Physica D 42 (1990) 85–98
North-Holland

THE COMPLEX BEHAVIOR OF SIMPLE MACHINES

Rona MACHLIN
Relational Technology, Park 80 West Plaza 1, Saddle Brook, NJ 07662, USA

and

Quentin F. STOUT[1]
Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, USA

This paper interprets work on understanding the actions of Turing machines operating on an initially blank tape. While this is impossible for arbitrary machines, complete characterizations of behavior are possible if the number of states is sufficiently constrained. The approach combines normalization to drastically reduce the number of machines considered, human-generated classification schemes, and computer-generated proofs of behavior. This approach can be applied to other computational systems, giving complete characterizations in sufficiently small domains. This is of interest in the area of emergent systems since the properties of such systems are often difficult to determine. By using computers to eliminate multitudes of machines with well understood behavior, some unanticipated exotic machines with complex behavior were discovered. These exotic machines show that it is quite difficult to estimate the number of states needed to produce a given behavior, and hence subjective estimates of complexity may be poor approximations of the true complexity.

1. Introduction

This expository paper discusses work on understanding the possible actions of a single simple machine interacting with a simple input. The machines are Turing machines, defined below, which have only a few states, and the input is an all-blank tape. Depending on one's background, this may either seem to be a very easy task, since the machines have very simple descriptions, or an impossible task, since among computer scientists it is well known that one cannot even decide whether or not an arbitrary Turing machine will halt. We show that it is instead a possible, but difficult, task, as long as the number of states is suitably restricted.

We believe that the techniques used to understand small Turing machines may prove to be useful in understanding other "simple" systems, especially if one wants to produce provably complete classifications of behavior in suitably restricted classes. Since this work is largely unknown outside of computer science, and in fact is not even well known within computer science, we have taken a mainly expository approach in order to reach a wider range of researchers. This work may also revise notions of interesting or desirable behavior in Turing machines. Further, while we are successful in characterizing sufficiently small Turing machines, we show that a single Turing machine can be viewed as an emergent system, and thus any attempt at an unrestricted classification of the behavior of all emergent systems in any sufficiently powerful class is doomed to failure. This limitation needs to be more widely understood.

Finally, we show that the behavior of small Turing machines is far more complicated than most people would guess, and that exhaustive search can locate machines that are exceedingly difficult to create on one's own. Because such

[1]Partially supported by *Incentives for Excellence* grant from Digital Equipment Corp.

0167-2789/90/$03.50 © Elsevier Science Publishers B.V. (North-Holland)

machines are quite unexpected, people tend to significantly overestimate the number of states needed to produce their behavior. This creates a false impression of complexity, leading one to believe that a system has many more components that it really does, rather than understanding that the complexity can come from the repeated application and interaction of a few simple, carefully chosen rules.

In section 2, we define Turing machines, and define the busy beaver and halting probability problems. These problems motivated work in classifying the behavior of small Turing machines. In section 3 we introduce the notion of tree normalization, which is used to drastically reduce the number of cases that must be considered. In section 4 we show the techniques used to classify Turing machines that are in infinite loops, which completes the computation of small busy beaver numbers. Section 5 shows how to apply this work to estimate the halting probability, and in section 6 we offer some concluding remarks.

2. Background

Turing machines are an attempt to formalize the notion of effective computation. While it is impossible to prove that one has correctly captured the intuitive notion of effective computation, all other attempts have yielded systems that can compute only functions computable by Turing machines, and hence there is fairly widespread acceptance of the Church–Turing thesis that Turing machines do indeed compute all functions that are effectively computable [14].

For our purposes, a (deterministic) *Turing machine* has an input–output *alphabet* of $\{0,1\}$, which writes and reads from a 2-way infinite tape of *squares*. The 0's and 1's are called *symbols*, and each tape square contains exactly one symbol. The 0 is thought of as being equivalent to blank.

A Turing machine has some finite number, k, of internal states, labeled $1,\ldots,k$, and a read/write head connecting it to the tape. At each time unit the read/write head is positioned under some square of the tape, and based on the symbol read and the current state, the Turing machine will write a (perhaps different) symbol at the square, move the read/write head left or right one square, and switch into a (perhaps different) state. See fig. 1.

For each possible pair of current state and symbol read there is a unique instruction specifying the symbol printed, head movement, and new state. Such instructions will be given as

(state, symbol, new symbol,

head movement, new state),

where L or R are used to indicate head movement to the left or right, respectively. We assume that the tape is initially all 0 (all blank), and that the Turing machine is initially in state 1.

A Turing machine continues to execute its instructions until it encounters an instruction specifying a new state of 0, in which case it prints the symbol, moves the head, and then halts. Fig. 2 illustrates this action, which the subscripts on the tape indicate the state of the Turing machine and the location of the head.

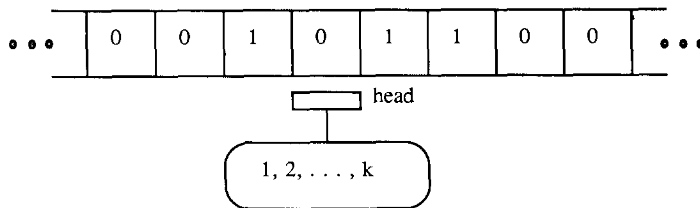

Fig. 1. A Turing machine.

Turing machine instructions

State	Symbol read 0	Symbol read 1
1	1R2	1L3
2	1L1	1R2
3	1L2	1R0

n	Tape status after n steps					
1				1	0_2	
2				1_1	1	
3			0_3	1	1	
4		0_2	1	1	1	
5	0_1	1	1	1	1	
6	1	1_2	1	1	1	
7	1	1	1_2	1	1	
8	1	1	1	1_2	1	
9	1	1	1	1	1_2	
10	1	1	1	1	1	0_2
11	1	1	1	1	1_1	1
12	1	1	1	1_3	1	1
13	1	1	1	1	1_0	1

Subscripts denote machine state and position of read/write head

Fig. 2. Turing machine execution.

