A New-Millenium Attack on the Busy Beaver Problem*

Kyle Ross, Owen Kellett, Bram van Heuveln, Selmer Bringsjord
Rensselaer AI & Reasoning (RAIR) Lab
Department of Cognitive Science
Department of Computer Sciencee
Rensselaer Polytechnic Institute (RPI)
Troy NY 12180 USA
{rossk2,kelleo,heuveb,selmer}@rpi.edu

draft 11.06.03 last updated by Owen

Abstract

Various computationally-based approaches to making progress on Rado's Σ function (the Turing-unsolvable "Busy Beaver" Problem: BBP) have been taken: brute force searches, genetic algorithms, heuristic behaviour analyses, etc. As a result, candidate BBP champions have been reported with a high degree of confidence for both the quintuple and quadruple formulations of the problem. However, what these previous research efforts lack is a definitive proof which explicitly confirms that these candidate machines are in fact Busy Beavers. The present paper proposes and explores the merits of combining tree-normalization search techniques with specific non-halt detection routines to explicity confirm BBP for small values of n. The start of our planned multi-year attack on BBP, this work establishes a foundation for exploiting a form of distributed computation used previously at our institution on the twin-prime problem and provides a fertile testbed for exploring both "visual" reasoning and possible super-Turing computation.

1 The "Busy Beaver" Problem

Rado's (1963) Σ ("Busy Beaver") function has become a classic focus of study in the theory of computing. Although certainly directly related to the halting problem (Church 1936), BBP poses an alternative formulation of the concept of non-computability for Turing machines: given a fixed-size alphabet and a limited number of internal states, create the most "productive" Turing machine that halts when run on an empty (all-0) input tape; $\Sigma(n)$ denotes the maximal productivity among all n-state Turing machines.

Lin and Rado (1964) define the problem:

> Consider, for a fixed positive integer n, the class K_n of all the n-card [state] binary [alphabet] Turing machines ... Let M be a Turing machine in this class K_n. Start M, with its card 1 [i.e. in the start state], on an all-0 tape. If M stops after a while, then M is termed a valid entry in the BB-n contest ... and its score $s(M)$ is the number of 1's remaining on the tape at the time it stops ... [the set of s-values] has a (unique) largest element which we denote by $S(n)$ ... It is practically trivial that this function $S(n)$ is not general recursive [i.e. is non-computable]... [but] it may be possible to determine the value of $S(n)$ for particular values of n.

Lin and Rado also define a function $\sigma(n)$ which is closely related to $\Sigma(n)$: given all valid entries in the BB-n contest (i.e., machines that halt after one or more steps), consider each machine in this set and let $\sigma(n)$ be the maximal number of shifts required for all machines to halt.

In the Lin and Rado formulation of the problem, methods for computing the problem are phrased in very abstract terms. It is our starting suggestion that Busy Beaver be viewed as a search-based optimization problem. The search space for BB-n is the set of all n-state Turing machines and an optimal machine (i.e. the machine that the search aims to "find") is a machine which, upon halting, contains precisely $\Sigma(n)$ 1's on its tape. This terminology (viz., the problem as a search) will be used throughout the remainder, and viewing Busy Beaver as such will facilitate understanding of our discussion.

[*] We are indebted to .... Part of this research is being suppored by the National Science Foundation (award # 0323324).

2 Turing Machine Formalization

There are two distinct Turing machine transition formulations: the quintuple (as defined by Turing) and the quadruple.[1]

For us herein, a Turing machine $M = (T, C)$ can be defined defined as a two-way infinite tape, T, and a finite state sequential control mechanism, C.

The former comprises a sequence of *cells*, or squares, and a *read-write head*. For idiomatic purposes, we term the square currently being scanned by the read-write head the *current square* and the symbol in this square the *current symbol*.

The read-write head can perform the following four operations:

1. 0 (write a 0 onto the current square)
2. 1 (write a 1 onto the current square)
3. L (move one square left)
4. R (move one square right)

For our work, the latter is defined by the quintuple $C = (Q, \Gamma, \delta, q_0, F)$ where Q is a nonempty finite set of *states*, Σ is a nonempty finite *alphabet*[2], δ is a mapping called the *transition function*, $q_0 \in Q$ is the *start state* and $F \subseteq Q$ is the set of *final states* (i.e., halting states).

δ is defined for quintuple transition Turing machines as a transition from the current state of C and the current symbol on T to a new state of C, a new symbol to be written on the current square, and a move of the read-write head (either to the left or to the right),[3] that is, δ is a partial function

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

For quadruple transition machines we modify the definition slightly: δ is defined similarly, except that only a write or a move, but not both, is allowed for any transition,[4] that is, δ is a partial function

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \cup \{L, R\}$$

It can be shown[5] that for any quintuple Turing machine, M, there exists an equivalent quadruple Turing machine M'; for the remainder of this paper, we consider only quadruple transition machines.

The operation of a Turing machine is as follows, assuming we have variables $q_{current}$ representing the current state of the finite control mechanism, C, and $r_{current}$ representing the current symbol on the tape, T: while $r_{current} \notin F$ and $q_{current} \times r_{current} \xrightarrow{\delta} q_{new} \times a$ for some $q_{new} \in Q$ and some $a \in \Gamma \cup \{L, R\}$ (i.e., while δ is defined for the current state and the symbol currently being scanned), perform the action defined by a and set $q_{current} \leftarrow q_{new}$.

3 Variants of the Busy Beaver Problem

There are a large number of variants of the Busy Beaver problem. For the purposes of this paper, only those based on a binary alphabet (viz., $\{0, 1\}$) will be considered. Eight different formulations of the problem can be created through combinations of the values of three variables: transition type (discussed supra), halting type, and output restriction type.

3.1 Halting Type

In the theoretical formulations of Turing machines (both quadruple and quintuple) described in sect. 2, transitions to $q \in F$ (final states) are exactly as normal transitions: given a current state and current symbol, such a transition involves either a write or a move (but not both) in the quadruple formulation and both a write and a move in the quintuple formulation. This describes the explicit halting machine.

The alternative is implicit halting in which a transition to a halt state involves neither a write nor a move. In many discussions, this is phrased without a halt state: the machine halts when it is in a state q reading symbol r and δ is undefined for $q \times r$. In this paper, a halt state is used to provide uniformity[6] between explicit and implicit formulations and because a halting state simplifies discussion without any semantic alteration to the Turing machines.

It is important to note, for the purposes of studying the Busy Beaver problem, that an n-state Turing machine has n states *plus* exactly one halt state (i.e. $|F| = 1$ and the halt state is not counted as one of the machine's states); we shall call this state H. The

[1] See (Révész 1983, Linz 1997) for a much more complete discussion of Turing machines and formal languages.

[2] Most formal definitions of Turing machines make a distinction between the input alphabet and the working alphabet; for our purposes, both are the set $\{0, 1\}$, so we omit the additional parameter.

[3] Traditionally, when a transition is applied, the write is performed, then the move of the read-write head, and, finally, the transition to the new state of the finite state control mechanism.

[4] Again, the action (i.e., either a write or a move) occurs prior to the state-change of the finite state control machine.

[5] See (Ross 2003) for a constructive proof.

[6] The benefit of this uniformity is that generic discussion—discussion applicable to both implicit and explicit machines—can be made regarding the problem at hand.

Turing machines considered in our research have the restriction that there can be no outgoing transitions from the halt state—when the machine is in the halt state, it cannot be run any further.[7]

3.2 Output Restriction Type

The final variable that gives rise to variants of the problem is one placed only on machines that have halted—an additional "candidacy requirement." In the standard position formulation, a machine must be halted with its read head scanning the leftmost of a contiguous series of 1's on the tape and, except for this contiguous series, there must be no other 1's on the tape.[8] In the non-restricted formulation (which this paper calls the "non-standard" formulation), any machine that has halted is a candidate for the Busy Beaver contest; this is, of course, a proper superset of the candidates that would be considered under the standard position requirement.

4 Taxonomy of the Busy Beaver Functions

Most previous work on the Busy Beaver problem has dealt with some variant of the binary alphabet quintuple formulation. There has, however, been some previous study of several of the quadruple variants. The following summary of that work and the terminology referring thereto is adapted from (van Heuveln et al n.d.).

