EuroGP'2006 Pierre Collet, Marco Tomassini, Marc Ebner and Steven Gustafson editors, Budapest, Hungary 10-12 April, 2006, LNCS, Springer-Verlag

The Halting Probability in von Neumann Architectures

W. B. Langdon and R. Poli

Department of Computer Science, University of Essex, UK

Abstract. Theoretical models of Turing complete linear genetic programming (GP) programs suggest the fraction of halting programs is vanishingly small. Convergence results proved for an idealised machine, are tested on a small T7 computer with (finite) memory, conditional branches and jumps. Simulations confirm Turing complete fitness landscapes of this type hold at most a vanishingly small fraction of usable solutions.

1 Introduction

Recent work on strengthening the theoretical underpinnings of genetic programming (GP) has considered how GP searches its fitness landscape [1,2,3,4,5,6]. Results gained on the space of all possible programs are applicable to both GP and other search based automatic programming techniques. We have *proved* convergence results for the two most important forms of GP, i.e. trees (without side effects) and linear GP [1,7,8,9,10]. As remarked more than ten years ago [11], it is still true that few researchers allow their GP's to include iteration or recursion. Indeed there are only about 50 papers (out of 4631) where loops or recursion have been included in GP. Without some form of looping and memory there are algorithms which cannot be represented and so GP stands no chance of evolving them.

We extend our results to Turing complete linear GP machine code programs. We analyse the formation of the first loop in the programs and whether programs ever leave that loop. Mathematical analysis is followed up by simulations on a demonstration computer. In particular we study how the frequency of different types of loops varies with program size. In the process we have executed programs of up to 16 777 215 instructions. These are perhaps the largest programs ever (deliberately) executed as part of a GP experiment. (beating the previous largest of 1 000 000 [12]). Results confirm theory and show that, the fraction of programs that produce usable results, i.e. that halt, is vanishingly small, confirming the popular view that machine code programming is hard.

The next two sections describe the T7 computer and simulations run on it, whilst Sections 4 and 5 present theoretical models and compare them with measurement of halting and non-halting programs. The implications of these results are discussed in Section 6 before we conclude (Section 7).

Table 1. T7 Turing Complete Instruction Set

Instruction	*#operands*	*operation*	*v* set	
ADD	3	A + B→C	v	Every ADD operation either sets or clears the overflow bit v.
BVS	1	#addr→pc if v=1		LDi and STi, treat one of their arguments as the address of the data. They allow array manipulation without the need for self modifying code. (LDi and STi data addresses are 8 bits.)
COPY	2	A→B		
LDi	2	@A→B		
STi	2	A→@B		
COPY_PC	1	pc→A		
JUMP	1	addr→pc		To ensure JUMP addresses are legal, they are reduced modulo the program length.

2 T7 an Example Turing Complete Computer

To test our theoretical results we need a simple Turing complete system. Our seven instruction CPU (see Table 1) is based on the Kowalczy F-4 minimal instruction set computer `http://www.dakeng.com/misc.html`, cf. appendix of [13]. T7 consists of: directly accessed bit addressable memory (there are no special registers), a single arithmetic operator (ADD), an unconditional JUMP, a conditional Branch if oVerflow flag is Set (BVS) jump and four copy instructions. COPY_PC allows a programmer to save the current program address for use as the return address in subroutine calls, whilst the direct and indirect addressing modes allow access to stacks and arrays.

Eight bit data words are used. The number of bits in address words is just big enough to be able to address every instruction in the program. E.g., if the program is 300 instructions, then BVS, JUMP and COPY_PC instructions use 9 bits. These experiments use 12 bytes (96 bits) of memory (plus the overflow flag).

3 Experimental Method

There are simply too many programs to test all of them. Instead we gather representative statistics about those of a particular length by randomly sampling programs of that length. Then we sample those of another length and so on, until we can build up a picture of the whole search space.

To be more specific, one thousand programs of each of various lengths ($30 \ldots 16\,777\,215$ instructions) are each run from a random starting point, with random inputs, until either they reach their last instruction and stop, an infinite loop is detected or an individual instruction has been executed more than 100 times. (In practise we can detect almost all infinite loops by keeping track of the machine's contents, i.e. memory and overflow bit. We can be sure the loop is infinite, if the contents is identical to what it was when the instruction was last executed.) The programs' execution paths are then analysed. Statistics are gathered on the number of instructions executed, normal program terminations, type of loops, length of loops, start of first loop, etc.