As fig. 2 shows, even though the individual instructions have simple, precise definitions, the overall behavior of the Turing machine can be quite complicated and not readily apparent from the individual instructions. Thus a single Turing machine, viewed as a collection of cooperating individual instructions, is an emergent structure according to the definition in ref. [7].

2.1. Halting problems and halting probability

Many problems have been posed involving Turing machines. The best known of these, the *Halting Problem*, asks for an algorithm with input consisting of a Turing machine M and an initial tape T, and which outputs "true" if M will eventually halt when started on input T, and outputs "false" otherwise. This problem is well known to be impossible in the sense that no Turing machine can provide such an algorithm (see any standard text in computability, such as ref. [14]). Assuming that the Church–Turing thesis is correct, this shows that no effective algorithm of any form can solve the halting problem. It is also well known that there is a fairly simple machine U, the universal Turing machine, such that it is impossible to provide an algorithm to decide if U halts on input T for arbitrary T. Similarly, the *restricted halting problem*, in which the machine varies but the initial tape is all blank, is also unsolvable.

A related problem, which we call the *halting probability problem*, can be intuitively phrased as "What is the probability that a random Turing machine will halt when started on an all-blank tape?" To formally define this probability, denoted Ω, one must assign probabilities to Turing machines. Since there are infinitely many Turing machines it seems that no assignment is completely natural, and we postpone such an assignment until section 3. However, one can show that for any nontrivial assignment Ω cannot be computed, where by an algorithm computing Ω we mean that given any $\epsilon > 0$, the algorithm will return a rational number which is within ϵ of Ω.

Apparently Chaitin was the first to formally define Ω [5], though his definition differs from that given in section 3. Ω has many interesting properties [9], and recently Chaitin used it in his significant transformation of Gödel's incompleteness theorem into a statement about the solutions of a specific exponential Diophantine equation [6].

2.2. Busy beaver problems

Tibor Rado felt that the arguments used to prove the uncomputability of the halting problem were not sufficiently intuitive, and posed the Busy Beaver Problem as a more concrete variation. To define this, let $H(k)$ denote the set of all k-state Turing machines which eventually halt when started on a blank tape. Note that $H(k)$ is finite since it is a subset of the set of all k-state Turing machines, and there are exactly $(2 \cdot 2 \cdot (k+1))^{2k}$ different sets of instructions for k-state Turing machines, i.e. there are $2k$ instructions that need to be supplied, one for each (state, symbol) pair, and for each instruction there are 2 choices of new

symbol, 2 choices for head direction, and $k+1$ choices for new state.

For a Turing machine M in $H(k)$, let $\sigma(M)$ denote the number of 1's left on the tape when M halts after starting on an all-blank tape, and let $s(M)$ denote the number of steps performed by M before halting. For example, for the machine in fig. 2, $\sigma(M)=6$ and $s(M)=13$. Rado [16] defined the kth *busy beaver number*, denoted $\Sigma(k)$, by

$$\Sigma(k)=\max\{\sigma(M)\colon M\in H(k)\},$$

that is, $\Sigma(k)$ is the maximum number of 1's left on the tape by any halting k-state Turing machine. Similarly, he defined $S(k)$ by

$$S(k)=\max\{s(M)\colon M\in H(k)\}.$$

The *busy beaver problem* is to give an algorithm which computes $\Sigma(k)$ for all values of k. Note that for any k, this problem merely asks for the maximum among a finite set of values.

Rado provided a nice proof that Σ could not be effectively computed by showing that if f is any function computable by a Turing machine, then there is a number n, depending on f, such that $\Sigma(n)>f(n)$. This also shows that S cannot be effectively computed since $S(k)\geq\Sigma(k)$ for all k. While Rado emphasized computing Σ, we will instead concentrate on the function S, for reasons which will become clearer in section 2.3. Work on computing S and Σ is discussed in refs. [1–3, 7, 10–13, 15–17].

2.3. Problem relationships

The halting problem, halting probability problem, and busy beaver problems are closely related, in that a solution to any one of them would yield a solution to each of the others (although the transformations may not have any practical usefulness). To illustrate this, suppose we have an algorithm which computes S, and want to solve the restricted halting problem. To decide if machine M halts on blank tape, merely count the numbers of states in M, call this k, and then simulate the running of M for $S(k)$ steps. If M has not halted in $S(k)$ steps then it must be that it will never halt, by the definition of S. Notice that $\Sigma(k)$ would not have been as useful since it may be that M sometimes writes 1's and sometimes erases them, making it difficult to guarantee that it will not suddenly erase all but $\Sigma(k)$ or fewer 1's and then halt.

To see how the halting probability problem can be used to solve the restricted halting problem, let M be some specified Turing machine, and suppose it has probability p. (It suffices to merely know that p is a nonzero lower bound on the probability of M.) Using any effective enumeration of the Turing machines, simulate running the first Turing machine for one step, then simulate the first Turing machine for two steps, then the second Turing machine for two steps, then the first Turing machine for three steps, then the second Turing machine for three steps, then the third Turing machine for three steps, and so on. (This stimulation process is known as *dove-tailing*.) Whenever a machine halts, add its probability to a running total. Eventually, either M halts, or else the running total becomes large enough so that if p were added to it then the total would exceed the halting probability. In this latter case M cannot halt.

All the other possible choices of using a solution of one problem to solve another can be done similarly, with the exception that it may not be obvious that an algorithm which computes $\Sigma(k)$ can be used to compute $S(k)$ (and hence to solve any of the other problems). Rado [16] noted that one could prove that

$$S(k)\leq(k+1)\Sigma(5k)2^{\Sigma(5k)}$$

(much better bounds are possible), and any upper bound T for $S(k)$ can be used to compute $S(k)$. To do this, one merely runs all k-state Turing

machines for T steps. Any machine which runs for T steps without halting must be in an infinite loop, so the largest number of steps used by a machine which halts in T or fewer steps must be $S(k)$.

