GECCO'2002 W. B. Langdon, *et. al.*, editors, pages 812–819, New York, 9-13 July, Morgan Kaufmann

Convergence Rates for the Distribution of Program Outputs

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
Tel: +44 (0) 20 7679 4436, Fax: +44 (0) 20 7387 1397

Abstract

Fitness distributions (landscapes) of programs tend to a limit as they get bigger. Markov chain convergence theorems give general upper bounds on the linear program sizes needed for convergence. Tight bounds (exponential in N, $N \log N$ and smaller) are given for five computer models (any, average, cyclic, bit flip and Boolean). Mutation randomizes a genetic algorithm population in $\frac{1}{4}(l+1)(\log(l)+4)$ generations. Results for a genetic programming (GP) like model are confirmed by experiment.

1 INTRODUCTION

We have shown that the fitness distribution of sufficiently large programs will converge eventually [Langdon and Poli, 2002]. We now use standard results from Markov chain theory, to give quantitative bounds on the length of random linear genetic programs, such that the distribution of their outputs is independent of their size. The bounds depend heavily on the type of computer, the fitness function, and scale with the size of the computer's memory. Proving general convergence rates for the fitness of programs requires detailed consideration of the interaction within random programs between different input values. In some cases we can do this, while in others we leave this for future work.

The next section summarises the Markov model of linear genetic programming (GP). Sections 3.1–3.5 describe a wide range of models of computers for running linear GP and prove their convergence properties. Sections 3.1 and 3.2 highlight the importance of internal coupling. Section 3.3 and 3.4 use Markov minorization to prove upper bounds for, firstly any computer (3.3) and secondly average computers (3.4). Section 3.5 is close to some practical GP systems [Banzhaf *et al.*, 1998], while Section 3.6 gives an interesting result on the convergence of bit string genetic algorithms. The application of our results are given in Section 4 and we conclude in Section 5.

2 MARKOV MODELS OF PROGRAM SEARCH SPACES

[Langdon and Poli, 2002] deals with both tree based and linear GP. For simplicity we will consider only large random linear programs. However we anticipate similar bounds also exist for large random trees.

In the following models, the computer is split in two. The random program and all control circuitry form one part, while the computer's *data* memory, inputs and outputs form the second. The memory is treated as a finite state machine (FSM) with 2^N states. (Where N is the number of bits of data in the machine.) Each time a program instruction is executed, data are read from the memory, the result is calculated and written into the memory. This changes the pattern of bits inside the memory. This is modelled as moving the FSM from one state to another. This is deterministic. Given a bit pattern and an instruction, the bit written to memory is also fixed. That is, given a particular state, executing a particular instruction will always move the FSM to the same state. Of course, in general, executing a different instruction will move the FSM to a different state.

Before starting a program, the memory is zeroed and the inputs are loaded[1]. As each instruction in the program is executed, the FSM is updated. If the program is l instructions long, the memory (FSM) is updated l times and then the program *halts*. The program's

[1] Some practical GPs, e.g. Discipulus, write protects the inputs.

answer is then read from the memory. I.e. the output is determined by the last state reached by the FSM.

If there are I instructions then there are I^l possible programs of length l. Suppose we calculate each one's fitness by running it on a fix set of tests and then compare its answers with the tests' target values. The fitness distribution is given by plotting a histogram of the number of programs (divided by I^l) with each fitness value. [Langdon and Poli, 2002] shows, for big enough l, the distribution for one length is pretty much the same as for any another.

It is usually impractical to generate every program of a given length. Instead we consider a large randomly drawn sample of the possible programs. To generate a random program of length l, we simply choose at random l times from the instruction set. When this program is executed, the FSM (i.e. the computer's memory) is updated randomly at each step. But note that the FSM can only move to one of a small number of possible states at each step. Which ones are possible depends *only* on its current state. These are the conditions for a Markov process.

Provided it is possible for a program to set any pattern of bits and there is an instruction which leaves a bit pattern unchanged (no-op), then the Markov process will converge. These conditions mean the FSM is connected, i.e. it is possible to move, in a finite number of steps, from any state to any other. The requirement for at least one no-op keeps the maths simple later by avoiding cycles but its not fundamental. If these conditions hold, then the process of randomly updating the FSM is a Markov process with nice limiting properties. For example, this means if we run the process for long enough (i.e. execute enough random instructions) the probability of the FSM being in any particular state will be a constant. I.e. the probability does not change as more random instructions are executed. (Although it may depend upon which state we are considering.) Secondly it does not depend on how the FSM was started. Since the program's answer is read from the memory, it is determined by the FSM final state. I.e. the probability of any particular answer being given by a random sequence of instructions does not depend on how many instructions there are (provided there are sufficient). As the probability is independent of starting conditions, which include the program's inputs, it does not depend on them either. However if the inputs are write protected, then they are external to the computer's data memory. In which case changes to the inputs have to be considered as changes to the state machine and hence may change the limiting distribution of its outputs.