- Boolos and Jeffrey (1989) have studied the quadruple, implicit halt, standard position formulation. It was in part this work that inspired the present research. Busy Beaver maximal productivity numbers for this formulation will be denoted $B(n)$ and maximal shift numbers will be denoted $b(n)$.
- A Portuguese group (Pereira, Machado, Costa & Cardoso n.d.) used a combination of genetic algorithms and hill-climbing techniques to study the quadruple, explicit halt, standard position variant of the problem. We call this function $P(n)$ and the associated shift function $p(n)$.
- A German group (Oberschelp, et al.) used probabilistic reasoning

  to research the quadruple, implicit halt, non-standard formulation. We define $O(n)$ and $o(n)$ to represent the productivity and shift champions, respectively, for this variant.
- The quadruple analogue to Rado's original problem (quadruple, explicit halt, non-standard) has not, to our knowledge, been studied previously, but we use $R(n)$ (for Rado) to denote the productivity champions for this version and $r(n)$ for the shift champions.

5 Difficulty of the Problem

Calculating Busy Beaver numbers is *very* difficult. For $n = 1$, the calculation is trivial, but not so for even $n = 2$. The following discussion is adapted from that in (van Heuveln et al n.d.). The difficulty of computing values for $\Sigma(n)$ derives from 3 sources: the non-computability of $\sigma(n)$, the enormous search space, and the non-computability of the halting problem.

First, an n-state Turing machine that halts (such a machine will henceforth be called an "halter") may take an inordinate number of steps to do so, as will be seen. Notice that for any value of n if $\sigma(n)$ were to be known, $\Sigma(n)$ would be computable via the following method: run each machine M for $\sigma(n)$ transitions. If the machine has not halted, it is known to be a non-halter and can be discarded. For the machines that *have* halted, find the maximal number of 1's on any of their tapes; this number is $\Sigma(n)$. For large values of n, if $\sigma(n)$ were known, it would *still* take a great deal of computational power to find $\Sigma(n)$ since $\sigma(n)$ grows quickly enough to *itself* be non-computable.

The second source of difficulty is the enormous size of the search space. For explicit formulations (viz. $P(n)$ and $R(n)$), $|M(n)| = (4n + 4)^{2n}$.[9]

Third, and most menacing, in order to evaluate a Turing machine (i.e., to determine if a machine is a valid BB-n candidate), one must find out whether the machine halts or not; this is precisely the halting problem described in (Church 1936)—a problem which is provably non-computable. For small values of n, halt-detection heuristics may be used to classify machines as halters or non-halters, and for slightly larger values of n, careful examination of the machines can determine halting behaviour.

[7] It may seem intuitive that there cannot be outgoing transitions from the final state, but many theoretical formulations allow leaving the accept or reject states.

[8] This is the method by which natural numbers are conventionally represented as input to and output from binary alphabet Turing machines.

[9] The derivation of this formula is as follows: there are a total of $2n$ possible transitions in an n-state binary machine (i.e. 1 per state per read symbol); for each of these, there are 4 possible actions times $n+1$ next states (including n internal states and the halt state). For implicit formulations (viz. $B(n)$ and $O(n)$), $|M(n)| = (4n + 1)^{2n}$. This formula is like that for the explicit formulations except that there is only one possible halt transition rather than four.

6 Conjectures

Maybe insert the table of previously-known results here(?)

I'm not sure how far out on the limb we're willing to go; I'll save this section until after we've discussed and agreed on this matter.

7 Optimization Methodologies

If we were to attempt the problem through a naïve brute-force approach, it would be entirely impractical to compute even small Busy Beaver values. For example (based on the formula from sect. 5): for $B(6)$ there would be $(4(6) + 1)^{2(6)} \simeq 5.96 \times 10^{16}$ machines to consider. The problem can, however, be partially simplified through a group of reductions that decrease the redundancy of the search space. The following discussion looks at the types of redundancy and presents optimizations to eliminate them from the search space being considered.

7.1 Tree Normalisation

7.1.1 Isomorphic Machines

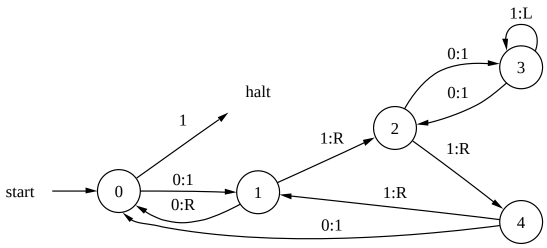


Figure 1: B(5) Champion

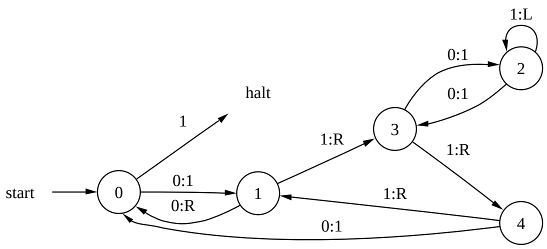


Figure 2: B(5) Champion Isomorph

The Turing machines illustrated in fig. 1 and fig. 2 are identical to one another except that states 2 and 3 have been interchanged. Clearly, the machines are functionally equivalent. For any n-state Turing machine there are $n!$ isomorphic machines (since there are $n!$ permutations of the n state-numbers), but we need to consider only one of these since their behaviour will be identical.

7.1.2 Unused-State Machines

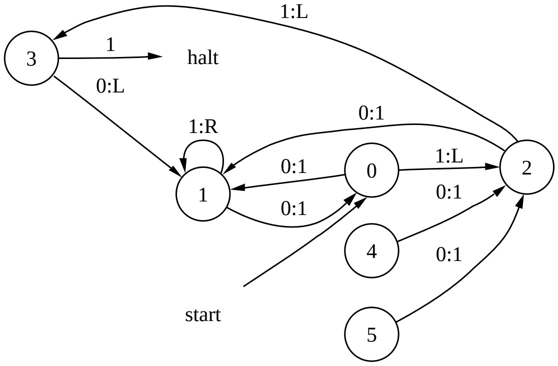


Figure 3: A 6-state Turing Machine with 2 unused States

The Turing machine illustrated in fig. 3 is a 6-state machine but only four of the states are reachable (i.e. because there are no transitions terminating in either state 4 or state 5). For every n-state machine that has 1 unreachable state, there is an equivalent machine with $n - 1$ states, so no such machines need to be considered for our search to be exhaustive.

7.1.3 Tree Normalisation Schema

Figure 4: O^{th} Level (Root) of the Normalised Tree

Partially- and Fully-Defined Machines For any Turing machine, it is either the case that there exists one or more transitions to the halt state or there does not exist such a transition. A partially defined machine is a machine that does not have a halt transition (i.e., a transition wherein the halt state is the "next state" portion of the transition); a fully defined

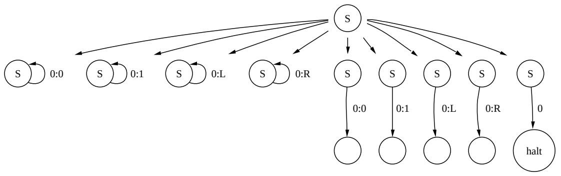


Figure 5: 0th and 1st Levels of the Normalised Tree

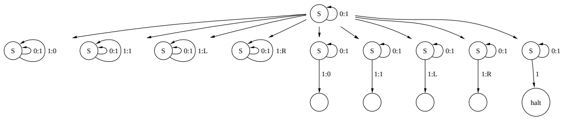


Figure 6: Part of the 2nd Level of the Normalised Tree

machine is a machine that *does* have a transition to its halt state. The importance of the distinction between partially- and fully-defined machines will become evident during the following discussion of the tree normalisation method.

Tree Normalisation Schema One solution to the problems posed by isomorphs and unused-state machines is a generative method known as tree normalisation. The root level of the tree (fig. 4) is a single-state machine with no transitions defined; this, of course, is a partially-defined machine.

To continue generating the tree, choose an unexpanded internal node representing a normalisation candidate. Such a machine is characterised by the following behaviour: when run on an all-0 input tape,[10] the machine halts after one or more transitions and *is not* in the halt state at such time. In order to determine if a machine is a normalisation candidate or not, we must determine whether the machine halts or not. The heuristic mechanisms that we employ for non-halt detection are described in sect. 8.

