Genetic Programming and Evolvable Machines (2009) 10:5–36
DOI 10.1007/s10710-008-9065-y

Scaling of Program Functionality

W. B. Langdon

Preprint of Genet Program Evolvable Mach (2009) 10:5–36, *Revision* : 1.30

Abstract The distribution of fitness values (landscapes) of programs tends to a limit as the programs get bigger. We use Markov chain convergence theorems to give general upper bounds on the length of programs needed for convergence. How big programs need to be to approach the limit depends on the type of the computer they run on. We give bounds (exponential in N, $N \log N$ and smaller) for five computer models: any, average or amorphous or random, cyclic, bit flip and 4 functions (AND, NAND, OR and NOR).

Programs can be treated as lookup tables which map between their inputs and their outputs. Using this we prove similar convergence results for the distribution of functions implemented by linear computer programs. We show most functions are constants and the remainder are mostly parsimonious.

The effect of ad-hoc rules on genetic programming (GP) are described and new heuristics are proposed.

We give bounds on how long programs need to be before the distribution of their functionality is close to its limiting distribution, both in general and for average computers. The computational importance of destroying information is discussed with respect to reversible and quantum computers.

Mutation randomizes a genetic algorithm population in $\frac{1}{4}(l+1)(\log(l)+4)$ generations.

Results for average computers and a model like genetic programming are confirmed experimentally.

Keywords search landscapes, evolutionary computation, genetic algorithms, genetic programming

Mathematical and Biological Sciences and Computing and Electronic Systems,
University of Essex, CO4 3SQ, UK

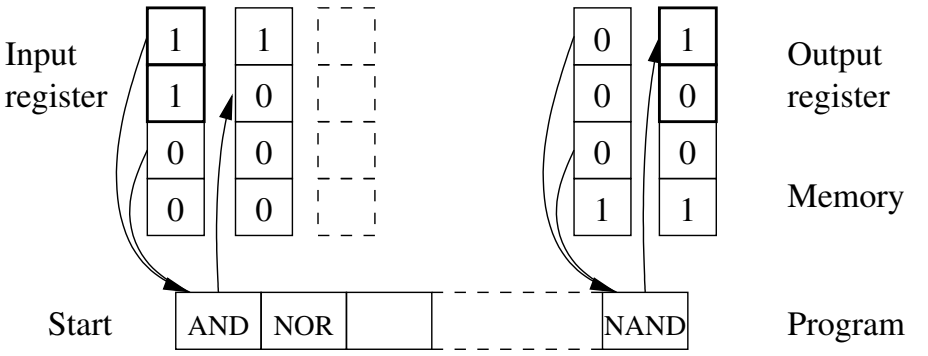

Fig. 1 A linear program. Each instruction reads from memory, operates on the data and then writes the result back to memory. The initial contents of memory and after each instruction are shown above the program. In this example, the top two bits are used both for initial input (1,1) and output (0,1).

instruction	in1	in2	out
AND	2	0	1

Fig. 2 An example instruction. The instruction read two bits from memory (from addresses 2 and 1), ANDs them, and writes the result back to memory (address 1).

1 Introduction

Figures 1 and 2 shows the operation of a program and the layout of its instructions. Each instruction in the program can be seen principally by its use of and effect on the contents of the computer's memory. As we shall see, by treating finite computer programs simply as the sequence of computer instructions they execute and the resulting changes to the computer's memory, powerful convergence results can be obtained.

Our approach can be criticised for its very generality. We prove results about programs in general. This can be likened to the oft quoted "No free lunch" theorem (Wolpert and Macready 1997), which tells us about average performance but does not tell us anything about search performance on specific instances. Similarly results proving problems are in NP apply to the problem in general. Specific instances of such problems can be easy to solve. Nevertheless both NFL and NP have been very useful results.

Given the reader has constructed or evolved a program our approach has only a little to say about that specific program. Instead we aim to tackle programming in general. That is, what are the characteristics of the space of all possible programs? How difficult is it to search this space? How do the characteristics change with different types of computer architecture? Such analysis can cast light on the role of ad hoc rules buried in current commercial GP (Foster 2001) and how they improve the search space. Are there special features of some spaces which help one type of search more than another?

Fitness landscapes are an appealing metaphor for evolutionary and local search (Reidys and Stadler 2002). 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 a tiny fraction of

small benchmark problem spaces. Our approach is to provide quantified metrics for the general case 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 sufficiently large programs, their fitness distribution will converge. We shall answer the question: how long do programs have to be in practice to get reasonably close to this limit. In (Langdon 2002a) 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. While in (Langdon 2002b) we provided similar results on the distribution of functions.

Although our initial motivation has been the need to put GP on firmer theoretical foundations, these results apply to the space that GP searches. They are not specific to how it searches or its fitness landscape. Since they apply to the space, rather than the search, they are applicable to other automatic programming techniques. Indeed our results are general and apply to specific classes of functions and may be of use to anyone interested in those functions.

The next section summarises the Markov model of linear programs, while Section 3 describes total variation distance as a convergence metric. Minorization is summarised in Section 4 and used to provide an upper bound for *any* computer (Section 5). Section 6 gives an example of exponentially slow convergence. However Section 7 gives an example where convergence is much faster than the general upper bound. Section 8 proposes an average or random computer which could serve as a model for amorphous computing. Minorization is applied to it, yielding upper bounds indicating on most computers fitness distributions converge rapidly. Those interested in results for genetic programming, rather than the analysis technique, may jump directly to Section 9.

Section 9 describes detailed models for the distribution of outputs and functions implemented by a computer similar to that used in linear genetic programming. These results compare well with measurements. To highlight the computational importance of destroying information, Section 10 briefly considers reversible computers as a special case of quantum computing.

Section 11 is a slight digression. In it we give explicit convergence results for linear bit string genetic algorithms. This is followed by a discussion, which includes suggestions for improving GP (Section 12), and our conclusions (Section 13). A complete list of results is given in Appendix A.

2 Markov Models of Program Search Spaces

This section explains how we can use standard Markov modelling techniques to prove facts about the distribution of programs in terms both of their outputs and their functionality. In particular how both distributions change with changes in program length.

In (Langdon and Poli 2002) we deal 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.

2.1 Markov Models of Program Outputs

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 itself is not stored in the computer's data memory.) 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 ourselves to fixed programs that run through a given number of instructions and stop. They do not loop or modify themselves and they always terminate (Langdon 2006). 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 et al. 1998). Section 12 briefly discusses possible extensions to loops, programming without a program counter (Banzhaf 2005) and any time algorithms.

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, (Feller 1957, 1966; Haggstrom 2002). 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. See Table 1.

When we consider random programs, we mean every possible instruction within the program is equally likely. (Thus if there are I instructions, the chance of choosing any one of them is $1/I$. See Table 2 for a list of frequently used symbols.) 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.

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 or perhaps the design includes a NOP.) Given (1) and (2) the Markov process is ergodic and irreducible. This means that after a large number of random updates the probability of the computer's memory holding a particular pattern of bits will settle into a limiting distribution. Once the limit is reached, additional random updates, continue to change bits but on average the distribution of bit patterns in memory does

Table 1 Schematic of simple Markov matrix. For illustrative purposes, the computer has four memory bits and two instructions: OR (∘) and AND (×). Both read from bits 0 and 1 and write to bit 3. The 16 possible memory states are represented in hexadecimal. For example, see the second row, OR(0,1) sets bit 3 (next state 1001). That is, OR takes the computer from state 1 to state 9. In contrast AND (×) leaves the computer in state 1. (Instructions which don't change the state effect the matrix's diagonal.) If we chose between our two instructions at random the matrix elements (∘) and (×) are 0.5 (⊗ = 1.0) and the matrix becomes a Markov matrix representing the probabilities of moving from one state to another. Of course in a real example, the matrix will be far far bigger and there will be many many instructions.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	⊗									∘						
1		×									∘					
2			×									⊗				
3					⊗									∘		
4						×									∘	
5							×									⊗
6	⊗									∘						
7		×									∘					
8			×									⊗				
9					⊗									∘		
A						×									∘	
B							×									⊗
C	⊗									∘						
D		×									∘					
E			×									⊗				
F					⊗									∘		

Table 2 Common Symbols

β	is the sum of the minimum values of the entries in each column of a matrix. I.e. $\beta = \sum_j \min_i M_{ij}$. β is used in minorization conditions.
I	number of different computer instructions (from which programs can be constructed). For simplicity, we usually assume, in random programs, each instruction is equally likely, although some results, e.g. Markov limiting behaviour, do not require the probability to be $1/I$, only that it be fixed.
l	length of a program, i.e. number of instructions in it, or the l^{th} instruction from the start of a program. In GAs (Section 11 only) the number of bits in the chromosome.
N	size of computer's memory in bits.
n	size of computer's input register in bits.
m	size of computer's output register in bits.

not change any more. Also, while the distribution may depend upon the bit pattern, it does not depend upon the program's inputs.

2.2 Markov Models of Program Functionality

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.

Table 3 contains a very simple example of a six bit computer. It has a three bit input register, allowing each program to process values 0 to 7. The computer's output register occupies two bits. Thus a program can output 0, 1, 2 or 3. Thus it can implement $4^8 = 65\,536$ functions. Since the number of programs rises exponentially with their

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

Before						After r4=AND(r0,r2)					
Memory						Memory					
			Input							Out	
r5	r4	r3	r2	r1	r0	r5	r4	r3	r2	r1	r0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	1	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	1	0	1	0	1	0	1	0	1
0	0	0	1	1	0	0	0	0	1	1	0
0	0	0	1	1	1	0	1	0	1	1	1

length, the number of possible programs considerably exceeds the number of functions. Table 3 shows the action of a program containing a single instruction. It implements the identity function (i.e. the output equals the lower two bits of the input). Obviously on this computer there are many different programs which implement the identity function.

