GECCO'2002 Late Breaking paper E. Cantu-Paz, editor, New York, 9-13 July

Random Search is Parsimonious

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 in detail the distribution of Boolean functions implemented by random non-recursive programs, similar to linear genetic programming. Most functions are constants, the remainder are mostly simple.

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.

1 Introduction

This late breaking paper presents extensions to work in the main proceedings [Langdon, 2002]. Much of the background is described in the main proceedings.

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 the main proceedings we put numbers to and defined scaling laws for the length of linear genetic programs, so that the distribution of program outputs produced by random programs is effectively independent of their size. Here we provide similar results on the distribution of *functions*.

The next section summarises our Markov model of linear GP, total variation distance (Section 1.2) and minorization (Section 1.3). Minorization is used to provide a weak upper bound for any computer (Section 2) and application to an average computer is discussed (Section 2.1). Section 3 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 4) and our conclusions (Section 5).

1.1 Markov Model of Programs

We want to know about every possible program. Our approach is to sample 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. The major interest is to generate samples of different sized programs to see how the search space changes.

We limit ourself to programs that run through a given number of instructions and stop. They do not loop.

The computer's data memory is zeroed. The program's inputs are loaded into the input register (a memory cell). The program runs and in the process reads and write to memory. When it stops, its answer is read from the output register (a memory cell). It is well known that a program can be interrupted (check pointed) and restarted later without ill effect, provided it is restarted from where it got to and the memory is restored to the condition that it was in before the program was suspended. The program counter and memory are all the state that is needed. If we treat the program as a random sequence of instructions, then we have a Markov process. This is important because there are 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.

1.2 Convergence Metric

In the next two 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

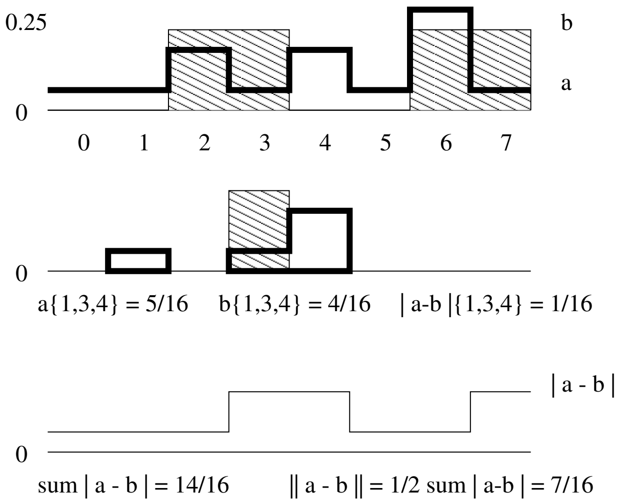

Figure 1: Total variation distance between probability distributions a and b. The probability of being in subset $\{1, 3, 4\}$ with a and b are compared. $||a - b||$ is the largest difference across all 256 possible subsets.

x, not just single points. See Figure 1. In mathematical terminology $||a - b|| = \sup_{x \subset \chi} |a(x) - b(x)|$ [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).

1.3 Markov Minorization

A Markov transition matrix P can be thought of as telling us how a randomly chosen operation mixes things up. The computer's memory for each input to the program defines the *function* the program implements at this point in its execution. Executing one randomly chosen instruction will update the pattern and so to the function. In practical computers the number of possible instructions is large but much less that the number of possible bit patterns in memory and so very much less than the number of possible functions. Thus after one instruction the implemented function can only be one of a small fraction of the total. 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 of functions can increase. After $a > 2$ steps even more functions can be reached. Minorization [Rosenthal, 1995] is a way of quantifying this mixing (or at least provides 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. I.e. β is given by adding together the chance of getting, in a steps, from one function to the least likely other function. I.e. β is the sum of the minimum values of the entries in each column of the matrix $P \times P \times \cdots \times P = P^a$.

