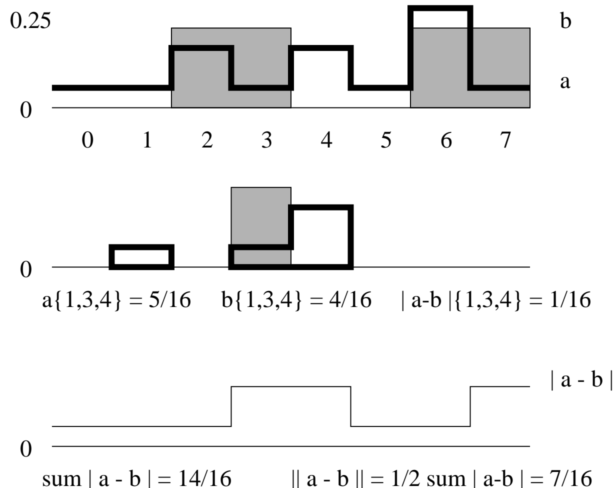

Figure 1 Total variation distance between probability distributions a and b. Top panel shows a and b. Middle compares the probability of being in subset $x = \{1, 3, 4\}$ with a and with b and calculates their difference, $_x|a(x) - b(x)| = 1/16$. $||a - b||$ is the largest difference across all 256 possible subsets. In the discrete case $||a - b|| = 1/2 \sum |a - b|$ (Diaconis 1988).

The transition matrix P can be thought of as telling us how a randomly chosen operation mixes things up. As described at the end of Section 2, the computer's memory after a program has reached instruction l with each legal input specifies the function the program implements at this point in its execution. Executing one randomly chosen instruction will update the pattern and so potentially also update the function. In practical computers the number of possible instructions is large but much less than the number of possible bit patterns in memory. Thus after one instruction only a small fraction of memory states can be reached. That is, in one step the Markov matrix can only mix things up a little. However if we consider executing two instructions in sequence the number will increase. After many steps even more can be reached. Minorization (Rosenthal 1995) is a way of quantifying this mixing (or at least providing a lower bound on it). Specifically the difference between the actual μ_k and limiting π distributions obeys:

$$||\mu_k - \pi|| \leq (1 - \beta)^k$$

where

$$\beta = \sum_j \min_i P^a_{ij}$$

and k is the number of groups of a instructions. β is the sum of the minimum values of the entries in each column of the matrix P^a ($P^a = P \times P \times \cdots \times P$). To find β, we start by finding the chance of changing the contents of the inputs $\times$ memory table, using a instructions, from each initial value (i) to the least likely ($\min_i$) other possibility j (for that value i). Then β is given by adding together all these probabilities.

4.1 Convergence Bound on Functions Implemented on Any Computer

Let a be the minimum number of program instructions required to set the inputs$\times$memory table to an arbitrary value. That is, from the initial starting condition, s, there is at least one program of a instructions which sets the table to any of its 2^{N2^n} values. This means we have a minorization condition for P^a. If there are I instructions, the number of programs of length a is I^a. So $P^a_{s,j} \geq I^{-a}$. Therefore β is at least I^{-a} and so for any computer:

$$||\mu_l - \pi|| \quad \leq \quad (1 - I^{-a})^{l/a} \tag{1}$$

I.e. we are guaranteed that the difference between the probability distribution of functions implemented by programs of length l and its limiting distribution (i.e. as $l \to \infty$) falls geometrically as the length increases.

Setting $|| \cdot ||$ to 10% yields a convergence length l for any computer with I instructions $l \leq 2.3aI^a$. Where a is the minimum number of instructions needed to set the inputs $\times$ memory table to any value.

4.1.1 Weakness of Minorization Bound

Note (1) leads to an exponential bound. In fact we can devise examples where convergence in terms of both outputs and implemented functions does require exponentially long random programs. (In the case of a cyclic computer at least $0.8\frac{3}{4\pi^2}2^{2N}$ instructions are needed (Langdon 2002, 3.1).) However a major difficulty is that Inequality (1) can be a very weak upper bound.

Consider a very small example of a useful computer (of the type to be described in Section 5) with two input bits ($n = 2$) and one output bit $m = 1$. There are $2^{m\times 2^n} = 16$ functions. The most difficult of these is parity, which needs three instructions. Thus $a = 3N$ instructions should be sufficient. There are four operation codes, two memory address are read in each instruction, and the output is written back to memory, so the number of instructions I is $4 \times N^3$. Therefore $l \leq 2.3\ 3N\ 4^{3N} N^{3\times 3N}$. If $N = 8$, $a = 24$ and $I = 2048$ and so programs need not exceed $7.2\ 10^{7399}$ for the distribution of their functions to be near the limit.

We can get more realistic estimates by considering the computer in more detail. Even so, in this example, the minorization bound is many times the actual convergence rate. (Near convergence is seen by programs of about 25 instructions, cf. Figure 4.)

4.2 The Average Computer

We have been considering computer instructions as moving the computer from one state to another. We can define an average computer with I independent instructions, to be representative of the whole class of such computers if they are average instructions. An instruction chosen at random from the set of all possible instructions will be similar to the average over all possible instructions. So in (Langdon 2002) we argued that a randomly connected computer is an average computer. We used the minorization process (described above in Section 4) to prove the distribution of outputs produced by such an average computer converges and calculate only $l \leq \frac{15.3+2.3\ N}{\log I}$ (Langdon 2002, (1)) random instructions are needed to get close to the limit, cf. Figure 2.