4 Terminating Programs

The introduction of Turing completeness into GP raises the halting problem, in particular how to assign fitness to a program which may loop indefinitely [14]. We shall give a lower bound on the number of programs which, given arbitrary input, stop, and show how this varies with their size.

The T7 instruction set has been designed to have as little bias as possible. In particular, given a random starting point a random sequence of ADD and copy instructions will create another random pattern in memory. The contents of the memory is essentially uniformly random. I.e. the overflow v bit is equally likely to be set as to be clear, and each address in memory is equally likely. (Where programs are not exactly a fraction of a power of two long, JUMP and COPY_PC addresses cannot completely fill the number of bits allocated to them. This introduces a slight bias in favour of lower addresses.) So, until correlations are introduced by re-executing the same instructions, we can treat JUMP instructions as being to random locations in the program. Similarly we can treat half BVS as jumping to a random address. The other half do nothing. We will start by analysing the simplest case of a loop formed by random jumps. First we present an accurate Markov chain model, then Section 4.2 gives a less precise but more intuitive mathematical model. Section 4.3 considers the run time of terminating programs.

4.1 Markov Chain Model of Non-Looping Programs

The Markov chain model predicts how many programs will not loop and so halt. This means it, and the following segments model, do not take into account those programs which are able to escape loops and do reach the end of the program and stop. As a program runs, the model keeps track of: the number of new instructions it executes, if it has repeated any, and if it has stopped. The last two states are attractors from which the Markov process cannot escape. State i means the program has run i instructions without repeating any. The next instruction will take the program from state i either to state $i+1$, to SINK or to HALT. In our model the probabilities of each of these transitions depends only on i and the program length L, see Figure 1. We construct a $(L+2) \times (L+2)$ Markov transition matrix T containing the probabilities in Figure 1. The probabilities of reaching the end of the program (HALT) or the looping (SINK) are given by two entries in T^L. Figure 2 shows our Markov chain describes the fraction of programs which never repeat any instructions very well.

4.2 Segment Model of Non-Looping Programs

As before, we assume half BVS instructions cause a jump. So the chance of program flow not being disrupted is 11/14. Thus the average length of uninterrupted random sequential instructions is $\sum_{i=1}^{L/2} i\,(11/14)^{i-1}\,3/14$. We can reasonably replace the upper limit on the summation by infinity to give the geometric distribution (mean of $14/3 = 4.67$ and standard deviation $\sqrt{14^2/3^2 \times 11/3} = 8.94$).

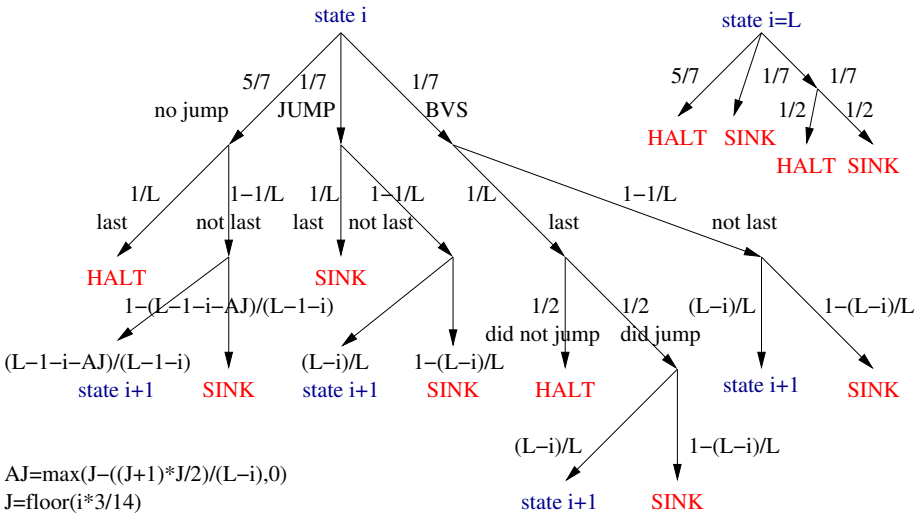

Fig. 1. Probability tree used to create Markov model of the execution of random Turing complete programs. HALT indicates a terminating program, while SINK means the start of a loop.