2 Convergence Bound on Functions Implemented on Any Computer

Let a be the minimum number of program instructions required to implement every one of the functions. That is, from the initial starting condition, s_0, for each of the possible functions, there is at least one program of a or fewer instructions which implements the function. We will gloss over one difficulty, which is that here we consider only functions from the input register to the output register. I.e. we ignore the $N - m$ bits of memory not in the output register. Ignoring this potential violation of the Markov property for the time being, we have a minorization condition for P^a. In fact $P^a_{s_0,j} \geq I^{-a} > 0 \ \forall j$. (Where I is the number of instructions provided by the computer.) Therefore $\beta \geq I^{-a}$ and so for any computer:

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

I.e. we are guaranteed that the difference between the probability distribution for our computer after executing a random program of length l instructions and its limiting distribution (after ∞ instructions) 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 number of instructions to implement any of the possible functions.

Now we come to the major difficulty, Inequality (1) can be a very weak upper bound. Even for the simplest possible instance of a useful computer (cf. Section 3) the bound is too weak to help. E.g. 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. However only three instructions are needed, i.e. $a = 3$. In fact there are $32(N - 1)(N - 2)$ programs of length three that implement even-2 parity. So we can improve our bound on β from $1/I^3$ to $32(N - 1)(N - 2)/I^3$. There are four operation codes, two memory address are read in each instruction and the output is written back to memory so $I = 4 \times N^3$.

$$||\mu_l - \pi|| \leq \left(1 - \frac{32(N-1)(N-2)}{(4 \times N^3)^3}\right)^{l/3}$$

Even for a 1 byte ($N = 8$) computer this gives a convergence length of $l \approx 13.8N^9/(N-1)(N-2) \approx$ 44 million. In fact near convergence is seen by programs of about 25 instructions cf. Figure 3.

2.1 The Average Computer

[Langdon, 2002] argues that each randomly connected computer is representative of all computers. It uses the minorization process (described above in Section 1.3) to prove the distribution of outputs produced by it converges and how may instructions are needed to get close to the limit. An informal argument is presented that convergence of the distribution of functions converges 2^n times slower. It may be possible to strengthen this result.

3 AND NAND OR NOR Computer

The model used in the main proceeding is used again but we look at the functions implemented by programs and give new examples.

3.1 Convergence of Random Outputs

In this model 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.

We look at the distribution of memory patterns that are produced by running all programs of a given length, l, and by considering a large number of random programs length l. I.e. programs with l randomly chosen instructions.

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.

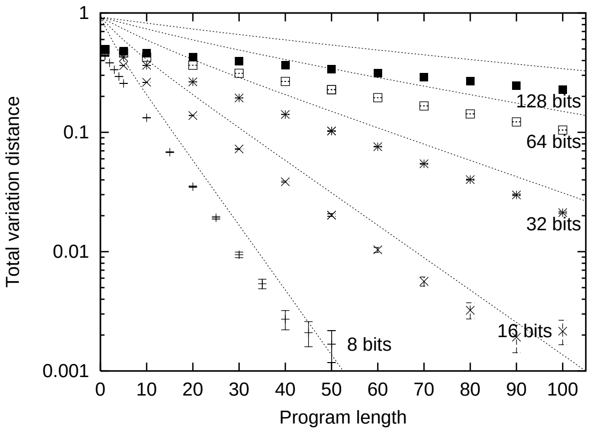

Figure 2: Convergence of outputs of random 4 bit Boolean (AND, NAND, OR, NOR) linear programs with different memory sizes. Note the agreement with upper bound $\sqrt{1/2(\exp(me^{-2l/N})-1)}$ (dotted lines).

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 are, 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 and as random programs (length l) get bigger

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

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 2 confirms this. If m or N are quite small, one needs to be caution about extracting exact values. However the bound gets tighter with larger memory.

3.2 Convergence of Random Functions

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 we can consider every program as a function from the inputs $(0..2^n-1)$ to

Table 1: 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							
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

the complete memory $(0..2^N-1)$. We assume here every possible input value is tested. We will be primarily interested in just the output register but for the time being let us consider the whole memory. We can build a table N bits wide with a row for each input. Each row contains the memory pattern after the program halts after having started with the corresponding input, cf. Table 1.

We know (cf. Section 3.1) in the limit of long random programs, that for any given input pattern each bit is as likely to be set as clear. I.e. any row of the table is a uniform random pattern. 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 finite state 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 output is $x+p \bmod 2^m$. Where x is the input and p is a constant (specific to that program). That is, each row in our 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. In contrast useful computers do not 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 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.