This convergence applies to the whole of the computer's memory. We expect shorter random programs (i.e. fewer instructions) to be needed if less memory is used. Therefore, depending upon the type of the computer, we might expect much shorter programs to be sufficient to give convergence of just the (small) output register. Indeed, in some cases, we can prove this.

3 CONVERGENCE RESULTS

3.1 SLOW CONVERGENCE EXAMPLE

[Rosenthal, 1995] gives several results on the number of random steps needed by a Markov process to reach equilibrium. In this section we chose what appears to close to a worst case, in order to show an example where the random programs have to be very long indeed. The example is a frog's random walk around a circle of W lily pads. At each time step, the frog can only jump clockwise, anti-clockwise or stay still. [Rosenthal, 1995] uses Markov analysis to show that after sufficient time steps the frog may be found on any lily with equal probability ($\frac{1}{W}$) and to show $\mathrm{O}(W^2)$ steps are needed before the chance of any of them being occupied is approximately the same.

We shall use the total variation distance between two probability distributions to indicate how close they are. The total variation distance between probability distributions a and b is defined as $||a - b|| = \sup_{x \subseteq \chi} |a(x) - b(x)|$. I.e. the largest value (supremum) of the absolute difference in the probabilities [Rosenthal, 1995]. The sup is taken over *all subsets*, i.e. every possible grouping of states x, not just single points. (Otherwise it would be small as long as $a(x)$ and $b(x)$ are both always small, even if the distributions a and b are not similar).

Suppose there are three instructions: do nothing, add one to memory and subtract one from memory. We have carry over from one memory word to the next and wrap around if all memory bits are set or all are clear. This corresponds to the frog jumping from lily pad W to 1 or 1 to W. (Remember the lilies are arranged in a circle.) Part of the memory is loaded with inputs and part designated the output register. (Read only inputs are not permitted in this example.)

[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 (sup) 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 $\mathrm{O}(2^{2N})$ for the distribution of FSM 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, programs with between 4,000 and 23,000 random instructions need to be considered before each state is equally likely.

The output is read from part of the whole computer's memory (the m bit output register). Since in the limit each of the 2^N states is equally likely, each of the 2^m possible answers is also equally likely. The special instruction set means, the distribution of answers takes just as long to converge as does the whole of the computer. This is despite the fact that the output register only occupies a fraction of the whole of the computer.

The output of any program is $x + p \bmod 2^m$, where x is the input and p is a constant (specific to that program). Note this computer can only implement 2^m 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.

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

3.2 FAST CONVERGENCE EXAMPLE

The second example also uses results from [Rosenthal, 1995] (Bit flipping) [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. Again input (n bits) and output (m bits) registers are defined (and read only inputs are forbidden).

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 $||\mu_k - \pi|| \leq 10\%$. I.e. random programs of length $\frac{1}{4}(N+1)(\log(N)+4)$ will be enough to ensure each bit of the computer is equally likely to be set as to be clear, regardless of the programs' inputs. (Section 3.5 explains why $c_1 = 4$ is sufficient.) Again in the limiting distribution each state is equally likely.

Returning to our computer with 1 byte of memory, programs with no more than 14 random instructions are needed to ensure 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 3.5).

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 non-overlapping bits of the input register are discarded.) 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. Therefore 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. Finally the distribution of program fitnesses converges at least as fast. However, since a given fitness function need not treat each of the m output bits equally, its limiting distribution need not be uniform and it can converge faster.

This suggests 9 random instructions will be enough to ensure the output of a 1 byte Boolean (i.e. one bit) computer is random. Further that every Boolean fitness function will also be close to its limiting distribution. Note this does not depend upon the number of input bits n (although n cannot exceed 8 of course).

3.3 ANY COMPUTER

This section 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.

The general Markov minorization condition [Rosenthal, 1995] is fairly complex. Fortunately for this proof (and Section 3.4) we can use a simplified special case.