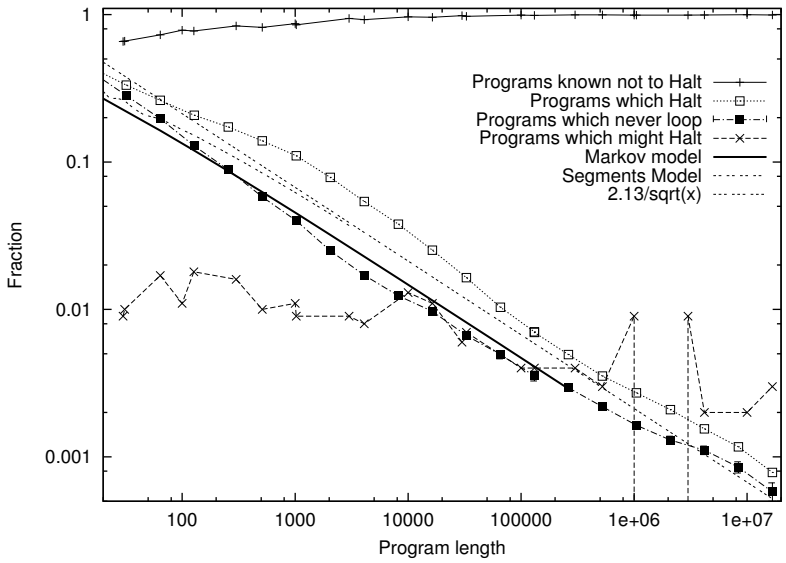

Fig. 2. Looping + and terminating (□ ■) T7 programs To smooth these two curves, 50 000 to 200 000 samples taken at exact powers of two. (Other lengths lie slightly below these curves). Solid diagonal line is the Markov model of programs without any repeated instructions. This approximately fits ■, especially if lengths are near a power of two. The other diagonal line is the segments model and its large program limit, $2.13\,\text{length}^{-\frac{1}{2}}$. The small number of programs which did not halt, but which might do so eventually, are also plotted ×.

For simplicity we will assume the program's L instructions are divided into $L/4.67$ segments. Two thirds end with a JUMP and the remainder with an active BVS (i.e. with the overflow bits set). The idea behind this simplification is that if we jump to any of the instructions in a segment, the normal sequencing of (i.e. non-branching) instructions will carry us to its end, thus guaranteeing the last instruction will be executed. The chance of jumping to a segment that has already been executed is the ratio of already executed segments to the total. (This ignores the possibility that the last instruction is a jump. We compensate for this later.)

Let i be the number of instructions run so far divided by 4.67 and $N = L/4.67$. At the end of each segment, there are three possible outcomes: either we jump to the end of the program (probability $1/N$) and so stop its execution; we jump to a segment that has already been run (probability i/N) so forming a loop; or we branch elsewhere. The chance the program repeats an instruction at the end of the ith segment is

$$= \frac{i}{N}(1-\frac{2}{N})(1-\frac{3}{N})\dots(1-\frac{i}{N})$$

I.e. it is the chance of jumping back to code that has already been executed (i/N) times the probability we have not already looped or exited the program at each of the previous steps. Similarly the chance the program stops at the end of the ith segment is

$$\begin{aligned}\frac{1}{N}(1-\frac{2}{N})(1-\frac{3}{N})\dots(1-\frac{i}{N}) = \frac{1}{N^i}\frac{(N-2)!}{(N-i-1)!} = \frac{(N-2)!}{N^{N-1}}\frac{N^{N-1-i}}{(N-i-1)!}\\ = (N-2)!N^{1-N}e^N\mathrm{Psn}(N-i-1,N)\end{aligned}$$

Where $\mathrm{Psn}(k,\lambda) = e^{-\lambda}\lambda^k/k!$ is the Poisson distribution with mean λ.

The chance the program stops at all (ignoring both the possibility of leaving the first loop and of other loops for the time being) is simply the sum of all the ways it could stop

$$\sum_{i=1}^{N-1}(N-2)!N^{1-N}e^N\mathrm{Psn}(N-i-1,N) = (N-2)!N^{1-N}e^N\sum_{j=0}^{N-2}\mathrm{Psn}(j,N)$$

For large mean ($N \gg 1$) $\sum_{j=0}^{N-2}\mathrm{Psn}(j,N)$ approaches $1/2$ (see Figure 3). Therefore the chance of long programs not looping is (using Gosper's approximation $n! \approx \sqrt{(2n+1/3)\pi}\, n^n e^{-n}$ and that for large x $(1-1/x)^x \approx e^{-1}$):

$$\approx 1/2(N-2)!N^{1-N}e^N \approx 1/2\sqrt{2\pi/N}\left(1+\frac{37}{12N}\right)$$

That is (ignoring both the possibility of leaving the first loop and of other loops for the time being) the probability of a long random T7 program of length L stopping is about $1/2\sqrt{2\pi 14/3L}\,(1+\frac{37\times 14}{36L}) = \sqrt{7\pi/3L}\,(1+259/18L)$. As mentioned above, we have to consider explicitly the $3/14$ of programs where the last instruction is itself an active jump. Including this correction gives the chance of a long program not repeating any instructions as $\approx 11/14\sqrt{7\pi/3L}\,(1+259/18L)$. Figure 2 shows this $\sqrt{\text{length}}$ scaling fits the data reasonably well.