A solution to any of these problems would, at least theoretically, provide an effective means of solving mathematical problems. For example, to solve Fermat's last theorem, which asserts that there are no positive integers x, y, z, i such that $i > 2$ and $x^i + y^i = z^i$, one could write a program which uses dove-tailing to try all possible choices of x, y, z, and i, and which halts if it ever finds equality. Therefore the solution to Fermat's last theorem is reduced to the restricted halting problem for this program.

A more general reduction of mathematics to halting problems can be obtained by noticing that a proof is merely a finite sequence of symbols which can be generated, and verified, by a computer. Given any mathematical statement one is curious about, construct a program which generates all possible proofs and then verifies if it is a proof of the desired statement, halting whenever such a proof is found. Using this procedure, the provability of any mathematical statement is "reduced" to the problem of deciding if a specific Turing machine halts. This fact is central to the work in ref. [6].

3. Tree normalization

Despite the fact that the halting probability and the busy beaver problem cannot be solved, one can ask about partial solutions. For the halting probability problem one could ask for upper and lower bounds on the probability, and for the busy beaver problem one might determine exact values for some values of k, or bounds on the values. This approach was taken by Rado in his classes, and has been pursued by many others since [1, 2, 7, 10–13, 17]. We will emphasize the approaches to the busy beaver problem since that is where most of the work has been performed, though we were first attracted to working on the halting probability problem.

To evaluate $\Sigma(k)$ or $S(k)$ for small values of k, one immediately encounters the problem of having a large number of possible machines. As was noted above, there are $[4(k+1)]^{2k}$ k-state Turing machines, which, for example, is 25 600 000 000 when $k = 4$. However, many of these are equivalent or their behavior is readily apparent. Lin and Rado [12] noted that, since one starts in state 1 reading a zero, if the instruction is to go to state 0 then the machine will halt after only one step, while if the instruction is to go to state 1 then the program will be an infinite loop. This observation alone classifies the behavior of 10 240 000 000 4-state machines. Therefore the only unknown behavior is to go to a new state, and since the labels of the states are arbitrary we may as well call it state 2. Further, if the machine makes its first head movement to the left it will just be a mirror image of an equivalent machine with left and right head movement reversed for all instructions. Therefore one can assume that the first head movement is to the right.

Since Lin and Rado emphasized calculating $\Sigma(k)$, they could make a final reduction, namely that the first step prints a 1. This is because if it does not print a 1, imagine following the machine's operation until it first prints a 1, and starting it instead at the instruction that printed that 1. This new start will eventually halt if and only if the original one did, and both will produce the same number of 1's. Thus Lin and Rado could assume that the original instruction was (1, 0, 1, R, 2). However, the new start will use fewer steps than the original, and hence this normalization may underestimate $S(k)$ by as much as $k-1$. This can be corrected by first using the Lin and Rado normalization to obtain a lower bound S' of $S(k)$. Then, noting that $S' + k - 1$ is an upper bound on $S(k)$, one can utilize the approach described in section 2.3 to use this upper bound to determine the exact value of $S(k)$.

The Lin and Rado approach can be extended (though they did not do so) to the viewpoint that

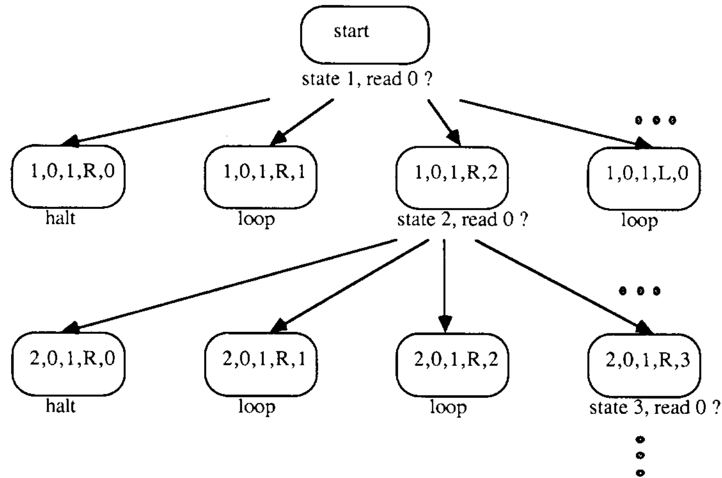

Fig. 3. Tree-normal programs.

one deals with incompletely specified Turing machines, only filling in instructions as they are needed. For example, if the initial instruction was (1, 0, 1, R, 2), then the machine will go to state 2 and again encounter a 0, so now a new instruction is needed. For this, either a 0 or 1 can be printed, the head can be moved either L or R, and either the machine halts (state 0), goes to a state already used (1 or 2), or goes to a new state, which we can relabel as 3. Thus there are $2 \cdot 2 \cdot 4 = 16$ effectively different choices for being in state 2 and seeing a 0, given the prior choice of instruction (1, 0, 1, R, 2). Four of these choices halt, four go into infinite loops (printing 0 or 1, moving R, and going to state 1 or 2), while the remaining eight each will then encounter a situation requiring that yet another instruction be generated. This approach was utilized in refs. [1, 2, 11].

The generation of instructions as they are needed yields a tree representation of the machines generated, as illustrated in fig. 3. The machines generated are said to be in *tree-normal* form. Notice that a single machine in tree-normal form may represent many Turing machines. For example, the Turing machine with instructions (1, 0, 1, R, 2) and (2, 0, 1, R, 2) represents $[4(k+1)]^{2k-2}$ k-state Turing machines, each of which will go into an infinite loop.

3.1. Probabilities

Tree-normal form can be used to assign probabilities to Turing machines. We say that the root of the tree has probability 1, and whenever a node has children (i.e. when it eventually encounters a situation where a new instruction is needed) then its probability is evenly divided among all of its children. In general, if a node with probability p represents a partial Turing machine where states $1, \ldots, i$ have been explicitly referenced so far (with the convention that the root explicitly references state 1), if a new instruction is needed then there are exactly $2 \cdot 2 \cdot (i+2)$ children, each of which has a probability $p/[4 \cdot (i+2)]$.