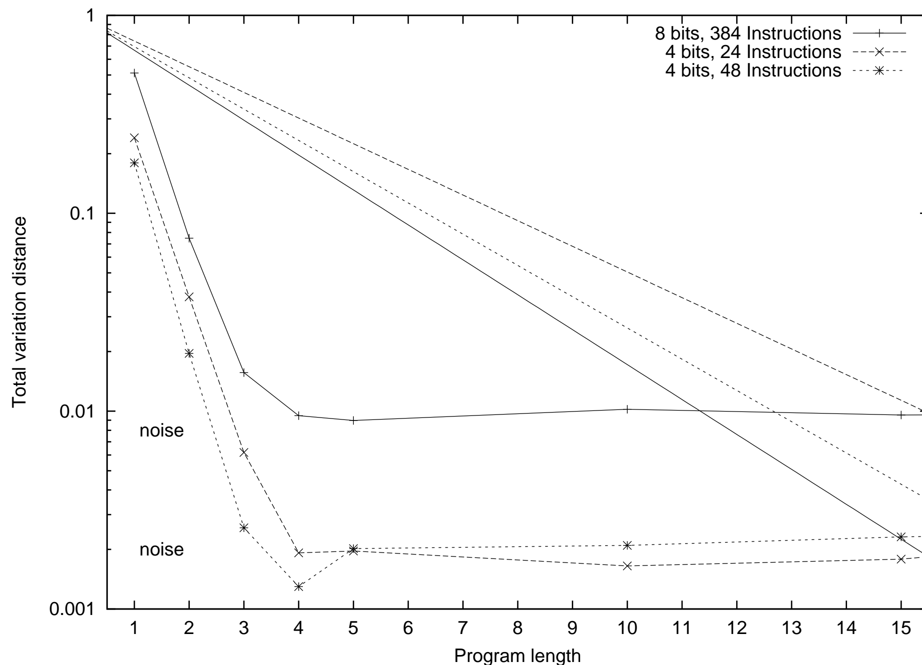

Figure 2 Convergence of three random machines measured with a million random programs at each length. Note 1) allowing for noise, measurements are within theoretical upper bound (Langdon 2002) (straight lines) and 2) large instruction sets are needed to ensure randomly connected computers can access all their memory states.

If two rows of the inputs × memory table are different, executing any of the instructions will change them both to (probably different) random contents. However if the two rows have the same contents, after any instruction they will both still have the same contents (albeit probably different from their previous one). I.e. if two rows become synchronised, they will remain synchronised no matter how many more instructions are executed. (We exclude conditional branches and loops.) We shall assume there are a large number of independent random instructions. This ensures the transition matrix is almost certain to be connected, i.e. the computer can implement all functions. Given a long enough program, chosen at random, all the rows in the table will be synchronised. This means in the limit of very long programs, the contents of memory will be the same no matter what the program's inputs were. I.e. almost all long programs implement one of the 2^m constants. Further each constant is equally likely.

The chance of two rows, which had different contents, becoming synchronised by the next random instruction is 2^{-N}. Therefore the number of random instructions before two rows become synchronised is exponentially distributed, with mean and standard deviation 2^N.

To totally synchronise all inputs, $2^n - 1$ rows must be synchronised with the first. The expected number of instructions for any row to synchronise with the first is $2^N/(2^n - 1)$. The expected number for the second is $2^N/(2^n - 2)$, and so on. Until rows synchronise, they are independent of each other. So the mean for all rows to synchronise is the sum of $2^n - 1$ individual means. I.e. the total for all rows is $2^N \sum_{i=1}^{2^n-1} 1/i$. (For large h, the harmonic number, $\sum_{i=1}^{h} 1/i$, can be approximated by $\log h + \gamma$, where γ is Euler's constant.) So, for large n, the expected program length for complete synchronisation is $(\log(2^n - 1) + \gamma)2^N = (.58 + .69n)2^N$ Similarly the variance is also given by summing, $2^{2N} \sum_{i=1}^{2^n-1} 1/i^2 \rightarrow 2^{2N} \frac{\pi^2}{6}$. So, for large n, the standard deviation tends to $\frac{\pi}{\sqrt{6}} 2^N = 1.29\ 2^N$(cf. the coupon collector's problem (Stirzaker 1999, pages 274–275)). Note the distribution has an approximately exponential tail so programs longer than a modest multiple of $n2^N$ are almost certain to return a constant value regardless of their inputs.

5 AND NAND OR NOR Computer

The model used in (Langdon 2002) is used again but we look at the functions implemented by programs and give new examples.

5.1 Convergence of Random Boolean Outputs

This model is close to actual (linear) GPs (Nordin 1997). The computer comprises N bits of memory and the CPU. The memory contains the input register (n bits) and the output register (m bits). In our experimental work, the input and output registers overlap. The CPU has 4 Boolean instructions: AND, NAND, OR and NOR. Before executing any of these, two bits of data are read from the memory. Any bit can be read. The Boolean operation is performed on the two bits and a one bit answer is created. The CPU then writes this anywhere in memory, over writing what ever was stored in that location before.

Note the instruction set is complete in the sense that, given enough memory, the computer can implement any Boolean function.

Each time a random instruction is executed, two memory locations are (independently) randomly chosen. Their data values are read into the CPU. The CPU performs one of the four instructions at random. Finally the new bit is written to a randomly chosen memory location.