Consider what happens when the CPU reads two columns that are not in the limiting distribution. That is, one (or both) contains a mixture of zeros and ones. Consider one of the elements which is not in the majority class. Tables 2, 3 and 4 show the state of each output bit generated by each instruction, both in terms of its value and also how it compares to the other bits within the same column. In particular if it is in the majority or not.

Tables 2, 3 and 4 show if the elements of a particular row in both input columns both contain the majority bit value (i.e. 0 or 1) then that row of the output column will always have the majority value, for that column. Conversely, if both elements are of the minority class then (depending on their values) the output row may be in the majority class. Table 4 shows that when one element contains a majority bit and the other a minority and given equal chance of the four Boolean operations, each output is equally likely. Therefore overall there is a consistent basis towards generating output columns with an increased number of bits that

Table 2: Output of Boolean AND on Function table. Note operations between bits in the majority^M always yield outputs in the majority and on average the majority tends to increase.

AND	0^M	0	1^M	1
0^M	0^M	0^M	0^M	0^M
0	0^M	0	0	0^M
1^M	0^M	0	1^M	1
1	0^M	0^M	1	1

Table 3: Output when both input bits are not in the majority^M. Note if the bit values are different, the output is in the majority.

AND	0	1
0	0	0^M
1	0^M	1

OR	0	1
0	0	1^M
1	1^M	1

NAND	0	1
0	1	1^M
1	1^M	0

NOR	0	1
0	1	0^M
1	0^M	0

Table 4: Output when one input bit is not in the majority^M. Note if AND, NAND, OR and NOR are equally likely, the output is in the majority 50% of the time.

AND	0^M	1^M
0	0^M	0
1	0^M	1

OR	0^M	1^M
0	0	1^M
1	1	1^M

NAND	0^M	1^M
0	1^M	1
1	1^M	0

NOR	0^M	1^M
0	1	0^M
1	0	0^M

are the same. I.e. each random operation tends to move the function table closer to one of the patterns in the limiting distribution.

3.3 Model of Convergence of Functions

This is a some what empirical model of convergence of the function implemented by random Boolean programs. 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 worry about this.)

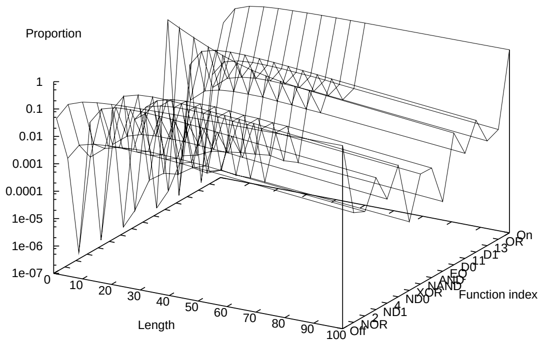

Figure 3: Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits). Half long programs implement Always-on and half Always-off. Short programs are more varied and many implement the identity function "D0". Longer programs are needed to achieve convergence of functions than of outputs

Initially, where the input and output registers overlap, the outputs are equal to the inputs. 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 converges continues but at slower exponential rate. We construct two models to explain this.

3.4 Initial Convergence of Boolean Functions

Initially the output register column contains the identity function and the rest is all zeros. 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 $2.3N/m$.

In Figure 4 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 an $\approx 1/2\ n/N$ correction term.

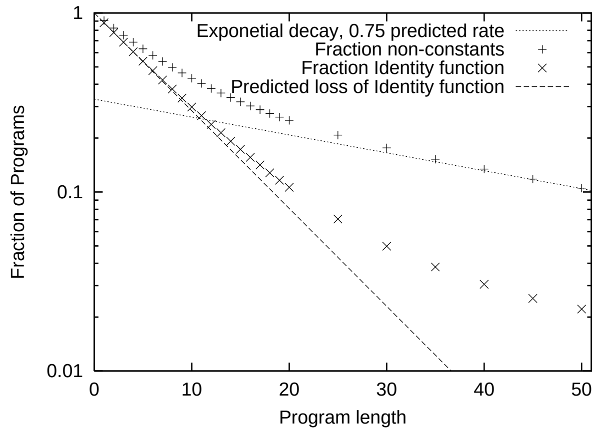

Figure 4: Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits) Steep line is prediction of exponential overwriting of identity function, "D0".