A program's external functionality is defined by its actions on the I/O registers. However to prove convergence results across all programs we must consider the whole of the computer's memory. In general we must also consider other elements of its state, such as internal CPU registers and flags (such as the carry flag). For simplicity, we shall treat CPU state as though registers, flags etc. are simply special parts of memory. (Cf. the memory mapped registers of the PDP-11).

Since we treat all state as part of memory, the truth table of inputs v. memory completely defines a program. (The right hand side of Table 3 is an example of such a truth table.) In other words, it is only necessary to consider the state of the computer after it has run a program on each input. So if another program produces the same truth table it will be totally equivalent. (In our example, the truth table occupies $6 \times 8 = 48$ bits, so there are $2^{48} = 2.8\ 10^{14}$ possible tables.)

Suppose we test a program by running it on each possible input in a predefined sequence and each time it stops we record the computer's state before reinitialising the computer and running the program on the next input. This gives us a complete inputs v. memory truth table for that program. If another program produces the same truth table, the two programs are exactly equivalent, even if they contain different instructions or caused the computer to pass through different states whilst they were running. Since the states are always the same at the end: 1) the output register will be the same, 2) they will implement the same function and 3) if they are extended by appending the same code fragment, so that they run both run the same additional instructions, then the extended programs will remain locked in step and produce the same answer and implement identical functions.

Previously, when we considered running a program just once, it was only necessary to consider the row of the input v. memory truth table for the program's input. In general, the rows of the table are not independent. Therefore, if we run the program multiple times (to find the function it implements) we must consider the whole truth table. The whole table contains all the relevant state information, so execution of

any of the computer's instructions moves the computer to a new state which is fully determined by the current state (truth table). I.e. each instruction deterministically moves from one state of the $(2^N)^{2^n}$ possible states to another. So we can again do our trick of considering a random program to be made of a randomly chosen sequence of instructions, each performing a state transition and treat these as a Markov process.

Again we need to impose weak constraints on the computer to ensure the new Markov process is both ergodic and irreducible. One way of doing this is to make sure that our computer can implement any Boolean function of the inputs and write this to any part of memory. Given enough instructions, the truth 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.) Secondly we also need 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. Having made sure the Markov process on the inputs v. memory truth table is ergodic and irreducible, we know that, given enough random instructions, the probability of each possible inputs $\times$ memory table will settle into a limiting distribution π.

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. Note the difference is taken over *all subsets*. i.e. every possible grouping of states x, not just single points. In maths: $||a - b|| = \sup_{x \subseteq \chi} |a(x) - b(x)|$ (see Figure 3). (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, Sections 6 and 7, we will show that the distribution of functions converges at the same rate as the distribution of outputs. However in neither case is the computer able to do general computation. In Section 9 we consider more useful models.

4 Markov Minorization

When considering finite 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).

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

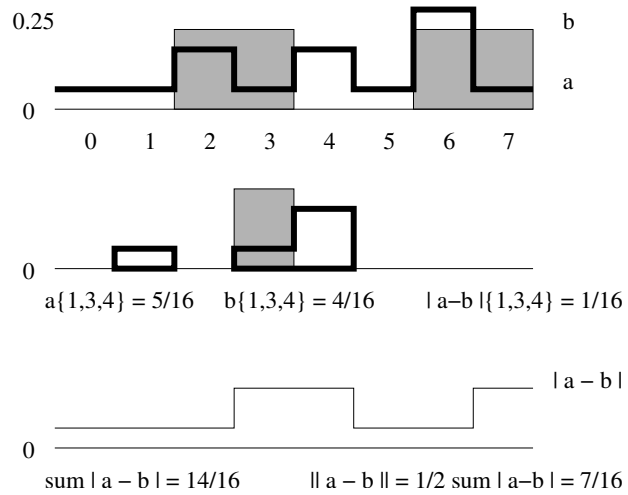

Fig. 3 Total variation distance between probability distributions a and b. Top panel shows a (solid line) and b (shaded). Middle compares the probability of being in subset $x = \{1, 3, 4\}$ with a and with b and calculates their difference, $_x|a(x) - b(x)| = 1/16$. $||a - b||$ is the largest difference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

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 distribution after $l = ka$ instructions μ_{ka} and limiting π distribution obeys:

$$||\mu_{ka} - \pi|| \leq (1 - \beta)^k$$

where

$$\beta = \sum_j \min_i P_{ij}^a$$

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.

If we can find a value for β that is greater than zero, not only does a limiting distribution exist but also we can place an upper limit on the deviation between the actual distribution after l steps and the limit. Further the difference between the actual and limiting distributions falls exponentially or faster with l.

5 Any Computer

The first subsection (5.1) gives a quantitative upper bound on the convergence of the distribution of outputs produced by any computer which fits the general framework given in Section 2. Subsection 5.2 provides the corresponding bounds for functions.

5.1 Convergence Bound on Outputs of Any Computer

Again let P_{ij} be the probability that starting in state i the next operation will take us to state j. (If j cannot be reached from i in one move, then $P_{ij} = 0$.) In order to get a minorization condition we consider P^a (rather than P). This means, instead of looking at the available state transitions if each of the I instructions is used once, we consider the transitions possible when they are used a times. For any given state there are up to I^a states the computer's memory could be in after a instructions. (Ignoring overlaps, each is equally likely.) So if $I^a \geq 2^N$ it is now possible that in at least one column of P there will be no zero entries.

From the way that we constructed our computer, it is possible, eventually, to move from the starting state s_0 to any state y. Let a_{data} be the number of steps required. This meets the minorization condition for $P^{a_{\text{data}}}$. In fact $P^{a_{\text{data}}}(s_0, y) \geq I^{-a_{\text{data}}} > 0$ $\forall y$. Therefore $\beta \geq I^{-a_{\text{data}}}$ and so for any computer:

$$||\mu_l - \pi|| \leq (1 - I^{-a_{\text{data}}})^{\lfloor l/a_{\text{data}} \rfloor} \tag{1}$$

Let us say when the actual distribution is within 10% of the limit, i.e. $||\cdot|| = 10\%$, we are close enough. Rearranging gives, for any computer with I instructions, an upper limit on the convergence length l of $2.30 a_{\text{data}} I^{a_{\text{data}}}$ Where a_{data} is the number of instructions to reach any state. Note this is an upper bound on l, the distribution μ_l may approach its limit π for much shorter programs.

Note, given a set of minimal programs which together can set the computer's memory to anything, we know: 1) the distribution of the computer's outputs after running a randomly selected program has a limit, and 2) the length of the longest program in the set a_{data} gives an upper limit on the length of programs needed to guarantee the output of random programs approaches this limit. And that both are true for any computer.

5.2 Convergence Bound on Functions Implemented on Any Computer

To establish a minorization bound on the distribution of functions implemented by any computer, let $a_{\text{functions}}$ 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_0, ensure that there is at least one program of $a_{\text{functions}}$ instructions which sets the table to any of its 2^{N2^n} values. If there are I instructions, the number of programs of length $a_{\text{functions}}$ is $I^{a_{\text{functions}}}$. So $P^{a_{\text{functions}}}(s_0, j) \geq I^{-a_{\text{functions}}}$. Therefore β is at least $I^{-a_{\text{functions}}}$ and so for any computer:

$$||\mu_l - \pi|| \leq (1 - I^{-a_{\text{functions}}})^{\lfloor l/a_{\text{functions}} \rfloor} \tag{2}$$

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 $l \leq 2.30 a_{\text{functions}} I^{a_{\text{functions}}}$. Where here $a_{\text{functions}}$ is the minimum number of instructions needed to set the inputs × memory table to any value.

As in Section 5.1, the length of the longest program $a_{\text{functions}}$ in the set of minimal programs (which together can set the computer's inputs v. memory to anything) tells us how the computer will converge. Firstly we are sure that the distribution of the functions implemented by randomly selected programs has a limit. Secondly $a_{\text{functions}}$ gives an upper limit on the length of programs needed to guarantee the functionality of random programs approach this limit. Again both are true for any computer.

5.3 Weakness of Minorization Bound

Note both (1) and (2) lead to an exponential bound. Section 6 shows we can devise examples where convergence in terms of both outputs and implemented functions does require exponentially long random programs. However a major difficulty is that in other cases Inequalities (1) and (2) provide very weak upper bounds.

Consider a very small example computer (of the type to be described in Section 9). Assume it has two input bits ($n = 2$), one output bit $m = 1$ and N memory bits. There are four operation codes (AND NAND OR NOR). Each has three address fields: two memory addresses for inputs and one for output. Each field can independently address every bit of memory. I.e. they can each take N values. Thus the number of instructions I is $4 \times N \times N \times N$.

