Revision : 1.31 To be submitted to Special Issue of Natural Computation

Mapping Non-conventional Extensions of Genetic Programming

W. B. Langdon and R. Poli

Department of Computer Science, University of Essex, UK

Abstract. Conventional genetic programming research excludes memory and iteration. We have begun an extensive analysis of the space through which GP or other unconventional AI approaches search and extend it to consider explicit program stop instructions (T8), including Markov analysis and any time models (T7). We report halting probability, run time and functionality (including entropy of binary functions) of both halting and anytime programs. Irreversible Turing complete program fitness landscapes, even with halt, scale poorly however loops lock-in variation allowing more interesting functions.

1 Introduction

Recent work on strengthening the theoretical underpinnings of genetic programming (GP) has considered how GP searches its fitness landscape [1]. 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. Few researchers allow their GP's to include iteration or recursion. Indeed there are only about 60 papers (out of 4000) where loops or recursion have been included in GP [2, Appendix B]. Without some form of looping and memory there are algorithms which cannot be represented and so GP stands no chance of evolving them.

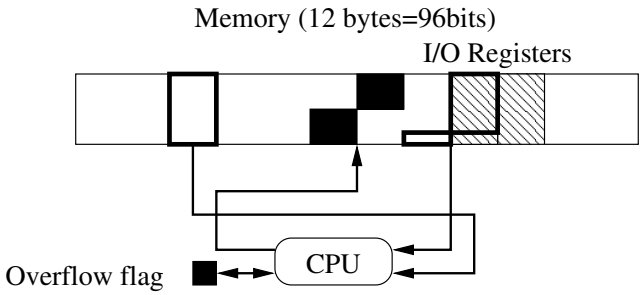

Fig. 1. T7 and T8 contain bit addressable random access memory including memory mapped input–output registers.

We have recently shown [3] in the limit of large T7 programs (cf. Figure 1):

- The T7 halting probability falls sub-linearly with program length. Our models suggest the chance of not looping falls as $O(\text{length}^{-1/2})$. Whilst including both non-looping and programs which escape loops we observe $O(\text{length}^{-1/4})$.
- Run time of terminating programs grows sub-linearly with program length. Again both mathematical and Markov models are confirmed by experiments and show run time of non-looping programs grows as $O(\text{length}^{1/2})$. Similarly, including both non-looping and programs which exit loops, for the T7, we observe run time $\leq O(\text{length}^{3/4})$.
- Despite the fraction of programs falling to zero, the shear number of programs, means the number of halting T7 programs grows exponentially with their size.
- Experimentally the types of loop and their length varies with the size of T7 program. Long programs are dominated by programs which fall into, and cannot escape from, one of two types of loop. In both cases the loops are very tight. So (in our experiments) even for the longest programs (we considered programs of up to 16 million instruction) on average no more than a few hundred different instructions are executed.

It is important to stress that these, and our previous results, apply not only to genetic programming, but to any other unconventional computation embedded in the same representation.

While the T7 computer is Turing complete, [2, Appendix A], these results are not universal. The T7 was chosen since it is a minimal Turing complete von Neumann architecture computer with strong similarities with both real computers and linear genetic programming [4]. At the 2006 Dagstuhl "Theory of Evolutionary Algorithms" [Seminar 06061] the question of the generality of the T7 was raised. In Section 4 we shall show that the impact of the addition of an explicit halt instruction is, as predicted, to dramatically change the scaling laws. With the T8 computer (T7+halt) almost all programs stop before executing more than a few instructions.

In Section 5 an approximate Markov model of the T8 is presented. We shall see its predictions compare well with experiment.

Sections 6 and 7 consider a third alterative halting technique: the any time algorithms [5]. In this regime, each program is given a fixed quantum of time and then aborted. The program's answer is read from the output register regardless of where its execution had reached. These last two experimental sections (6 and 7) consider program functionality, rather than just if they stop or not and show constants and (in these experiments) the identity function and random functions are common.

Table 1. T8 Turing Complete Instruction Set

Instruction	*Number of args*	*operation*	
ADD	3	A + B→C	v set
BVS	1	#addr→pc if v=1	
COPY	2	A→B	
LDi	2	@A→B	
STi	2	A→@B	
COPY_PC	1	pc→A	
JUMP	1	addr→pc	
HALT	0	pc→end	

Each operation has up to three arguments. These are valid addresses of memory locations. Every ADD either sets or clears the overflow bit v.
COPY_PC and JUMP use just enough bits to address each program instruction.
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 4 or 8 bits.)
JUMP addresses are moded with program length.
Programs terminate either by executing their last instruction (which must not be a jump) or by executing a HALT.

2 T7 and T8 – Example Turing Complete Computers

To test our theoretical results we need a simple Turing complete system. In [3] we introduced the T7 seven instruction CPU, itself based on the Kowalczy F-4 minimal instruction set computer `http://www.dakeng.com/misc.html`, cf. appendix of [2]. The T8 adds a single halt instruction to the T7 instruction set.

The T8 (see Figure 1 and Table 1) 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, four copy instructions and the program halt. 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.

In Section 4 eight bit byte data words are used, whilst Sections 6 and 7 both use four bit nibbles. 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 and the overflow flag.

3 Experimental Method

There are too many programs to test all of them, instead we gather representative statistics about those of a particular length by randomly sampling.

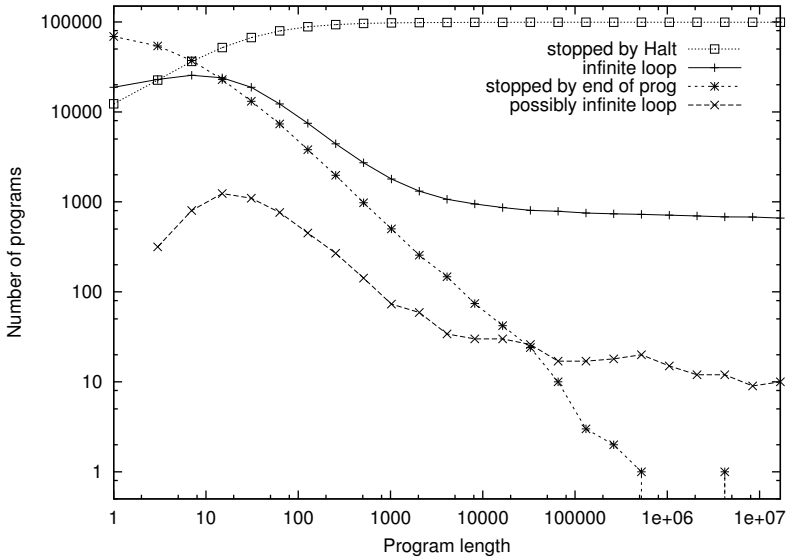

Fig. 2. Almost all short T8 programs are stopped by reaching their end (*). This proportion falls rapidly (about $\propto 1/\text{length}$) towards zero. Longer programs are mostly stopped by a HALT instruction □. The fraction of programs trapped in loops (+ and ×) appears to settle near a limit of 1 in 150.

By sampling a range of lengths we create a picture of the whole search space. Note we do not bias the sampling in favour of short programs.

One hundred thousand programs of each of various lengths (1...16 777 215 instructions) are each run from a random starting point. (NB not necessarily from the start.) Each program is given random inputs. They run until: either they execute a HALT, 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 program's 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 T8 Programs

Figure 2 shows, as expected, inclusion of the HALT instruction dramatically changes the nature of the search space. Almost all T8 programs stop, with only a small fraction looping. This is the opposite of the T7 (where most programs loop).

