Unconventional Computation UC'06 York, 4-8 September 2006,

Mapping Non-conventional Extensions of Genetic Programming

W. B. Langdon

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) and any time models (T7). We report halting probability, run time and functionality (including entropy of binary functions) of both halting and anytime programs. Turing complete program fitness landscapes, even with halt, scale poorly.

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.

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

– The T7 halting probability falls sub-linearly with program length. Our models suggest the chance of not looping falls as O(length$^{-1/2}$). Whilst including both non-looping and programs which escape loops we observe O(length$^{-1/4}$).

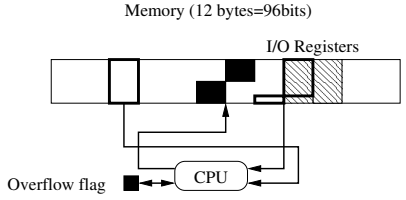

Fig. 1. T7 and T8 have the same bit addressable memory and input–output

– 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.

Sections 5 and 6 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 (5 and 6) consider program functionality, rather than just if they stop or not.

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.

Table 1. T8 Turing Complete Instruction Set

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. (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.

Instruction	*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

In Section 4 eight bit byte data words are used, whilst Sections 5 and 6 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 (plus 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. 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 ($1 \ldots 16\,777\,215$ instructions) are each run from a random starting point (NB not necessarily from the start) with random inputs, 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 (most programs loop).

Figures 3 and 4 show the run time of terminating T8 programs. In both programs stopped by reaching their end (Figure 3) and by a HALT instruction (Figure 4), the fraction of programs falls exponentially fast with run time. In both cases, it falls most rapidly with short programs and appears to reach a limit of $(7/8)^{-\text{length}}$ for longer programs. A decay rate of 7/8 would be expected if

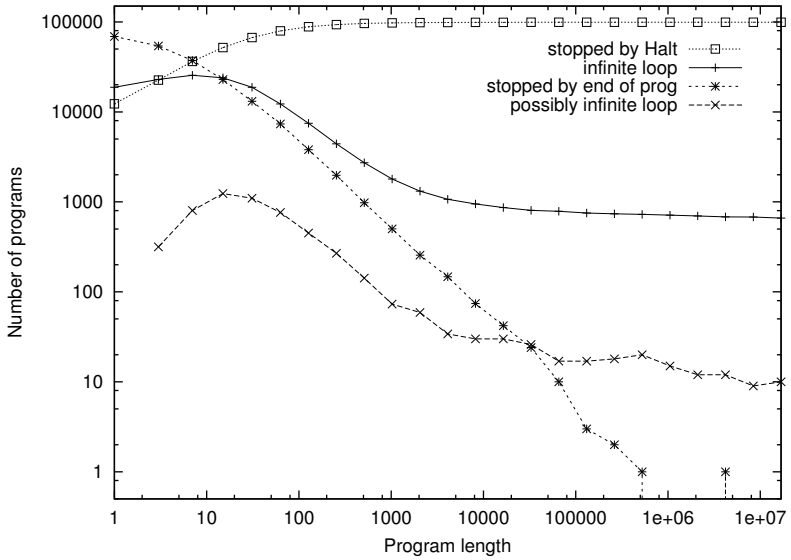

Fig. 2. Almost all short T8 programs are stopped by reaching their end (*). This proportion falls rapidly (about $\propto 1/\text{length}$) towards zero. Longer programs are mostly stopped by a halt instruction □. The fraction of programs trapped in loops (+ and ×) appears to settle near a limit of 1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

programs ran until they reach a HALT instruction. 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 5 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 perhaps may do something more useful.

5 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 [6]. Here we look at any time algorithms [5] 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 53 instructions. (53 was chosen since by then we expect all but 0.1% of non-looping T8 programs to have stopped, cf. Figure 4.)