A program of length $\leq N$ instructions can be written to set the memory to any desired pattern. I.e. $a_{\text{data}} = N$. From (1) convergence length $\leq 2.30 N 4^N N^{3N}$. We can also consider convergence of functions, of which there are $2^{m \times 2^n} = 16$. The most difficult of these is parity, which needs three instructions. Thus, in terms of convergence of functions, $a_{\text{functions}} = 3N$ instructions should be sufficient. Therefore, using (2), program length $\leq 2.30\ 3N\ 4^{3N} N^{3\times 3N}$. If $N = 8$, $a_{\text{data}} = 8$, $a_{\text{functions}} = 24$ and $I = 2048$ and so programs need not exceed $1.9\ 10^{6785}$ ($7.2\ 10^{7399}$ for functions) for the distribution 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 length. (Near convergence is seen by programs of about 13 instructions (25 instructions for functions) cf. Figure 8.)

6 Slowly Converging Example (Cyclic Computer)

In Section 5 we proved exponentially large convergence bounds for any computer. To counter the argument that these upper bounds are too weak we construct an example computer which does indeed require programs to be exponentially long before convergence is seen. (Section 7 to 9 give examples where convergence occurs very much faster.)

The cyclic computer has N bits of memory arranged as a single register. It has three instructions: do nothing, increment memory and decrement memory. Increment

and decrement act upon the whole of memory. E.g. if the memory holds the value 15 (001111_2), then after increment the value will be 16 (010000_2). Note the memory register wraps around. E.g. on an 8 bit computer, incrementing 255 produces 0. So a program of 2^N increments (or 2^N decrements) leaves the computer in the same state it started in. (The computer takes its name from this cyclic behaviour.)

If we subtract the number of decrement instructions from the number of increments (to give p) the output of a program is $(x+p) \bmod 2^N$, where x is the input. Therefore, given a program's output on one input, we can say directly what its output will be on any other input. Note, there are only 2^N functionally distinct values for p and so the cyclic computer can only implement 2^N functions.

The probability distribution of functions clearly follows the distribution of outputs. So when l is long enough to ensure each output is equally likely, then so too is each function.

By using Rosenthal's Markov analysis (Rosenthal 1995, circles of lily pads) we can show that after sufficiently many random instructions every memory pattern is equally likely. Secondly that $O((2^N)^2)$ instructions are needed before the chances become approximately the same. Rosenthal's approach is based upon an eigen analysis of random walks on finite Abelian groups.

Rosenthal (1995) shows that the actual probability distribution μ_l after l random instructions is exponentially close for large l to the limiting distribution π (in which each of the 2^N states is equally likely). Actually (if there more than two bits of memory, i.e. $N > 2$) we have both lower and upper bounds on the maximum difference between the actual distribution of outputs of length l random programs and the uniform 2^{-N} distribution:

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

That is the programs have to be longer than $O(2^{2N})$ for the distribution of memory states to be very close to the limiting distribution. E.g. to make $||\mu_l - \pi|| < 0.1$, the lower bound says l must exceed $0.8 \frac{3}{4\pi^2} 2^{2N}$, while the upper bound says it need not exceed $\log(101) \frac{3}{4\pi^2} 2^{2N}$.

For a computer with 1 byte of memory ($N = 8$ bits), programs with between 4,000 and 23,000 random instructions need to be considered before each state is equally likely.

In general, the distribution of program fitnesses will also take between $0.06\ 2^{2N}$ and $0.35\ 2^{2N}$ to converge. Of course specific fitness functions may converge more rapidly.

7 Fast Convergence Example (Bit Flip Computer)

This second example shows a different architecture where not only does the computer still converge, but it converges very much faster than the upper bound proved for every computer in Section 5. Section 7.1 described the bit-flip computer and the convergence of random programs' outputs on it. Section 7.2 proves results for the convergence of their functionality.

7.1 Convergence of Bit Flip Outputs

This example uses Rosenthal's results on random bit flipping (Rosenthal 1995). See also (Diaconis 1988, pages 28–30). Assume a computer with N bits of memory and $N+1$ instructions. The zeroth instruction does nothing (no-op) while each of the others flips a bit. I.e. executing instruction i, reads bit i, inverts it and then writes the new value back to bit i. There is an input register of n bits and an output register of m bits. (The instruction set treats every bit of memory uniformly, so the input register is read-write rather than read only.)

Once again the limiting distribution is that each of the states of the computer is equally likely. However the size of programs needed to get reasonably close to the limit is radically different. Only $\frac{1}{4}(N+1)(\log(N)+c_1)$ program instructions are required to get close to uniform (Diaconis 1988, page 28) (Rosenthal 1995). In fact, for large N, it can also be shown that, in general, convergence will take more than $\frac{1}{4}(N+1)(\log(N)-c_2)$ instructions.

Using the upper bound and setting $c_1 \geq 4$ will ensure we get sufficiently close to convergence. Since then $||\mu_l - \pi|| \leq 10\%$. I.e. random programs of length $\frac{1}{4}(N+1)(\log(N)+4)$ will be long enough to ensure each bit of the computer is equally likely to be set as to be clear, regardless of the programs' inputs. (Section 9 explains why $c_1 = 4$ is sufficient.) Again in the limiting distribution each state is equally likely.

Only $m/(N+1)$ bit flips actually effect the output, so $\frac{1}{4}(N+1)(\log(m)+4)$ random instructions will suffice for the each of the 2^m outputs to be equally likely (cf. Section 9).

Returning to our computer with 1 byte of memory ($N = 8$), programs with no more than 14 random instructions are needed to ensure each state is equally likely. While if we run programs with a Boolean output (where the output register occupies a single bit $m = 1$) random programs need only contain nine instruction to make either output equally likely.

7.2 Convergence of Bit Flip Functions

Since, in this bit flip computer, those parts of the input register which are not also part of the output register have no impact on it, we can ignore them. Assume s bits are shared by the input and output registers. We can construct a truth table for each program. It will have 2^s rows. The zeroth row gives the output of the program (in the range $0 \ldots 2^m - 1$) when all s bits of the input register are zero. Each bit of the row is equal to the number of times the corresponding memory bit has been swapped by the program, modulo two. Each of $2^s - 1$ other rows is determined by the zeroth row. I.e. the complete table and hence the complete function implemented by a program, is determined by its output with input zero. Section 7.1 considered the convergence of program outputs, so we know that: 1) for large programs, each of the 2^m functions is equally likely and 2) the distribution of functions converges with the distribution of outputs. I.e. by $\frac{1}{4}(N+1)(\log(s)+4)$ random instructions.

Frequently the value or fitness of a program is determined by its functionality. Therefore the distribution of fitness values will also converge to a limit. Also it will converge to its limit at least as fast as the distribution of functionality converges. Many fitness functions are based on the errors programs make. Usually they do not care on which input errors occurred. Instead, they often either count the number of errors or sum the sizes of the errors. In such cases, many different functions have identical fitness.

Therefore the distribution of fitness values typically may converge faster (i.e. require shorter programs) than the distribution of program functionality.

Returning to our 1 byte ($N = 8$, $s = 1$) example. Nine random instructions are enough to ensure all of the possible Boolean functions are equally likely. Furthermore every Boolean fitness function will also be close to its limiting distribution.

8 The Average/Random/Amorphous Computer

In Computer Science we are used to the notion that computers are highly designed, precision engineered artifacts. For example, instruction sets are deliberately chosen to have highly ordered effects on the state of the machine, particularly the contents of its memory. Nevertheless we can theoretically analyse more amorphous computing devices. Indeed nanotechnology may be a route to their practical construction and use. Here we consider an abstract machine which can be considered as an average or typical machine, since its instruction set, rather than being designed, is chosen at random.

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 we suggest that a randomly connected computer is an average computer, and vice-versa.

While such a random connection machine might seem perverse, and we would expect it to be hard for a human to program, on the face of it, it could well be Turing complete (taking into account its finite memory).

The next subsection considers the convergence of the outputs of an average computer, while subsection 8.2 considers convergence of the functions it implements.

8.1 Convergence of Outputs of Amorphous Programs

The following paragraphs construct the state transition matrix $P(x, y)$ for our random computer. From $P(x, y)$ we show the distribution of outputs of all programs also converges as programs get longer. Then we calculate the rate at which it converges. We show, depending upon details, programs as short as five instructions may ensure convergence.

Suppose given any possible data in memory each of the I instructions independently randomises it. Since the number of states 2^N is much bigger than the number of instructions, from any given current state of the machine, in one instruction, it is impossible to reach most others. I.e. most elements of the transition matrix $P(x, y)$ will be zero. However each instruction must put the machine into a new state (which, of course, could be the same as the current state). So at least one element in each column of $P(x, y)$ will be bigger than zero. With I instructions, the new state could be upto I different states. That is upto I elements in each column of $P(x, y)$ can be non zero. If two instructions change the current state to the same next state, then, assuming each instruction is used with equal probability, this state is twice as likely as those which can arise by only one instruction. However only $I - 1$ states would be accessible. If three instructions produce the same outcome, then the corresponding element of $P(x, y)$ will be three times as big, and so on. If each of the I instructions is

equally likely to be used $P(x, y)$ must be an integer multiple of $1/I$. I.e. for any state x, $P(x, y) = 0$ or $1/I$ or $2/I$ or $\ldots$ or I/I.

Since every column of P has 2^N elements, for a single instruction, the chance of any given $P(x, y)$ being zero is $(1 - 2^{-N})$. Considering all I independent instructions gives the chance of any given $P(x, y)$ being zero is $(1 - 2^{-N})^I$. Consider two instructions chosen at random. $P^2(x, y) = 0$, or $1/I^2$ or $\ldots$ or $2I/I^2$. The chance of any given element of $P^2(x, y)$ being zero is $(1 - 2^{-N})^{2I}$.