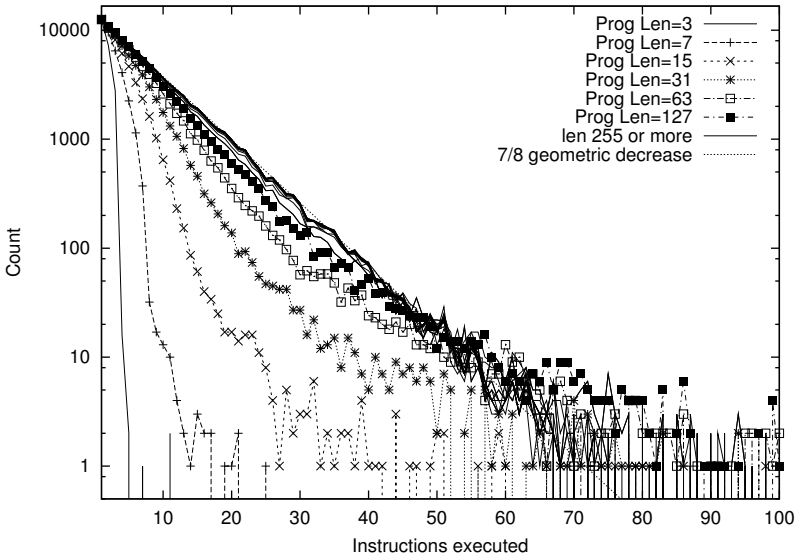

Fig. 3. Distribution of frequency of HALTed T8 programs by their run time. There is an geometric fall in frequency at all lengths, however for lengths < 127 the decay is faster than (1-1/8). For longer random T8 programs, the decay constant tends to 7/8.

Figure 3 shows the run time of T8 programs terminated by a HALT instruction. We see the fraction of programs falls exponentially fast with run time. It falls most rapidly with short programs but reaches a limit of $(7/8)^{-\text{length}}$ for longer programs. A decay rate of 7/8 would be expected if programs ran until they reach a HALT instruction. Since the distribution of run time of programs which stop by reaching their ends is also dominated by the chances of avoiding running into a HALT, they have similar distributions, even if they are finally stopped by the alternative mechanism, cf. [6, Fig. 3]. I.e. to a first approximation, run time of long terminating T8 programs can be estimated by ignoring the possibility of loops. This gives a geometric distribution and so an expected run time of 8 instructions regardless of program size. For all but very short programs, Figure 4 confirms the mean is indeed about 8. For a geometric distribution the standard deviation is 7.48 (also consistent with measurements) so almost all T8 programs terminate after executing no more than 31 instructions (mean+3σ). Again this is in sharp contrast with the T7, where long terminating T7 programs run many instruction, and so it might have been hoped would do something more useful. We shall return to the utility of random T7 programs in Section 7.

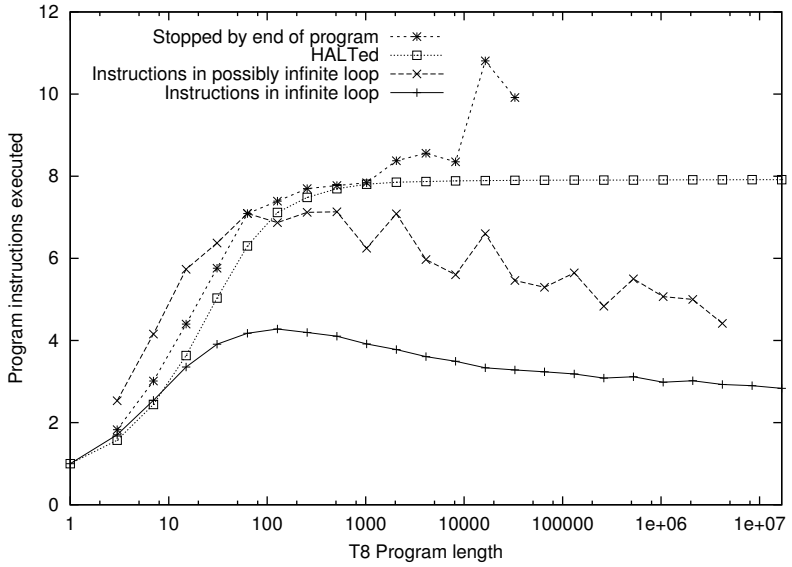

Fig. 4. Mean number of T8 instructions obeyed by terminating programs ($*$ and $\square$) and length of loops ($\times$ and $+$) v. program size. Noisy data suppressed. Data are consistent with long random T8 programs having a geometric distribution, with mean of 8 and thus standard deviation is $\sqrt{8^2 7/8} = 7.48$. Possibly due to additional competition with HALT instructions, on average, final loops in non-terminating T8 programs are even tighter than in the T7 [3, Fig. 9].

5 T8 Markov chain model

We extend our T7 model (see [7] for details) of the process of executing instructions in a program as a Markov chain to the T8. The states of the chain represent how many instructions have been visited so far (i.e., how many different values the program counter has taken) and whether or not the program has looped or halted. (Both types of T8 termination are combined into one state). The state transition matrix represents the probability of moving from one state to another.

The model starts with the system being in state 0 (no instruction has been visited). From that state, we can either go to state 1, where one instruction has been obeyed, or to the halt state. From state 1 the system can do three different things: 1) it can halt (this can happen if the current instruction is a HALT or it is the last in the program), 2) it can perform a jump and revisit the single instruction we have just executed, or 3) it can proceed, visiting a new instruction (thereby reaching state 2). Naturally, if the system halts there is no successor possible other than the halt state again. If instead the system reaches a new instruction, the process repeats: from there we can halt, revisit an old instruction or visit a new instruction.

Note that if the system revisits an instruction, this does not automatically imply that the system is trapped in a loop. However, determining the probability that a program will still be able to halt if it revisits instructions is very difficult, and, so, we will assume that all programs that revisit an instruction will not halt. For this reason, our model will provide an underestimate of the true halting probability of a program. We call this the "sink" state. Like the halt state, if the system is in the sink state at one time step it will remain in the sink state in all future time steps.

Every time step the state number is incremented (we cannot "unvisit" visited instructions), unless the system halts or revisits. There is a limit to how many times this can happen. In programs of length L, no more than L new instructions can visited. So, the states following state $i = L - 1$ can can only represent revisiting or halting.

5.1 Markov chain model: transition probabilities

To calculate the probability of moving from one state to visit a new instruction, halt or reach the sink, we need to make a number of assumptions which we can represent diagrammatically with the probability tree in Figure 5.

Apart from the fact that the T8 contains eight instructions rather than seven, the probabilities p_1–p_4 are the same in the new T8 model as for the T7. These are described in detail in [7]. The new probability, p_5, is the probability that a new next instruction is not a HALT instruction. This is simply 7/8.

The calculation of p_2 and the assumptions that underly it are described in [7]. Chief amongst these is the assumption that the destination for a jump is a randomly chosen address within the program. This seems reasonable because