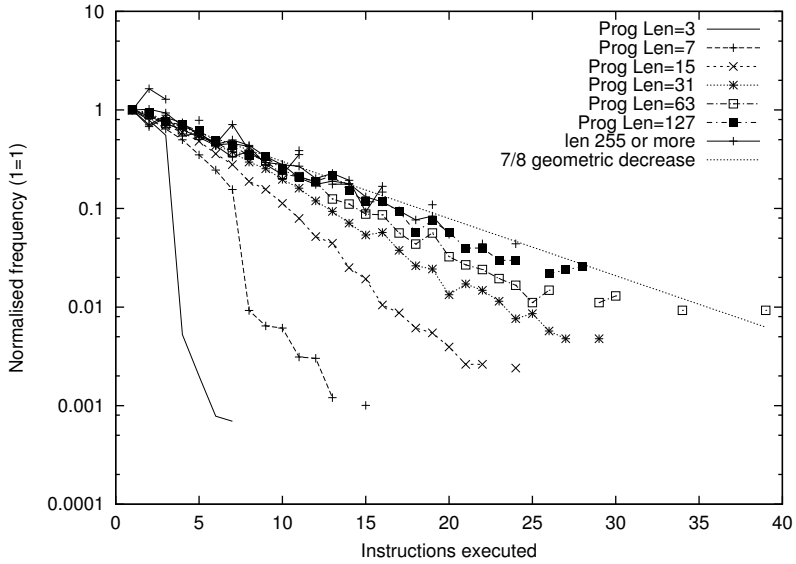

Fig. 3. Distribution of frequency of normally stopping T8 programs by their run time. Noisy data suppressed. To ease comparison, all data has been vertically rescaled so that data at the left lie on top of each other. Distributions fit geometric decay. As program gets longer the decay constant tends to 7/8.

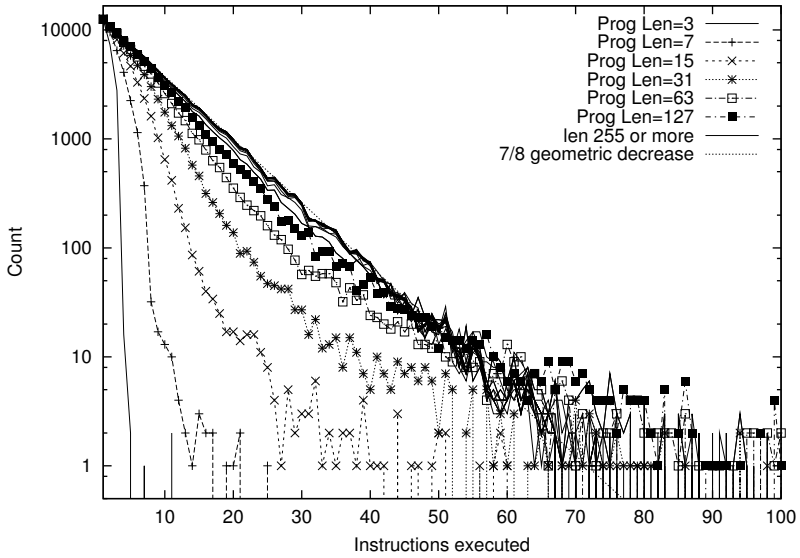

Fig. 4. Distribution of frequency of HALTed T8 programs by their run time. There is an geometric fall in frequency at all lengths, however for lengths < 127 the decay is faster than (1-1/8). For longer random T8 programs, the decay constant tends to 7/8.

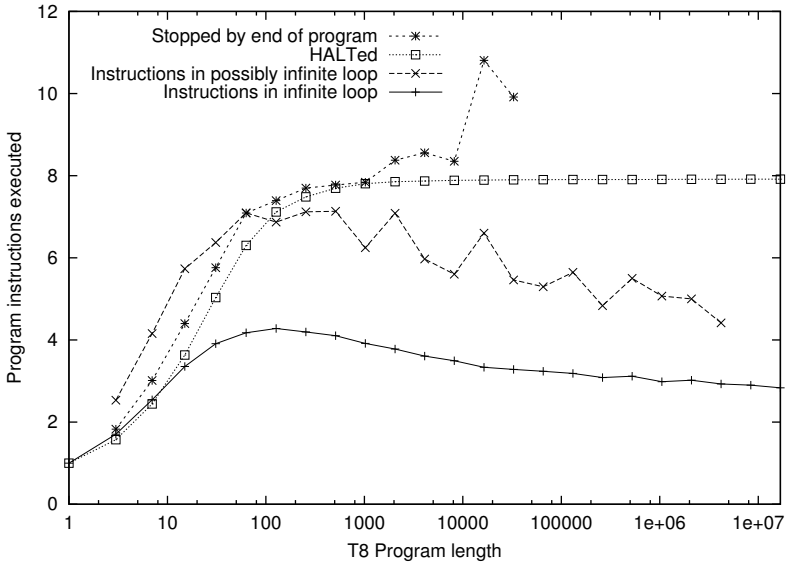

Fig. 5. Mean number of T8 instructions obeyed by terminating programs ($*$ and $\square$) and length of loops ($\times$ and $+$) v. program size. Noisy data suppressed. Data are consistent with long random T8 programs having a geometric distribution, with mean of 8 and thus standard deviation is $\sqrt{8^2 7/8} = 7.48$. Final loops in non-terminating programs are even tighter than in T7 [3, Fig. 9].