Assume that we have a Turing machine M which is intended to be a candidate in the BB-n contest; further assume that M has been determined to be a normalisation candidate via the method previously discussed, that it currently has m states, and that these states are labelled $M_0, M_1, \dots M_{m-1}$. Run the machine until it halts in state C_i reading symbol R_i. Let $s = \min(m+1, n)$. M will have precisely $4s+1$ children if it is an implicit halt machine or $4s+4$ children if it is an explicit halt machine. For each state M_i (where $i \in \{0, 1, 2, \dots m\}$) of the machine create a child of M with the additional transition $C_i \times R_i \overset{\delta}{\rightarrow} M_i \times 0$, a child with the transition $C_i \times R_i \overset{\delta}{\rightarrow} M_1 \times 1$, a child with the transition $C_i \times R_i \overset{\delta}{\rightarrow} M_i \times L$, and a child with the transition $C_i \times R_i \overset{\delta}{\rightarrow} M_i \times R$. This defines a total of $4m$ children. If $m < n$ then create 4 children of M with an *additional* state M_m. For each of these children distribute one of the following transitions: $C_i \times R_i \overset{\delta}{\rightarrow} M_m \times 0$, $C_i \times R_i \overset{\delta}{\rightarrow} M_m \times 1$, $C_i \times R_i \overset{\delta}{\rightarrow} M_m \times L$, $C_i \times R_i \overset{\delta}{\rightarrow} M_m \times R$. We have now defined $4s$ children. For implicit halt machines, create a single child with the additional transition from C_i reading R_i and terminating in the halt state. For explicit halt machines, create 4 children and distribute one of the following transitions to each $C_i \times R_i \overset{\delta}{\rightarrow} H \times 0$, $C_i \times R_i \overset{\delta}{\rightarrow} H \times 1$, $C_i \times R_i \overset{\delta}{\rightarrow} H \times L$, $C_i \times R_i \overset{\delta}{\rightarrow} H \times R$. We have now enumerated either $4s+1$ children of M if M is an implicit halt machine or $4s+4$ children of M if M is an explicit halt machine. 2 levels of this normalised tree generation are shown in fig. 5 and fig. 6.

It can be shown that the tree normalisation method eliminates both of the problems discussed here (viz., isomorphs and unused-state machines) and that it is complete and optimal—the BB-n productivity numbers created by considering only the machines generated by the tree normalisation method will be equally productive as would be the maximal productivity machine found by exhausting the search space; proofs of these assertions can be found in (Ross 2003).

7.1.4 Empty Tape Machines

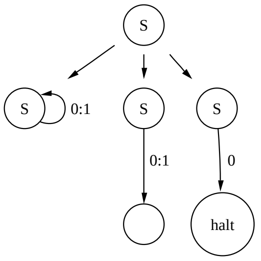


Figure 7: Normalisation Tree (Implicit) Pruned Based on Forcing First Write

Consider a Turing machine M that, after one or more transitions, again has a blank (all-0) tape. Let C_i be the state that M is in at after the *last* such shift is made. Create a machine M' that is identical

[10] We will assume henceforth that machines being run start with a blank input tape so as to avoid redundantly declaring that this is the case.

to M but *starts* in state C_i. Clearly, M' will behave exactly as M does. Since M' has a different first transition than does M, M' is non-isomorphic to M and, based on the completeness of normalisation, will be generated (possibly by proxy through a machine representing its behaviour) at some time during the generation of the tree. Hence, empty tape machines need not be considered.

Notice that machines whose first action (i.e. the action taken from the start state when reading a 0) is *anything* other than “write a 1 on the tape” is an empty tape machine.

To partially solve this problem, we simply *define* the first action taken by any machine to be “write a 1”. This reduces the branching factor between the 0th and 1st levels of the tree from 8 to 2,[11] as shown in fig. 7; compare this against fig. 5.

We also implement the generalised solution to the problem. To do this, we, subsequent to the first transition, simply track how many non-blank symbols are on the tape; if this number ever reaches 0, then we discard the machine and its successors in the tree as blank-tape machines. Since the first move has been defined to be “write a 1”, we are assured that all blank-tape machines have been eliminated from consideration.

7.2 Nonproductive Transitions

7.2.1 Simple Nonproductive Transitions

We term any transition of the form $i \times s \xrightarrow{\delta} j \times s$ a *simple nonproductive transition.* Such a transition reads a symbol s and then writes the same symbol back onto the tape.

It can be shown that for every machine with a simple nonproductive transition generated by the tree normalisation schema, there exists a functionally equivalent machine (that is also generated during construction of the tree) which has no such transition.[12]

7.2.2 Complex Nonproductive Transitions

Another type of nonproductive transition takes the 2-part form of $i \times s \xrightarrow{\delta} j \times s'$ followed by $j \times s' \xrightarrow{\delta} k \times s$ —effectively, the first transition replaces the symbol s with the symbol s' and the second transition reverses this action. We term such transitions *complex nonproductive transitions.*

As with simple nonproductive transitions, it can be shown that a machine’s complex nonproductive transitions can be eliminated from our search without affecting our results.[13]

7.3 Mirror Machines

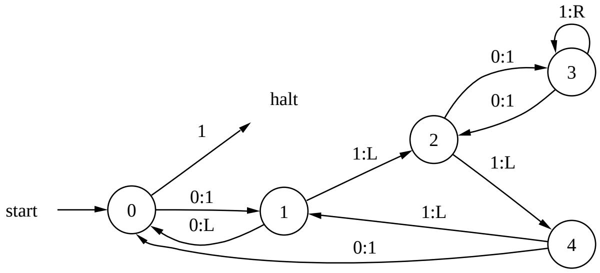


Figure 8: $B(5)$ Champion Mirror

Consider an arbitrary Turing machine M. Replace every left move in M with a right move and every right move with a left move; call this new machine M'. A moment’s thought will reveal that M' behaviour is not isomorphic to M’s but that their productivities are equal and, were we to watch them operate in parallel, the behaviour of M' would be precisely as if we were watching M operate via a mirror. TWo such machines are shown in fig. 1 and fig. 8. Hence, there is no need to consider both M and M' since their behaviour (both in terms of shifts and productivity) is equal—M is a maximal productivity machine for a BB-n contest if and only if M' is; thus it will suffice to consider only one of the two.

We can eliminate mirror machines from consideration if we require that the first move made by any machine during its operation be to the right. This leaves us with the three machines shown in fig. 9 and it is with the substantially pruned tree shown in this diagram that searches for Busy Beaver numbers ought to be started.

8 Non Halt Detection

Clearly the greatest barrier in calculating the value of the Busy Beaver function for a particular value of n using this approach is the determination of whether

[11] We ignore the fully-defined (halting) machines in this count since these will have no children and, thus, have trivial sub-trees.

[12] A proof that this is the case can be found in (Ross 2003).

[13] It can be shown that for every machine with a complex nonproductive transition there exists an equivalent machine with neither simple nor complex nonproductive transitions which will be generated during normalisation.

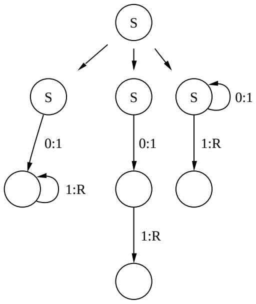


Figure 9: The 3 Start Machines

a machine halts. Despite the existance of the halting problem, as stated in sect. 5, specific non-halt behaviors can be used to classify a subset of machines as non-halters.

The following adapts elements of discussions in both (Machlin & Stout 1990) and (Brady 1983). We use a modified notion from the one found in (Machlin & Stout 1990) to represent Turing machines and their state at particular points in time: M and M^c are used to represent a Turing machine and its corresponding mirror machine (see sect. 7.3) respectively. A *word* or *tape component* is defined as an arbitrary length continuous sequence of characters on the tape and is represented by $[X]$[14]. If a machine is in state s, it is represented on the tape as a subscript to indicate its location: $[_sX]$ represents a machine in state s reading the left most symbol of X; $[X_s]$ represents a machine in state s reading the right most symbol of X; $_s[X]$ represents a machine in state s reading the symbol directly to the left of the leftmost symbol of X; and likewise $[X]_s$ represents a machine in state s reading the symbol directly to the right of the rightmost symbol of X. 0^* represents an infinite sequence of 0's and $[X]^i$ represents a sequence of i X's concatentated together.