Now it considerably simplifies the argument, to note, that the four instructions are symmetric. In the sense that no matter what the values of the two bits read, the CPU is as likely to generate a 0 as a 1. That is each instruction has a 50% chance of inverting exactly one bit (chosen uniformly) from the memory and a 50% chance of doing nothing. In (Langdon 2002) we proved that in there is a limiting distribution π in which each possible output is equally likely, further the distance between the actual distribution of programs of length l, μ_l is bounded by

$$||\mu_l - \pi||^2 \quad \leq \quad \frac{1}{2}\left(e^{me^{-\frac{2l}{N}}} - 1\right) \qquad (2)$$

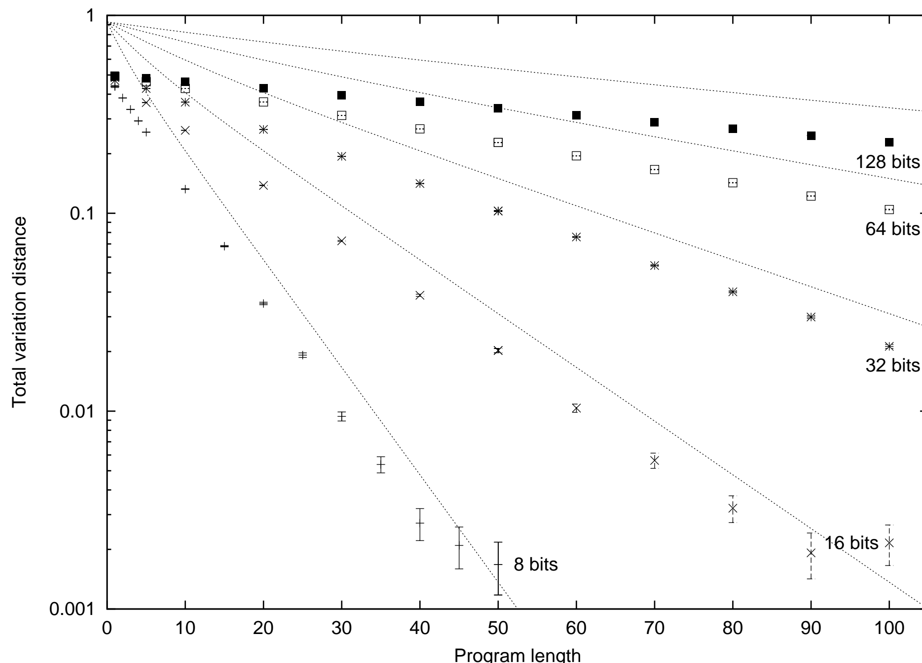

Figure 3 Convergence of outputs of 4 input bit Boolean (AND, NAND, OR, NOR) linear programs with different memory sizes. Note the agreement with upper bound $\sqrt{1/2(\exp(me^{-2l/N})-1)}$ (dotted lines).

We can quantify how long programs need to be for the actual probability distribution to be reasonably close to the large program limit by requiring the total variation distance, $||\mu_l - \pi||$, not to exceed 10%. This is met by programs that are longer than $\frac{1}{2}N(\log(m)+4)$. Figure 3 confirms this and shows, as expected, Inequality (2) is quite a tight bound.

5.2 Convergence of Random Boolean Functions

As discussed in Section 2, a given program can be thought of as a transformation from the initial memory pattern to the memory pattern when the program terminates. Now when the program starts, most of the memory is zero. Only the input register can be non-zero. So again we can consider every program as a transformation from the inputs $(0..2^n-1)$ to the complete memory $(0..2^N-1)$. In this section all 2^n possible input values are tested. We will be primarily interested in just the output register but for the time being let us consider the whole memory. Again we can build a inputs$\times$memory table N bits wide with 2^n rows, one for each input. Each row contains the memory pattern after the program halts after having started with the corresponding input, cf. Tables 1 and 2.

We know (cf. Section 5.1) in the limit of long programs, that for any given input pattern each bit is as likely to be set as clear. I.e. after a long ($\frac{1}{2}N(\log(N)+4)$) random sequence

Table 2 Example of memory contents after running a programs. Each row gives contents (8 bits) after starting with corresponding input (2 bits, 2 left hand columns). Note correlation between rows.

Input		Memory							
								I/O	
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
1	1	1	1	0	0	0	0	1	1

of instructions, any row of the table will hold a uniform random pattern of 0 and 1s. However the rows of the table are not independent. In the large program limit, each of the rows is identical.

This is a fairly general property of this type of computer. However we can devise machines that do not have it. For example, a machine which only allows increment or decrement of the whole memory in a cyclic manner (Langdon 2002, Sect. 3.2). After running any program on a cyclic machine the memory holds the value $(x + p) \bmod 2^N$. Where x is the input and p is a constant (specific to that program). Since x, and therefore the number of rows, cannot exceed 2^N, each row in the table is unique.

If we have 2^n identical copies of the computer. We run them in step. Each runs the same program but with a different input. They start in a different states. No matter how long the program is, the computers will never synchronise. They are also reversible, in the sense that the original input can be calculated from the program and its output. Recently there has been interest in reversible computers since quantum computers are reversible, and also for use in high integrity systems (Bishop 1997). In contrast typical computers (and also random computers, cf. Section 4.2) do synchronise, they are not reversible and they destroy information during the course of a program run.

It is possible to devise a program for our four Boolean operators computer that behaves like the cyclic machine, however if we look at the average long program it does loose information and the rows in its table are not unique.

To prove this we note that we can consider our table not just as the final state of the computer but as its current state at each instruction of the program. We trace how the table changes from being empty (apart from the input register) to the state when the program halts. Now each row represents the complete state when the program was started with the corresponding input. Each time an instruction is executed, two bits are read and one bit is updated. Since there are no branch instructions in this computer, which memory locations are read and which is updated is the same no matter what the program's inputs were. So we can think of each instruction reading two *columns* of our table, performing the same logical operation on a number of pairs of bits and writing the results back to a *column* of our table.

In the limiting case, both of the columns read by any instruction contain either all zeros or all ones, as does the second. Therefore no matter which operation the CPU performs, the

whole of the output column will be either all zeros or all ones. That is, the 2^N patterns are attractors. If the computer ever reaches one of them, it will remain in one of them. Note this means the original input has been erased, and any program will return a constant no matter what its input was.

Each row of the table corresponds to the action of the program with a specific input. We have shown (Langdon 2002) that in the limit of large random programs each memory pattern is equally likely. Therefore in the limit each of the 2^N attractors are also equally likely. I.e. each of the 2^m possible constants are equally likely. We consider how long it will take to get close to the limiting distribution by constructing a number of models of the convergence process.