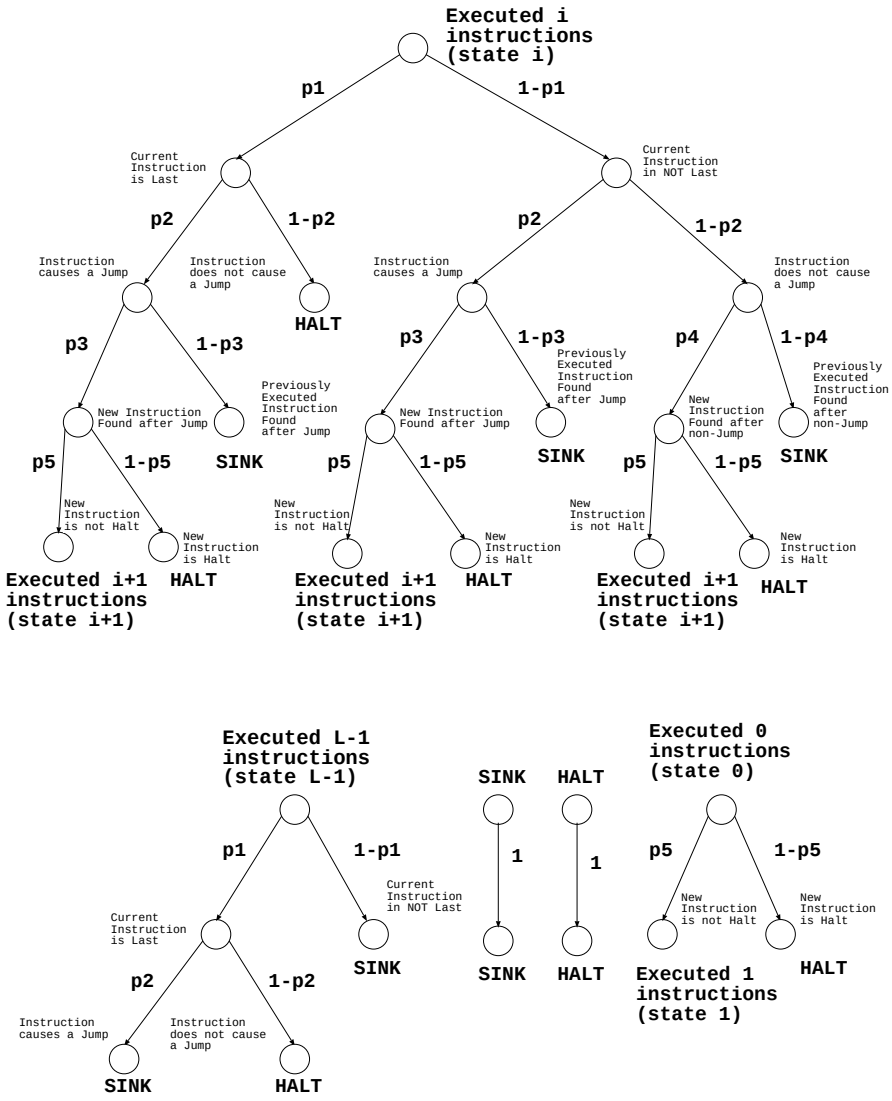

Fig. 5. Event diagram used to construct Markov chain model of T8 program execution and halting processes.

destinations taken from memory are liable to have been randomised between jumps. However where two jumps are close to each other, so there is little random overwriting of memory between them, the second has a small chance ($< 1/96$) of using exactly the same target address as the first, so forming a loop. For long T8 programs, we actually observe 1/150. It appears this has little effect in the T7, since a loop was likely to form anyway, but to have more impact in the T8, since loops are rare.

There are a number of special cases. These are shown below the main tree in Figure 5. For the T8 they are calculated using the methods described above or in [7].

5.2 Transition matrix

The above model, summarised by Figure 5, enables us to fill in all the values of the Markov transistion matrix. The Markov process for a program of L instructions has $L + 2$ states: states 0 to $L - 1$ represent the number of instructions executed so far, whilst state L is reserved for the sink and state $L + 1$ indicates the program has halted.

The Matrix has a particully simple form. Since, except for $L - 1$ and L, the state always increases, all values either above or along the diagonal are zero (except, $M_{L+1,L+1} = 1$ and $M_{L+2,L+2} = 1$). The only non-zero elements below the diagonal are either in the sub-diagonal, $M_{i,i+1}$, or in the last two rows: $M_{i,L+1}$ and $M_{i,L+2}$. This structure allow efficient implementation in C. In particular the halting probablity can be calculated in a few minutes for programs containing millions of instructions. The Markov transition matrix for programs of seven T8 instructions is:

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.875 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.729 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.677 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.611 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.518 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.362 & 0 & 0 & 0 \\
0 & 0.050 & 0.110 & 0.185 & 0.292 & 0.471 & 0.884 & 1 & 0 \\
0.125 & 0.220 & 0.213 & 0.203 & 0.190 & 0.168 & 0.116 & 0 & 1
\end{pmatrix}$$

Figures 6 and 7 show, except for the tiniest programs, there is extraordinarily good agreement between the theory and experiment.

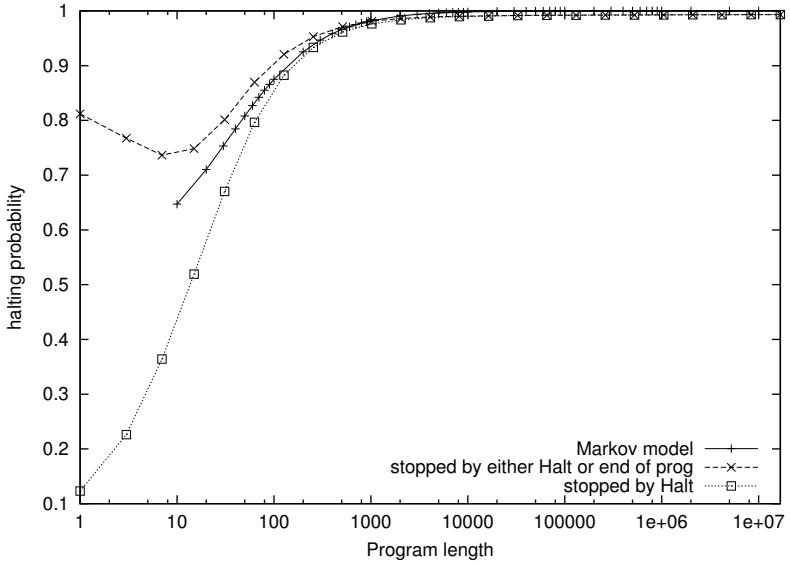

Fig. 6. Markov chain model halting probability vs. experiment for T8.

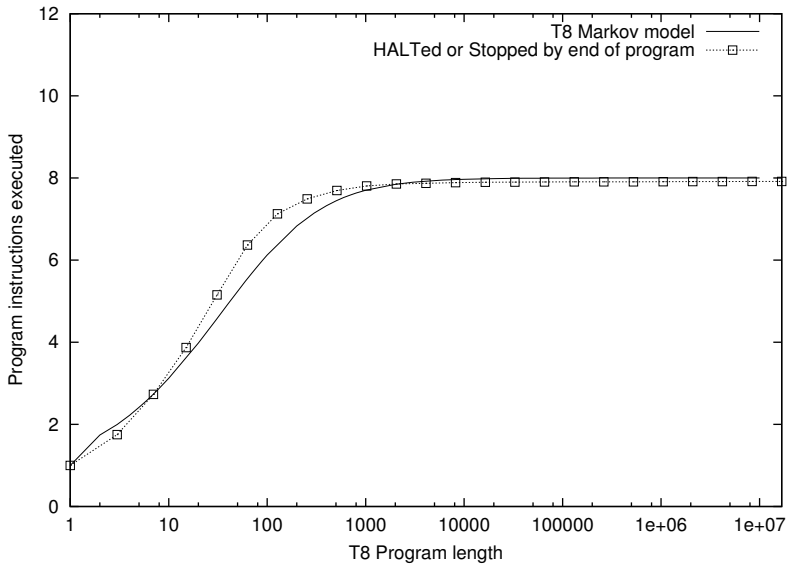

Fig. 7. Estimate of the mean number of instructions executed by halting programs computed using Markov chain model vs. experiment for T8.

6 T8 Functions and Any Time Programs

The introduction of Turing completeness into genetic programming raises the halting problem, in particular how to assign fitness to a program which may loop indefinitely [8]. Here we look at any time algorithms [5,9] implemented by T8 computers. I.e. we insist all program halt after a certain number of instructions. Then we extract an answer from the output register regardless of whether it terminated or was aborted. (The input and output registers are mapped to overlapping memory locations, which the CPU treats identically to the rest of the memory, cf. Figure 1.) We allow the T8 1000 instructions.

In this section and Section 7 we look at functions of two inputs, by defining two input registers (occupying adjacent 4 bit nibbles) and looking at the data left in memory after the program stops (or is stopped). In these sections, the data word size is 4 bits. Each random program is started from a chosen random starting point 256 times just as before, except the two input registers are given in turn each of their possible values. To avoid excessive run time and since we are now running each program 256 times (rather than once) the number of programs tested per length is reduced from 100 000 to 1000.

In addition to studying the random functions generated by the T8, we also study the variation between individual programs runs with each of the 256 different inputs and see how this varies as the program runs. We use Shannon's [10] information theoretic entropy measure $S = -\sum_k p_k \log_2(p_k)$, to quantify the differences, at a given time, between the state of the T8 on different runs. The state includes the programme counter, the overflow bit and all the memory. Remember each run has different inputs.