8.1 Back Tracking

The objective of the backtracking non-halt detection algorithm is to prove that a machine can never reach a set of conditions in which it does, or could potentially halt. Given the particular definition of a Turing machine used in the Busy Beaver problem (n-states defined on a binary alphabet), there are exactly $2n$ possible {state, symbol} pairs for which a transition can possibly be defined. In addition, the only possible way in which a machine can halt is if it reaches a set of conditions (represented as a {state, symbol} pair) for which the transition is either a transition to the halt state, or in the case of partially defined machines, does not exist at all.[15] The backtracking algorithm, therefore, iterates through every {state, symbol} pair which has one of these two properties and "backtracks" to see if it is possible to reach these conditions.

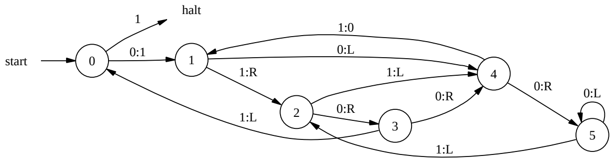


Figure 10: Back Tracking example

8.1.1 Back Tracking Example

Consider the machine in fig. 10. Given the synopsis of the backtracking algorithm stated above, we immediately turn our attention to those {state, symbol} pairs for which there is either a transition to the halt state defined or no transition at all. This particular machine is fully defined, so there are therefore no pairs for which there is no transition. There is one transition to the halt state however, (in state 0, reading a 1) and therefore our test set of conditions contains only one pair, namely {0, 1}

A simple visual analysis of this machine induces the realization that if the machine is to halt, at some point during execution, the following must hold:

- the machine must be in state 3
- the machine must be reading a 1 at the read head
- a 1 must be to the direct left of the read head on the tape
- we combine these three conditions into a single representation of the tape at this time: 11_3[16]

[14] any symbol can be used to replace X in this word and naturally the same holds for any other mentioned component of the machine

[15] In the case of partially defined machines, if a transition does not exist for a particular {state, symbol} pair, a child machine could easily be created with a transition to the halt state defined on that pair

[16] From herein, we will represent partial tape configurations in this format. Namely, the contents of the tape is some sequence of 1's and 0's, and the current state is represented as a subscript at the position of the current read head on the tape

This is so because these conditions qualify as the only conditions which will induce a transition into state 0, reading a 1 on the tape. We therefore have "backtracked" to the pair {3, 1}. Applying the same procedure to this pair as we applied to {0, 1} above, we establish the new state: 0_21, which the machine must enter at some point during execution.

At this point, however, we must undergo an additional step to confirm whether or not these conditions will in fact lead to the halt state. Running the machine for one step on this partial tape configuration yields yet another partial tape configuration: 01_3. This configuration, however, does not match the partial configuration 11_3 from which we originally backtracked. 01_3 is the only tape configuration which will induce a transition into state 3 reading a 1. However, it does not produce the correct tape configuration necessary to continue on to the halt state. The machine, therefore, can never reach the halt state and is a proven non-halter.

8.1.2 Back Tracking Formalization

Now that we have seen how the backtracking algorithm works in practice, we can define the concrete algorithm implemented in our program:

```
foreach {state, symbol} pair $a$ in the set of pairs as
    specified in sect. 8.1, do the following:
        1. Construct a local tape configuration $x$
           which must exist in order for the machine
           to perform the transition defined for $a$
        2. Execute (3) substituting $a$ for $b$ and $x$ for $y$
        3. Let $b$ be a {state, symbol} pair parameter
           and $y$ be a local tape configuration parameter
           foreach {state, symbol} pair $c$ for which
               there is a transition defined that terminates
               in the state of $b$, do the following:
               Construct a local tape configuration
                   $z$ such that
                   a) the transition defined for $c$ will
                   be performed on this tape, and
                   b) after the transition is performed,
                   the resulting local tape matches $y$
               if such a tape cannot possibly be created,
                   return false
               else Execute (3) substituting $c$ for $b$
                   and $z$ for $y$
if every {state, symbol} pair in the above set produces
    a false return value when executing (3),
    the machine cannot possibly halt and is therefore
    classified as a non-halter.
else the proof fails and the results are inconclusive.
```

Now as Machlin and Stout so gracefully put it in (Machlin & Stout 1990): "While backtracking can be useful, it cannot be guaranteed to always stop since otherwise it would supply a solution to the halting problem." As a result of this problem, we are forced to specify a step limit pertaining to how far the procedure is allowed to "backtrack." If this limit is reached, the results are also inconclusive.

8.2 Subset Loops

The subset loop detection heuristic is perhaps the simplest of the non-halt routines because it requires absolutely no knowledge of the input tape or execution of the machine. A representation of the state/transition diagram is all that is necessary. Formally, a machine can be classified as a subset loop if all of the following hold:

- There is a set of states s such that for each state in s, a transition for each symbol in the alphabet (in our case just the binary alphabet {0,1}) is defined.
- Every transition defined from a state in s terminates in a state in s.
- At some point during execution, the machine enters one of the states in s.

This definition is very intuitive and it is easy to see that setting $s = \{1, 2, 3, 4, 5\}$ in fig. 11 satisfies the first two conditions. The third condition, however, on the surface appears to require execution of the machine to confirm. This is not the case:

Recall from sect. 7.1.3 our tree normalization schema. Because of the mechanisms employed with this approach, for every machine generated by our solution, all defined transitions are guaranteed to be used at some point during execution. As a result, if a set s such as the one above exists, at some point during execution each transition defined from a state in s is guaranteed to be used. The third condition, therefore, trivially follows from this fact. In the interests of preventing redundancy, refer to sect. 7.1.3 for additional clarification.

8.3 Simple Loops

Generally speaking, a Turing machine that can be classified as a simple loop moves the read head in

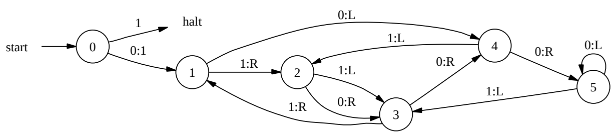
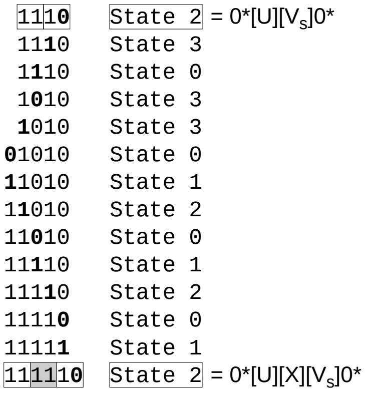


Figure 11: Subset Loop machine



11|1**0**	State 2	= 0*[U][V$_s$]0*
11**1**0	State 3	
1**1**10	State 0	
1**0**10	State 3	
1010	State 3	
01010	State 0	
11010	State 1	
1**1**010	State 2	
11**0**10	State 0	
11**1**10	State 1	
111**1**0	State 2	
1111**0**	State 0	
1111**1**	State 1	
11|11|1**0**	State 2	= 0*[U][X][V$_s$]0*

Figure 12: Christmas Tree execution

a generally leftward or generally rightward direction in some infinite, repeatable fashion. More formally, a Turing machine M is classified as a simple loop if it, or its corresponding mirror machine M^c has the following properties:

1. At some point a during execution, the following tape configuration is reached: $0^*[C][X_s][Y]0^*$.

2. One of the following properties holds:

   - The same tape configuration is reached at some later point.

   - The following tape configuration is reached at some later point b: $0^*[C][V][X_s][Y]0^*$ *and* Between points a and b, the read head never moves past the left edge of the initial X (which would incidentally be the same position as the left edge of the resulting V.

The first case is quite simple to grasp. If the tape, read head, and current state are all identical at two different points during execution, it is an obvious infinite loop and will never halt. The second case is similarly intuitive. Consider the final condition which states that the read head may never move past the left edge of the initial X. Because of this, it is clear that the machine will iteratively generate additional V elements ad infinitum.