Define P_{ij} to 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 $P_{ij} = 0$.) The complete matrix P formed from all the P_{ij} is known as the transition matrix. A simple Markov minorization condition is that there is at least one state which can be reached from all the others in one step. That is, there is at least one column of the transition matrix P whose entries are all positive (not zero). Given this the corresponding Markov chain converges geometrically quickly [Rosenthal, 1995].

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

where

$$\beta = \sum_{y=1..2^N} \min_{x=1..2^N} P(x, y)$$

I.e. β is the sum of the minimum values of the entries in each column of P.

All fine and dandy, however, there are 2^N elements in each column of P but only a small number I of possible instructions. Thus there will be at least $2^N - I$ elements in each column of P that are zero. Thus $\beta = 0$. This does not mean that the Markov process will not converge or even that it will take a long time. It just means the simple application of a minorization condition does not take us very far.

One way round this difficulty is to replace P by P^k in the minorization condition. 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 k times. For any given state there are up to I^k states the FSM could be in after k instructions. (Ignoring overlaps, each is equally likely.) So if $I^k \geq 2^N$ 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 be the number of steps required. This meets the minorization condition for P^a. In fact $P^a(s_0, y) \geq I^{-a} > 0 \; \forall y$. Therefore $\beta \geq I^{-a}$ and so for any computer:

$$||\mu_k - \pi|| \leq (1 - I^{-a})^{\lfloor k/a \rfloor}$$

Setting $|| \cdot ||$ to 10% yields a convergence length k for any computer with I instructions $k \leq 2.3025851 a I^a$. Where a is the number of instructions to reach any state. ($a < 2^N$).

3.4 AVERAGE COMPUTER MODEL

Suppose given any possible data in memory each of the I instructions independently randomises it.

Thus for any state x $P(x, y) = 0$ or $1/I$ or $2/I$ or $\ldots$ or I/I. Most elements of the transition matrix $P(x, y)$ will be zero but between 1 and I elements in each column will be non zero. 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 l instructions, each element of $P^l(x, y)$ will be a multiple i (possibly zero) of I^{-l}. 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 $= I^N$. So the distribution of i's mean is $I^l/2^N$ and its standard deviation is $\sqrt{I^l \times 1/2^N \times (1 - 1/2^N)}$. For large I^l the distribution will approximate a Normal distribution. If $I^l \gg 2^N$, 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 $I^l/2^N - 5\sqrt{I^l \times 1/2^N}$. So β will be at least $2^N I^{-l}(I^l/2^N - 5\sqrt{I^l \times 1/2^N})$. I.e. $\beta \geq (1 - 5\sqrt{I^{-l} \times 2^N})$.

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

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

Now we have a practical value of β we can use the minorization condition on P^l to give

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

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

$$\begin{aligned}
\alpha^{\lfloor k/l \rfloor} &\geq ||\mu_k - \pi|| = 0.1 \\
\lfloor k/l \rfloor \log \alpha &\geq -2.3025851 \\
k &\leq \frac{-2.3025851 \; l}{\log \alpha} \\
&= \frac{-2.3025851 \; (-2\log(\alpha/5) + N \log 2)}{\log \alpha \log I}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{-2.3025851\ (2\log 10 + N\log 2)}{-\log 2\log I} \\
k &\leq \frac{15.298044 + 2.3025851\ N}{\log I} \qquad (1)
\end{aligned}
$$

Note this predicts quite rapid convergence for our randomly wired computer. E.g. if it has 8 instructions $k \approx 7 + N$. That is for a one byte 8 random instruction computer programs longer than 16 will be close to the computer's limiting distribution.

Inequality (1) 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 (1) becomes $k \leq (15.298044 + 2.3025851\ m)/\log I$. E.g. for Boolean problems ($m = 1$). Only about 9 random instructions are need for an 8 random instruction computer to have effectively reached the programs' outputs limiting distribution.

As in Section 3.2, we can construct a look up table for a particular program which contains the value it yields for each input. It will have 2^n rows, each of which can have one of 2^m values. As in Section 3.2, the relationship between each row is determined by the program. However, the more powerful architecture means that each row can have an apparently independent value. So there are $(2^m)^{2^n}$ possible tables (and hence $2^{m\times 2^n}$ possible functions). For a given input (i.e. row in the lookup table) each output is equally likely. If each row were independent then every complete table (and hence each function) would be equally likely. A loose argument says, we can fill the table by running k random instructions and storing the output register in the table. We then re-use the current contents of the memory (first noting the contents of the input register). We run another k random instructions. This yields another random output value, which is effectively independent of the first. This is stored in the table row corresponding to the intermediate value of the input register. It will take at least 2^n such operations to fill the table but each row will be independent and so each of the $2^{m\times 2^n}$ possible tables will be equally likely. I.e. running $\mathrm{O}(2^n m/\log I)$ random instructions will ensure each function is equally likely (cf. no free lunch, NFL [Wolpert and Macready, 1997]). Finally the distribution of program fitness' will also have converged by this point (though its distribution need not be uniform and, for a specific fitness function, it may have converged more quickly).