In this section and Section 6 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 how this varies as the program runs. We use Shannon's [7] information theoretic entropy measure $S = -\sum_k p_k log_2(p_k)$, to quantify the difference between the state of the T8 (programme counter, overflow bit and memory) on different runs, with different inputs, at the same time.

5.1 Any Time T8 Entropy

We have two confounding effects. We measure the change in the variation between T8 processors as they run the same program (from different inputs), so a major influence is whether a particular processor is still running or has obeyed a HALT instruction or reached the end of its program. For simplicity when cal-

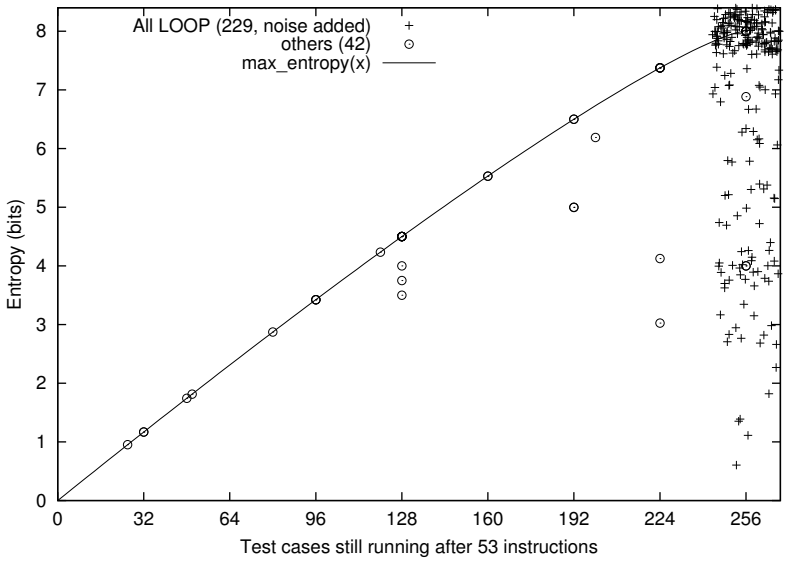

Fig. 6. Entropy of T8 programs of 7 instructions still running at least one test case after 53 time steps. Noise is added to + data to spread them.

culating the entropy of a mixture of stopped and still executing programs, the stopped programs are treated as if they were all in the same stopped state. This means in the graphs that follow, the number of runs of a program that reach the t^{th} instruction has an impact of the entropy as well as (for example) the difference between the contents of memory. (Remember apart from the input register, each of the 256 runs, start in the same, random, state.) To illustrate this, Figure 6 shows the 229 (+) of 1000 T8 programs of 7 instructions which always get stuck in a loop tend to have a high entropy. In contrast the entropy of the remaining 42 programs *circ* strongly depends upon how many runs (test cases) are still executing. (Figure 6 takes a snap shot after 53 instructions have been obeyed). It is clear the number of active test cases has a strong influence on the variation in memory contents etc. between test cases. I.e. entropy, in most short T8 programs, is as large as possible, given the number of test cases still running. This is consistent with the fact that most short programs implement the identity function, see Figure 9, which has maximum entropy.

Figure 7 shows the average number of test cases still running up to instruction 53. The shortest programs tend to stop or loop immediately. Only those still looping show on Figure 7. Short programs which loop on one test case tend to loop on all of them, giving the almost constant plots for short programs seen in Figure 7. Longer random programs, tend to run for longer and have more variation between the number of instructions they execute on the different test cases. Figure 8 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