8.4 Christmas Trees

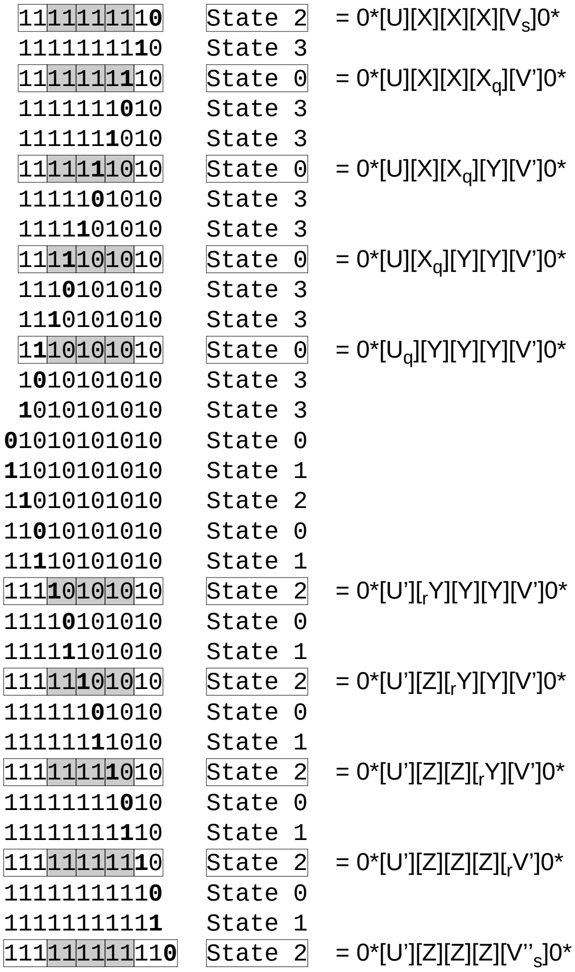
"Christmas Tree" machines, studied rigorously by both Brady and Machlin and Stout in (Brady 1983) and (Machlin & Stout 1990) respectively, are classified as non-halters due to a repeatable back and forth sweeping motion which they exhibit on the tape. Observably, the pattern of the most basic form of Christmas Trees is quite easy to recognize. The machine establishes two end components on the tape and one middle component. As the read head sweeps back and forth across the tape, additional copies of the middle component are inserted while maintaining the integrity of the end components at the end of each sweep.

Consider the partial execution of a 4-state Christmas tree machine denoted in fig. 12.[17] This illustration represents the state of the tape between two successive right extremum in the machine's back and forth sweeping pattern. Clearly, at the first extremum point, the tape contains two end components (respectively labeled U and V); after one complete sweep, an additional X component is generated while the original U and V components remain intact. Also, the machine is in the same state (2) at both points. Examination of additional successive sweeps yields the finding that this pattern continues to hold. Extraction of this phenomenon alone, however, does not constitute a proof of non-haltingness, it only presents the possibility. Further investigation is required:

8.4.1 Christmas Tree Behavior

Establishing the above behavior satisfies the first step towards "Christmas Tree" detection. Let us now consider the same machine referenced above except at a later point during its execution in fig. 13. At this point we see that the machine has introduced three X components capped by the U and V components respectively on each end. As the read head sweeps across the tape in the leftward direction, it methodically transforms each X component into identical Y components, entering the next X component in the

[17] Each line of this figure represents the tape contents as well as the current state of the machine. The tape is denoted in abbreviated form, under the assumption that the remainder of the tape to the left and to the right consists of infinite sequences of 0's. The character displayed in bold indicates the position of the current read head.



```
 1111111110   State 2   = 0*[U][X][X][X][Vs]0*
 1111111110   State 3
 1111111110   State 0   = 0*[U][X][X][Xq][V']0*
 1111111010   State 3
 1111111010   State 3
 1111111010   State 0   = 0*[U][X][Xq][Y][V']0*
 1111101010   State 3
 1111101010   State 3
 1111101010   State 0   = 0*[U][Xq][Y][Y][V']0*
 1110101010   State 3
 1110101010   State 3
 1110101010   State 0   = 0*[Uq][Y][Y][Y][V']0*
 1010101010   State 3
 1010101010   State 3
01010101010   State 0
11010101010   State 1
11010101010   State 2
11010101010   State 0
11110101010   State 1
11110101010   State 2   = 0*[U'][rY][Y][Y][V']0*
11110101010   State 0
11111101010   State 1
11111101010   State 2   = 0*[U'][Z][rY][Y][V']0*
11111101010   State 0
11111111010   State 1
11111111010   State 2   = 0*[U'][Z][Z][rY][V']0*
11111111010   State 0
11111111110   State 1
11111111110   State 2   = 0*[U'][Z][Z][Z][rV']0*
11111111110   State 0
11111111111   State 1
111111111110  State 2   = 0*[U'][Z][Z][Z][V''s]0*
```

Figure 13: Christmas Tree execution2

same state q each time. Additional X components inserted in the middle, therefore, would be transparently passed over and have no effect on the general behavior of the machine. The same properties hold as the machine sweeps back across the tape transforming each Y component into a Z component, while maintaining the same state of the machine r as it enters each successive Y component.

At the completion of this sweep, however, we find ourselves in a somewhat useless state of $0^*[U'][Z][Z][Z][V_s'']0^*$. We have confirmed that additional X components will generate the same, one sweep pattern; however, this pattern leaves us with all of the X components translated into Z components and two alternate caps on each end. It turns out that the key to the "Christmas Tree" behavior is the make-up of the U' and V'' components after such a sweep as the one shown in fig. 13. At the completion of the sweep, the U' component must be made up of the original U as well as an auxiliary Z_1 component and the V'' component must be made up of an auxiliary Z_2 component as well as the original V. In addition, $[Z_1][Z][Z][Z][Z_2] = [X][X][X][X]$ must hold.

8.4.2 Christmas Tree Formalization

Considering the above behaviors, we are now ready to outline the complete requirements for a machine to be classified as a "Christmas Tree" non-halter as specified by Machlin and Stout in (Machlin & Stout 1990).

> 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 number of steps):
>   - $[X][V_s]0^* \Rightarrow_q [X'][V']0^*$
>   - $[X_q][X'] \Rightarrow_q [X'][Y]$
>   - $0^*[U_q][X'] \Rightarrow 0^*[U'][Y']_r$
>   - $[Y'][_rY] \Rightarrow [Z][Y']_r$
>   - $[Y'][_rV'] \Rightarrow [Z][V_s'']$
> 3. $[U'][Z]^i[V''] = [U][X]^{i+1}[V]$ for all $i \geq 1$.

Note the addition of the X' and Y' components. While they are often identical to the Y and Z components respectively, they allow for a more robust

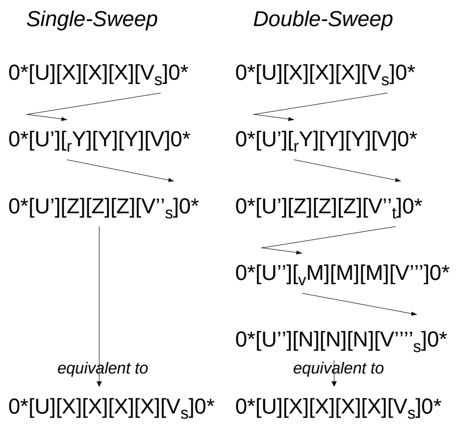


Figure 14: Comparison of cycles between single-sweep and double-sweep Christmas trees

transformation from X cells to Y cells and from Y cells to Z cells. This increases the scope of the detection routine.

8.5 Christmas Tree Variations

8.5.1 Multi-sweep Christmas Trees

Multi-sweep Christmas Trees exhibit the same "Christmas Tree" like behavior as the above described Christmas Trees. As their name suggests, however, these machines require multiple sweeps back and forth across the tape before a repeatable pattern is established. Consider the basic cycle of a single sweep Christmas Tree: an arbitrary length sequence of X's is capped on each end by a U and a V component with the read head on the right edge of V. The machine sweeps across the tape to the left, converting each X into a Y, and on the return trip, converts the Y's into Z's. This completes one cycle as the resulting configuration is equivalent to the original configuration with an additional X inserted into the center.