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). However since it lacks any particular regularities, we would anticipate random search to be as effective as any other technique (such as genetic programming) at programming it.

3.5 FOUR BOOLEAN INSTRUCTION COMPUTER

This model is the closest to actual (linear) GPs. 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, overwriting 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 before, we look at the distribution of memory patterns that are produced by running all programs of a given length, l, by considering a large number of random programs of that length. 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.

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. Thus we can update the analysis in Section 3.2 based on [Diaconis, 1988, pages 28–30] and [Rosenthal, 1995].

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

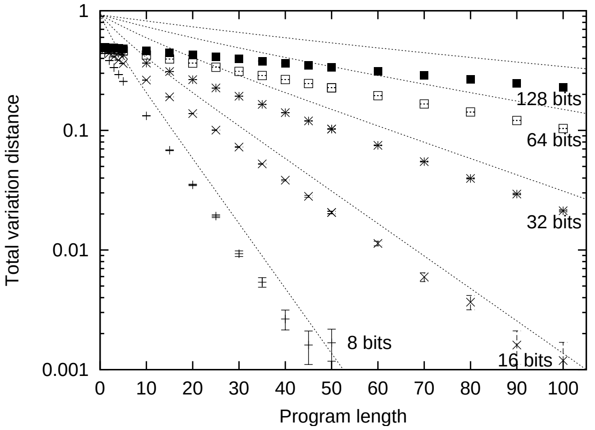

Figure 1: Convergence of outputs of random 3 bit Boolean (AND, NAND, OR, NOR) linear programs with different memory sizes. Note the agreement with upper bound $\sqrt{1/2(\exp(me^{-2l/N})-1)}$ (dotted lines).

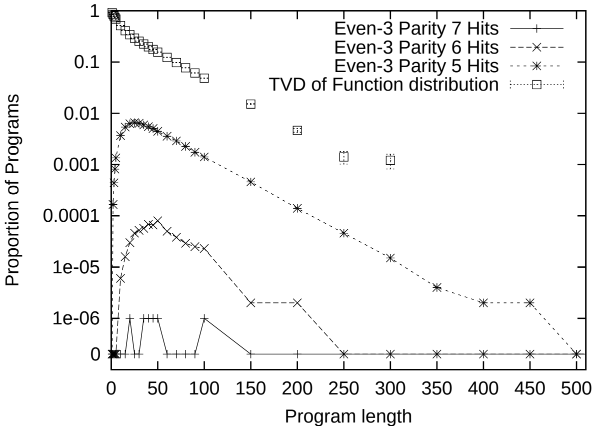

Figure 2: Convergence of Even-3 parity fitness. Even in the limit some functions are more common than others. Longer programs are needed to achieve convergence of functions than of outputs, (Figure 1, 8 bits)

$$< \quad \frac{1}{2}\sum_{j=1}^{\infty}\frac{N^j}{j!}e^{-\frac{2j}{N}l}$$

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

Requiring $||\mu_l - \pi||$ not to exceed 10% gives the upper bound $l \leq \frac{1}{2}N(\log(N)+4)$. That is, programs need only be twice as long on this computer (which is capable of real computation) as on the simple bit flipping computer of Section 3.2.

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 (2) 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 1 confirms this.

On this computer, the output of a program given one input is strongly related to its output with another input. This means the loose lookup table argument of Section 3.4 breaks down. The distribution of functions does converge (albeit more slowly than the distribution of outputs) but in the limit each of the $2^{m\times 2^n}$ possible functions are not equally likely (see Figure 2). Detailed modelling of this is left to further work.

How long it takes for a fitness distribution to converge will depend upon the nature of the fitness function. Once its function distribution has converged, the fitness distribution must also converge. However, it could require substantially shorter programs.

3.6 CONVERGENCE IN BIT STRING GAs

The bit flipping model (in Section 3.2) is very close to standard mutation in bit string genetic algorithms (GAs). The principle difference is in GAs the number of bits flipped follows a Poission 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 $O(l\log(l))$ 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, 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.