6.1 Any Time T8 Entropy

Figure 8 shows the average number of test cases still running. The shortest programs tend to stop or loop immediately. Short programs which loop on one test case tend to loop on all of them, giving, within a few instructions, the almost constant plots for short programs seen in Figure 8. Longer random programs, tend to run for slightly longer and have more variation between the number of instructions they execute on the different test cases. Figure 9 shows there is a corresponding behaviour in terms of variation between the same program given different inputs. I.e., loops are needed to keep small programs running and compact loops mean small programs tend to keep their variability. This gives the almost constant high entropy plots for short programs seen in Figure 9. However longer random programs tend to run for longer and use more instructions. More random instructions actually means that the memory etc. tends to behave the same on every input and this convergence increases as the programs run for more time. Indeed there is also less variation in average behaviour with longer random T8 programs. Leading to the general

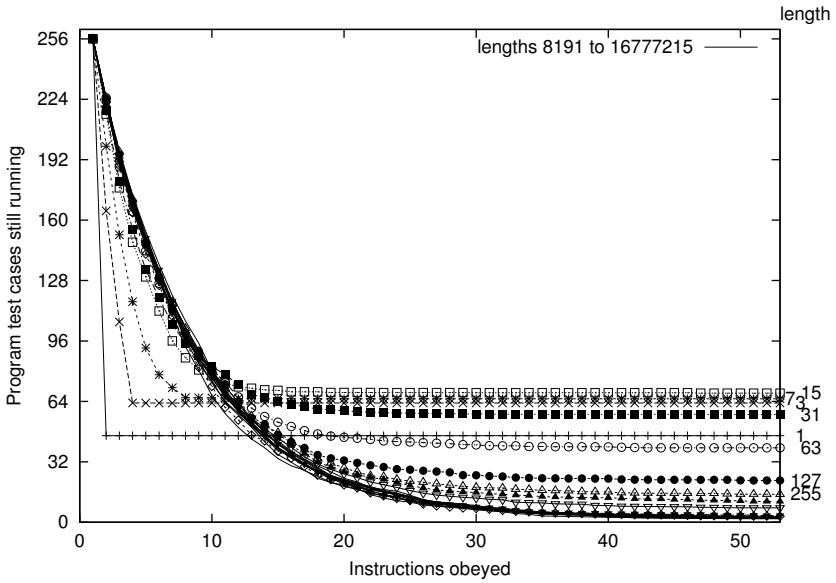

Fig. 8. Mean number of test cases where T8 program has not HALTed or stopped. 1000 random T8 programs of different lengths. (Plots continue out to 1000 instructions are almost flat.)

decrease in entropy with run time seen in Figure 9.[1] In the next section we will restrict ourselves to just looking at the I/O registers rather than the whole of memory, that is the notion of programs as implementing functions which map from inputs to output. However we shall see the two views: entropy and functionality, are consistent.

6.2 Any Time T8 Convergence of Functions

There are $256^{2^4 \times 2^4} = 3.23\ 10^{616}$ possible functions of two 4 bit inputs and an 8 bit output. However, as shown by Figure 10, uniformly chosen random programs of a given length sample these functions very unequally. (This is also true of the T7, cf. Figures 19 and 20). In particular the identity function and the 256 functions which return constants are much more likely than others. Figure 10 also plots two variations on the identity (where the least significant or most significant nibble implement a 4 bit identity function) and two cases of 4 bit constants. In these four cases the other 4 bits are free to vary. Note that while they represent a huge number of functions they are less frequent than either of their 8 bit namesakes.

[1] In Figure 9 we include the entropy of those T8 programs which have stopped or HALTed leading to higher values than we reported in [6, Fig. 8] since there we combined them into a single state.

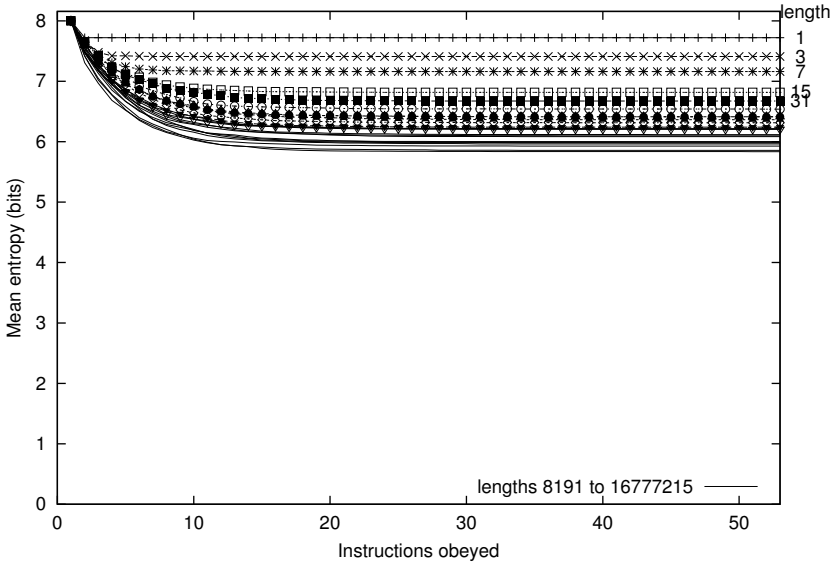

Fig. 9. Evolution of variation between test cases in 1000 random T8 programs of different lengths. Same programs as in Figure 8.

We define a function's entropy by ignoring its inputs and just considering the numbers of each different output. Identity has unique outputs and so its entropy is maximal (8 bits). Whilst a constant function has a single output value and so has zero entropy. The mean entropy of the $3.23\ 10^{616}$ possible 8-bit functions is 7.17. The functions whose entropy is near the average are plotted in Figures 10 and 10 as "random" functions.

The rise in the frequency of random functions and fall in the identity function with increasing program size (Figure 10) are consistent with the corresponding fall in entropy seen in Figure 9. (The same will be seen in the next section for the T7, cf. Figures 19 and 13.)

7 T7 Functions and Any Time Programs

Having shown the success of the any time approach, we return to the T7. The measurements in this section are based on running the T7 on 256 test cases as in Section 6.

Figure 11 confirms removing HALT does indeed mean most long programs run up to the any time limit. Figure 11 relates to all 256 runs of each random program. In short programs the distribution of run times versus input is bi-modal. One peak corresponding to T7 programs quickly reaching their end and stopping on all inputs and the other to looping until the any time limit, again on all 256 inputs. As programs get longer, the peak at the maximum

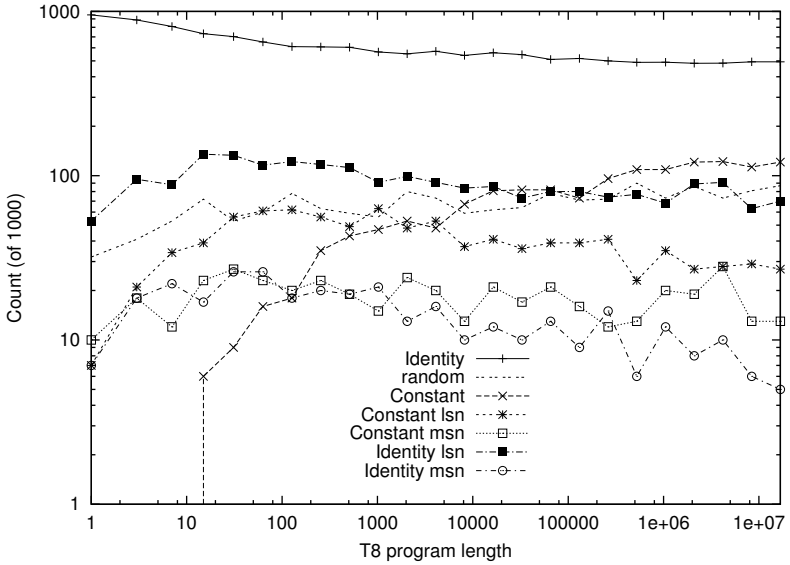

Fig. 10. Frequency of common functions implemented by random T8 programs of different lengths by 1000 instruction cycles. Data are noisy due to sample size, but the trend to falling proportion of functions, except the constant and random fuctions, with program size can be seen.

allowed time increases at the expense of the peak near one instruction. However the distribution also spreads, with more runs executing between 1 and 1000 instructions. However, the fraction of looping programs also increases, so that with even longer programs, the peak at 1000 becomes even stronger and the spread in run time falls.[2] Only at intermediate lengths can the input switch random programs looping/halting behaviour.