5.3 Model of Convergence of Boolean Functions

This is a some what empirical model of convergence of the function implemented by random Boolean programs. Instead of total variation distance we take into account that we know the limiting distribution for the inputs $\times$ memory table is a uniform random combination of columns of all zeros or all ones. We consider the fraction of programs that implement a function which is not in the limiting distribution. I.e. those which do not yield a constant. (The distribution of constants converges rapidly to each being equally likely, and we do not consider this further.)

Initially, where the input and output registers overlap, the outputs are equal to the inputs. I.e. the output register part of the inputs $\times$ memory table contains the Identity function. As random operations are performed, the proportion of non-constant functions falls (see Figure 4).

Initially convergence is rapid and dominated by the removal of the Identity function. As programs get longer convergence continues but at slower exponential rate. We construct two models to explain this.

5.4 Initial Convergence of Boolean Functions

Initially the output register columns of the inputs $\times$ memory table contains the Identity function and the other columns are all zeros (cf. Table 1). Only m/N instructions write their output to the output register. If this proportion were fixed the proportion of Identity functions would fall as $(1 - m/N)^l$. Rearranging, and assuming $N \gg m$ (so $\log(1 - m/N) \approx -m/N$) we can get a lower bound on program length for convergence of the distribution of functions of $2.3N/m$.

In Figure 5 we have further refined this model by taking advantage of the fact that we know which of the operations overwrite the output register with the Identity function given they are reading either from memory which still contains zero or from the output register itself. For our four function instruction set and $m = 1$, this introduces a $\approx n/(2N)$ correction.

As other parts of the memory are over written the rapid convergence due to loss of the identify function (cf. D0 in Figure 4) slows. Cf. "knee" in observations as program length passes N. Nevertheless, Figures 5, 8 and 9 show $(1 - m/N)^l$ is a very good model for

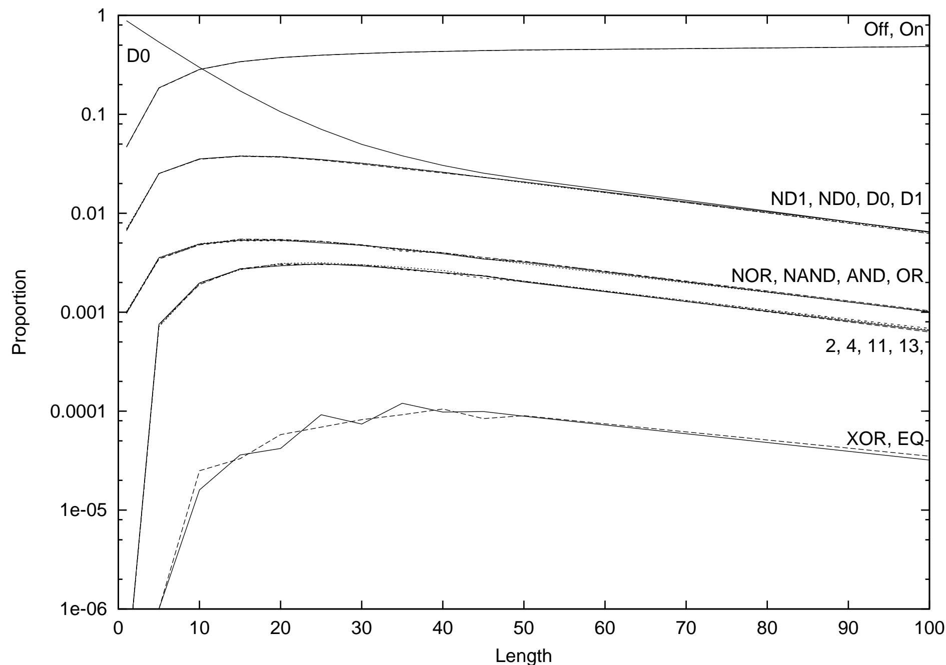

Figure 4 Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits). Almost half long programs implement Always-on and nearly half implement Always-off. Short programs are more varied and many implement the Identity function "D0".

the whole search space. It is only when we want to consider the distribution of very rare non-trivial functions, that a more detailed model is needed.

5.5 Later Convergence of Boolean Functions

After $O(N)$ instructions the majority of the memory is likely to have been overwritten and to be weakly correlated with the input data. In this section we provide a crude model of the distribution of functions in long linear programs. The first thing to stress is that we do see convergence towards the limiting distribution (equal numbers of each of the 2^m constants). Figures 4, 5, 8 and 9 show the proportion of non-constant functions falls exponentially, albeit with a smaller exponent than predicted in the previous section. It is also worth pointing out that since the exponent is much smaller than that associated with the increase in numbers of programs with increase in their length, that the total number of interesting functions continues to increase dramatically, cf. Figure 6.

To model the small fraction of interesting functions we note they can only arise in programs that have either avoided over writing the whole of the input register with a constant or have saved (one or more) copies of it elsewhere in memory. Computer instructions reading