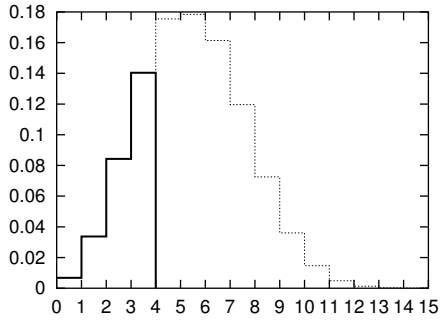

Fig. 3. Poisson distribution, with mean=5. Note region 0 to mean-2 corresponding to the segments model of non-looping programs.

4.3 Average Number of Instructions run before Stopping

The average number of instructions run before stopping can easily be computed from the Markov chain. This gives an excellent fit with the data (Figure 4). However, to get a scaling law, we again apply our segments model.

The mean number of segments evaluated by programs that do halt is: $\frac{\sum_{i=1}^{N-1} i/N \ \prod_{j=2}^{i}(1-j/N)}{\sum_{i=1}^{N-1} 1/N \ \prod_{j=2}^{i}(1-j/N)}$.Consider the top term for the time being

$$= 1/N \sum_{i=1}^{N-1} i \exp\left(\sum_{j=2}^{i} \log(1-j/N)\right) < 1/N \sum_{i=1}^{N-1} i \exp\left(\sum_{j=2}^{i} -j/N\right)$$

$$= 1/N \sum_{i=1}^{N-1} i \exp\left(-\frac{i(i+1)-2}{2N}\right) < 1/N e^{\frac{1}{N}} e^{-\frac{1}{2N}} \sum_{i=1}^{N-1} i \exp\left(-\frac{i^2}{2N}\right)$$

$$\approx e^{\frac{1}{2N}} 1/N \int_{1/2}^{N-1/2} x e^{-x^2/2N} dx = e^{\frac{1}{2N}} \left[e^{-x^2/2N}\right]_{N-1/2}^{1/2} \approx e^{\frac{3}{8N}}$$

Dividing $e^{\frac{3}{8N}}$ by the lower part (the probability of a long program not looping) gives an upper bound on the expected number of segments executed by a program which does not enter a loop $\approx \frac{e^{3/8N}}{1/2\sqrt{2\pi/N}\left(1+\frac{37}{12N}\right)}$ $\approx (1+\frac{3}{8N})(1-\frac{37}{12N})\sqrt{2N/\pi} \approx (1-\frac{65}{24N})\sqrt{2N/\pi}$. Replacing the number of segments N ($N = 3L/14$) by the the number of instructions L gives, to first order, $14/3 \times \sqrt{2 \times (3L/14)/\pi} = \sqrt{28L/3\pi} = 1.72\sqrt{L}$. Figure 4 shows, particularly for large random programs, this gives a good bound for the T7 segments model. However, as Figure 2 confirms, the segments model itself is an over estimate.

Neither the segments model, nor the Markov model, take into account derandomisation of memory as more instructions are run. This is particularly acute since we have a small memory. JUMP and COPY_PC instructions introduce correlations between the contents of memory and the path of the program counter. These make it easier for loops to form.