The diagonal line shows that, for program shorter than 4 000 000, the fraction of runs which stop falls approximately as $1/\sqrt{\text{length}}$, as expected. (For even longer T7 programs, the 1000 instruction limit aborts a few programs even though they are not stuck in loops.)

Figure 12 shows that the fraction of programs which never loop, falls as $O(1/\sqrt{\text{length}})$, as we found previously [3]. Figure 12 plots combined behaviour over 256 test cases rather than a single run and the data word size is half that reported in [3]. However, initially we have similar results: the fraction of T7 programs which do not loop falls with program length. Notice that for longer programs the fraction does not continue towards zero. This is because we now use the any time approach to abort non-terminated programs and so a few programs (19–47 out of 1000) are stopped early, when they might have continued to find themselves in loops.

[2] It is this increase–fall-increase in the spread of run time which gives rise to the curious non-monotonic curve in the average *minimum* run time seen in Figure 11.

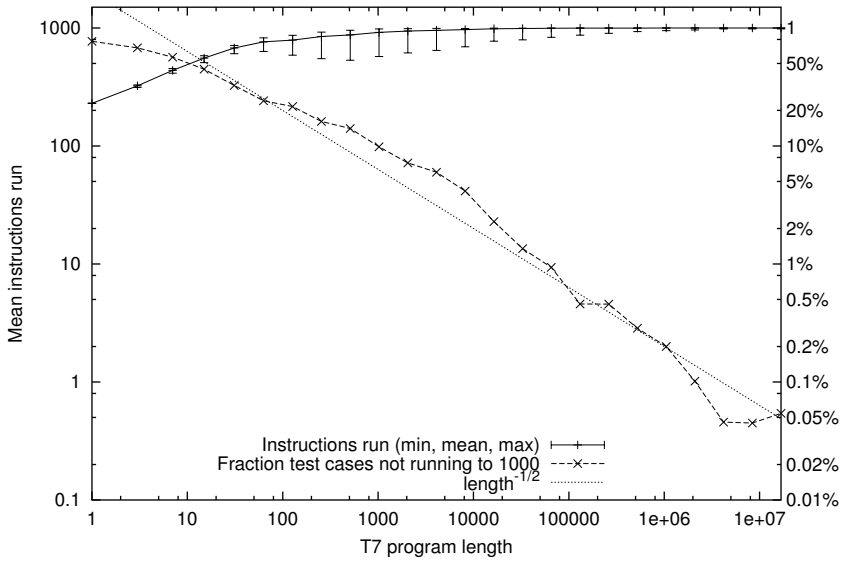

Fig. 11. Run time on 256 test cases of 1000 random T7 programs of each of a variety of lengths. Some short programs run to their end and stop but almost all longer programs run up to the limit of 1000 instructions on all test cases.

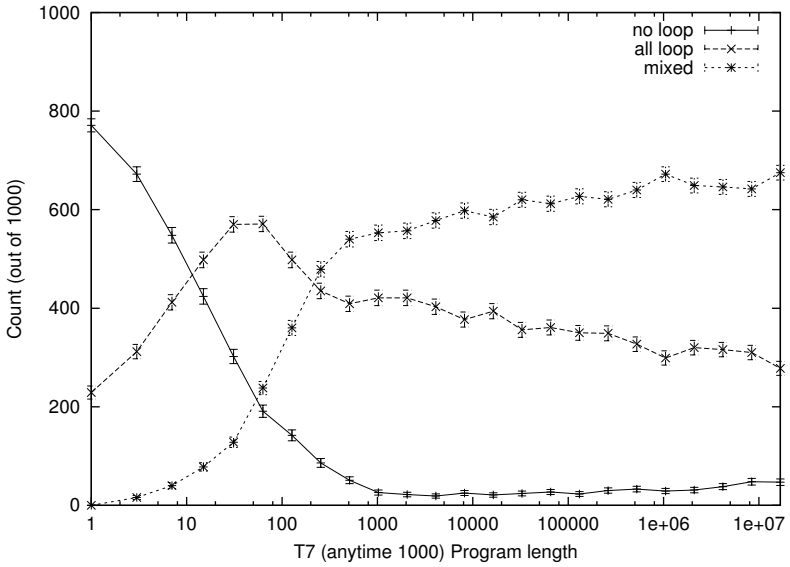

Fig. 12. Looping characteristics of Random T7 programs on 256 test cases. All programs still running are aborted after 1000 instructions. Error bars show standard error.

7.1 T7 Convergence

Before looking at the functionality of random T7 programs (Section 7.3) we will report variation in both the T7's memory (including the overflow bit) and routes taken though the program (Section 7.2). At the start of each program, the memory is initialised to be in one of 256 states and the program counter (PC) is set to one randomly chosen value. Therefore the entropy is always exactly 8 bits across the 256 test cases. It cannot increase. Figure 13 shows on average it falls rapidly in the first few instructions. That is, after a handfull of instructions, many test cases cause the T7 to enter identical configurations. Since the T7 is deterministic, once a pair of test cases are syncronised in this way at the next time stop they will both update both their memory's and their PC's in the same way and so enter the same state. I.e. once syncronised they remain syncronised. Hence variability across the test cases can only fall. We see this in Figure 13, however note the initial steep fall in entropy, depending on program length, quickly bottoms out, and no further convergence occurs.

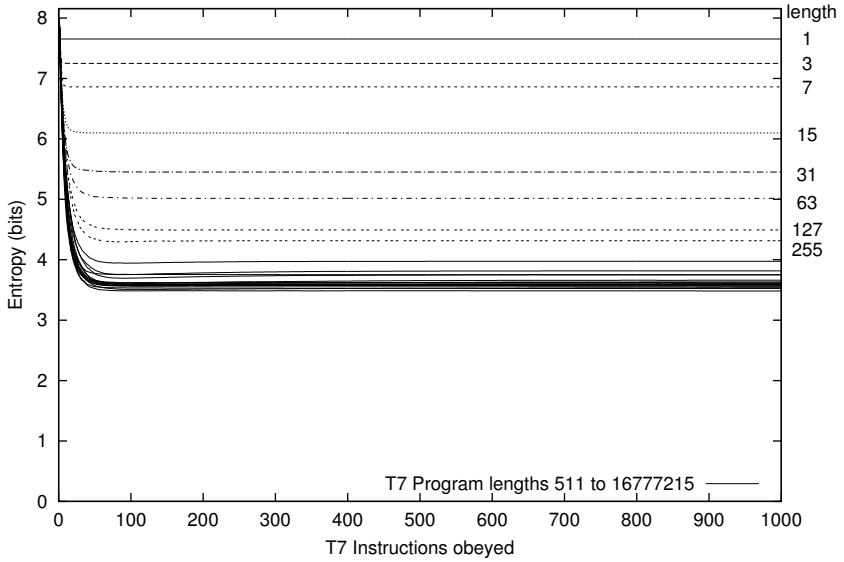

Fig. 13. Average reduction in variation between test cases for 1000 random T7 programs of each of a variety of lengths run on 256 test cases, up to 1000 steps. The smooth averages conceal strong peaks in the data at 0 (constants), 1, 2, 3, 4, 5 and 8 bits (e.g. identity). Cf. Figure 15.

The final state of the T7 includes its output register and hence the mapping from input to output. I.e. the functionality of the program. If program implements the identity function (or a permutation) then the output register for each test case is different, and therefore so too most be the state of the

whole machine, and the entropy will be 8 bits. Note the variation of the output register cannot exceed that of the whole machine. So a final entropy of 4 bits implies the entropy of the function must be 4 bits or less.