For a instructions, each element of $P^a(x, y)$ will be a multiple i (possibly zero) of I^{-a}. The values of i will be randomly distributed and follow a binomial distribution with $p = 1/2^N$, $q = 1 - p$ and number of trials $= I^a$. So the mean of the distribution of i is $I^a/2^N$ and its standard deviation is $\sqrt{I^a \times 1/2^N \times (1 - 1/2^N)}$. For large I^a the distribution will approximate a Normal distribution. If $I^a \gg 2^N$, even for large 2^N, practically all i will lie within a few (say 5) standard deviations of the mean. I.e. the smallest value of i in any column will be more than $I^a/2^N - 5\sqrt{I^a \times 1/2^N}$. So β will be at least $2^N I^{-a}(I^a/2^N - 5\sqrt{I^a \times 1/2^N})$ (cf. Table 2 and Section 4). I.e. $\beta \geq (1 - 5\sqrt{I^{-a} \times 2^N})$.

Let $\alpha = 5\sqrt{I^{-a} \times 2^N}$. So $\beta \geq (1 - \alpha)$. Next chose a particular number of instructions a so that α is not too small. E.g. set $\alpha = 0.5$ so $\beta \geq 0.5$.

$$\begin{aligned}
\alpha &= 5\sqrt{I^{-a} \times 2^N} \\
\sqrt{I^{-a} \times 2^N} &= \alpha/5 \\
0.5(-a \log I + N \log 2) &= \log(\alpha/5) \\
a &= \frac{-2 \log(\alpha/5) + N \log 2}{\log I}
\end{aligned}$$

Now that we have a practical value of β, we can use the minorization condition on P^a to give a bound on the difference between the actual probability distribution of memory states after running a randomly chosen program of l instructions on a randomly constructed computer and its limiting distribution:

$$\begin{aligned}
||\mu_l - \pi|| &\leq \left(1 - (1 - 5\sqrt{I^{-a} \times 2^N})\right)^{\lfloor l/a \rfloor} \\
&= \left(5\sqrt{I^{-a} \times 2^N}\right)^{\lfloor l/a \rfloor} \\
&= \alpha^{\lfloor l/a \rfloor}
\end{aligned}$$

Choosing a target value of $||\mu_l - \pi||$ of 10% gives:

$$\begin{aligned}
\alpha^{\lfloor l/a \rfloor} &\geq ||\mu_l - \pi|| = 0.1 \\
\lfloor l/a \rfloor \log \alpha &\geq -2.30 \\
l &\leq \frac{-2.30\ a}{\log \alpha} \\
&= \frac{-2.30\ (-2 \log(\alpha/5) + N \log 2)}{\log \alpha \log I} \\
&= \frac{-2.30\ (2 \log 10 + N \log 2)}{-\log 2 \log I} \\
l &\leq \frac{15.3 + 2.30\ N}{\log I} \qquad (3)
\end{aligned}$$

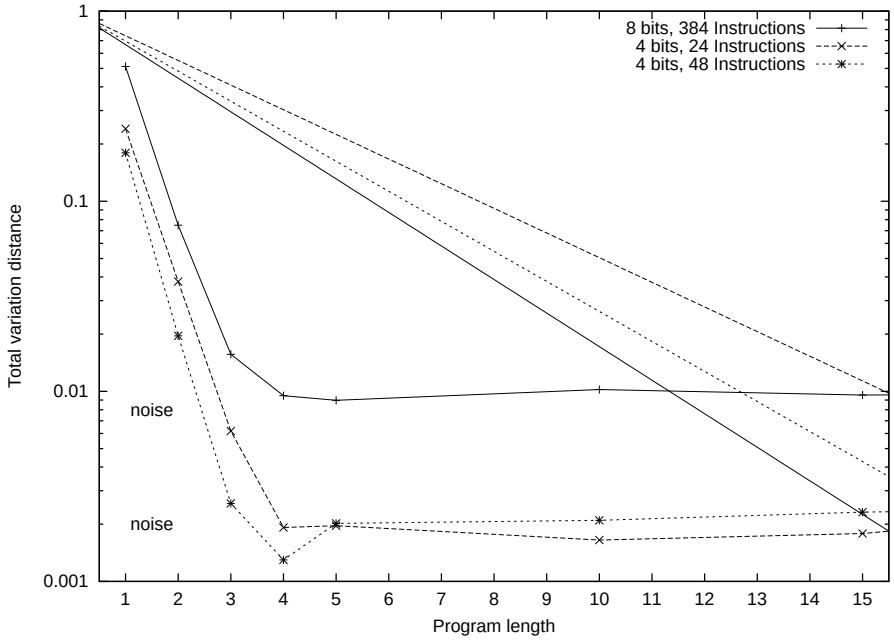

Fig. 4 Convergence of three random machines, measured with a million random functions at each length. Note allowing for noise, measurements are within theoretical upper bound, Inequality (3) (straight lines). Also note large instruction sets are needed to ensure randomly connected computers can access all their memory states.

Note this predicts quite rapid convergence for our randomly wired computer. E.g. if it has 8 instructions $l \approx 7 + N$. For a one byte ($N = 8$) computer with 8 random instructions ($I = 8$), programs longer than 16 instructions will be close to the computer's (approximately uniform) limiting distribution.

Inequality (3) bounds the length of random programs need to be to ensure, starting from any state, the whole computer gets close to its limiting distribution. Again we define parts of the memory as input and output registers. Each program's output is given by m output bits. Due to the random interconnection of states, on average we can treat each of the 2^m states associated with the output register as projection of 2^{N-m} states in the whole computer, so Inequality (3) becomes $l \leq (15.3 + 2.30\ m)/\log I$.

E.g. for Boolean problems ($m = 1$). Only about 9 random instructions are needed for an 8 random instruction computer to have effectively reached the programs' outputs limiting distribution.

Figure 4 confirms our analysis experimentally. The length of programs run on amorphous computers in Figure 4 are actually even shorter than our calculated bound. Doubtless this is in part due to our choice of five standard deviations. If need be, somewhat tighter bounds could be established by more detailed consideration of β.

8.2 Convergence of Functions on Amorphous Computers

Let us consider the truth table of the function that each program implements (cf. Sections 2.2 and 7.2). 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 at identical points in the program, they will remain synchronised no matter how many more instructions are executed. 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.

By totally synchronised, we mean by the end of the program, the computer's memory content is the same for all 2^n possible inputs. Therefore the program will always output the same value. Next we use the "coupon collector" argument (Stirzaker 1999, pages 274–275) to calculate how long this will take.

For total synchronisation, $2^n - 1$ 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 $\gamma \approx 0.5772$ 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 the variance for each row. I.e. the variance is $2^{2N} \sum_{i=1}^{2^n-1} 1/i^2$. As the upper limit increases, this sum converges rapidly to $2^{2N} \frac{\pi^2}{6}$. So, for large n, the standard deviation tends to $\frac{\pi}{\sqrt{6}} 2^N = 1.29\ 2^N$

The "coupon collector" distribution has an approximately exponential tail. Therefore amorphous programs longer than $(6.5 + 0.69n)2^N$ are almost certain to return a constant value regardless of their inputs.

9 Linear GP: AND NAND OR NOR Computer

Since this model is close to actual (linear) GPs (Nordin 1997) (see Figure 5) we shall examine it in more detail than the other computers.

The computer comprises N bits of memory and the CPU. The memory contains the input register (n bits) and the output register (m bits). 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

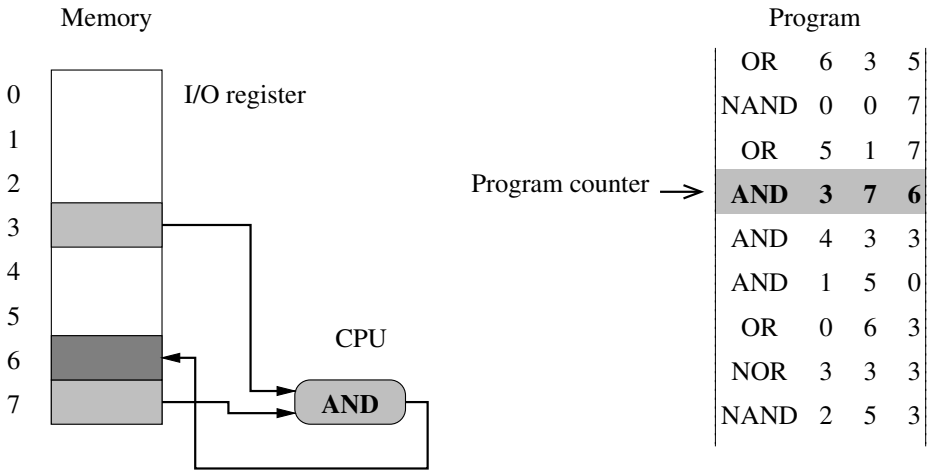

Fig. 5 The four function computer. On the left the CPU is executing an AND instruction. On the right we see an example program.

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.

As with the initial random population of a linear GP system, each of the available instructions can occur at any point in the program and which memory locations it reads from and which it writes to are chosen randomly. Of course, as evolution proceeds, a GP system will rapidly home in on particularly high scoring combinations of instructions. Nevertheless Monte Carlo sampling tells us about the underlying space which GP (or other methods) is searching. Such information might be used to improve GP, e.g. with better fitness functions (see Section 12) or explain a GP's operation. As we shall see, this architecture has several features which eases the analysis of random streams of 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.