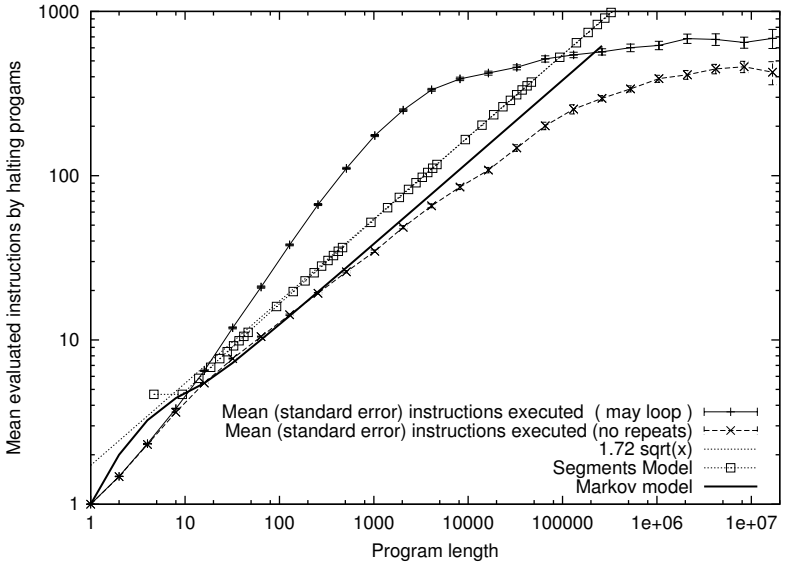

Fig. 4. Instructions executed by programs which halt. Std dev≈mean suggests geometric distribution. As Fig. 2, larger samples used to increase reliability. Models ok for short programs. However as random programs run for longer, COPY_PC and JUMP derandomise the 96 bit memory, so easing looping.

5 Loops

5.1 Code Fragments which Form Loops

If a BVS or an unconditional JUMP instruction jumps to an instruction that has been previously obeyed, a loop is formed. Unless something is different the second time the instruction is reached (e.g. the setting of the overflow flag) the program will obey exactly the same instruction sequence as before, including calculating the same answers, and so return to start of the loop again. Again, if nothing important has changed, the same sequence of instructions will be run again and an infinite loop will be performed. Automated analysis can, in most cases, detect if changes are important and so the course of program execution might change, so enabling the program to leave the loop.

We distinguish loops using the instruction which formed the loop. I.e. the last BVS or JUMP. There are two common ways JUMP can lead to a loop: either the program goes to an address which was previously saved by a LOAD_PC instruction or it jumps to an address which it has already jumped to before. E.g. because the two JUMP instructions take their target instruction from the same memory register. A loop can be formed even when one JUMP address is slightly different from the other. Therefore we subdivide the two types of JUMP loops into three sub-classes: those where we know the address register has not been modified, those where the least significant three bits might have been changed, and the rest. See Figure 5.

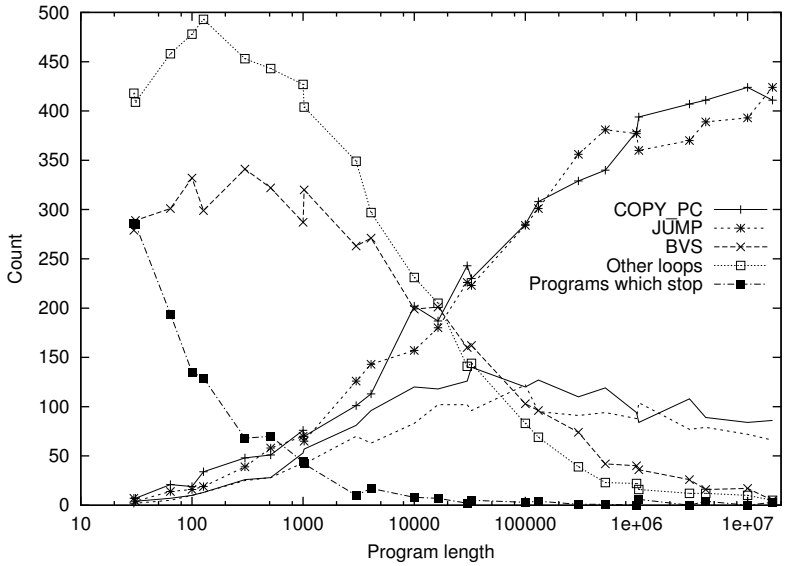

Fig. 5. Types of first loop in T7 random programs. In large programs most loops have a last branch instruction which is either COPY_PC (+) or JUMP (∗) with unsullied target addresses. In a further 20–30% of programs, the lower three bits of the target address may have been modified (plotted as solid and dashed lines). Since BVS (×) jumps to any address in the program, it is less likely to be responsible for loops as the program gets larger. Mostly, the fraction of unclassified loops (□) also falls with increasing program size.