This is the notion of Turing machine probability that we will use for the halting probability problem, so to determine Ω we need to find the sum of the probabilities of all leaf nodes corresponding to a new instruction sending the machine to state 0. Similarly one could define the *infinite loop probability* as the sum of the probabilities of all leaf nodes corresponding to machines in infinite loops. There is a slight technical question of whether the sum of these two probabilities is 1, since the tree has infinite height and one can show that there exist such trees, with probabilities assigned in the same manner, for which the sum of

the probabilities of the leaves is less than 1. However, it is easy to show that in our case these two probabilities do indeed sum to 1.

It is important to note that our definition of Ω differs somewhat from that in refs. [5, 6, 9], and that this definitional change alters the value. While the definition in refs. [5, 6, 9] has some properties that make it simpler to use for theoretical purposes, we believe our definition is somewhat more natural and easier to understand.

3.2. Variations

It was the problem of defining, and then estimating, the halting probability that lead us to tree normalization. However, we discovered that it had been used earlier by Brady [1] in his work on the busy beaver problem, and so his terminology was incorporated into the work reported in ref. [11] (written by the first author under her maiden name). The work in ref. [11] is an independent confirmation of Brady's results, which is important since the sheer volume of human and computer work involved raises the possibility of error. The work in ref. [1] was eventually published in ref. [2], while the work in ref. [11] has not been previously published.

There are four differences between tree normalization used to find $\Sigma(k)$ or $S(k)$ and the version used to define Ω. In each of these, the trees used for busy beaver problems are smaller than the tree for the halting probability. First, as was noted earlier, the initial instruction for finding $\Sigma(k)$ can be taken to be (1, 0, 1, R, 2), ignoring the other 11 possibilities. As was stated above, this may slightly underestimate $S(k)$, but a post-mortum check can be used to correct it. Second, no node need be generated which sends the machine into a state labeled $k+1$ or higher. Third, if a node represents a partial machine in which only states $1,\dots,i$ have been explicitly referred to so far, and if $2i-1$ instructions have already been defined, then the new instruction is the last instruction involving states $1,\dots,i$. If the new instruction sends the machine to states $1,\dots,i$ then the machine will be an infinite loop, so to compute busy beaver numbers one need only consider instructions sending the machine to states 0 or $i+1$. Finally, fourth, the instructions sending the machine into the halt state need only print a 1 and move R, since any other options would produce the same value for $S(k)$ and either the same or smaller values for $\Sigma(k)$.

Table 1
Number of Turing machines to be analyzed.

k	Tree-normal	$[4(k+1)]^{2k}$
2	41	20736
3	3936	16777216
4	603712	25600000000

All counts of numbers of nodes will be in terms of the tree normalization used to find $S(k)$, though estimates for Ω will use the correct tree normalization for it. All counts are taken from ref. [11], and when infinite loops are classified these counts differ slightly from those in ref. [2]. These small differences are due to slight variations in the definitions used, number of steps stimulated, and the order in which the tests were applied.

Table 1 shows the number of tree-normal machines generated for the busy beaver problems, as opposed to the number of Turing machines which are formally different. This clearly shows the significant reductions accomplished through the use of tree normalization. This table was made by a back-tracking program which simulated each node until it halted, was in an infinite loop, or reached a situation where another definition was needed, in which case the appropriate children were generated. The task of determining when a program is in an infinite loop is discussed in section 4.

4. Infinite loops

The major effort in calculating busy beaver numbers and estimates for the halting probability lies in proving that large numbers of machines are

in infinite loops. The approach taken in refs. [1, 11, 12] is to examine some of these machines by hand, elicit a common behavior which insures that a machine is in an infinite loop, and then write a program which examines candidate machines and proves that some of them do indeed have that behavior. This process tends to iterate, with the researcher constantly trying to reduce the number of unclassified machines by either generalizing types of behavior earlier searched for, or by discovering new types of behavior.

In the end, a small enough number of machines remain so that they can be manually examined and verified to be in infinite loops. In ref. [12], only one type of behavior was needed in determining $\Sigma(3)$, with only 40 machines which needed to be verified by hand. In refs. [1, 11], four types of behavior were used in determining $\Sigma(4)$ and $S(4)$, along with a couple of hundred machines which were verified by hand.

One byproduct of this approach is that, while the busy beaver and halting probability problems are defined in terms of machines that halt, the interesting work involves machines that do not halt. In a certain sense, this approach treats all halting programs as equivalent, only needing to record probabilities, number of ones produced, or number of steps used, while the programs that do not halt must be more carefully examined and characterized.

The remainder of this section is based on the work in ref. [11], which was in turn based on ref. [1]. At each stage, machines not yet classified are called *holdouts*. Using tree normalization and allowing machines to run a couple of hundred steps produces 1364 3-state machines which halt, and 2572 3-state holdouts, and 182 604 4-state machines which halt, along with 421 108 4-state holdouts. The holdouts were run through a program to see if they could be proven to have a behavior known as a simple loop. The holdouts from the simple loop test were then put through a back-tracking analysis to see if it could be shown they were in infinite loops without determining which type of loop they were. (Actually, a fast test for

Table 2
Classification of tree-normal machines.

	3-state	4-state
total	3936	603 712
halted	1364	182 604
infinite loop	2572	421 108
simple loop	2495	404 733
back-track	50	10 363
Christmas tree	25	5 144
shadow Christmas		241
counter	2	417
holdout		210

simple loops was combined with the simulation program, and a more thorough simple loop test was run after back-tracking analysis, but logically there was no need to do so. The numbers reported are the sum of those found by the different simple loop tests, with the vast majority found by the fast test.) The remaining holdouts were run for a while, and based on the rate at which new tape squares were visited, they were tentatively classified as being a Christmas tree or a counter. For each of these classes, a program was developed which in most cases could prove a candidate was of the indicated type and was indeed in an infinite loop. Finally, the remaining holdouts were examined by hand.