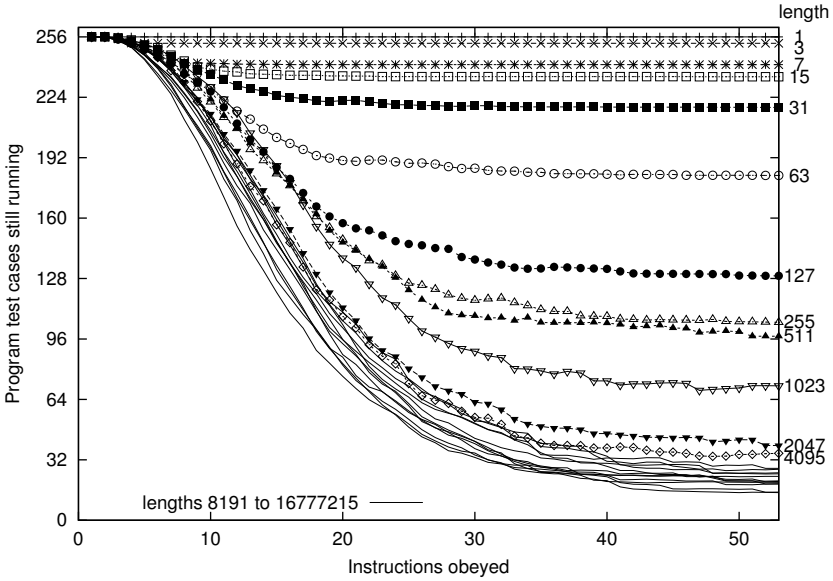

Fig. 7. Mean number of test cases where T8 program has not HALTed or stopped. 1000 random T8 programs of different lengths.

their variability. This gives the almost constant high entropy plots for short programs seen in Figure 8. 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 decrease in entropy with run time seen in Figure 8. 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.

5.2 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 9, uniformly chosen random programs of a given length sample these functions very unequally. (This is also true of the T7, cf. Figures 12 and 13). In particular the identity function and the 256 functions which return constants are much more likely than others. Figure 9 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.

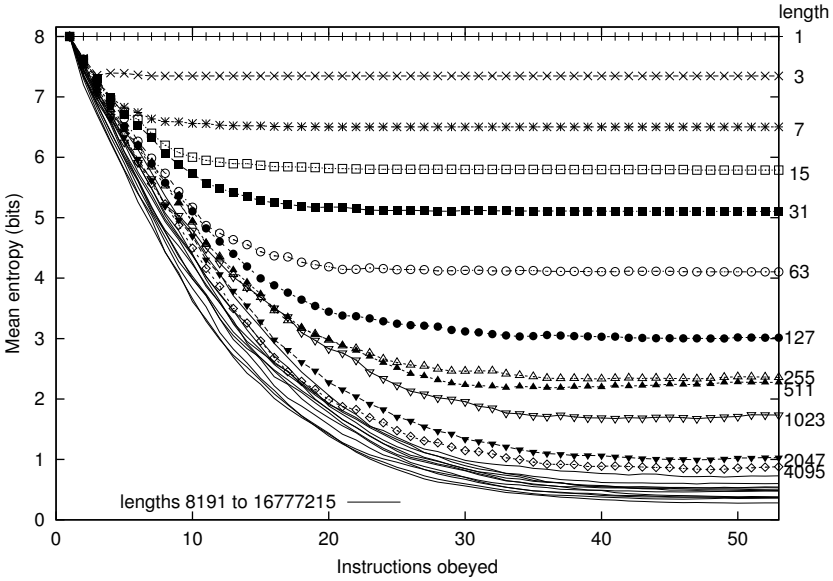

Fig. 8. Evolution of variation between test cases in 1000 random T8 programs of different lengths. Same programs as in Figure 7.