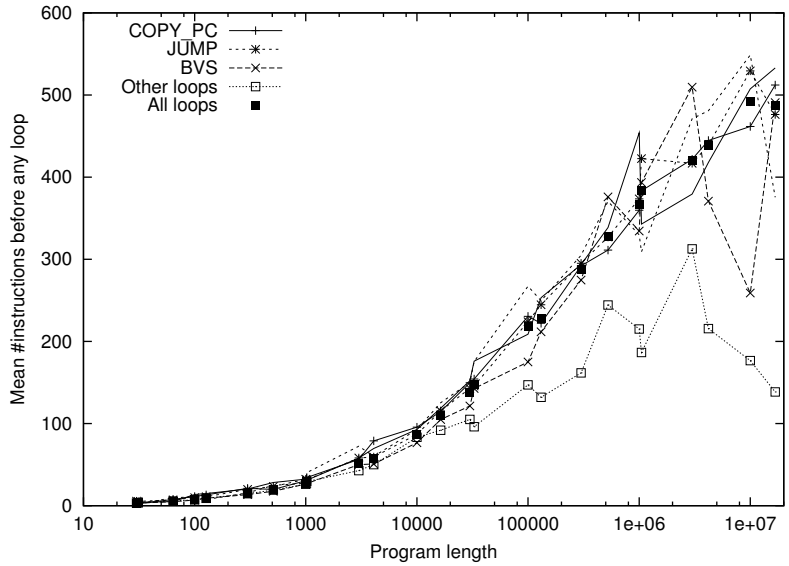

Fig. 6. Number of instructions before the first loop in T7 programs.

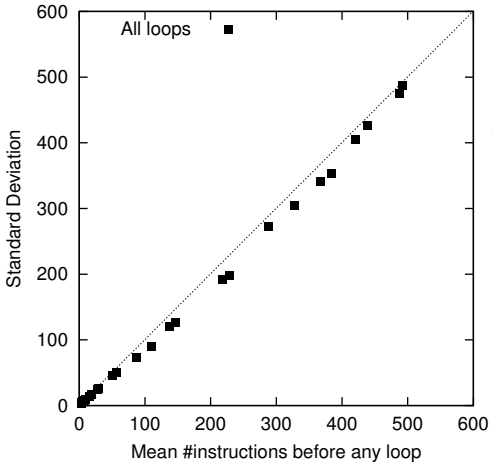

Fig. 7. Mean and std dev of the number of T7 instructions before the first loop. In a geometric distribution the mean and std dev are ≈ equal (dotted line).

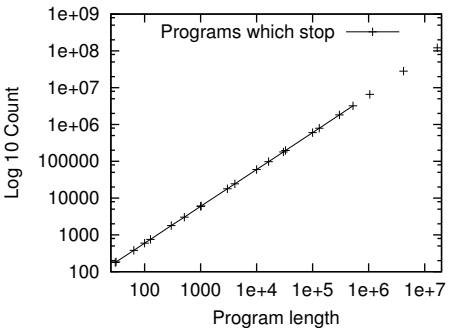

Fig. 8. The number of Halting programs rises exponentially with length despite getting increasingly rare as a fraction of all programs. Note log log vertical scaling.

5.2 Number of Instructions Before the First Loop

Since we are stopping on the first loop there is competition between the different types of loop and only the fastest to form are observed. So the observed mean number of instructions before a loop is formed is pretty much independent of loop type (cf. Figure 5). With bigger programs, BVS loops get longer and so might be expected to appear later in a program's execution. This apparent contradiction is resolved by noting BVS loops become a smaller fraction of first loops.

5.3 COPY_PC and JUMP Loops

As Figure 5 shows, almost all long programs get trapped in either a COPY_PC or a JUMP loop. We can approximately model the lengths of both types of loops. In both cases very short loops are predicted. We would expect, since there is less chance to disrupt memory, tight loops to be more difficult to escape.

Let M = #bits = 96, A = size of program address, D = data size = 8. Assume the chance of a loop containing i instructions = chance appropriate JUMP × (chance loop not already formed and memory not disturbed)$^{i-1}$. It is very difficult to calculate the probability of another loop forming before the one of interest. Instead we will just model the random disruption of the address stored in memory by a COPY_PC instruction. There are seven instructions, four of which write D bits and COPY_PC which writes A bits. The effect of a random update not changing overwritten data, is as if the target was shrunk to $\approx A-2$ bits. Thus the chance of a random instruction modifying the address is $(4(A+D-3)+2A-3))/7M$. Therefore the chance of a COPY_PC-JUMP loop being exactly i instructions long is $\approx \frac{1}{7M}\left(1-(6A+4D-15)/7M\right)^{i-1}$. This is a