The following subsections explain this process in more detail, and table 2 shows the number of machines classified at different states. Given a Turing machine M, we use M^c to denote the machine formed from M by changing all R moves to L, and vice versa. By a *word* we mean a (perhaps empty) finite string of 0's and 1's. For a word W and state r we use W_r to mean that the machine is in state r examining the rightmost symbol of W; ${}_rW$ means that the machine is in state r examining the leftmost symbol of W; $W\ {}_r$ means that the machine is in state r examining the first symbol to the right of W, and ${}_r\ W$ means that the machine is in state r examining the first symbol to the left of W. We use 0^* to mean infinite occurrences of 0, and W^i to mean i concatenated copies of W.

State	Symbol read 0	Symbol read 1
1	1R2	1L3
2	1R3	
3	0L1	0R1

$1\underline{0}$	$110\underline{0}1$	$11111101\underline{0}$
$11\underline{0}$	$11\underline{0}01$	$111111011\underline{0}$
$1\underline{1}$	$111\underline{0}1$	$11111101\underline{1}$
$\underline{1}1$	$1111\underline{1}$	$1111110\underline{1}1$
$\underline{1}$	$11110\underline{0}$	$11111100\underline{1}$
$\underline{0}1$	$111101\underline{0}$	$1111110\underline{0}1$
$\underline{0}01$	$1111011\underline{0}$	$111111\underline{0}01$
$1\underline{0}1$	$111101\underline{1}$	$1111111\underline{0}1$
$11\underline{1}$	$11110\underline{1}1$	$11111111\underline{1}$
$110\underline{0}$	$111100\underline{1}$	$111111110\underline{0}$
$1101\underline{0}$	$11110\underline{0}1$	$1111111101\underline{0}$
$11011\underline{0}$	$1111\underline{0}01$	$11111111011\underline{0}$
$1101\underline{1}$	$11111\underline{0}1$	$1111111101\underline{1}$
$110\underline{1}1$	$111111\underline{1}$	$111111110\underline{1}1$
$1100\underline{1}$	$1111110\underline{0}$	$1111111100\underline{1}$

Underlining indicates position of read/write head

Fig. 4. A simple loop.

4.1. *Simple loops*

Fig. 4 shows a simple loop in operation. A Turing machine M is called a *simple loop* if either M or M^c satisfy one of the following:

(1) Some tape configuration is repeated infinitely often. That is, there is a nonzero state s and words X and Y such that at some time step the tape configuration is $0^* X_s Y 0^*$, and the same tape configuration is reached at some later time step.

(2) M periodically moves to the right, in that there is some nonzero state s, words X and Y and a nonempty word V, such that at one time the tape configuration is $0^* X_s Y 0^*$, at some later time the tape configuration is $0^* V X_s Y 0^*$, and between these times M never moved left of the left edge of the initial X.

It is clear from the definition that a machine classified as a simple loop is indeed in an infinite loop.

Initial checks for simple loops were done by maintaining a table containing the tape configuration, position, time step, and state of the most recent occurrence of each state–symbol pair. After each step the table was consulted to see if a simple loop condition could be detected. Holdouts from the initial check were run through a second version which maintained a table of the conditions each time the Turing machine scanned a square at the edge of the critical portion of the tape (i.e. the nonzero portion). Of the initial 2572 3-state holdouts, all but 77 were proven to be simple loops, while of the initial 421 108 4-state holdouts, all but 16 375 were simple loops.

State	Symbol read 0	Symbol read 1
1	1R2	1L1
2	0L1	1L3
3	1R4	1L3
4		0R2

Fig. 5. A machine analyzable via back-tracking.

4.2. *Back-tracking*

One straightforward way to prove that a program is in an infinite loop is to directly prove that it cannot reach the halt state. For example, consider the tree-normalized machine in fig. 5, which has only one unspecified instruction, namely being in state 4 while scanning a zero. This machine can halt only if it reaches this instruction, i.e. it must reach a local tape configuration of 0_4. To get there, it must have been in state 3 scanning a zero to the right of this zero, i.e. it must have been in configuration $0_3 0$. The only instructions which move to state 3 are for state 2 input 0, or for state 3 input 1, so the tape must have been in the configuration $0\,1_2$ or $0\,1_3$. However, both of these would produce $0_3 1$, which is not what was needed. Therefore the configuration 0_4 can never be reached, and the machine must be in an infinite loop.

While back-tracking can be useful, it cannot be guaranteed to always stop since otherwise it would supply a solution to the halting problem. As with all of the heuristics we discuss, one must make some decision as to how long to run this technique

State	Symbol read 0	Symbol read 1
1	1R2	1L1
2	1L1	1R3
3		1R1

$1\underline{0}$	$11111\underline{0}$	$11111\underline{1}11$
$\underline{1}1$	$1111\underline{1}1$	$1111\underline{1}111$
$\underline{0}11$	$111\underline{1}11$	$111\underline{1}1111$
$1\underline{1}1$	$11\underline{1}111$	$11\underline{1}11111$
$11\underline{1}$	$1\underline{1}1111$	$1\underline{1}111111$
$111\underline{0}$	$\underline{1}11111$	$\underline{1}1111111$
$11\underline{1}1$	$\underline{0}111111$	$\underline{0}11111111$
$1\underline{1}11$	$1\underline{1}11111$	$1\underline{1}1111111$
$\underline{1}111$	$11\underline{1}1111$	$11\underline{1}111111$
$\underline{0}1111$	$111\underline{1}111$	$111\underline{1}11111$
$1\underline{1}111$	$1111\underline{1}11$	$1111\underline{1}1111$
$11\underline{1}11$	$11111\underline{1}1$	$11111\underline{1}111$
$111\underline{1}1$	$111111\underline{1}$	$111111\underline{1}11$
$1111\underline{1}$	$1111111\underline{0}$	$1111111\underline{1}1$
	$111111\underline{1}1$	$11111111\underline{1}$

Underlining indicates position of read/write head

Fig. 6. A Christmas tree.

before abandoning it. When applied with a 15-step limit to the 3-state holdouts only 27 holdouts were left. When applied with a 10-step limit on the 4-state holdouts only 6012 remained as holdouts.