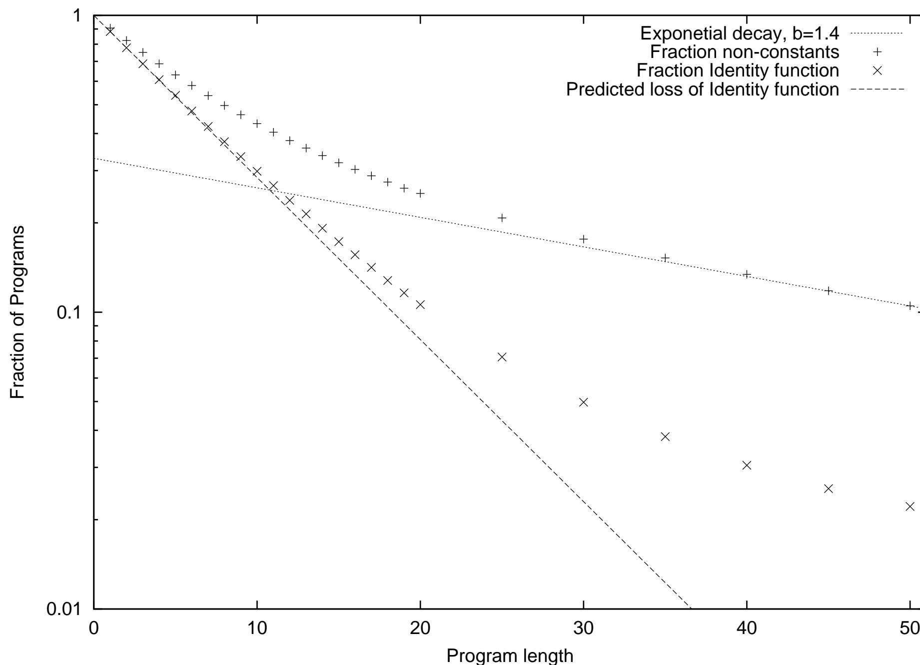

Figure 5 Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits). The steep straight line is the predicted effect of exponential loss of the Identity function "D0" caused by overwriting of the output register. Initially this explains almost all the convergence (Sect. 5.4). While the dotted line is experimental fit (Sect. 5.5).

(parts of) these copies of the input register may lead to interesting (i.e. non-constant) functions. The proportion of each such functions, appears to, fall dramatically with the number of operations needed to create them from the input data. However the ratio of frequencies of these functions appears to stabilise when programs are long, cf. Figure 4. This could allow us to estimate the proportion of functions which are too low to measure, either from measurement or models of more frequent functions. However we start by modelling the number of copies of the input register held in memory and then use this to predict the fraction of non-trivial programs.

Let x be the fraction of columns in the inputs $\times$ memory table which are not all zero or all one, i.e. not constants. Almost all of these (cf. Figure 4) will contain copies (or inverted copies) of one of the bits of the input register. After executing a randomly chosen instruction, the number of non-constant columns will be the same, one higher or one lower.

The only way the number of non-constants can increase, is by selecting one or more non-constants as input, selecting a constant for output (probability=$(1 - x)$), and then the operation not generating a constant. If one input column is constant and the other is a

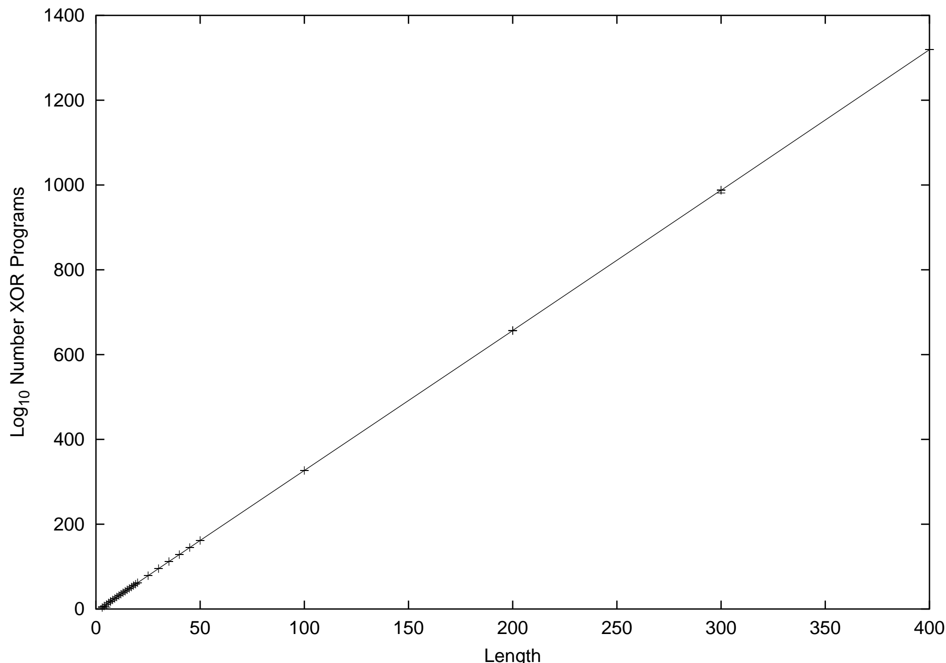

Figure 6 Number of programs which implement 2-bit odd parity (AND, NAND, OR, NOR, 8 bits). Cf. Section 4.1.1.

copy (or inverted copy) of an input bit and a random operation (from our set of four, AND NAND, OR and NOR) is chosen, the chance of creating a constant is 50%.

If both columns are copies (or inverted copies) of different bits of the input register, then the output will not be constant. If they are copies of the same bit, then the output will be a copy (or inverted copy). The only way to get a constant is if one input is an inverted copy of the other, which will always produce a constant. In practice, the distribution of copies of the input register is very nonuniform. If only one bit of the input register has been copied (or inverted), the chance of generating a constant column from two non-constant columns is 1/2. If every bit of the input register has equal number of copies the chance falls to $1/2n$. Let us define b as the effective average number of bits of the input register copied, so the chance is $1/(2b)$, where $1 \leq b \leq n$. So

$$\begin{aligned} P(x \to x+1/N) &= (1/2)2(1-x)x(1-x) + (1-1/(2b))x^2(1-x) \\ &= x - (2b+1)/(2b)x^2 + 1/(2b)x^3 \end{aligned}$$

Similarly the number of non-constants can fall by selecting a non-constant as output, and either selecting constants for both input columns or selecting one or more non-constants but generating a constant from them. Again we assume almost all non-constants are copies