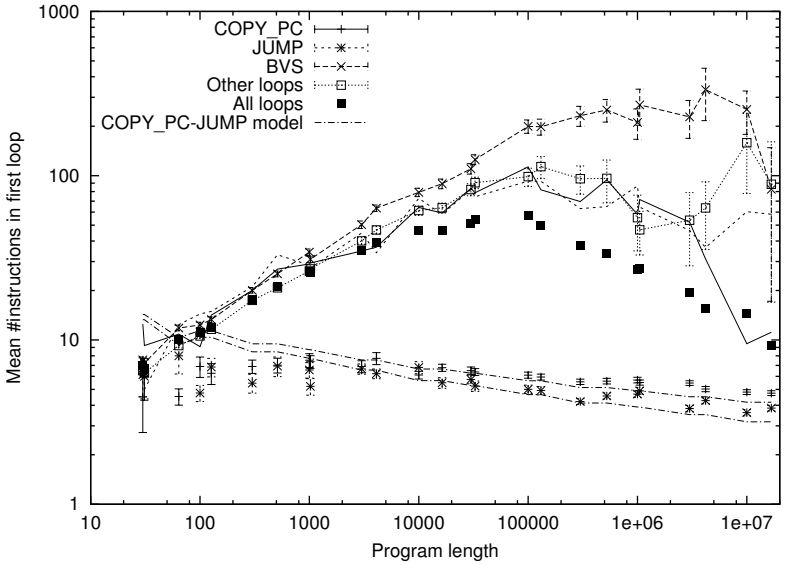

Fig. 9. Length of the first loop. Note the average (■) initially tracks "other" (□), BVS (×) etc. but as the number of these types of loop, cf. Figure 6, falls the mean begins to resemble the length of first loops created by unsullied COPY_PC (+) and JUMP (∗) instructions. The theoretical curves lie close by.

geometric distribution, with mean $7M/(6A+4D-15)$. For the longest programs $A = 24$, suggesting the mean length will be 161/672=4.17. In fact, we measure 4.74 ± 0.16, cf. Figure 9. The mean length for JUMP-JUMP loops will be one less (lower curve in Figure 9). The simple model is quite good but does not fully capture the competition between different loops. Note the vast majority of programs in the whole search space (which is dominated by long programs) fall into loops with fewer than 20 instructions.

6 Discussion

Of course the undecidability of the Halting problem has long been known. More recently work by Chaitin [15] started to consider a probabilistic information theoretic approach. However this is based on self-delimiting Turing machines (particularly the "Chaitin machines") and has lead to a non-zero value for Ω [16] and postmodern metamathematics. Our approach is firmly based on the von Neumann architecture, which for practical purposes is Turing complete. Indeed the T7 computer is similar to the linear GP area of existing Turing complete GP research.

While the numerical values we have calculated are specific to the T7, the scaling laws are general. These results are also very general in the sense that they apply to the space of all possible programs and so are applicable to both GP and any other search based automatic programming techniques.

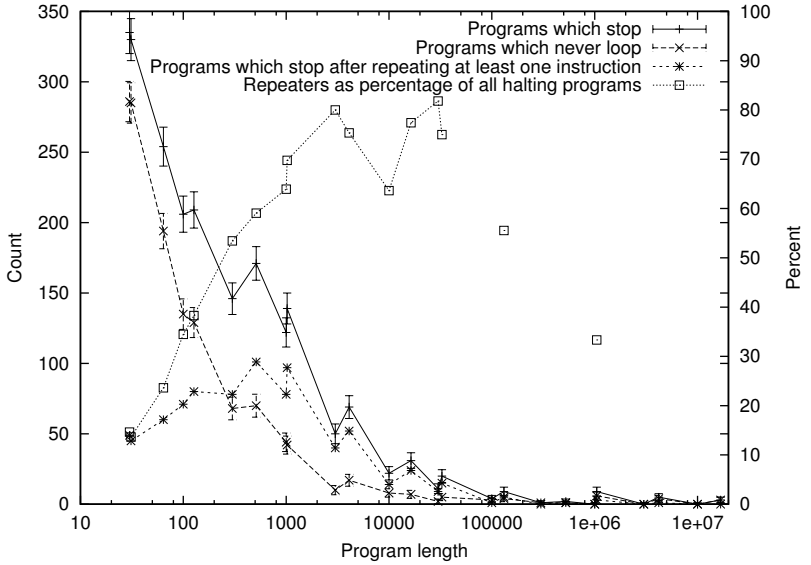

Fig. 10. Number of T7 programs (out of 1000) which escape the first loop and subsequent loops and then terminate successfully. Error bars are standard errors, indicating the measured differences ($*$) and percentages ($\Box$) become increasingly noisy with programs longer than 3000. Nevertheless the trend for terminating programs to loop but escape from the loop can be seen.