4.3. Christmas trees

When Lin and Rado analyzed 3-state Turing machines, they applied some of the initial stages of tree normalization, and wrote programs to detect simple loops. They ended up with 40 machines that they analyzed by hand. Most of these exhibited the back-and-forth sweeping motion shown by the machine in fig. 6. Brady called this behavior a *Christmas tree*. While the behavior is more complex than that of a simple loop, it is still clearly repetitive and in an infinite loop.

Formally, a Turing machine M is a *Christmas tree* if either M or M^c satisfy the following conditions for some nonzero state s:

(1) There are nonempty words U, V, and X such that the tape configuration at some time is $0^*U\,V_s\,0^*$, and at some later time is $0^*\,U X V_s\,0^*$.

(2) The following conversions hold, where X, X', Y, Y', Z, V, V', V'', U, and U' are nonempty words and q and r are nonzero states (the symbol $\Rightarrow$ means that M transforms the left-hand side into the right-hand side after some numbers of steps):

(a) $X V_s 0^* \Rightarrow {}_q X' V' 0^*$;
(b) $X_q X' \Rightarrow {}_q X' Y$;
(c) $0^* U_q X' \Rightarrow 0^* U' Y'_r$;
(d) $Y'_r Y \Rightarrow Z Y'_r$;
(e) $Y'_r V' \Rightarrow Z V''_s$;

(3) $U' Z^i V'' = U X^{i+1} V$ for all $i \geq 1$.

While this definition is somewhat complicated, it just guarantees that the machine sweeps back and forth, growing a periodic middle part of the tape configuration.

Again it can be proved that any Christmas tree must be in an infinite loop. To detect these, a program was written which ran a holdout for a couple of hundred steps to overcome startup effects, and which then cut the nonblank part of the tape in half to obtain candidates for V and U. Then it ran the machine until a back-and-forth sweep was observed. If after the sweep the new tape had a right-hand portion that matched V and a left-hand that matched U, then the remainder in the middle was taken to be X. This process was continued to find values for X', Y, etc., and to verify the conditions. If the program ever ran too many steps without finding the desired behavior, or could not successfully determine appropriate words, then the machine remained a holdout.

There are many variations of Christmas trees, so the initial program was modified to detect more of the variants. Brady called one variation an *alternating Christmas tree*, for it takes two back-and-forth sweeps to complete its cycle. Another variation, a *shadow Christmas tree*, illustrated in fig. 7, creates an increasing "shadow" at one edge, which it never scans past.

After running the 3-state holdouts through the various Christmas tree programs, only 2 holdouts remained, while for the 4-state machines only 627 holdouts remained.

State	Symbol read 0	Symbol read 1
1	1R2	1L1
2	0L1	0R3
3	1R4	1R3
4	0R2	

1$\underline{0}$	1101$\underline{0}$	11101110$\underline{0}$	1111011110$\underline{0}$
$\underline{1}$	11011$\underline{0}$	1110111$\underline{0}$	111101111$\underline{0}$
$\underline{0}$1	110110$\underline{0}$	11101111$\underline{0}$	1111011111$\underline{0}$
1$\underline{1}$	11011$\underline{0}$	1110111$\underline{1}$	111101111$\underline{1}$
10$\underline{0}$	110111$\underline{0}$	11101$\underline{1}$1	11110111$\underline{1}$1
101$\underline{0}$	11011$\underline{1}$	1110$\underline{1}$11	1111011$\underline{1}$11
1010$\underline{0}$	1101$\underline{1}$1	1110$\underline{1}$111	111101$\underline{1}$111
101$\underline{0}$	110$\underline{1}$11	111$\underline{0}$1111	11110$\underline{1}$1111
1011$\underline{0}$	11$\underline{0}$111	111$\underline{1}$1111	1111$\underline{0}$11111
101$\underline{1}$	111$\underline{1}$11	11110$\underline{1}$11	11111$\underline{1}$1111
10$\underline{1}$1	1110$\underline{1}$1	111101$\underline{1}$1	111110$\underline{1}$111
1$\underline{0}$11	11101$\underline{1}$	1111011$\underline{1}$	1111101$\underline{1}$11
11$\underline{1}$1	111011$\underline{0}$	11110111$\underline{0}$	11111011$\underline{1}$1
110$\underline{1}$	1110111$\underline{0}$	111101111$\underline{0}$	111110111$\underline{1}$

Underlining indicates position of read/write head

Fig. 7. A shadow Christmas Tree.

State	Symbol read 0	Symbol read 1
1	1R2	
2	1L3	1R1
3	0R1	0L3

1$\underline{0}$	$\underline{0}$101	$\underline{0}$010001	11$\underline{0}$101
$\underline{1}$1	1$\underline{1}$01	$\underline{0}$10001	111$\underline{1}$01
$\underline{0}$01	11$\underline{0}$1	1$\underline{1}$0001	1111$\underline{0}$1
$\underline{0}$1	111$\underline{1}$	11$\underline{0}$001	11111$\underline{1}$
1$\underline{1}$	1111$\underline{0}$	111$\underline{0}$01	111111$\underline{0}$
11$\underline{0}$	11111$\underline{0}$	11$\underline{1}$101	1111111$\underline{0}$
111$\underline{0}$	1111$\underline{1}$1	1$\underline{1}$0101	111111$\underline{1}$1
11$\underline{1}$1	111$\underline{1}$01	$\underline{1}$00101	11111$\underline{1}$01
1$\underline{1}$01	11$\underline{1}$001	$\underline{0}$000101	1111$\underline{1}$001
$\underline{1}$001	1$\underline{1}$0001	$\underline{0}$00101	111$\underline{1}$0001
$\underline{0}$0001	$\underline{1}$00001	1$\underline{0}$0101	11$\underline{1}$00001
$\underline{0}$001	$\underline{0}$000001	$\underline{1}$10101	1$\underline{1}$000001
1$\underline{0}$01	$\underline{0}$00001	$\underline{0}$010101	$\underline{1}$0000001
$\underline{1}$101	1$\underline{0}$0001	$\underline{0}$10101	$\underline{0}$00000001
$\underline{0}$0101	$\underline{1}$10001	1$\underline{1}$0101	$\underline{0}$0000001

Underlining indicates position of read/write head

Fig. 8. A counter.