9.1 Convergence of 4 Instruction Program Outputs

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. Thus we can update the analysis in Section 7 based on (Diaconis 1988, pages 28–30) and (Rosenthal 1995). The difference between the actual distribution μ_l

of outputs for programs of length l and the long program limit π is bounded by

$$
\begin{aligned}
||\mu_l - \pi||^2 &\leq \frac{1}{4}\sum_{j=1}^{N} \frac{N!}{j!(N-j)!}\left|1-\frac{j}{N}\right|^{2l} \qquad (4)\\
&= \frac{2}{4}\sum_{j=1}^{\lceil \frac{N+1}{2} \rceil} \frac{N!}{j!(N-j)!}\left(1-\frac{j}{N}\right)^{2l}\\
&< \frac{1}{2}\sum_{j=1}^{\infty} \frac{N^j}{j!} e^{-\frac{2j}{N}l}\\
||\mu_l - \pi||^2 &\leq \frac{1}{2}\left(e^{Ne^{-\frac{2l}{N}}} - 1\right)
\end{aligned}
$$

Suppose that we want to find the shortest length for which programs have a distribution μ_l which is within 10% of the limiting distribution π for infinitely big programs. (I.e. $||\mu_l - \pi|| < 0.1$.) Rearranging the above formula gives the upper bound $l \leq \frac{1}{2}N(\log(N) + 4)$.

Note, programs need only be twice as long on this computer (which is capable of real computation) as the simple bit flipping programs of Section 7. This result is for convergence of the whole computer. Next we show that programs can be shorter, if we consider only convergence of programs' outputs (i.e. of the output register).

In this computer the chance of updating the output register is directly proportional to its size. So the number of instructions needed to randomised the output register is given by its size (m bits). But we need to take note that most of the activity goes on the other $N - m$ bits of the memory. Therefore Inequality (4) becomes

$$
\frac{1}{4}\sum_{j=1}^{m} \frac{m!}{j!(m-j)!}\left|1-\frac{j}{N}\right|^{2l}
$$

which leads to $l \leq \frac{1}{2}N(\log(m)+4)$. Figure 6 shows we have excellent agreement between theory and experiment and that the theoretical upper bound is only slightly larger than the observed convergence rate.

For example, on a four function computer with an eight bit memory, we observe convergence to within 10% for programs longer than 12 instructions. Whilst the theoretical bound is 16 instructions.

9.2 Convergence of 4 Instruction Program Functions

Having correctly predicted convergence of program outputs, we next show that the distribution of programs' functionality also converges. Almost all large programs implement one of the constant functions and each constant is equally likely. We next consider more useful programs. In Sections 9.3 and 9.4, we model the behaviour of shorter programs. Section 9.5 shows how they tend to the large program limit.

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

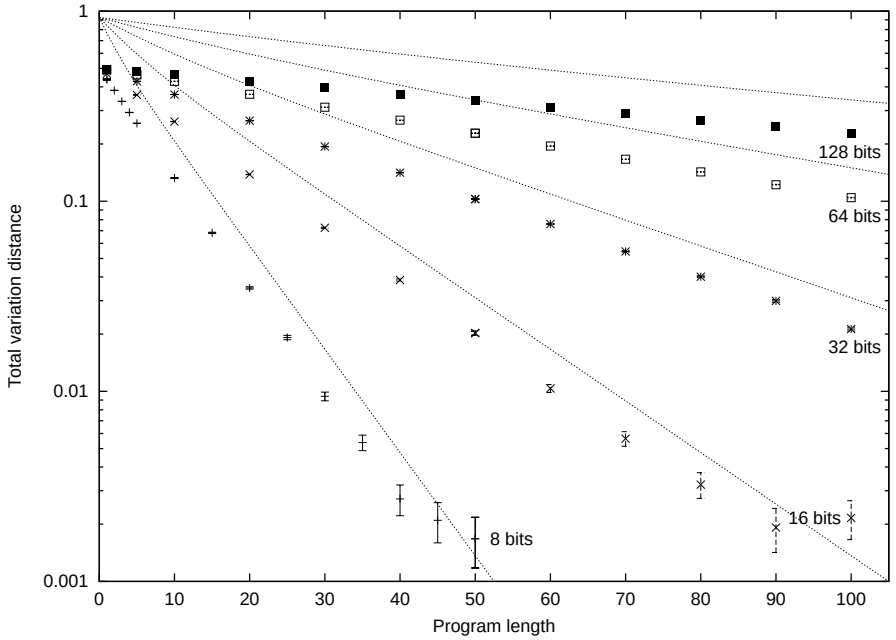

Fig. 6 Convergence of outputs of 4 input bit Boolean (AND, NAND, OR, NOR) linear programs with different memory sizes. Note the agreement with upper bound $\sqrt{1/2(\exp(me^{-2l/N})-1)}$ (dotted lines).

Table 4 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

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 an 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 3 and 4.

We know (cf. Section 9.1) 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 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. We would like to use a "coupling" argument, such as given by (Propp and Wilson 1996) but since our random program is fixed we cannot assume the state transitions are independent. Instead we note that the Markov process for updating the memory may or may not have

a synchroniser. (A synchroniser is a sequence of instructions which takes the machine from every possible state to one state.) If there is a synchroniser, a long enough random program is bound to contain that sequence. It does not matter where the synchroniser is. Once synchronised the machine will remain synchronised, and so its output will be the same no matter how it started. I.e. each of the rows in the inputs × memory table will be identical. This means long random programs will be irreversible. Conversely if there is no synchroniser, programs will map each starting state to a different final state, and the program is reversible. Section 10 briefly considers reversible computation. In particular, Section 10 shows that if two identical reversible computers running the same program are started with different inputs at the same time, they do not synchronise. I.e. their memories will never be identical. In contrast typical computers (and also random computers, cf. Section 8) 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 a typical long program, it does loose information and the rows in its table are not unique. I.e. the AND NAND OR NOR computer is not reversible.

To prove this, we construct a synchroniser. Setting r_0 to the opposite of r_1 followed by AND(r_0,r_1) will always generate a zero (regardless of their initial values). So r_0 =NAND(r_1,r_1) followed by $N-2$ AND(r_0,r_1) instructions can be used to clear memory cells $\mathrm{r}_2 \ldots \mathrm{r}_{(N-1)}$. We then use r_0 =AND(r_2,r_2) r_1 =AND(r_2,r_2) to clear r_0 and r_1: ensuring the whole of memory is clear regardless of its initial state. I.e. we have a synchroniser of $N+1$ instructions.

A very long sequence of random instructions is bound to contain our synchroniser, so at some point it will clear the entire machine's memory. From this point on the initial input makes no difference and whatever the program does to the memory will be the same no matter which input value is considered. (I.e. the rows of inputs × memory table are all the same.) So we need only consider one case. However Section 9.1 has already covered this and we know that in the limit of large random programs each memory pattern is equally likely. I.e. each of the 2^m possible constants are equally likely. Also (after synchronisation) only a further $\frac{1}{2}N(\log(N)+4)$ random sequence of instructions are needed to approach the limit. (We could use the length of our synchroniser to give an upper bound on the length of random programs needed to be reasonably sure that they contain it ($\approx 2.3(N+1)I^{N+1} = 2.3(N+1)4^{N+1}N^{3(N+1)}$). However this turns out to be a very weak bound on convergence of the distribution of functions.) Figure 7 plots the convergence of the fitness distribution for a parity problem, in which the input registers is not write protected. Note the fraction of all high fitness programs falls rapidly towards zero. We consider how long it will take to get close to the limiting distribution by constructing a model of the convergence process.

9.3 Model of Convergence of 4 Instruction Program Functions

By considering the 4 instruction computer is more detail, Sections 9.4 and 9.5 are able to model the number and functionality of interesting shorter programs.

Instead of total variation distance we take into account that we know the limiting distribution for the programs' functionality is that they are overwhelmingly likely to return the same value for each input. In the long program limit, since the contents of the memory is the same no matter what the input is; the inputs × memory table is a uniform

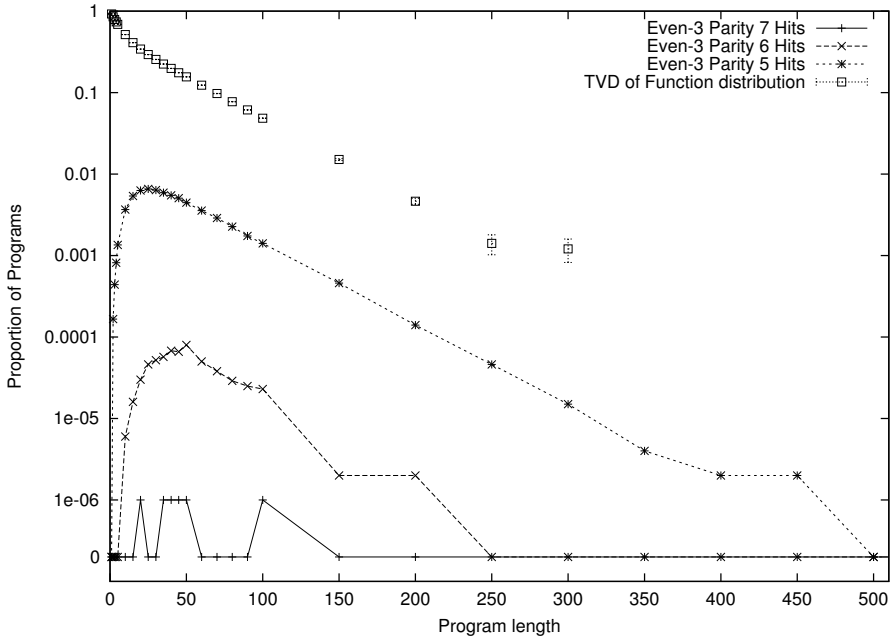

Fig. 7 Convergence of Even-3 parity fitness. (AND, NAND, OR, NOR, I/O register may be overwritten.) Partial success (5 hits) is much more frequent than higher fitness (6 and 7 hits). No parity (8 hits) solutions were found. Even near the limit some functions are more common than others. Longer programs are needed to achieve convergence of functions than of outputs (cf. Figure 6, 8 bits).