Consider the following example. Suppose there have been no conditional branches and that the input has not been saved elsewhere then if the third instrauction overwrite the input register with 27, this will immediately remove all variation between the test cases and entropy will fall to zero. In this way a random sequence of writes to memory progressively removes information about how the program started. However this is fairly slow [11]. Figure 13 shows in most cases entropy remains well above zero.

Figure 14 shows, if we exclude programs suspected to be infinite loops, then for longer programs the variation between test cases does indeed fall to near zero. The long flat times in Figures 13 and 14, lead us to suggest that in most cases, T7 programs find themselves in tight loops containing few instructions, which after a few iterations have destroyed all the information they can. Repeated execution of the same instructions (upto the anytime limit) cannot destroy any more information and the machine cycles through the same small set of states endlessly. We suggest non-looping programs, on average, execute more random instructions and so entropy falls to lower levels. Longer, non-looping, T7 programs execute tend to execute more random instructions before terminating and so tend to have lower entropy after 1000 instructions. Suggesting a greater number of uninteresting functions.

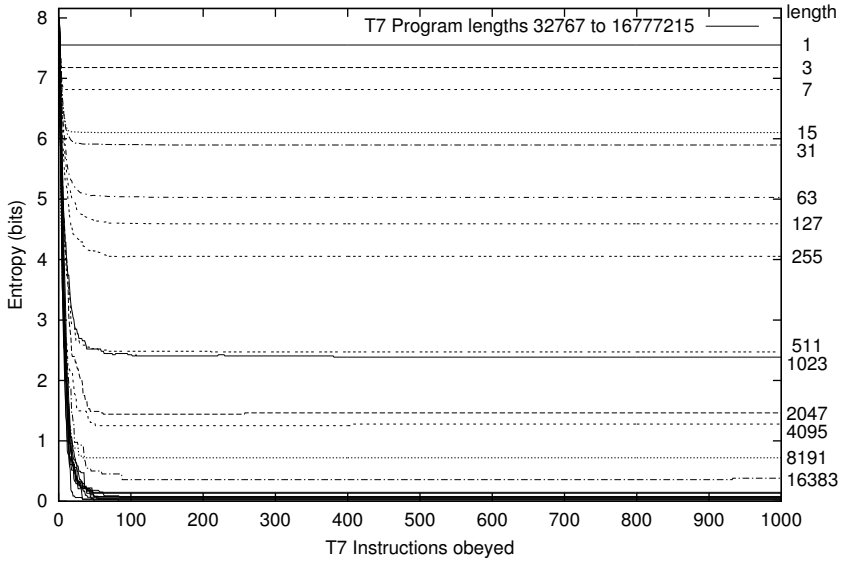

Fig. 14. As Figure 13 except we include only runs which did not loop infinitely on any test case. NB for simplicity we extend plots to 1000, even if all runs stopped before 1000.

Figure 15 shows there is a wide distribution of entropy in random T7 programs at all lengths. With strong clusters at integer values. The center (median) of the distribution shows similar values to the mean (cf. Figure 13). One would expect to find similar clusters in the entropy of random T7 functions.

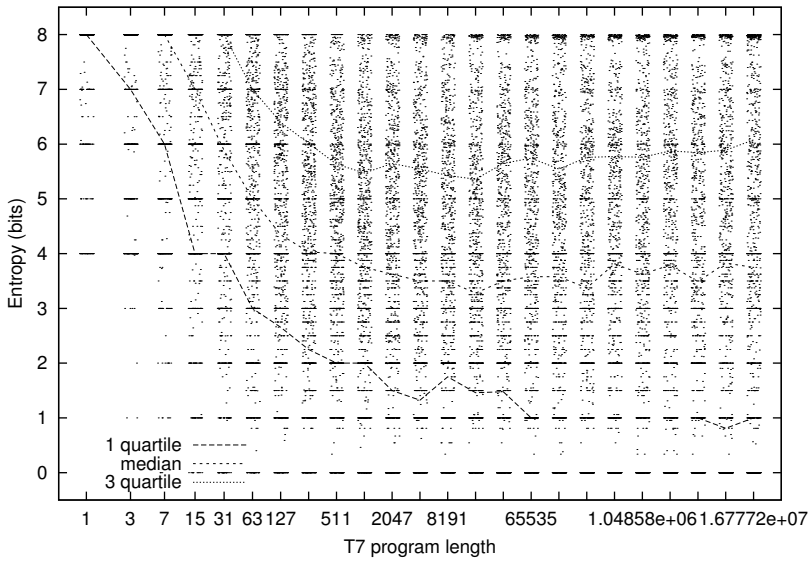

Fig. 15. Variation of T7 machine across 256 test cases at end of 1000 random T7 programs of a variety of lengths. (Anytime limit=1000).

7.2 T7 Convergence of program paths

One of the ways used to verify programs in practise is to run them multiple times with different inputs. The goal being to execute every path (and hence instruction) in the program. Correct operation in these tests giving reassurance that the program is correct. In Figure 16 we consider running all test cases in parallel and at each time step looking at the variation in the location of the PC in the program. (Again we use entropy to measure variation.) All runs start at the same point, so PC entropy is zero. Unlike total entropy, PC entropy can increase and fall. E.g. a branch conditional on the input will cause the program to take two different routes on different test cases, leading to an increase in entropy. While two divergent paths can come together. E.g. two simultaneous jumps to the same target. However we seldom see this and PC entropy rarely falls.

Figure 17 shows that distribution of the variation in routes through random T7 programs. Again the centre of the distribution is near the mean and

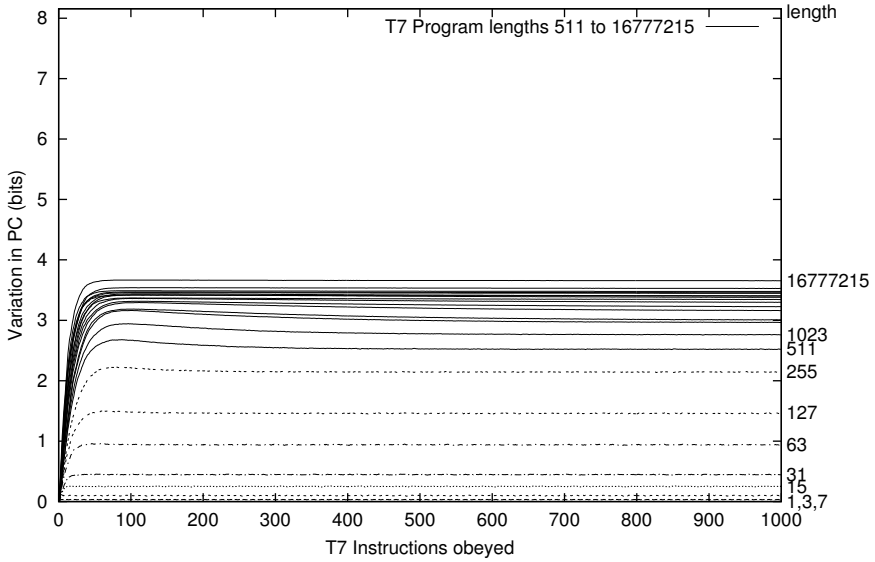

Fig. 16. Mean variation in program counter across 256 test cases whilst program runs. As Figure 13 but we plot only the program counter's contribution to entropy.

there are large clusters at integer values. NB less than a quarter of large programs execute the same final instruction on all 256 test cases even though they are looping. This could mean they are in the same loop but have some how become out of phase or that the test cases get trapped in completely separate loops.

Assuming PCs mainly become desynchronised by programs running along different paths[3], then we suggest randomly running long T7 program seldom forces them to execute different paths. Running the average program 256 times, gives a PC entropy of 4 bits. (16 different values out of 256 gives 4 bits.) In other words, we would expect to have to run a typical T7 program 16 additional times to force it to test a new path.

The relationship between complete variation and PC variation for any time T7 programs of length 1023 is plotted in Figure 18. As we have seen, shorter programs usually finish at the same point (i.e. PC entropy 0) regardless of the accumulated differences in their memory. While long program finish points have greater variation and this is highly correlated with the total variability of both memory and PC.