(or inverted copies) of bits of the input register. So

$$\begin{aligned} P(x \to x - 1/N) &= x(1-x)^2 + (1/2)2xx(1-x) + 1/(2b)x^3 \\ &= x - x^2 + 1/(2b)x^3 \end{aligned}$$

Notice that there is a slight imbalance between these. So, as expected, random operations tend over time to reduce the fraction of copies of the input register in memory. Let $f(x)$ be the average decrease in x.

$$\begin{aligned} f(x) &= P(x \to x - 1/N) - P(x - 1/N \to x) \\ &= x - x^2 + \frac{x^3}{2b} - \left((x - 1/N) - \frac{2b+1}{2b}(x - 1/N)^2 + \frac{(x-1/N)^3}{2b} \right) \\ &= +1/N + (2b+1)/(2bN^2) + 1/(2bN^3) \\ &\quad - \big((2b+1)/(bN) + 3/(2bN^2) \big)\, x \\ &\quad + \left(1/(2b) + 3/(2bN)\right) x^2 \\ f(x) &= A + Bx + Cx^2 \end{aligned} \tag{3}$$

Equation (3) is small but never negative, cf. Figure 7. The expected fractional decrease is $(1 - f(x)/x)$. If this were constant, the convergence would be exponential. The convergence rate will be dominated by the slowest stage, i.e. the decay rate closest to zero.

Differentiating $(1 - f(x))/x$ and setting this to zero yields

$$\begin{aligned} 1/x f'(x) - 1/x^2 f(x) &= 0 \\ 1/x(B + 2Cx) - 1/x^2(A + Bx + Cx^2) &= 0 \\ -A + Cx^2 &= 0 \end{aligned}$$

$$\begin{aligned} \min &= \sqrt{A/C} \\ &= \sqrt{\frac{1/N + (2b+1)/(2bN^2) + 1/(2bN^3)}{1/2b + 3/(2bN)}} \\ &= \sqrt{\frac{2b + (2b+1)/N + 1/N^2}{N+3}} \approx \sqrt{\frac{2b}{N+3}} \end{aligned} \tag{4}$$

Substituting the minimum given by Equation (4) gives an estimate of the convergence rate for long programs.

$$\begin{aligned} \frac{f(\min)}{\min}\frac{1}{N} &= 2\sqrt{AC}/N + B/N \\ &\approx \sqrt{2/(bN^3)} \end{aligned}$$

Figures 5, 8 and 9 indicate our model gives a rough approximation to the observed exponential decay rate. Interestingly, the average number of non-constants in 128 bits of memory used by programs implementing function D1 was 14. (Equation (4), assuming $b = 1$, suggests the minimum rate of loss of non-constant functions is when the memory contains 16 columns which are not all zero or all one.)

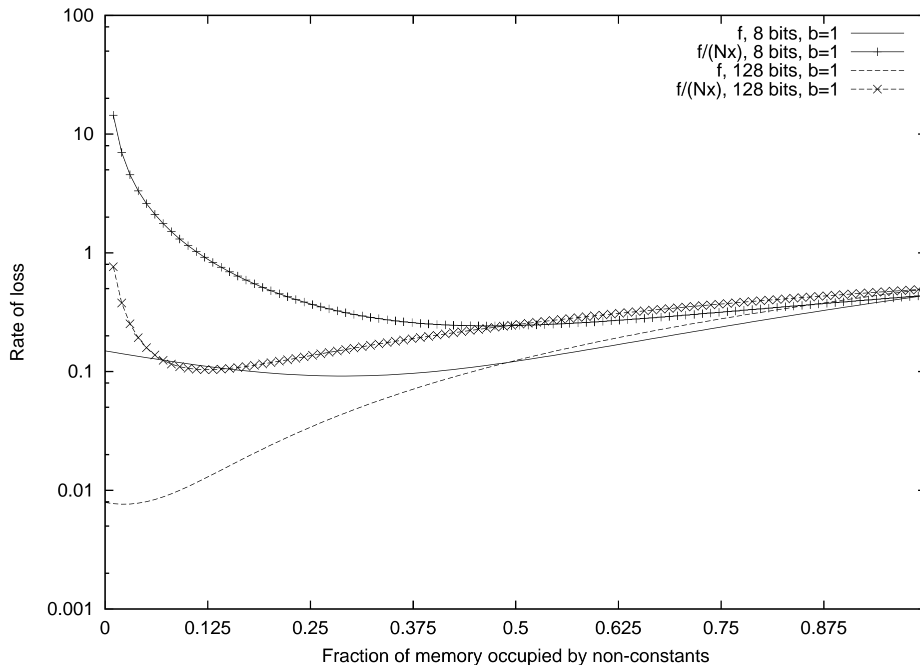

Figure 7 Dependence of expected rate of loss of non-trivial functions depends upon number of copies in memory. Note semi-stable region at $\approx \sqrt{2/N}$.

6 Discussion

The apparent slower convergence of functions implemented by random programs than of their outputs can be ascribed to two causes. Firstly all the popular non-trivial functions yield outputs that exactly match the limiting distribution of outputs. Thus considering only outputs can indicate near convergence even when there is a high fraction of non convergence when looking at the functions. More fundamentally there are many more functions that can be implemented than outputs that can be returned but exactly the same number of computer instructions to mix things up. So the minimum program size needed to implement every function is much longer than that needed to return every output value. Markov Minorization can be used to formalise this with lower bounds, unfortunately so far the bounds appear to be too weak to be useful.