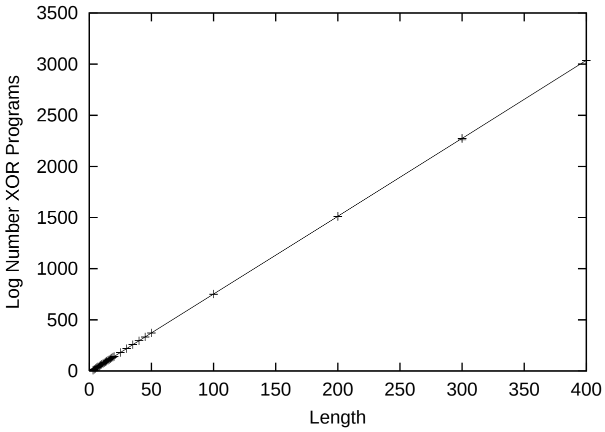

Figure 5: Number of programs which implement 2-bit odd parity (AND, NAND, OR, NOR, 8 bits).

As other parts of the memory are written to the rapid convergence due to lose of the identify function (cf. D0 in Figure 3) slows. Cf. "knee" in observations as program length passes N.

3.5 Later Convergence of Boolean Functions

After O(N) instructions the majority of the memory is likely to have been overwritten and (initially) is 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, 7 and 8 show the proportion of non-constant functions falls exponentially, albeit with a smaller exponent. 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 5.

In order to model the small fraction of interesting functions we assume that they only arise in programs which have avoided over writing all of the input register with a constant before saving multiple copies of it elsewhere in memory. Computer instructions reading (parts of) these copies of the input register lead to other interesting (i.e. non-constant functions). The proportion of each such function, 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 3. 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.

Let x be the fraction of memory containing copies (or inverted copies) of one of the bits of the input register. After executing one randomly chosen instruction, the number of copies will be the same, one higher or one lower.

The only way the number can increase, is by selecting one or more copies as input, selecting a non-copy for output, and then the operation not generating a constant. Given either one or two random non-constants as input and a random operation (from our set of four, AND NAND, OR and NOR), the chance of creating a constant is 50%. So $P(x \to x + 1/N) = 1/2(1 - (1 - x)^2)(1 - x) = x - 3/2x^2 + 1/2x^3$

Similarly the number of copies can fall by selecting a copy as output, and either selecting constants for both input columns or selection one or more copies but generating a constant from them. So $P(x \to x - 1/N)$ $= x((1 - x)^2 + 1/2(1 - (1 - x)^2)) = x - x^2 + 1/2x^3$

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.

$$\begin{aligned} f(x) &= P(x \to x - 1/N) - P(x - 1/N \to x) \\ &= 1/N + 3/2/N^2 + 1/2/N^3 \\ &\quad -3(1/N + 1/2/N^2)x \\ &\quad +(1/2 + 3/2/N)x^2 \\ f(x) &= A + Bx + Cx^2 \end{aligned}$$

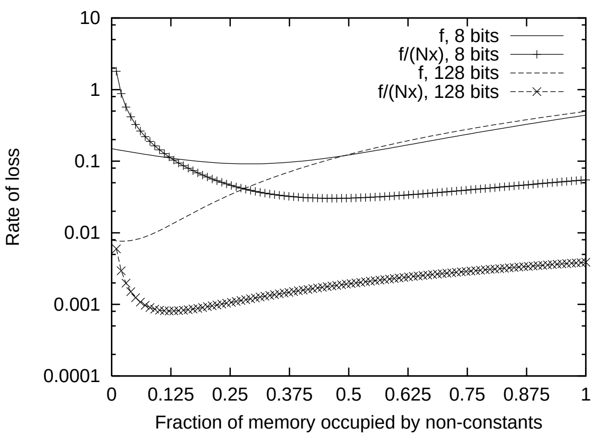

Figure 6: Dependence of expected rate of loss of non-trivial functions depends upon number of copies in memory. Note semi-stable region at $\approx \sqrt{2/N}$.