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 × memory table contains the Identity function. As random operations are performed, the proportion of non-constant functions falls (see Figure 8).

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.

9.4 Short 4 Instruction Program Functions

Initially the output register columns of the inputs × memory table contains the Identity function and the other columns are all zeros (cf. Table 3, page 6). 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 $2.30N/m$.

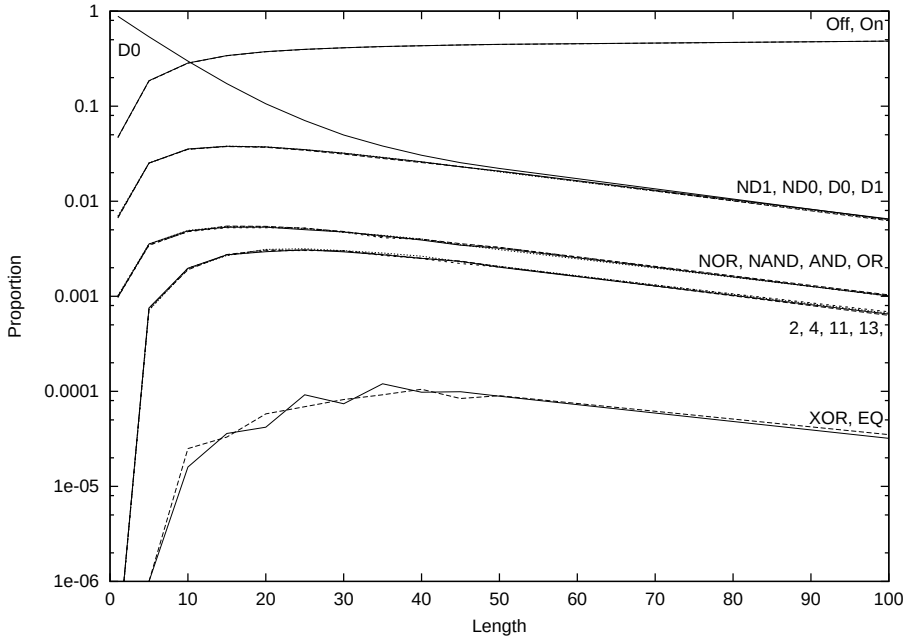

Fig. 8 Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits). Almost half long programs implement Always-on and nearly half implement Always-off. Short programs are more varied and many implement the Identity function "D0".

In Figure 9 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 initial rapid convergence, due to loss of the identify function (cf. D0 in Figure 8), slows. Cf. "knee" in observations as program length passes N. Nevertheless, Figures 9 and 11 show $(1 - m/N)^l$ is a very good model for 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.

9.5 Longer 4 Instruction Program 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 8, 9 and 11 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 10.

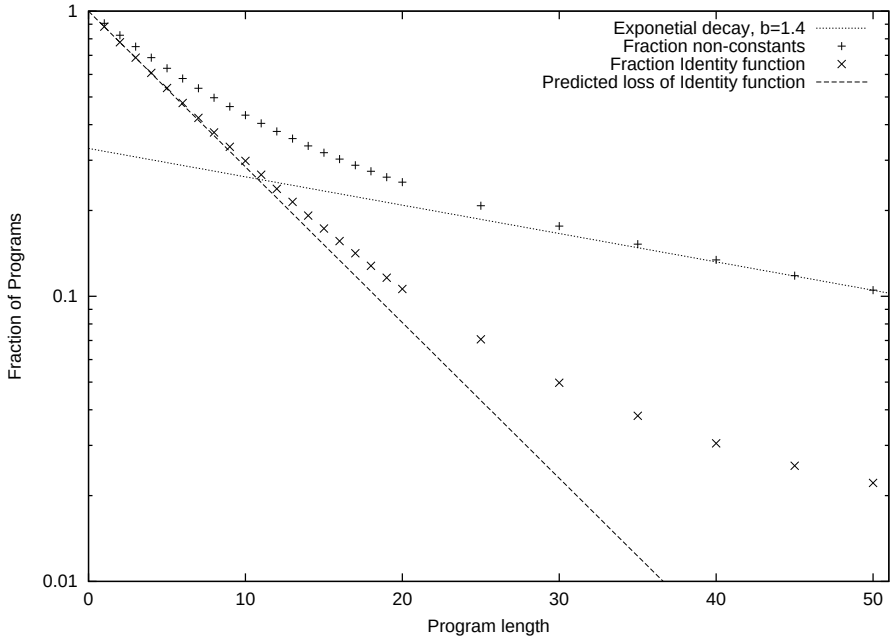

Fig. 9 Convergence of binary Boolean random linear functions (AND, NAND, OR, NOR, 8 bits). The steep straight line is the predicted effect of exponential loss of the Identity function "D0" caused by overwriting of the output register. Initially this explains almost all the convergence (Section 9.4). While the dotted line is experimental fit (Section 9.5).

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 (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 8. 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.

In (Langdon 2002b, pages 194–197) we presented an informal model which suggests the proportion of interesting functions implemented on our four Boolean function linear computer falls exponentionally with program length with an exponent $O(N^{-3/2})$. Figures 9 and 11 shows empirical evidence that the fraction of interesting functions does indeed have this two stages behaviour.

10 Reversible/Quantum Computers

A reversible computer is one where every program's input can be inferred from its output (Langdon 2003). This is usually imagined as feeding the program's outputs back into the program and running it backwards from its end to its start. (Hence the name reversible). Once it reaches its start, it will have calculated its inputs. Recently

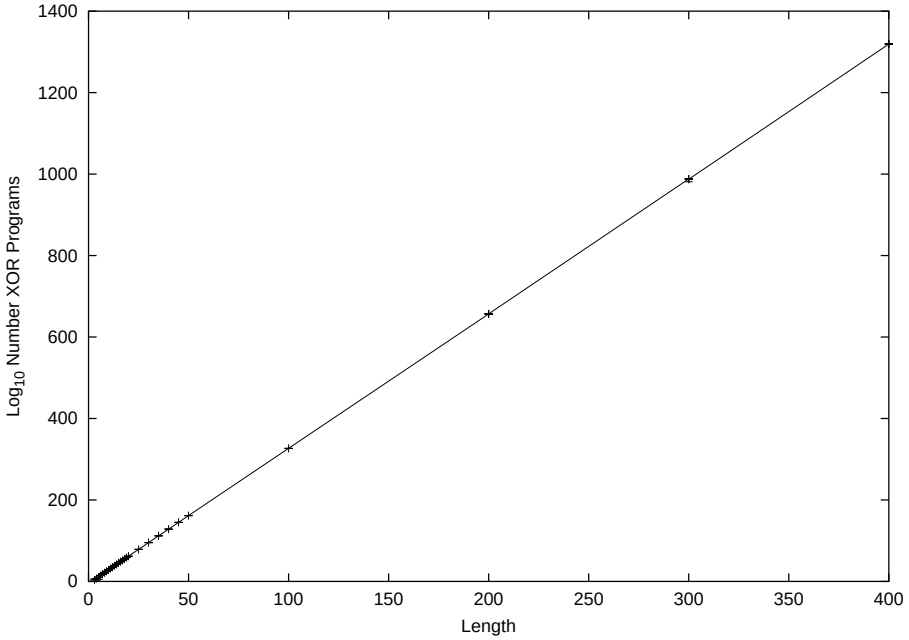

Fig. 10 Number of programs which implement 2-bit odd parity (AND, NAND, OR, NOR, 8 bits). Cf. Section 5.3.

there has been interest in reversible computers for use in high integrity systems (Bishop 1997).

The exponential speed up in theory possible from quantum computers comes from simultaneous processing of many quantum-bits which are held in a carefully constructed superposition of multiple quantum states (qubits). The states of the system are described by complex numbers and the transformation are treated as complex unitary matrices. Until measurements are made to collapse the entangled qubits, each matrix has an inverse and quantum computations can be reversed. Note simulations of quantum computing proceed by taking a complex vector as the starting state and progressively multiplying it by unitary matrices. (Each matrix represents a transformation brought about by a quantum gate.) Apart from the use of special complex matrices, this can be thought of as similar to our process of modelling Markov processes by multiplying by matrices.

Reversible computers can be considered as a special case of quantum computers. The matrices contain only real numbers (i.e. non-complex). A unitary matrix whose elements are all real numbers is known as an orthogonal matrix.

For us to be able to reverse a computation, we cannot destroy information at any step, so our programs must be composed only of reversible instructions. (This is directly analogous to quantum computers. Which, except when measurements are taken, must be composed only of reversible quantum transformations.) The reversible instructions can move data about memory but compressing information in one region (e.g. the output register) will imply corresponding expansion elsewhere in memory. To run our program backwards, we will need to know the state of the whole memory when it stopped, not just the output register.