4 APPLICABILITY

The results in Section 3 refer to specific types of computation, nevertheless we feel they are useful, particularly for common varieties of genetic programming.

The model does not cover programs that contain instructions that are executed more than once. I.e. no loops or backward jumps. (Forward jumps are in principle acceptable, as long as the number of executed instructions remains large.) 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]. The difficulty for the *proofs* is that, in general, repeating (a sequence of) random instructions does not give, on average, the same results as the same number of random instructions chosen independently. (If the loop contains enough random instructions to reach the limiting distribution then the problem does not arise because the input to the next iteration to the loop is already in the limiting distribution and so will remain there.) Similarly, there is no problem if the loop is followed by a large number of random instructions.

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 intervals, allowing the program to continue (or to be aborted). These have been called "any time algorithms". They have been used in GP, e.g. [Teller, 1994].

The dominant factors in determining length required for near convergence are the type of computer considered and the size of its (data) memory. The scaling law is given by the type. Comparing the four types in Section 3 suggests that the degree of interconnections in the state space is the important factor. 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 lead to exponential scaling. We suggest that the "bit flipping" and "4 Boolean Function" models are more typical and so we suggest $O(N \log N)$ would be found on real computers.

Most computers support random access at the byte or word level. This would suggest N should be the number of bytes or words in the data memory. However then we would expect the individual bits in each byte or word to be highly correlated, and so we would anticipate the simple $O(N \log N)$ law would break down. I.e. further random instructions will be required to randomise them. This might result in a multiplicative factor of $8 \log 8$ or $32 \log 32$ but this yields the same scaling law $((8 \log 8)N/8 \log N/8 = \mathrm{O}(N \log N))$ possibly with different numerical values.

Some linear GP systems write protect their inputs. 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.

All of the calculations in Section 3 have been explicitly concerned with the distribution of answers produced by the programs and the functions implemented by them. In principle we can use the Markov arguments to consider the distribution of functions implemented by the programs in other types of computer. The Markov process now becomes a sequence of changes in function. We start with the identity function and the distribution of functions rapidly spreads through the $2^{N^{2^N}}$ functions. An obvious difficulty is that the size of the transition matrixes increases exponentially (from $2^N \times 2^N$ to $2^{N^{2^N}} \times 2^{N^{2^N}}$). This might lead to an exponential (or worse) increase the upper bound scaling laws.

The random computer (cf. Section 3.4) gives an interesting model. Indeed it represents the average behaviour over all possible computers (of this type).

Finally 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.

5 CONCLUSIONS

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 ($2.31aI^a$, where I is the number of instructions and a is the length programs needed to store every possible value in the computer's memory, Section 3.3). 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 3.1) converges most slowly, $\leq 0.35\ 2^{2N}$, for large N. In contrast the bit flip computer (Section 3.2) takes only $\frac{1}{4}(N+1)(\log(m)+4)$ random instructions (m bits in output register). However in both, the distributions of outputs and of functions converge at this same rate to a uniform limiting distribution.

In Section 3.4 we introduced a random, model of computers. This represents the average behaviour over all computers (cf. NFL [Wolpert and Macready, 1997]). It takes less than $(15.3 + 2.3m)/\log I$ random instructions to get close to the uniform output limit. However a less formal arguments suggests a multiplicative factor of 2^n needs to be included before the distribution of functions is also close its limit.

Section 3.5 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.

Section 3.6 shows the number of generations ($\frac{1}{4}(l + 1)(\log(l) + 4)$) needed for mutation alone to randomise a bit string GA (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 3.2) and four Boolean instruction (Section 3.5) models. We speculate rapid O(|test set|$N \log m$) convergence in fitness distributions may be observed.

It is ten years since Jaws 1, these are the first general quantitative scaling laws on the space that genetic programming searches. They provide theoretical support for some pragmatic choices made in GP.

Acknowledgments

I would like to thank Jeffrey Rosenthal, David Corney, Tom Westerdale, James A. Foster, Riccardo Poli, Ingo Wegener, Nic McPhee, Michael Vose and Jon Rowe.

References

[Bäck, 1996] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York, 1996.

[Banzhaf *et al.*, 1998] 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.

[Blickle, 1996] Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis.* PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996.

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

[Gao, 1998] Yong Gao. An upper bound on the convergence rates of canonical genetic algorithms. *Complexity International*, 5, 1998.

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

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

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

[Teller, 1994] 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.

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