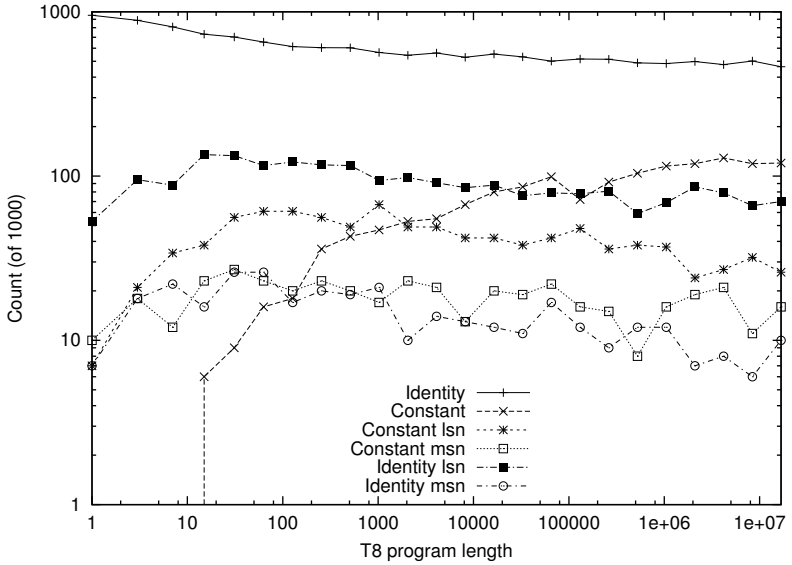

Fig. 9. Frequency of common functions implemented by random T8 programs of different lengths by 53 instruction cycles. Data are noisy due to sample size, but the trend to falling proportion of functions, except the constants, with program size can be seen.

The rise in the frequency of the constant functions and fall in the identity function with increasing program size (Figure 9) are consistent with the corresponding fall in entropy seen in Figure 8. (The same will be seen in the next section for the T7, cf. Figures 12 and 14.)

6 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 5. However since without a HALT instruction, the T7 programs tend to run for much longer, we increase the any time limit from 53 to 1000 instructions.

Figure 10 confirms removing HALT does indeed mean most programs run up to the any time limit. Figure 10 relates to all 256 runs of each random program. Whilst the error bars (top solid line) show on average there is some variation between identical programs starting with different inputs for middle sized programs both very large and very small programs have the same (any time) run time. This suggests most big T7 programs loop regardless of their input. Whilst short programs halt whatever their input. 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 11 shows that the fraction of programs which never loop, falls as $O(1/\sqrt{\text{length}})$, as we found previously [3]. Figure 11 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.

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 12 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.

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 12, 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

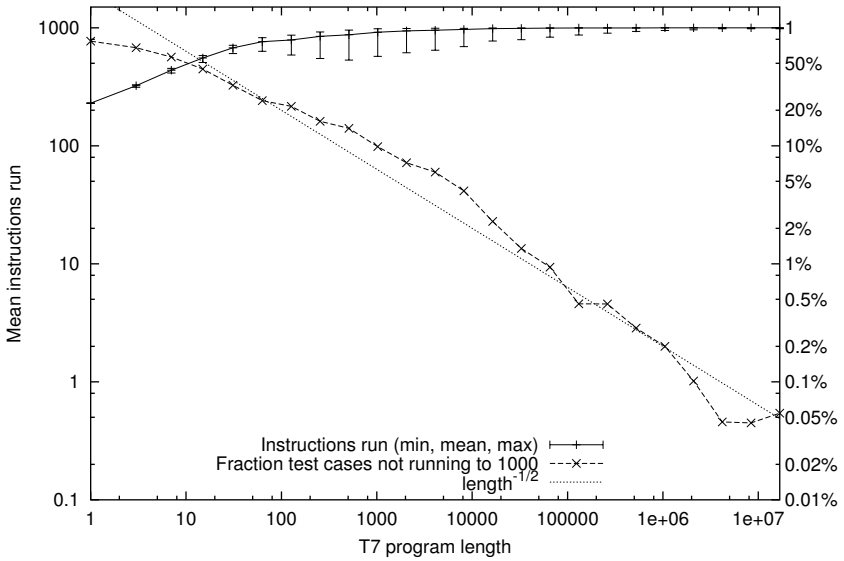

Fig. 10. Run time on 256 test cases of 1000 random T7 programs of each of a variety of lengths. Some short programs run to their end and stop but almost all longer programs run up to the limit of 1000 instructions on all test cases.