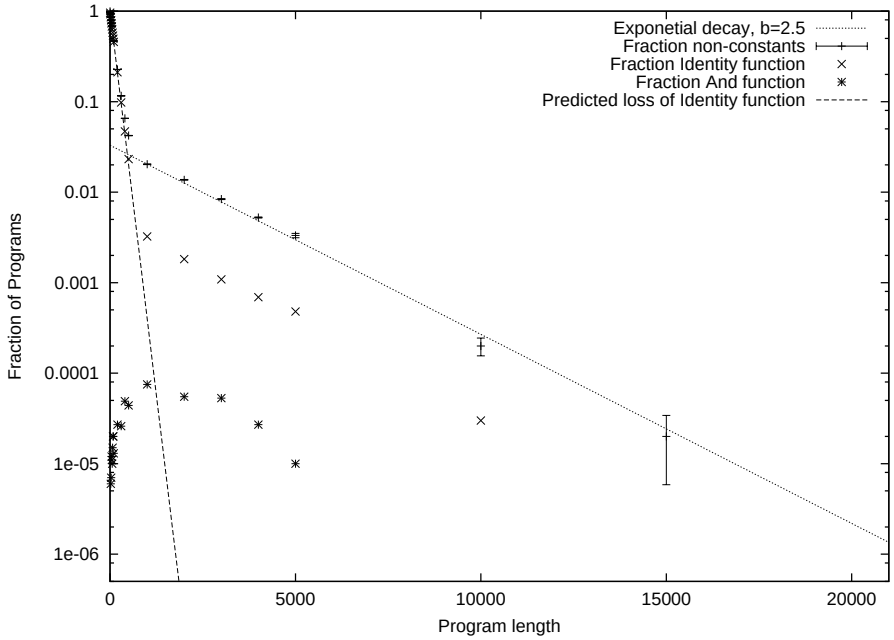

Fig. 11 Fraction of non-constants in 4 input bit Boolean random linear functions (AND, NAND, OR, NOR, 128 bits). Steep line is prediction of exponential overwriting of Identity function. While dotted line is slower exponential decrease in non-constant functions. Experimental fit is near that predicted in (Langdon 2002b) Note the proportion of all non-constant functions appears to decrease with the same exponent.

We are now familiar with the idea that a program is an inputs × memory table, which maps inputs to final memory. In the case of a reversible computer, we expand this to cover the whole of memory, so the table becomes $\text{memory}_{\text{start}} \times \text{memory}_{end}$ transformation matrix.

Since we can always uniquely determine which input gave rise to the final contents of memory, in a reversible computer no two inputs can give the same final memory. This means none of the rows of the table can be identical. In other words reversible programs permute the states. (A permutation matrix is a special kind of orthogonal matrix). Similarly every reversible instruction can be represented by a permutation matrix. The permutation matrix of the complete program is given by multiplying the matrices of its component instructions together, in the same sequence that they occur when the program is run.

As an example, consider the cyclic machine described in Section 6. 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.

Suppose we have 2^n identical copies of a reversible computer. We run them all in step. Each runs the same program but with a different input. They start in different states. No matter how long the program is, the computers will never synchronise. In contrast typical computers (and also random computers, cf. Section 8) do synchronise,

they are not reversible and they destroy information during the course of a program run.

11 Convergence in bit string genetic algorithms

The bit flipping model (in Section 7) is very close to standard mutation in bit string genetic algorithms (GAs). The principle difference is in standard GAs the number of bits flipped follows a Poisson distribution (unit mean is often recommended (Bäck 1996)). Thus 0.38 (rather than $1/(l+1)$) of chromosomes are not mutated and 0.26 (rather than zero) chromosomes have two or more bits flipped. (In this section, the length of the bit string chromosome is denoted by l.) Ignoring these differences, it takes only $\frac{1}{4}(l+1)(\log(l)+4)$ mutations to scramble a chromosome from any starting condition.

It is no surprise to find *asymptotic bounds* of $\mathrm{O}(l\log(l))$ reported before (Garnier et al. 1999), but note that $\frac{1}{4}(l+1)(\log(l)+4)$ is quantitative and does not require $l \to \infty$. Also it is a reasonably tight bound in the sense that replacing "+4" by a modest negative constant leads to a lower bound. However we include this section mainly because the answer comes straight from standard results without hard work.

Since each chromosome in a GA population is mutated independently, and the variance is small, the time taken to scramble an entire GA population is scarcely more than to scramble each of its chromosomes. Crossover makes the analysis more complex but since it moves bit values rather than changing them, we do not expect it to radically change the time needed (Gao 1998). E.g. for a GA population of 32 bit strings, mutation alone (note we turn off selection) will scramble it within about 61 generations. (For standard mutation the value may be slightly different.) Notice this is independent of population size, in contrast the number of generations taken by selection to unscramble the population depends on the size of the population but not l (Blickle 1996). According to (Bäck 1996, Table 5.4) binary tournament selection (without mutation or crossover) takes only 9 generations to remove all diversity from a population of 100.

12 Discussion

The results in Sections 5–10 refer to several specific types of computation. Nevertheless we they are useful, particularly for common varieties of genetic programming (GP), other unconventional programming techniques or cases where knowing how functions are distributed is important. For example, we now know that in many cases the distributions converge very rapidly so that the length of programs used by real GPs can be greater than the convergence length. That is, in some cases (even without excessive bloat (Langdon et al. 1999)) evolved programs lie in the "infinitely long" part of the search space. Indeed in some cases some of the longer programs in the initial random population can effectively be treated as if they were infinitely long. This may be a partial explanation for the anti-bloat shrinkage often seen in the first generations of GP runs (Langdon 1998, Figure 4.16) as GP selectively removes long low fitness random programs. Knowing this and facts about the behaviour of long programs, our results can be used as a to guide in the redesign of the way the initial GP population is created.

The model does not cover programs that contain instructions that are executed more than once. I.e. no loops or jumps. This is, of course, a big restriction. However,

many problems have been solved by GP systems without such loops or recursive function calls (Banzhaf et al. 1998). We have recently started to be able to map the search space of program with loops (Langdon 2006).

While the proofs suggests that the program will halt after l instructions, they can be made slightly more general by extracting the answer from the output register after l time steps. This is called an "any time algorithm". Any time algorithms have been used in GP, e.g. (Teller 1994). Another potential extension is to "programming without a program counter" (Banzhaf 2005). PC less programming treats each program as a set of instructions which the computer is free to execute in any order. Typically instructions are chosen at random. (Hence there is no need for a program counter (PC).) It should be possible to model such stochastic programs with our approach.

The dominant factors in determining the length required for near convergence are the type of computer considered and the size of its (data) memory. Comparing the four types in Sections 5–9 suggests that the degree of interconnections in the state space is the important factor in determining the form of the scaling law. That is the type of scaling is given by the nature of the sparsity of the Markov matrix. The ability to move directly from one memory pattern to another leads to linear scaling, while only being able to move to 2 adjacent data patterns leads to exponential scaling. We suggest that the "bit flipping" and "4 Instruction" models are more typical and so we suggest $O(N \log N)$ would be found on real computers.

The random computer (cf. Section 8) gives an interesting model. Indeed it represents the average behaviour over all possible linear program computers. While we are unaware of research in this area, it might be feasible to generate Turing complete "random" computing elements using nanotechnology.

An alternative view is to treat random instructions as introducing noise. Some instructions, e.g. clear, introduce a lot of noise, while others e.g. NAND, introduce less. So we start with a very strong, noise free, signal (the inputs) but each random instruction degrades it. Eventually, in the limiting distribution, there is no information about the inputs left. Thus the entropy has monotonically increased from zero to a maximum.

Practical linear GP systems write protect their inputs (Francone 2001; Langdon and Nordin 2001). Such systems can be viewed as like tree based GP (Langdon and Poli 2002). The proofs can be extended to cover this by viewing the read-only register as part of the CPU (i.e. not part of the data memory). Then we get a limiting distribution as before, but it depends on the contents of the read-only register, i.e. the programs' input. In general we would expect this to give the machine a very strong bias (i.e. an asymmetric limiting distribution) and in some cases this might be very useful. Indeed the limiting distribution will contain "Identity" functions and other "interesting" (i.e. non-constant) functions. 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 12). Possibly these may be related to GP's ability to find general solutions (Langdon 1998) and Occam's razor. Initialising the memory with multiple copies of the input register might also bring some benefit to GP runs.

Figures 9, 11, and 12, suggest it might be worthwhile investigating linear GP systems which impose a program length limit of about 10 N/m.