Practical linear GPs write protect the input register. This means, as with tree based GP, the frequency of "Identity" functions will be much higher and so too will that of other "interesting" (i.e. non-constant) functions. Indeed they will be present in the limiting distribution. Thus interesting functions in tree GP and protected input linear GP are more frequent, but there is still a bias in favour of simple functions (see Figure 10). Possibly these may be related to GP's ability to find general solutions (Langdon 1998) and Occam's

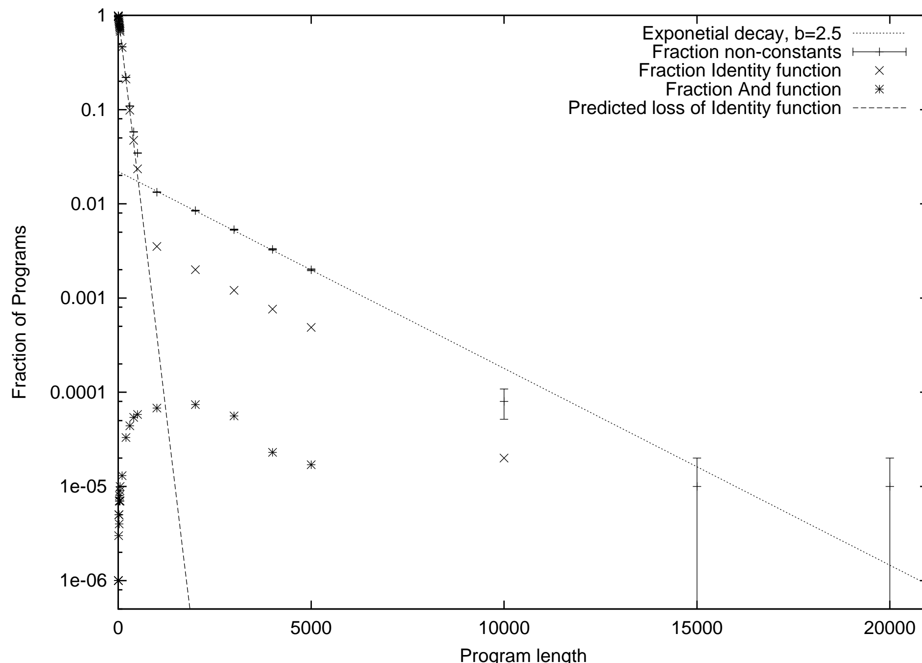

Figure 8 Fraction of non-constants in binary Boolean random linear functions (AND, NAND, OR, NOR, 128 bits). Steep line is prediction of exponential overwriting of Identity function. While dotted line is slower exponential decrease in non-constant functions. Experimental fit is slightly slower than predicted, as is shown by $b = 2.5$, while b should not exceed $n = 2$. Note the proportion of all non-constant functions appears to decrease with the same exponent.

razor. Initialising the memory with multiple copies of the input register might also bring some benefit to GP runs.

Figures 5, 8, 9, and 10, suggest it might be worthwhile investigating linear GP systems which impose a program length limit of about 10 N/m.

The numerical factors calculated by the model presented in Section 5.5 are only approximately correct. This warrants further analysis, perhaps based on a "gambler's ruin" approach. However our analysis stresses the role of memory in linear GP. Performance gains might be achieved by including analysis of the memory when programs terminate. Programs whose inputs $\times$ memory table was full of constant columns might be prevented from breeding, and those with many non-constants might receive a fitness bonus. While on-line monitoring of the non-constant count, could reduce run time if it were used to abort programs early when it falls to zero. However, even using machine code level parallelism (Poli and Langdon 1999), the monitoring overhead might be excessive. New fitness measures,

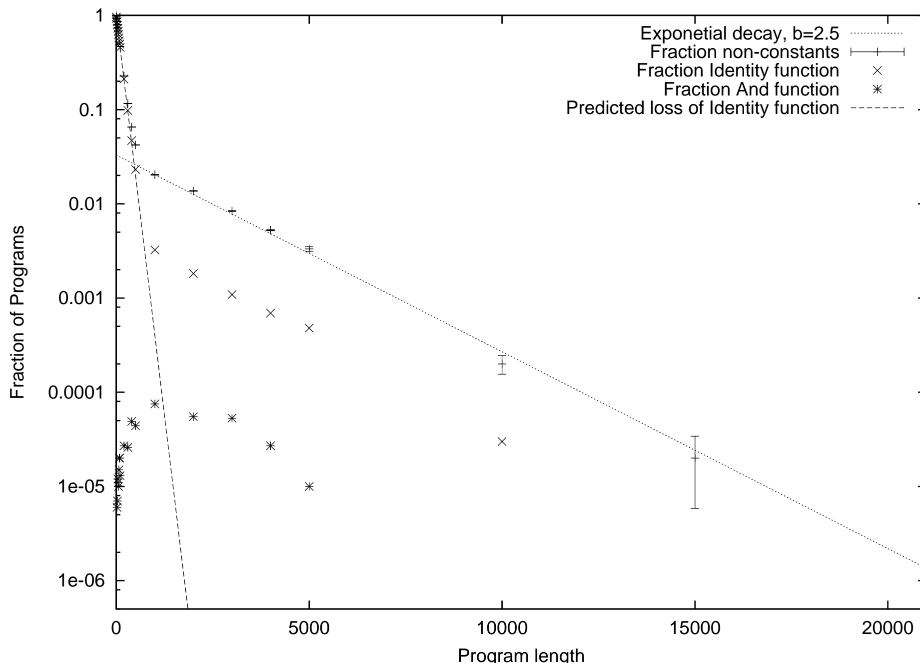

Figure 9 As Figure 8 but with 4 input bits rather than 2. Note experimental (dotted line) is more reasonable, since $n = 4$.

for problems like parity, might be devised where partial fitness would be credited using information content rather than Hamming distance to the target pattern.