Section 4 has accurately modelled the formation of the first loops in program execution. Section 5 shows in long programs most loops are quite short but we have not yet been able to quantitatively model the programs which enter a loop and then leave it. However we can argue recursively that once the program has left a loop it is back almost where it started. That is, it has executed only a tiny fraction of the whole program, and the remainder is still random with respect to its current state. Now there may be something in the memory which makes it to easier to exit loops, or harder to form them in the first place. For example, the overflow flag not being set. However, it may also contain previous values of the program counter (PC), which would tend to make it easier to form a new loop. Also initial studies indicate the memory and flag will become nearly random almost immediately. That is having left one loop, we expect the chance of entering another to be much the same as when the program started, i.e. almost one. Thus the program will stumble from one loop to another until it gets trapped by a loop it cannot escape. As explained in Section 5, we expect, in long programs, it will not take long to find a short loop from which it is impossible to escape.

Real computer systems lose information (converting into heat) [9]. We expect this to lead to further convergence properties in programming languages with recursion and memory.

7 Conclusions

Our models and simulations of a Turing complete linear GP system based on practical von Neumann computer architectures, show that the proportion of halting programs falls towards zero with increasing program length. However there are exponentially more long programs than short ones. In absolute terms the number of halting programs increases (cf. Figure 8) but, in probabilistic terms, the Halting problem is decidable: von Neumann programs do not terminate with probability one.

In detail: the proportion of halting programs is $\approx 1/\sqrt{\text{length}}$, while the average and standard deviation of the run time of terminating programs grows as $\sqrt{\text{length}}$. This suggests a limit on run time of, say, 20 times $\sqrt{\text{length}}$ instruction cycles, will differentiate between almost all halting and non-halting T7 programs. E.g. for a real GHz machine, if a random program has been running for a single millisecond that is enough to be confident that it will never stop.

Acknowledgements We thank Dave Kowalczy and EPSRC GR/T11234/01.

References

1. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. 2002.
2. N. F. McPhee and R. Poli. Using schema theory to explore interactions of multiple operators. In W. B. Langdon, *et al.*, eds., *GECCO 2002*, pp853–860.
3. J. Rosca. A probabilistic model of size drift. In R. L. Riolo and B. Worzel, eds., *Genetic Programming Theory and Practice*, pp119–136. Kluwer, 2003.
4. K. Sastry, U-M O'Reilly, D. E. Goldberg and D. Hill. Building block supply GP. In *Genetic Programming Theory and Practice*, pp137–154. Kluwer, 2003.
5. B. Mitavskiy and J. E. Rowe. A schema-based version of Geiringer's theorem for nonlinear genetic programming with homologous crossover. In A. H. Wright, *et al.*, eds., *Foundations of Genetic Algorithms 8*, *LNCS 3469*, pp156–175. 2005.
6. J. M. Daida, A. M. Hilss, D. J. Ward, and S. L. Long. Visualizing tree structures in genetic programming. *Genetic Programming & Evolvable Machines*, 6(1):79–110
7. W. B. Langdon. Convergence rates for the distribution of program outputs. In W. B. Langdon, *et al.*, eds., *GECCO 2002*, pp812–819, New York, 9-13 July 2002.
8. W. B. Langdon. How many good programs are there? How long are they? In K. A. De Jong, *et al.*, eds., *FOGA 7*, pp183–202, Morgan Kaufmann. Published 2003.
9. W. B. Langdon. The distribution of reversible functions is Normal. In *Genetic Programming Theory and Practise*, pp173–188. Kluwer, 2003.
10. W. B. Langdon. Convergence of program fitness landscapes. In E. Cantú-Paz, *et al.*, eds., *GECCO 2003*, *LNCS 2724*, pp1702–1714. Springer-Verlag.
11. A. Teller. Turing completeness in the language of GP with indexed memory. In *1994 IEEE World Congress on Computational Intelligence*, pp136–141.
12. W. B. Langdon. Quadratic bloat in genetic programming. In D. Whitley, *et al.*, eds., *GECCO 2000*, pp451–458.
13. W. B. Langdon and R. Poli. On Turing complete T7 and MISC F-4 program fitness landscapes. Technical Report CSM-445, University of Essex, UK, 2005.
14. S. R. Maxwell III. Experiments with a coroutine model for genetic programming. In *1994 IEEE World Congress on Computational Intelligence*, pp413–417a.
15. G. J. Chaitin. An algebraic equation for the halting probability. In R. Herken, ed., *The Universal Turing Machine A Half-Century Survey*, pp279–283. OUP, 1988.
16. C. S. Calude, M. J. Dinneen and C-K Shu. Computing a glimpse of randomness. *Experimental Mathematics*, 11(3):361–370, 2002.