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

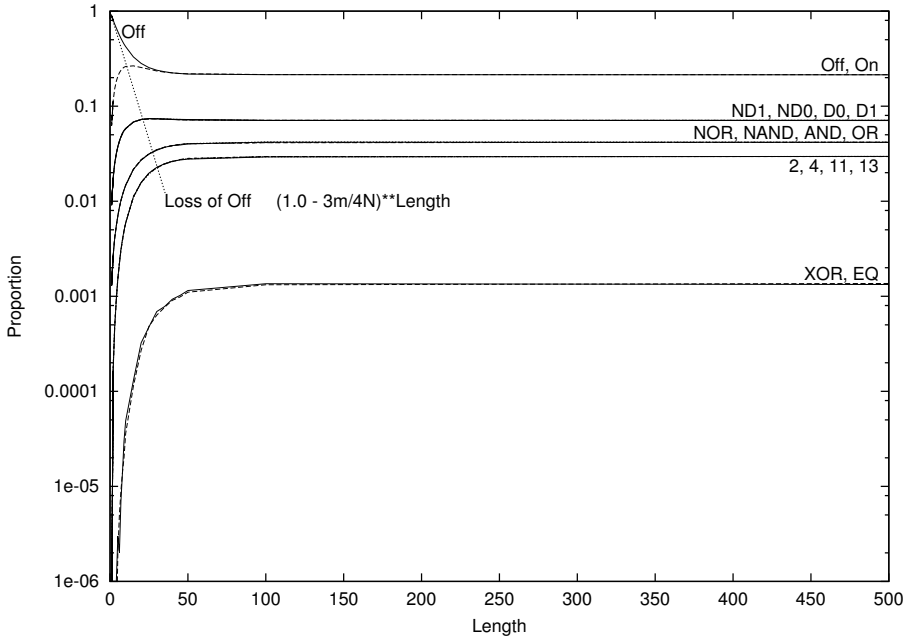

Fig. 12 As Figure 8 but note write protecting the input register means all functions occur in the long random program limit.

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, 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 9) 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 $2.30N/m$.)

13 Conclusions

Both the distribution of all programs' outputs and their functionality converge when we consider longer and longer programs. In Section 5 we showed an exponential upper bound on the length of programs for both distributions to approach their limits. To counter the argument that this upper bound is weak, we have shown an example

(Section 6) where we can calculate tight upper and lower bounds. Both are exponential. The exponential lower bound shows there are some cases which really do require exponentially long programs before either the distribution of outputs or functionality approaches its limit. That is, the general exponential bound is not weak.

We next showed, by establishing the speed of convergence, there are special cases where convergence occurs very much faster. For example, in Section 7, both the distributions of outputs and functionality of programs of length of only $O(N \log N)$ approach the distributions of the whole infinite program space. Another rapidly converging example is Amorphous computing. Section 8 showed barely a handful of random instructions can be enough for the distribution of outputs to get to within 10% of the limit.

Section 9 describes a linear GP system with the four common Boolean operations. We showed its space of programs also converges rapidly. Indeed programs need only be twice as long for this CPU (which can do real computation) than the rapidly converging example of Section 7. Section 9 describes in some detail how interesting functions approach their limits.

We have given results which map the vast majority of the infinite space of certain classes of programs. Our main results show that useful programs are exceedingly rare. Yet genetic programming has repeatedly demonstrated that progressive fitness based evolution is able to find them. This reiterates the power of evolution to solve problems which would be impossible for blind search.

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 the anonymous referees.

A Summary

The distribution of outputs produced by all computers converges to a limiting distribution as their (linear) programs get longer. We provide a general quantitative upper bound ($2.30aI^a$, where I is the number of instructions and a is the length of programs needed to store every possible value in the computer's memory, Section 5). Tighter bounds are given for four types of computer. There are radical differences in their convergence rates. The length of programs needed for convergence depends heavily on the type of computer, the size of its (data) memory N and its instruction set.

The cyclic computer (Section 6) converges most slowly, $\leq 0.35\ 2^{2N}$, for large N. In contrast the bit flip computer (Section 7) takes only $\frac{1}{4}(N+1)(\log(m)+4)$ random instructions (m bits in output register). In both computers, the distributions of outputs and of functions converge at the same rate to a uniform limiting distribution.

In Section 8 we introduced a random or amorphous, model of computers. This represents the average behaviour over all computers (cf. NFL (Wolpert and Macready 1997)). It takes less than $(15.3 + 2.30m)/\log I$ random instructions to get close to the uniform output limit. The limiting distribution contains only functions that are constants. Again convergence is exponential with 90% of programs of length $1.6\ n2^N$ yielding constants (n is the size of the input register in bits).

Section 9 shows the output of programs comprised of four common Boolean operators converges to a uniform distribution within $\frac{1}{2}N(\log(m)+4)$ random instructions. The importance of the pragmatic heuristic of write protecting the input register, is highlighted, since without it there are no "interesting" functions in the limit of large programs.

In Section 10 we showed quantum and reversible computers do not have synchronising sequences and consequently the behaviour of their long programs is radically different from that of conventional computers.

Section 11 shows the number of generations ($\frac{1}{4}(l+1)(\log(l)+4)$) needed for mutation alone to randomise a bit string genetic algorithm (chromosome of l bits).

Practical GP fitness functions will converge faster than the distribution of all functions, since they typically test only a small part of the whole function. Real GP systems allow rapid movement about the computer's state space and so appear to be close to the bit flipping (Section 7) and four Boolean instruction (Section 9) models. We speculate rapid $O(|\text{test set}|N \log m)$ convergence in fitness distributions may be observed.

The number of minimal solutions to XOR (even-2 parity) grows quadratically in memory size (cf. Section 5.3) but this corresponds to a rapid fall in proportion as memory is increased.

In the Boolean linear systems considered, complex functions are very rare even 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.

References

Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996. ISBN 0-19-509971-0.

Wolfgang Banzhaf. Challenging the program counter. In Susan Stepney and Stephen Emmott, editors, *The Grand Challenge in Non-Classical Computation: International Workshop*, York, UK, 18-19 April 2005. URL `http://www.cs.york.ac.uk/nature/workshop/papers/Banzhaf.pdf`.

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998. ISBN 1-55860-510-X.

Peter G. Bishop. Using reversible computing to achieve fail-safety. In *Proceedings of the Eighth International Symposium On Software Reliability Engineering*, pages 182–191, Albuquerque, NM, USA, 2-5 Nov 1997. IEEE. ISBN 0-8186-8120-9.

Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996. URL `http://www.handshake.de/user/blickle/publications/diss.pdf`.

Persi Diaconis. *Group Representations in Probability and Statistics*, volume 11 of *Lecture notes-Monograph Series*. Institute of Mathematical Sciences, Hayward, California, 1988. ISBN 0-940600-14-5.

William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley and Sons, New York, 2 edition, 1957. ISBN 0 471 25711 7.

William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley and Sons, New York, 1966.

James A. Foster. Review: Discipulus: A commercial genetic programming system. *Genetic Programming and Evolvable Machines*, 2(2):201–203, June 2001. doi: 10.1023/A:1011516717456.

Frank D. Francone. *Discipulus Owner's Manual*. 11757 W. Ken Caryl Avenue F, PBM 512, Littleton, Colorado, 80127-3719, USA, version 3.0 draft edition, 2001. URL `http://www.aimlearning.com/Discipulus%20Owners%20Manual.pdf`.

Yong Gao. An upper bound on the convergence rates of canonical genetic algorithms. *Complexity International*, 5, 1998. ISSN 1320-0682. URL `http://www.complexity.org.au/ci/vol05/gao/grateGao.html`.

Josselin Garnier, Leila Kallel, and Marc Schoenauer. Rigorous hitting times for binary mutations. *Evolutionary Computation*, 7(2):173–203, 1999.

Olle Haggstrom. *Finite Markov Chains and Algorithmic Applications*, volume 52 of *London Mathematical Society Student Texts*. Cambridge University Press, 2002. ISBN 0 521 890001 2.

William B. Langdon. Convergence rates for the distribution of program outputs. In William 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 2002a. ISBN 1-55860-878-8. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_gecco2002.pdf`.

William B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002b. Morgan Kaufmann. ISBN 0-12-208155-2. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_foga2002.pdf`. Published 2003.

William B. Langdon. The distribution of reversible functions is Normal. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 11, pages 173–188. Kluwer, 2003. ISBN 1-4020-7581-2. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_reversible.pdf`.

William B. Langdon. Mapping non-conventional extensions of genetic programming. In Cristian S. Calude, Michael J. Dinneen, Gheorghe Paun, Grzegorz Rozenberg, and Susan Stepney, editors, *Unconventional Computing 2006*, volume 4135 of *LNCS*, pages 166–180, York, 4-8 September 2006. Springer-Verlag. ISBN 3-540-38593-2. doi: 10.1007/11839132_14. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_uc2002.pdf`.

William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. ISBN 3-540-42451-2.

William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, Boston, 1998. ISBN 0-7923-8135-1.

William B. Langdon and Peter Nordin. Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 313–324, Lake Como, Italy, 18-20 April 2001. Springer-Verlag. ISBN 3-540-41899-7. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_handeye.ps.gz`.

William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, 1999. ISBN 0-262-19423-6. URL `http://www.cs.bham.ac.uk/~wbl/aigp3/ch08.pdf`.

Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

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, 1999. ISBN 0-262-19423-6. URL `http://cswww.essex.ac.uk/staff/rpoli/papers/Poli-AIGP3-1999.pdf`.

James Gary Propp and David Bruce Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9(1 & 2):223–252, 1996. doi: 10.1002/(SICI)1098-2418(199608/09)9:1/2⟨223::AID-RSA14⟩3.0.CO;2-O.

Christian M. Reidys and Peter F. Stadler. Combinatorial landscapes. *SIAM Review*, 44(1): 3–54, March 2002. doi: 10.1137/S0036144501395952.

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

David Stirzaker. *Probability and Random Variables A Beginner's Guide*. Cambridge University Press, 1999. ISBN 0-521-64445-3.

Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press. URL `http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Curiosities.ps`.

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