Fig. 14 illustrates the difference between the iterative cycle of a single-sweep machine and the corresponding cycle in a double-sweep machine. The first sweep looks nearly identical. However, when the read head reaches the right extremum after converting each Y component into a Z component, notice that it is no longer a requirement that the machine be back in the original state s. This is so because the machine sweeps back across the tape once more, converting Z's into M's and M's into N's before completing a full cycle and returning to the original state s. It is only after these two full sweeps that the machine reaches a state equivalent to the original with one additional X included.

Double-sweep Christmas Tree Formalization It is clear that double-sweep Christmas trees are intimately related to Christmas Trees. In fact, the rules which govern their behavior are essentially identically to two copies of the specification outlined in sec. 8.4.2 concatenated together (as one might expect):

A machine M is a *double-sweep 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, Z', M, M', N, V, V', V'', V''', V'''', U, U', and U'' are nonempty words and q, r, t, u, and v are nonzero states (the symbol $\Rightarrow$ means that M transforms the left-hand side into the right-hand side after some number of steps):
    - $[X][V_s]0^* \Rightarrow_q [X'][V']0^*$
    - $[X_q][X'] \Rightarrow_q [X'][Y]$
    - $0^*[U_q][X'] \Rightarrow 0^*[U'][Y']_r$
    - $[Y'][_rY] \Rightarrow [Z][Y']_r$
    - $[Y'][_rV'] \Rightarrow [Z][V''_t]$
    - $[Z][V''_t]0^* \Rightarrow_u [Z'][V''']0^*$
    - $[Z_u][Z'] \Rightarrow_u [Z'][M]$
    - $0^*[U'_u][Z'] \Rightarrow 0^*[U''][M']_v$
    - $[M'][_vM] \Rightarrow [N][M']_v$
    - $[M'][_vV'''] \Rightarrow [N][V''''_s]$
3. $[U''][N]^i[V''''] = [U][X]^{i+1}[V]$ for all $i \geq 1$.

Clearly the first five transformations refer to operations during the first sweep, while the second five transformation refer to operations during the second sweep. In addition, this definition could easily be extended for three sweeps, four sweeps, etc. Our current implementation generalizes the extension allowing us to detect multi-sweep machines of an arbitrary number of sweeps.[18]

8.5.2 Leaning Christmas Trees

Leaning Christmas trees escape the original Christmas tree detection routine because they *lean* in a sense that on each successive sweep, they transpose

[18] Our multi-sweep detection routine requires an argument to denote the number of sweeps to check for. It is not, therefore an all encompassing "multi-sweep detection routine" but instead a mechanism to individually check for double-sweep, triple-sweep, quadruple-sweep, etc. machines as needed.

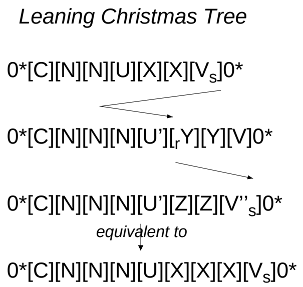

Leaning Christmas Tree
0*[C][N][N][U][X][X][V$_s$]0*
0*[C][N][N][N][U'][$_r$Y][Y][V]0*
0*[C][N][N][N][U'][Z][Z][V''$_s$]0*
equivalent to
0*[C][N][N][N][U][X][X][X][V$_s$]0*

Figure 15: Representative cycle of a leaning Christmas tree

the resulting configuration along the tape in a generally rightward (or leftward) direction. More specifically, imagine a tape configuration identical to that of which begins a sweep in a plain Christmas tree ($0^*[U][V_s]0^*$). Leaning Christmas trees begin with an additional constant component C on the leftmost boundary of the non-zero portion of the tape. After one complete sweep, $0^*[C][U][V_s]0^*$ is transformed into $0^*[C][N][U][X][V_s]0^*$. The machine then follows nearly the same transformation rules specified for Christmas trees. However, when the read head reaches the U component, it transforms a portion of itself into an additional N component. When it returns back to its new right extremum, it has effectively transposed the main components (U, V, and X's) rightward along the tape.

Refer to fig. 15 for further clarification on this leaning tree pattern. The set of transformations for leaning Christmas trees is identical to that of plain Christmas trees with the exception of one. $0^*[U_q][X'] \Rightarrow 0^*[U'][Y']_r$ is intuitively replaced with $[N][U_q][X'] \Rightarrow [N][N][U'][Y']_r$.

8.6 Counters

Counters manipulate the tape in a way such that at particular milestones during execution, dedicated portions of the tape are representative of a binary number. At successive milestones, the number is incremented; thus the machine simulates a binary counter which counts to infinity. Brady studies counters found in the 4-state, quintuple variation of the Busy Beaver problem. Our initial discussion is based closely on these studies:

The 1's and 0's of a binary number are represented as equal length words X and Y respectively in a binary counting Turing machine. In addition, we must establish the notion of an auxiliary word Z which is

```
0          State 0
1          State 1
10         State 2
10         State 2
10         State 1
100        State 3 = 0*[E][cB][B]*
1000       State 4
1001       State 0
1001       State 2
1001       State 2 = 0*[Er][Y][B]*
1001       State 2   checkpoint [1]
1001       State 1
1001       State 3 = 0*[E][cY][B]*
1001       State 4
1000       State 1
10000      State 3 = 0*[E][Z][cB][B]*
100000     State 4
100001     State 0
100001     State 2
100001     State 2 = 0*[E][Zr][Y][B]*
100001     State 2
100001     State 2 = 0*[Er][X][Y][B]*
100001     State 2   checkpoint [01]
100001     State 1
100001     State 3 = 0*[E][cX][Y][B]*
100001     State 4
100101     State 0
100101     State 2
100101     State 2 = 0*[Er][Y][Y][B]*
100101     State 2   checkpoint [11]
100101     State 1
100101     State 3 = 0*[E][cY][Y][B]*
100101     State 4
100001     State 1
100001     State 3 = 0*[E][Z][cY][B]*
100001     State 4
100000     State 1
1000000    State 3 = 0*[E][Z][Z][cB][B]*
10000000   State 4
10000001   State 0
10000001   State 2
10000001   State 2 = 0*[E][Z][Zr][Y][B]*
10000001   State 2
10000001   State 2 = 0*[E][Zr][X][Y][B]*
10000001   State 2
10000001   State 2 = 0*[Er][X][X][Y][B]*
10000001   State 2   checkpoint [001]
10000001   State 1
10000001   State 3 = 0*[E][cX][X][Y][B]*
```

Figure 16: Execution of a counter Turing machine

of equal length to X and Y and is used in the interim conversion of the tape from one binary number to the next. We also require the concept of a "blank" word B which consists of a sequence of 0's equal in length to X, Y, and Z. In this sense, the B word is often identical to the X word. Finally, the milestone as mentioned above comes in the form of an end word E which the read head uses as a checkpoint to begin and end the conversion of the tape from one binary number to the next.

Considering these definitions, a binary counter Turing machine converts an initially blank tape to a sequence that looks like the following: $0^*[E]_c0^*$. Incidentally, this can also be represented as follows: $0^*[E][_cB][B]^*$. It is at this point that our checkpoint has been established and the conversion of the tape begins. In order to satisfy the requirements of a binary counter, the following conversions must hold:

- $[_cX] \Rightarrow_r [Y]$
- $[_cY] \Rightarrow [Z]_c$
- $[Z_r] \Rightarrow_r [X]$
- $[_cB] \Rightarrow_r [Y]$
- $[E_r] \Rightarrow [E]_c$

Brady refers to these c and r states as "carry" and "return" signals. A carry signal sends the read head in a rightward direction along the tape and a return signal sends the read head back to the checkpoint. The transformations specified above guarantee that at each instance when the return signal returns to the checkpoint, the contents on the rest of the tape are representative of the next binary number in the sequence. The final transformation ensures that the checkpoint word E takes the return signal and reflects back a carry signal while maintaining the integrity of itself. Refer to fig. 16 for an illustration of this process.[19]

Counter Modifications Using the above specification (which is identical to that described by Brady) as a basis for our counter detection routine generates some curious results. Several machines whose execution appear to follow the above somehow escape the detection routine and are classified as holdouts. Further investigation reveals that the reflection of a carry signal by an X word generates an incorrect return signal in these particular machines. However, when this signal is sent to the Z word, the signal corrects itself. Similar behavior is observed on a few other machines in which a carry signal sent to a Y word sends an auxiliary return signal to the preceding Z word before receiving the correcy carry signal from the Z word.