4.4. *Counters*

The final class of loops for which programs were written were called counters by Brady. Fig. 8 illustrates a counter, and it is obvious that it is indeed acting as a type of binary counter.

Formally, a Turing machine M is a *counter* if either M or M^c satisfies the following conditions:

(1) There are nonempty words E, X, Y, Z, and Z', a nonzero state s, and a positive integer n such that at some time the tape configuration is $0^* E\, Y_s\, Z' Z^n X\, 0^*$.

(2) The following conversions hold, for some nonempty word X':

(a) $Y_s Z' \Rightarrow_s Z' Z$;

(b) $0^* E_s Z' \Rightarrow 0^* E X'_q$;

(c) $X'_q X \Rightarrow Y X'_q$;

(d) $X'_q Z \Rightarrow_s Z' X$;

(e) $X'_q 0^{|X|} \Rightarrow_s Z' X$

where $|X|$ denotes the length of X.

In this definition, the X acts as a "one", and the Z acts as a "zero", in a binary counter.

Using an approach similar to that used for Christmas trees, a counter detector program was developed. This program successfully classified the final two 3-state holdouts as counters, and when run on the 4-state machines left only 210 holdouts.

4.5. *Final holdouts*

The final 210 holdouts were examined by hand to verify that they were in infinite loops (ref. [2] reported 218 final holdouts). More than half were variations of counters, including base-3 and base-4 counters. Also discovered were further Christmas tree variations, such as alternating shadow trees and triple and quadruple sweep trees.

Brady noted an additional class of machines, which he called *tail-eating dragons*. They have a back-and-forth sweep, limited by the end of the "tail" they create. After each sweep they "bite off" a piece of the tail, and when it is completely consumed they create a new, larger tail. As with other classes, there are also variations on this behavior. Fig. 9 gives the instructions of a tail-eating dragon.

State	Symbol read 0	1
1	1R2	1L1
2	1R3	0R4
3	1L1	
4	0L1	1R4

Fig. 9. Instructions for a tail-eating dragon.

5. Halting probability

The known values of S can be used to estimate Ω. Tree normalization appropriate for Ω is used (see section 3.2) to generate a portion of the tree. A node corresponding to a machine of k states is simulated for at most $S(k)$ steps. If it does not halt in this many steps then it is in an infinite loop and its probability is added to a running total of infinite loop probabilities. If it reaches a point were a new instruction is needed, then the probabilities of all children which halt immediately (i.e. those in which the new instruction has a new state of 0) are added to a running total for the halting probability, and the remaining children are simulated.

For a node corresponding to a machine with a number of states for which the S value is unknown the machine is simulated for some predetermined number of steps. If the machine does not halt or need a new instruction, then it is abandoned, and its probability is part of the uncertainty in the knowledge of the halting probability. Because no node is simulated for more than the predetermined upper limit (counting all the steps leading to the node), only a finite portion of the tree is explored. However, to reduce the stack requirements, one may also abandon nodes which have more than some (significantly smaller) predetermined number of defined instructions. While such nodes add to the uncertainty, they have relatively small probability since the probability of a single node decreases rapidly with a number of defined instructions.

As in the use of the tree for finding busy beaver numbers, some additional simplifications can be incorporated. For example, suppose a node corresponds to a machine with k states, and all instructions but one have been defined. If the node reaches a point where this instruction is needed then any definitions which do not go to a new state (or 0) must yield an infinite loop, and hence their probabilities can be immediately added to the infinite loop probability. One can also use the classification routines discussed in section 4 to prove that machines are infinite loops, rather than just adding their probability to the uncertainty total.

Using the above techniques, including the knowledge of $S(k)$ for $k \leq 4$, but not using the classification routines on machines of more than four states, yielded the results in table 3.

Table 3
Turing machine probabilities.

Probability of halting	0.465
Probability of infinite loop	0.529
Uncertainty	0.007

6. Final comments

As the *Emergent Computations* conference demonstrated, there is a significant interest in the general problem of understanding the behavior of simple systems. Further, researchers working on such problems have a wide range of backgrounds. Because of this, we felt it useful to describe work that led to the complete characterization of Turing machines of four or fewer states, and which has also produced results such as provable bounds on the halting probability. We note that a single Turing machine is an emergent system, in that it satisfies all of the conditions set forth in ref. [8] for an emergent system, and it can indeed have a very complex observed behavior as it moves through time and space (along the tape). Thus this work shows that complete, provable characterization of a suitably restricted nontrivial class of emergent systems has been achieved.

This work is completely rigourous, as opposed to, say, mere statistical sampling. Further, by making extensive use of computers to prove that certain machines have well-understood behavior, the researchers were able to focus their attention on the final holdouts, discovering unexpected behavior such as base-4 counters and tail-eating dragons. Most people would be hard pressed to develop a 4-state machine with such behavior, just as they would be unlikely to develop a 4-state machine which moves for 107 steps before halting. The 210 final holdouts exhibit significant diversity, and such machines would probably not be found other than through careful use of computers to sift out machines with known behavior. The holdouts represent only about 0.3% of the tree-normal 4-state machines, and only about 0.0002% of the unnormalized 4-state machines. This approach may prove to be generally useful for researchers seeking simple emergent systems with unusual properties. Further, the success in "decompiling" all 4-state machines and provably deciding their behavior may make it an attractive approach for other researchers trying to "decompile" simple systems to obtain an understanding of their behavior.