Surprisingly, if we exclude loops, at all lengths, there is very little difference between test cases of the route taken through random T7 programs. This might be because the conditional branches needed to cause program paths to diverge, also often lead to loop formation.

[3] Rather than executing the same instructions at different times.

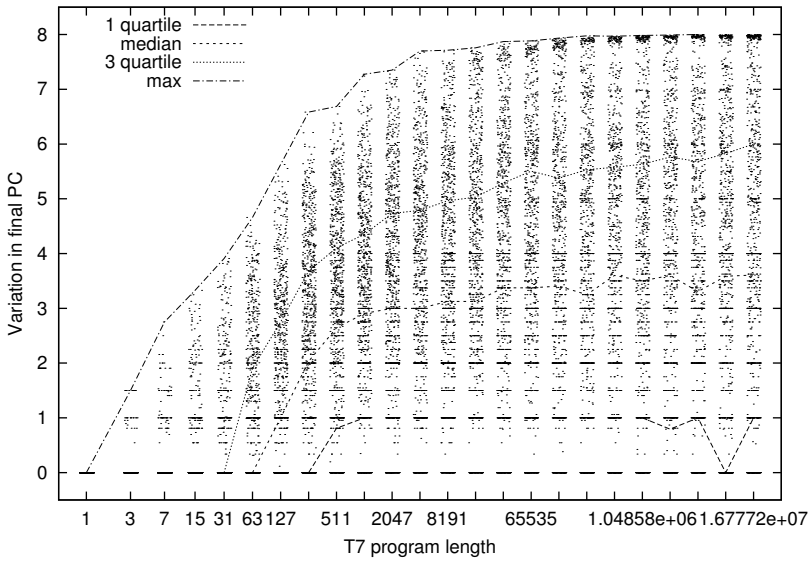

Fig. 17. Variation of in program counter (measured as entropy) across 256 test cases at end of 1000 random T7 programs. (Anytime limit=1000).

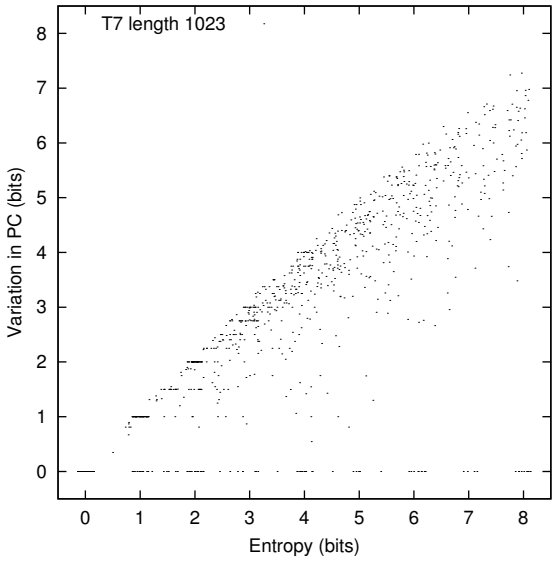

Fig. 18. Relationship between complete variation in T7 v. variation in only the PC. Shorter programs tend to cluster at (8,0) whilst longer ones lie even more tightly to the $x = y$ diagonal. (Anytime limit=1000, noise added to spread horizontally points).

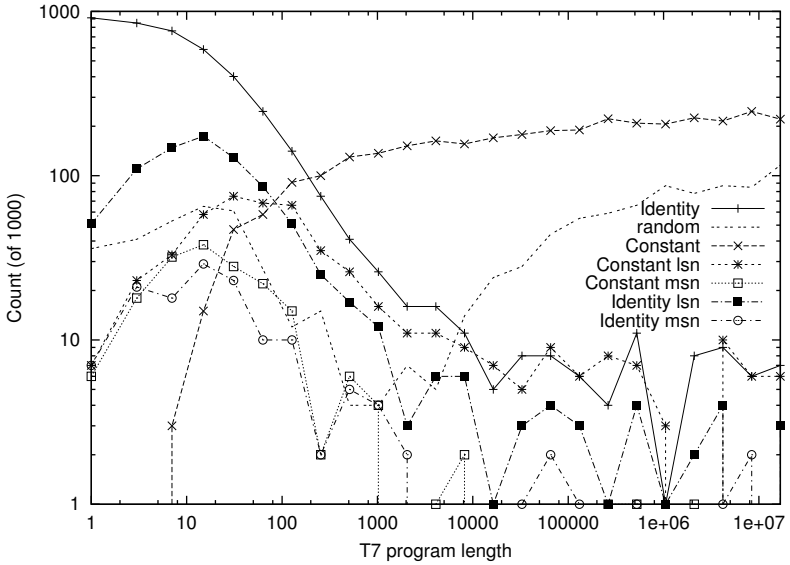

Fig. 19. Variation of some common functions implemented by T7 with program length. Note, except for the 256 constant functions and the random functions, after 1000 instructions most of the functions considered in Figure 10 become rare

7.3 Any Time T7 Convergence of Functions

As expected, when we allow the T7 to run for longer the variation between test cases reduces and there is an increased tendency for programs to become independent of their inputs. If a program's output does not depend on its input, i.e. all 256 test cases yield the same answer, then it effectively returns a constant. In Figure 19 the constant functions ($\times$) are those where the program's output (after up to 1000 instructions) does not depend upon its inputs. Notice the rise in the proportion of constants with program length, even though, in most cases, each program runs exactly 256×1000 instructions.

The variability of the output register cannot exceed the variability of the whole of memory. Therefore the variability of a program's function (again defined using entropy) cannot exceed the entropy of the whole of the T7. For short random program the function entropy can be much less, but for long programs they are often identical. Figure 21 compares them for intermediate length random T7 any time programs. Short programs tend to follow the same path on each test case yet, since memory has not yet been scrambled, yield more variable functions. (I.e. PC entropy 0. Cf. Figure 22.) However long programs follow more diverse paths, leading to higher PC entropy, and all three entropies (total, PC and function) being highly correlated. To a first approximation, we can say they are the same but (due to outputting repeated values) the entropy of random 8-bit functions is about 7.17 (rather than 8).

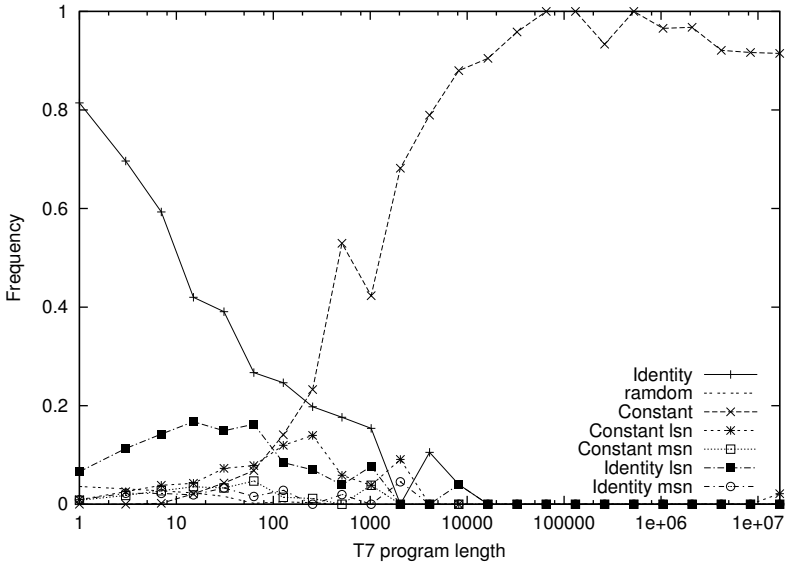

Fig. 20. Variation of some common functions implemented by T7 with program length. As Figure 19 except we include only programs which are not suspected of looping indefinitely on any test case.

Therefore long random programs with high entropy implement functions of slightly less entropy than their total or PC entropy.

In [3] we found that longer T7 programs tend to obey more random instructions (about $\sqrt{\text{length}}$) before they get stuck in tight loops. We suggest that the rise in constants with program length in Figure 19, is due to the greater loss of information in longer random sequence of non-looping instructions before a tight loop is entered. Also, the loss of knowledge about the input registers in the final loop is usually either small or repeating the same instructions many times does not loose any more information.