The expected decrease is small but never negative, cf. Figure 6. The expected fractional decrease is $(1 - f(x)/(Nx))$. 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 $(1 - f(x))/(Nx)$ and setting this to zero yields

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

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

Substituting the minimum given by Equation 2 gives an estimate of the convergence rate for long programs.

$$\begin{aligned} f(\min)/\min N &= 2\sqrt{AC}/N + B/N \\ &\approx 1.4N^{-3/2} \end{aligned}$$

Figures 4, 7 and 8 indicate our model gives a rough approximation (within 40%) 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 2 suggest the minimum rate of loss of non-constant functions is when the memory contains 16 columns which are not all zero or all one).

4 Discussion

The 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. We anticipate convergence in such systems and tree GP will be similar to "slow phase" (Section 3.5). However 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. Possibly these may be related to GP's ability to find general solutions [Langdon, 1998] and Occam's razor.

The numerical factors calculated by the model presented in Section 3.5 are only approximately correct. This warrants further analysis, perhaps based on a "gambler's ruin" approach.

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. However it could converge substantially faster. If we are right in suggesting each non-trivial function approaches convergence exponentially with the same exponent, then so too must the distribution of program fitness' for every non-trivial fitness function.

5 Conclusions

The use of variable length linear programs in genetic programming is wide spread 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

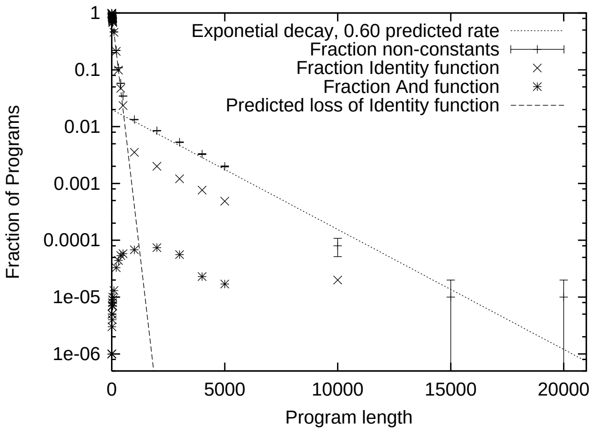

Figure 7: Fraction of non-constant in binary Boolean random linear functions (AND, NAND, OR, NOR, 128 bits) Steep line is prediction of exponential overwriting of identity function, "D0". While dotted line is slower exponential decrease in non-constant functions. Note the proportion of all non-constant functions appears to decrease with the same exponent.

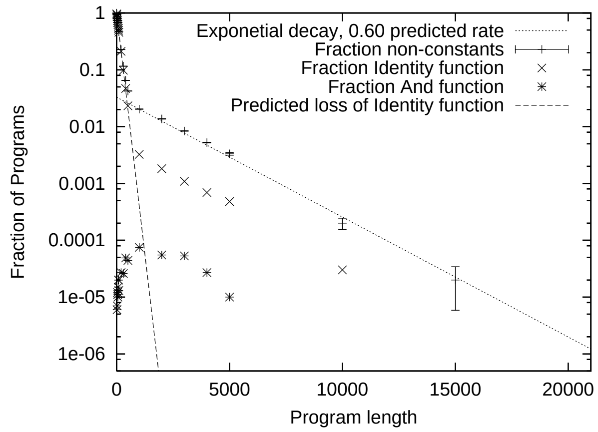

Figure 8: As Figure 7 but with 4 input bits rather than 2.

for arbitrary fitness functions. It also stresses the importance of some pragmatic choices which have been made in GP but without justification. For example, if the input register is not write protected, then the proportion of "interesting" functions decreases rapidly (i.e. within $\approx 2.3\ N/m$) to near zero as programs get longer.

However, in the Boolean linear systems considered, more complex functions are initially very rare and appear to reach a peak in their frequency near N/m. Followed by exponential decline, with the same exponent ($\approx \sqrt{2}/\sqrt{N^3}$) as the other non-trivial functions. 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.

References

[Diaconis, 1988] Persi Diaconis. *Group Representations in Probability and Statistics*, volume 11 of *Lecture notes-Monograph Series*. Institute of Mathematical Sciences, Hayward, California, 1988.

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

[Langdon, 1998] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming.* Kluwer, Boston, 24 April 1998.

[Langdon, 2002] W. B. Langdon. 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, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 812–819, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Poli and Langdon, 1999] Riccardo Poli and William B. Langdon. Sub-machine-code genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA, June 1999.

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