This situation can be remedied considering the following truth: when a carry signal is generated, the word immediately preceding the read head at this point is always either a Z word or an E word. Therefore, we can modify the transformations for a carry signal on X and Y by prepending these two possibilities. The first and second transformations defined above are thus replaced with the following:

- $[Z][_cX] \Rightarrow_r [X][Y]$
- $[E][_cX] \Rightarrow [E][_cY]$
- $[Z][_cY] \Rightarrow [Z][Z]_c$
- $[E][_cY] \Rightarrow [E][Z]_c$

This minor modification to the grammar increases the scope of Brady's original grammar as described above.

9 Implementation Methodology

We have chosen C++ for implementing our search programs. This decision was made for several reasons: the language is fairly architecture-independent,[20]

is well-known to be generate some of the fastest executable programs, and provides a robust object-oriented framework in which Turing machines can easily be modelled and simulated.

Depth-first search is the algorithm of choice for generation and exploration of the normalised tree. This algorithm is preferable to breadth-first search because of the lower memory requirements (vis-á-vis temporary storage of to-be-evaluated machines).[21]

Refer to fig. 17 for a graphical representation of the following discussion. We use a stack-based approach, beginning the stack with the three machines enumerated in fig. 9. At each step, the machine M at the top of the stack is popped off and sequentially sent through the non-halt detection routines described above. The order in which the routines are

[19] Each checkpoint in this machine is representative of the next number in a binary sequence if the sequence of X and Y words (i.e. 0 and 1 bits) are reversed. In this sense this machine could be considered a "mirror" counter machine.

[20] This was important during our research because our early tests were run on multiple platforms (i.e., Windows, Solaris, and Mac OS).

[21] See (Russell & Norvig 1994) for a good discussion of search techniques and the memory requirements thereof.

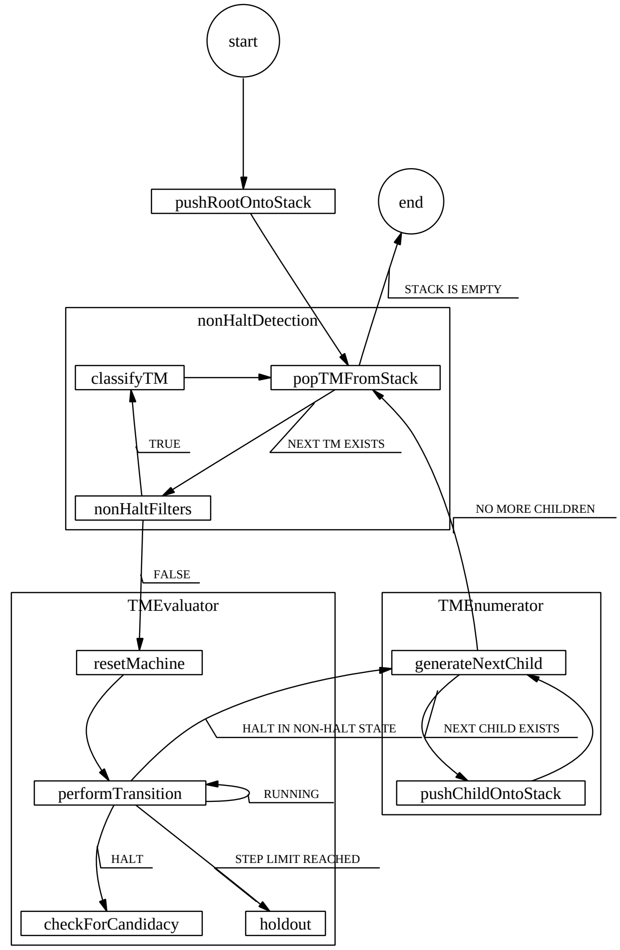


Figure 17: Representation of the program control sequence used in our implementation

applied is based largely on simulation tests to determine which routines are most efficient and which behaviors are most prominant in the search space. We define the sequence as follows:

1. Back tracking
2. Subset Loop
3. Simple Loop
4. Single-Sweep Christmas Tree
5. Double-Sweep Christmas Tree
6. Leaning Christmas Tree
7. Triple-Sweep Christmas Tree
8. Quadruple-Sweep Christmas Tree
9. Quintuple-Sweep Christmas Tree
10. Additional Multi-Sweep Christmas Trees as needed[22]
11. Counter

If any of the above routines confirm M as exhibiting its behavior specification, M is immediately classified as such and discarded. Otherwise, the machine is reset and passed on to the execution stage.

At this point, M is run until such time as it halts or it reaches a predetermined step limit at which time it is determined to be a holdout and not further considered. If M halts, then: If M is a fully-defined machine, its productivity is evaluated and standard positioning taken into account; if this is a new most-productive (or maximal shift) machine, then it is recorded; it is discarded otherwise. If, on the other hand, M is a partially-defined machine then its children in the normalised tree are generated and pushed (in theoretically arbitrary order) onto the stack. The search terminates when the stack is empty, at which time a search equivalent to exhausting the search space has been completed.

Our searches were run on a Macintosh Xserve server running MacOS X version 10.2 with 2.0GB main memory powered by twin 1Ghz PowerPC G4 processors with 256K L2 and 2MB L3 cache, per processor. GCC 3.2 was used with the “-O2” compiler flag.

[22] as n increases, additional sweep Christmas Tree detection routines are and will be required to account for every machine in the search space

n	Tree-Norm	Write-1	Move-R
2	93.94909%	96.40299%	99.13123%
3	99.47388%	99.72263%	99.92049%
4	99.96288%	99.98253%	99.99448%
5		99.99904%	99.99968%

n	s:s	s:s'-s':s	Empty Tape
2	99.31413%	99.40558%	99.40558%
3	99.96057%	99.97131%	99.97284%
4	99.99835%	99.99895%	99.99903%
5	99.99994%	99.99997%	99.99997%
6			99.9999999%

Table 1: Percentage Reduction Factor of Cumulative Optimizations

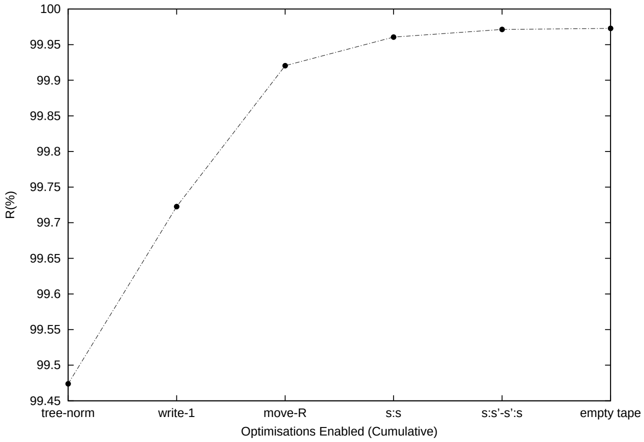


Figure 18: Optimization Efficacy for $n = 3$

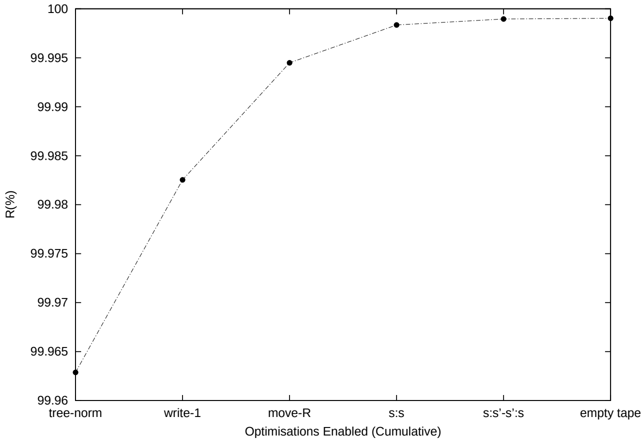


Figure 19: Optimization Efficacy for $n = 4$

10 Results and Records

10.1 Efficacy of Optimizations

Empirical analysis of the optimizations discussed in sect. 7 helps us see how much of the search space has been cumulatively reduced by the addition of each optimization. The optimizations were implemented and tested in a layered approach in the order that they were discussed previously.