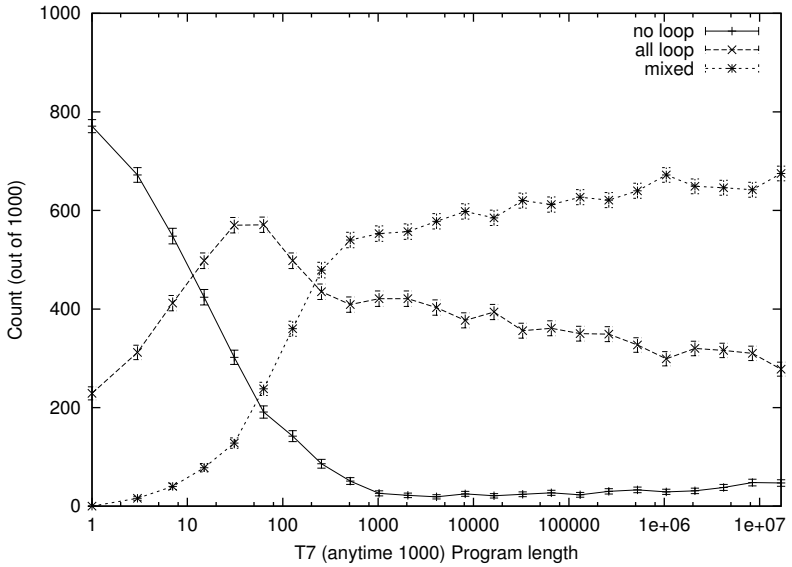

Fig. 11. Looping characteristics of Random T7 programs on 256 test cases. All programs still running are aborted after 1000 instructions. Error bars show standard error.

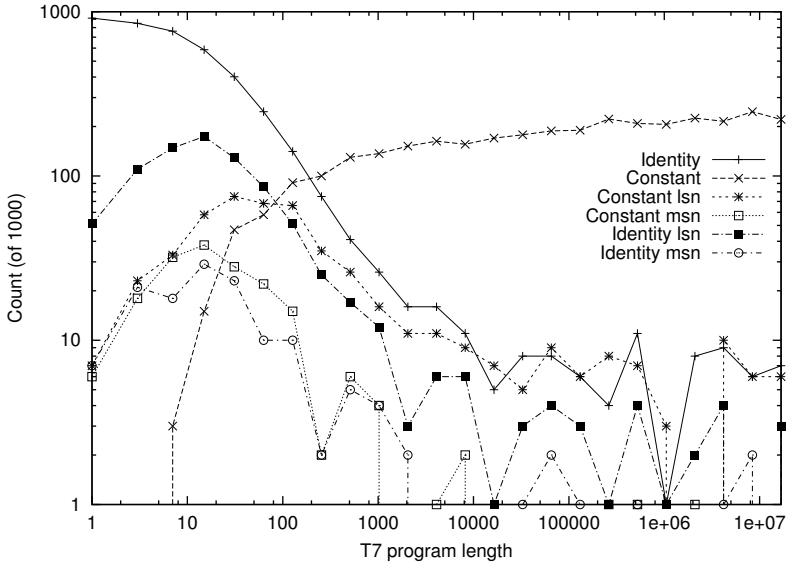

Fig. 12. Variation of some common functions implemented by T7 with program length. Note, except for the 256 constant functions, after 1000 instructions most of the functions considered in Figure 9 become rare

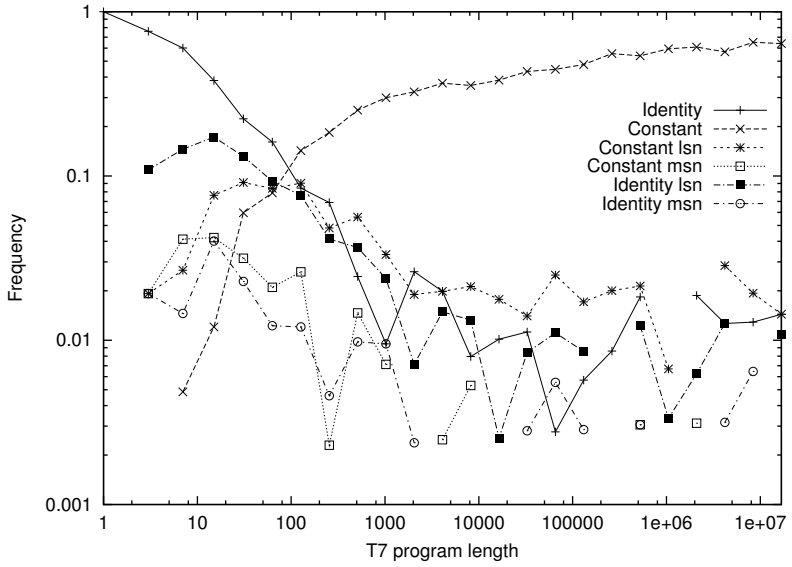

Fig. 13. Variation of some common functions implemented by T7 with program length. As Figure 12 except we include only programs which do not loop at all.

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 13 shows the rise in the proportion of constants with program length is even more pronounced. Whilst Figure 15 shows, if loops are excluded, in most cases variation between different input values falls rapidly to zero. Figures 14 and 15 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.