We have considered the convergence of the distribution of all functions. How long it takes for a specific fitness distribution to converge will depend upon the nature of the fitness function. Once the distribution of functions has converged the fitness distribution must also converge, but it could converge substantially faster. If each non-trivial function convergences exponentially with the same exponent $\mathrm{O}(N^{-3/2})$, then, provided we exclude the constants, so too must the distribution of program fitness' for every non-trivial fitness function. That is in the limit of long programs in simple linear systems (cf. Section 5) all such fitness distributions will converge exponentially fast to a limiting distribution at the same rate. I.e., if we exclude constant functions, the distribution of fitness will converge to a limit as program exceed $\mathrm{O}(N^{3/2})$. (If constants are allowed to dominate the fitness distribution, convergence will be seen by $2.3N/m$.)

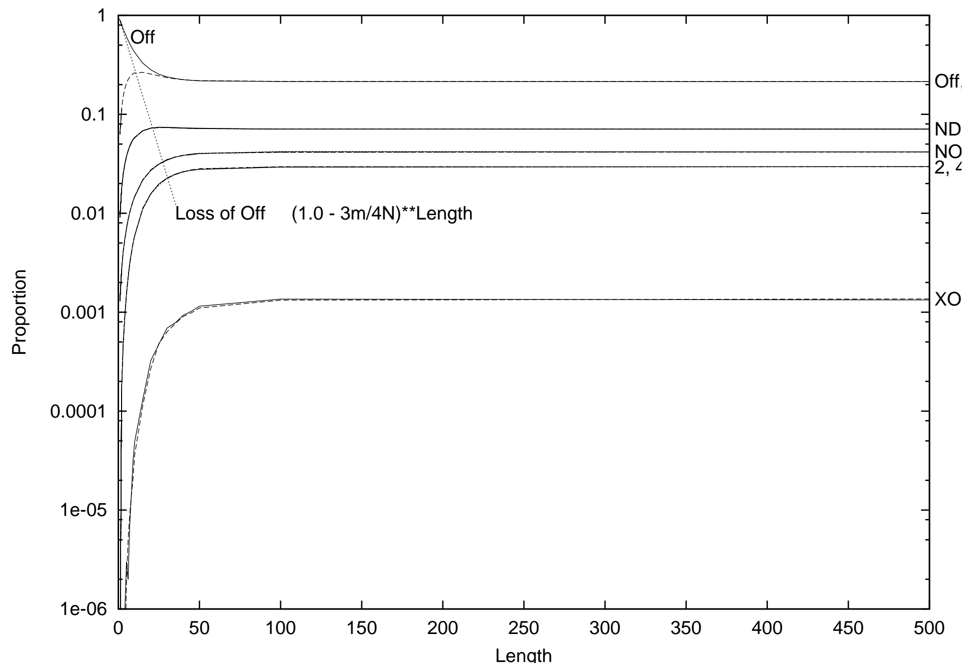

Figure 10 As Figure 4 but note write protecting the input register means all functions occur in the long random program limit.

7 Conclusions

The use of variable length linear programs in genetic programming (GP) is wide spread (Banzhaf, Nordin, Keller, and Francone 1998). We have investigated a model of such GP systems, which while restricted to Boolean operations at present, is not totally unrealistic. This allows us to predict the distribution of fitness for arbitrary fitness functions. It also stresses the importance of some pragmatic choices which have been made in GP. For example, if the input register is not write protected, then the proportion of "interesting" functions decreases rapidly, in the limit to zero, as programs get longer. (About 90% of programs of length 2.3 N/m are constants.)

The number of minimal solutions to XOR (even-2 parity) grows quadratically in memory size (cf. Section 4.1.1) but this corresponds to a rapid fall in proportion as memory is increased.

Section 4 provides an exponential bound ($2.3aI^a$) on program size for convergence of functions implemented by any linear computer. For an average linear computer without branches etc. and with a large instruction set, we show that the limiting distribution contains only functions that are constants. Again convergence is exponentially fast with 90% of programs of length 1.6 $n2^N$ yielding constants.

In the Boolean linear systems considered, complex functions are very rare in short programs and appear to reach a peak in their frequency near $l = N/m$. This suggests a size limit of $O(N/m)$ might be beneficial to linear GP. The peak is followed by exponential decline, with the same exponent ($\approx \sqrt{2/N^3}$) as the other non-trivial functions. Since $\sqrt{2/N^3} < I$, the number of solutions grows exponentially with program length l. It is also appears that the frequency of complex functions decreases dramatically as the number of operations needed to created them from the program's inputs increases. I.e. most functions are parsimonious.

Acknowledgements

I would like to thank Jeffrey Rosenthal, Tom Westerdale, James A. Foster, Riccardo Poli, Ingo Wegener, Nic McPhee, Michael Vose, Jon Rowe, Wolfgang Banzhaf, Tina Yu, and several anonymous referees.

References

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

Bishop, P. G. (1997, 2-5 Nov). Using reversible computing to achieve fail-safety. In *Proceedings of the Eighth International Symposium On Software Reliability Engineering*, Albuquerque, NM, USA, pp. 182–191. IEEE.

Diaconis, P. (1988). *Group Representations in Probability and Statistics*, Volume 11 of *Lecture notes-Monograph Series*. Hayward, California: Institute of Mathematical Sciences.

Langdon, W. B. (1998). *Genetic Programming and Data Structures*. Kluwer.

Langdon, W. B. (2002, 9-13 July). Convergence rates for the distribution of program outputs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (Eds.), *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, New York, pp. 812–819. Morgan Kaufmann Publishers.

Langdon, W. B. and R. Poli (2002). *Foundations of Genetic Programming*. Springer-Verlag.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. Ph. D. thesis, Universität Dortmund, Fachereich Informatik.

Poli, R. and W. B. Langdon (1999). Sub-machine-code genetic programming. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (Eds.), *Advances in Genetic Programming 3*, Chapter 13, pp. 301–323. MIT Press.

Rosenthal, J. S. (1995). Convergence rates for Markov chains. *SIAM Review 37*(3), 387–405.

Stirzaker, D. (1999). *Probability and Random Variables A Beginner's Guide*. Cambridge University Press.