This explanation is reinforced if we look at only the programs which did not loop. Figure 20 shows the rise in the proportion of constants with program length is even more pronounced. Whilst Figure 14 shows, if loops are excluded, in most cases variation between different input values falls rapidly to zero. Figures 13 and 14 also show test cases become more similar as the programs run. However programs which become locked into loops have less of a tendency to converge than those which are not looping. I.e. loops actually lock in variation and without them random programs are dissipative and so implement only the 2^n constant functions.

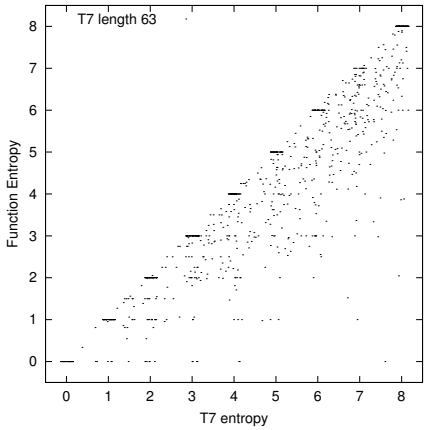

Fig. 21. Comparison of the entropy of functions implemented by 63 T7 instructions with total entropy.

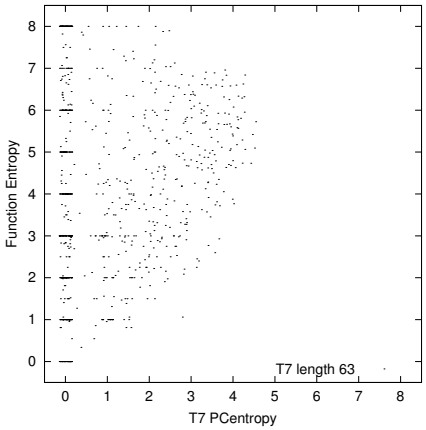

Fig. 22. As left, except compare with variability of last PC. (Anytime limit=1000. Noise added to spread x)

8 Discussion

Of course the undecidability of the Halting problem has long been known, however it appears to have become an excuse for not looking at unconventional approaches to evolve more powerful than O(1) functions. More recently work by Chaitin [12] 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 [13] and postmodern metamathematics. The special self-delimiting approach means halting programs cannot be extended and so each blocks out an exponentially large part of the search space. This can give very different statistics for the whole space. Our approach is firmly based on the von Neumann architecture, which for practical purposes is Turing complete. Indeed the T7 is similar to the linear genetic programming area of existing Turing complete genetic programming research.

Real computer systems lose information. We had expected this to lead to further convergence properties in programming languages with iteration and memory. However these results hint at strong differences between looping and non-looping programs. It appears that many tight loops are non-dissipative, in the sense that they cycle the computer through the same sequence of states indefinitely. In contrast, non-looping programs continue to explore the computer's state space but in doing so they become disconnected from where they started, in that they arrive at the same state regardless of where they started. This means they are useless, since they implement a constant.

Requiring input and output to be via fixed width registers is limiting. Variable sized I/O (cf. Turing tapes) is needed in general. Real CPUs achieve this by multiplexing their use of I/O registers. May be this too can be modelled.

It may be possible to obtain further results for the space of von Neumann architecture computer programs by separating the initial execution from looping. These initial experiments suggest the program path (i.e. conditional branches, jumps, etc.) of the program is initially not so important and that our earlier models on linear programs might be relevant. If, in other machines, most loops are also drawn from only a small number of types (in the case of T7 only two) it may be possible to build small predictive models of loop formation and execution.

8.1 Implications for Genetic Programming Research

These results help to characterise the search space explored by GP systems operating at the level of machine code. From earlier research we know that for those programs that terminate, as the number of instructions executed grows, their functionality approaches a limiting distribution. However it is only by computing the expected number of instructions actually executed by halting programs that we can assess this. For T7, for example, one can see that very long programs have a tiny subset of their instructions executed (e.g., of the order of 1,000 instructions in programs of $L = 1,000,000$). Therefore we think of the average number of instrauctions used as the *effective size* of programs of a particular length.

The introduction of Turing completeness into GP raises the problem of how to assign fitness to a program which may loop indefinitely [8]. Often, from a GP point of view, those programs that do not terminate are wasted fitness evaluations and they are given zero fitness. So, if the halting probability is very low, the fraction of the population that is really contributing to the search can be very small, thereby reducing GP ability to solve problems. Our theoretical model allows us to predict the effective population size for the initial generation. I.e. the number of non-zero fitness individuals within it. Since the initial population is composed of random programs only a fraction $p(\text{halt})$ are expected to halt and so have fitness greater than zero. We can use this to improve the size of the population or to put in place measures which ensure that a larger fraction of the programs in the population do terminate. This is particularly important because if not enough programs in the initial generation terminate, evolution may not even start.

In [7] we provided an efficient detailed Markov chain model for the execution and halting for T7 programs. In Section 5 we were able to support our earlier claim that it is a general model by applying it to another machine, specifically the T8.

9 Conclusions

Although these results are of a theoretical nature and aim at understanding the structure of the search space of computer programs, we feel this is a fundamental step for understanding and predicting the behaviour of systems that explore such a search space, such as a genetic programming system. Indeed, we were able to show that there are very clear implications of this research from the point of view of GP practice. For example, in [7] we provided recipes to ensure that enough programs in GP system actually terminate, recipes for halting non-terminating programs and recipes for assessing the run-time requirements of GP runs with Turing-complete assembly languages.

The Markov approach allows us to accurately estimate the halting probability and the number of instructions executed by programs that halt for programs including millions of instructions in just a few minutes.

Only a tiny fraction of the whole program is used. The rest has absolutely no effect. In future it may be possible to derive bounds on the effectiveness of testing (w.r.t. ISO 9001 requirements) based on code coverage.

The introduction of an explicit HALT instruction leads to almost all programs stopping. The geometric distribution gives an expected run time of the inverse of the frequency with which the HALT is used. This gives, in these experiments, very short run times and few interesting programs.

We also explored the any time approach, looking particularly at common functions and information theoretic measures of running programs. Entropy clearly illustrates a difference between non-dissipative looping programs and dissipative non-looping programs. There is some evidence that large random non-looping programs converge on the constant functions, however, possibly due to the size of the available memory, this is not as clear as we expected. This needs further investigation. We anticipate that detailed mathematical and Markov models could be applied to both the T7 and T8 any time approaches.

While genetic programming is perhaps the most advanced automatic programming technique, we have been analysing the fundamentals questions concerning the nature of programming search spaces. Therefore these results apply to any form of unconventional computing technique using this or similar representations which seeks to use search to create programs.

Acknowledgements We would like to thank Dagstuhl Seminar 06061. Funded by EPSRC GR/T11234/01.

References

1. W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
2. W. B. Langdon and R. Poli. On Turing complete T7 and MISC F-4 program fitness landscapes. Technical Report CSM-445, Computer Science, University of Essex, UK, December 2005.

3. W. B. Langdon and R. Poli. The halting probability in von Neumann architectures. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 225–237, Budapest, Hungary, 10 - 12 April 2006. Springer.
4. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann, 1998.
5. 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.
6. W. 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.
7. Riccardo Poli and William B. Langdon. Efficient markov chain model of machine code program execution and halting. In Rick L. Riolo, Terence Soule, and Bill Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 13. Springer, Ann Arbor, 11-13 May 2006.
8. Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.
9. Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In Hans-Georg Beyer, *et al.*, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.
10. Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, 1964.
11. W. 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 2002. Morgan Kaufmann. Published 2003.
12. Gregory J. Chaitin. An algebraic equation for the halting probability. In Rolf Herken, editor, *The Universal Turing Machine A Half-Century Survey*, pages 279–283. Oxford University Press, 1988.
13. Cristian S. Calude, Michael J. Dinneen, and Chi-Kou Shu. Computing a glimpse of randomness. *Experimental Mathematics*, 11(3):361–370, 2002.