7 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 [8] 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 [9] and post-modern 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.

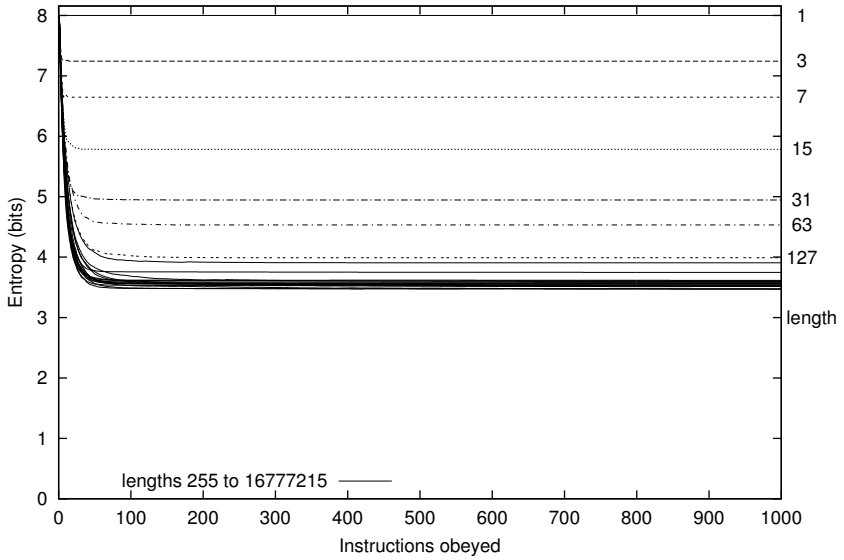

Fig. 14. Average reduction in variation between test cases for 1000 random T7 programs of each of a variety of lengths run on 256 test cases, up to 1000 steps. The smooth averages conceal strong peaks in the data at 0 (constants), 1, 2, 3, 4, 5 and 8 bits (e.g. identity).

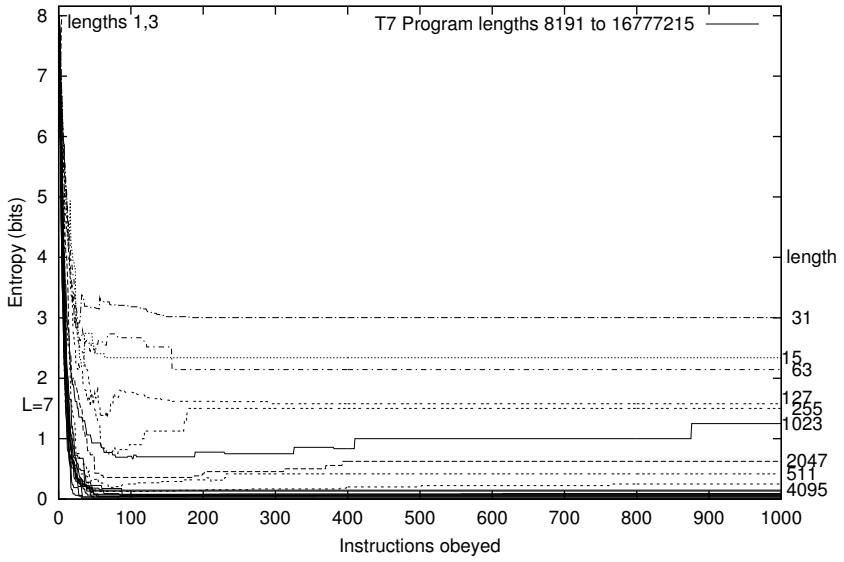

Fig. 15. As Figure 14 except we include only runs which did not loop on any test case.

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.

8 Conclusions

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

References

1. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer 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, Uni. Essex, UK, 2005.
3. The halting probability in von Neumann architectures. W. B. Langdon and R. Poli. In Collet *et al.* editors, *EuroGP-2006* LNCS 3905, pages 225–237. 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 *FLAIRS*, pages 270–274, Pensacola, Florida, 1994. IEEE Press.
6. S. R. Maxwell III. Experiments with a coroutine model for genetic programming. In *1994 IEEE World Congress on Computational Intelligence*, pp413–417a.
7. Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, 1964.
8. 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.
9. C. S. Calude, M. J. Dinneen and C-K Shu. Computing a glimpse of randomness. *Experimental Mathematics*, 11(3):361–370, 2002.