One crucial step in reducing the computational workload was the introduction of tree normalization. Tree normalization is a form of "lazy evaluation" which adds just those instructions which are needed, simulating the effects of a collection of instructions until a situation is encountered where a new instruction is needed. Classes of emergent systems other than Turing machines may also be numerically reduced through normalizations.

One way in which emergent systems research can impact upon work on Turing machines is by intensifying interest in the behavior of infinite loops. Computer scientists usually try to produce programs that rapidly complete their task and finish, rather than continue forever. (Operating systems are an important exception.) However, emergent systems research is most concerned with systems that have infinite, or very long, lifespans. Systems with short lifespans are usually easier to understand, just as it is trivial to see that a Turing machine which just moves right 10 steps and then halts (when given all-blank input) must have at least 10 states. It is much more complicated to understand or design infinite behavior, such as finding a minimal state Turing machine that has visited $\Theta(t^{1/5})$ tape squares after t steps. Even the work in refs. [1, 2, 11] did not completely classify all 4-state infinite loops, but rather resulted in hand analysis of a few holdouts. This analysis satisfied the authors that the machines were indeed in infinite loops, and they noted some interesting behavior, but they did not carefully describe all behaviors encountered, nor the number of machines with each behavior.

One caution for emergent systems researchers is that, while we believe formal approaches can be applied to other emergent systems models, we must emphasize that there are limits as to when exhaustive, provable characterization can be performed. Such characterizations must be tried with care and within appropriate parameter constraints. For example, as was noted previously, it is impossible to write a program which determines $S(k)$ for arbitrary k, and it is even impossible for any program to provide an upper bound for infinitely many k. Therefore the classification of all k-state Turing machines cannot be completed for arbitrary k, and similar statements can be made for almost all sufficiently general models.

This leaves, however, the interesting question of determining how far formal approaches can be pushed within emergent systems research. While one can easily prove that many problems involving infinite domain are unsolvable, it is not easy to delimit subdomains of solvable or feasible subproblems. For example, it is interesting to predict how far S will be determined. Such predictions are perilous, since, for example, in 1962 Rado felt that no known approach would yield $S(3)$, and that $S(4)$ was "entirely hopeless at present" [15]. Only two years later he and Lin published the solution for $S(3)$ [12], and by 1974 Brady had determined $S(4)$ [1].

In 1983 the largest known lower bound for $S(5)$ (i.e. the largest number of steps taken by a halting 5-state machine yet discovered) was 7707, and the

(a)

State	Symbol read 0	1
1	1R2	1R1
2	1L3	1L2
3	1R1	1L4
4	1R1	1L5
5	1L0	0L3

(b)

State	Symbol read 0	1
1	1R2	0L4
2	1L3	1R4
3	1L1	1L3
4	1R0	1R5
5	1R1	0R2

Fig. 10. Busy 5-state machines. (a) A machine that leaves 4098 1's. Discovered by Marxen and Buntrock [4]. (b) A machine that performs 23 554 768 steps before halting. Discovered by Marxen and Buntrock [4].

largest known lower bound for $S(6)$ was 13 488 [2]. In 1985, Uhling showed $S(5) \geq 2\,358\,063$, and $\Sigma(5) \geq 1915$ [7]. In 1989 Buntrock and Marxen [4] discovered that $S(5) \geq 23\,554\,768$ and $\Sigma(5) \geq 4098$ (see fig. 10). Based on Uhling's results, Brady [3] predicted that there will never be a proof of the values of $\Sigma(5)$ and $S(5)$. We are just slightly more optimistic, and are lead to recast a parable due to Erdös (who spoke in the context of determining Ramsey numbers): suppose a vastly superior alien force lands and announces that they will destroy the planet unless we provide a value of the S function, along with a proof of its correctness. If they ask for $S(5)$ we should put all of our mathematicians, computer scientists, and computers to the task, but if they ask for $S(6)$ we should immediately attack because the task is hopeless.

References

[1] A.H. Brady, UNSCC Technical Report 11-74-1 (November 1974).
[2] A.H. Brady, Math. Comp. 40 (1983) 647.
[3] A.H. Brady, in: The Universal Turing Machine: A Half-Century Survey, ed. R. Herken (Oxford Univ. Press. Oxford, 1988) p. 259.
[4] J. Buntrock and H. Marxen, personal communication (1989).
[5] G.J. Chaitin, J. Ass. Comput. Mach. 22 (1975) 329.
[6] G.J. Chaitin, Algorithmic Information Theory (Cambridge Univ. Press, Cambridge, 1987).
[7] A.K. Dewdney, Sci. Am. 251 (2) (August 1984) 19; 251 (5) (November 1984) 28; Sci. Am. 252 (4) (April 1985) 30.
[8] S. Forrest, Physica D 42 (1990) 1–11, these Proceedings.
[9] M. Gardner, Sci. Am. 241 (1979) 20.
[10] M.W. Green, Fifth IEEE Symposium on Switching Theory (1964) p. 91.
[11] R.J. Kopp, The busy beaver problem, M.A. Thesis, Mathematical Sciences, State University of New York at Binghamton (1981).
[12] S. Lin and T. Rado, J. Ass. Comput. Mach. 12 (1972) 196.
[13] D.S. Lynn, IEEE Trans. Computers 21 (1972) 894.
[14] M. Machtey and P. Young, An Introduction to the General Theory of Algorithms (North-Holland, Amsterdam, 1978).
[15] T. Rado, Symposium on Mathematical Theory of Automata (1962) p. 75.
[16] T. Rado, Bell Systems Tech. J. 91 (1962) 877.
[17] T.R.S. Walsh, Ass. Comput. Mach. SIGACT News 14 (1982) 38.