Table 1 shows the percentage reduction factor of the search space given the addition of each optimization. These data are identical for all four formulations B, O, P, and R. The label "tree-norm" refers to the basic tree normalisation (sect. 7.1), "write-1" to the forced first action (sect. 7.1.4), "move-R" to the forced first move (sect. 7.3), "s:s" to removal of simple nonproductive transitions (sect. 7.2.1), "s:s'-s':s" to removal of complex nonproductive transitions (sect. 7.2.2), and "empty tape" to the empty tape reduction (sect. 7.1.4).

The percentage reduction factor R is defined as the percentage reduction in the number of Turing machines evaluated: given the size of the entire search space, S, and the number of machines evaluated during a search, M, $R = 100(1 - M/S)$. The last 2 rows of the table (i.e. for $n = 5, 6$) are incomplete because it is impossible to practically run searches with these values of n without a fair number of optimizations in place.

Of particular interest is the case of $n = 6$ with all optimizations enabled. 99.999999% of the search space has been eliminated from consideration which sounds quite impressive. Considering that the full (i.e. completely unoptimised) search space is approximately 5.96×10^{16}, however, reveals that we must—given all of our optimizations and the immense reduction of the search space—evaluate 570 million Turing machines, still a daunting task. Even if we assume that the trend in efficacy from 2..6 continues, performing a search for $n = 7$ would entail, estimating *very* optimistically, evaluating on the order of 3×10^{11} machines.[23]

Fig. 18, fig. 19, and fig. 20 show in graphical form the same data as table 1; as before, R, the reduction factor, refers to the the percentage of the search space that has been reduced by *adding* an optimization to those that have already been implemented.

[23] This number was obtained by calculating the total size of the search space which is approximately 2.98×10^{20} times an (admittedly wishful) optimization factor of 99.9999999%.

Category	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Standard Halt	6	80	2264	103095	6640133
Non-standard Halt	2	76	3844	271497	24911677
s:s-transition	18	469	24425	1872797	189304589
s:s'-s':s-transition	10	237	11428	806981	76717404
Write-1	5	5	5	5	5
Move-R	4	5	5	5	5
Empty-tape	0	8	319	18527	1882827
Back-tracker	23	865	49481	4008364	403910416
Subset loop	0	0	146	11013	2582783
Simple loop	5	130	5605	381736	48492276
Christmas tree	0	2	156	13987	2166668
Double sweep Christmas Tree	0	0	23	2356	419598
Leaning Christmas Tree	0	0	0	69	23129
3-Sweep Christmas Tree	0	0	0	470	77740
4-Sweep Christmas Tree	0	0	0	76	17345
5-Sweep Christmas Tree	0	0	0	0	2156
6-Sweep Christmas Tree	0	0	0	0	1352
7-Sweep Christmas Tree	0	0	0	0	345
8-Sweep Christmas Tree	0	0	0	0	65
Counter	0	0	0	113	25678
Holdout	0	0	0	98	42166
Total	73	1877	97701	7491189	757218357

Table 2: Distribution of Normalized Machines for implicit formulations (B and O)

Category	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Standard Halt	13	229	7224	350979	23328811
Non-standard Halt	12	325	15389	1061240	96749364
s:s-transition	18	469	24425	1872797	189304589
s:s'-s':s-transition	14	286	12881	880534	82182812
Write-1	7	7	7	7	7
Move-R	6	7	7	7	7
Empty-tape	2	28	684	31122	2546483
Back-tracker	23	865	49481	4008364	403910416
Subset loop	0	0	146	11013	2582783
Simple loop	5	130	5605	381736	48492276
Christmas tree	0	2	156	13987	2166668
Double sweep Christmas Tree	0	0	23	2356	419598
Leaning Christmas Tree	0	0	0	69	23129
3-Sweep Christmas Tree	0	0	0	470	77740
4-Sweep Christmas Tree	0	0	0	76	17345
5-Sweep Christmas Tree	0	0	0	0	2156
6-Sweep Christmas Tree	0	0	0	0	1352
7-Sweep Christmas Tree	0	0	0	0	345
8-Sweep Christmas Tree	0	0	0	0	65
Counter	0	0	0	113	24678
Holdout	0	0	0	98	42166
Total	100	2348	116028	8614968	851873790

Table 3: Distribution of Normalized Machines for explicit formulations (P and R)

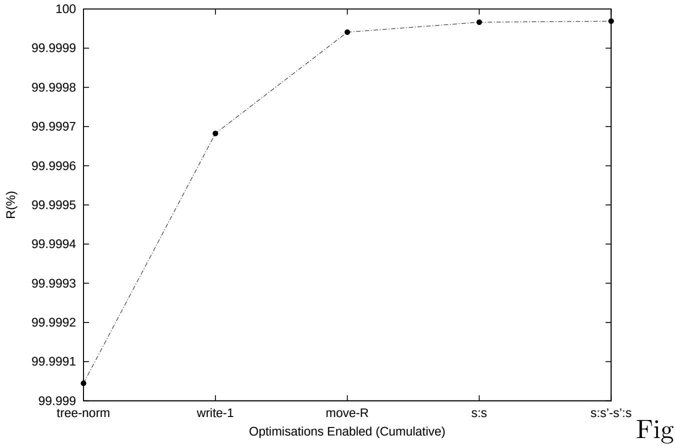


Figure 20: Optimization Efficacy for $n = 5$

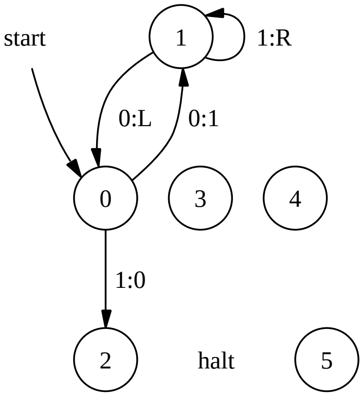


Figure 21: Partially defined machine pruned from the tree as an empty tape machine

10.2 Scope of Non-Halt Detection

Given the substantial reduction of the search space, we now turn to the overall distribution of the resulting set of machines in terms of halters, non-halters (classified according to the algorithms specified in sect. 8), and machines pruned from the tree (according to the rules outlined in sect. 7). Tables 2 and 3 respectively outline the overall statistics for the implicit formulations of the problem (B and O) and the explicit formulations (P and R).

Recall from sect. 9 the implementation methodology for the non-halt detection filter. When a machine is tested for non-haltingness, the detection routines are applied sequentially in the order that they appear in tables 2 and 3. As a result, non-halters are classified according to the routine for which they test positive first. Significant overlap, therefore, is likely among the categories. Regardless, confirmation of membership in any single one of the non-halting classficiations is conclusive evidence to render a machine a proven non-candidate which is sufficient given our specified goal. Additional analysis in this area may be required in order to optimize run-time and possibly re-sequence the non-halt filters when attacking the problem for higher values of n such as 7, 8, and beyond.

In any case, the category of most interest is clearly the Holdout category. Holdouts are machines which have tested negative for all non-halt detection filters, and then have also been run to the pre-defined step limit without halting. For a particular value of n, if the number of these "unaccounted for" machines equals 0, then the candidate champion for this n becomes no longer simply a candidate champion, but a proven Busy Beaver, whose productivity is the confirmed value of BB(n).[24] As is illustrated in Tables 2 and 3, the present effort has confirmed BB(n) for $n = 2, 3, 4$, and very nearly 5.

10.3 Extrapolation of Distribution via. Tree Normalization

The data illustrated in Table 2 describes only a subset of the entire search space for each particular value of n. According to the methods used to generate this subset, every machine not in this set should theoretically be equivalent in behavior to one or more machines in our pruned data. We wish to confirm this, and do so by making use of a representative statistic calculation to extrapolate the observed data set to one which is equal in size to the entire search space.

Consider the 6-state machine illustrated in fig. 21. This machine is pruned from the tree and classified as an empty tape machine before it is discarded. Notice that states 3, 4, and 5 are not even used in the definition of the machine. From the concept of isomorphic machines as outlined in sect. 7.1.1, we can calculate the number of isomorphs that this machine has using a simple permutations function: $Isomorphs(M) = (n-1)P(r-1)$[25] where n is the number of states in the machine M and r is the num-

[24] We generalize the four variants of the problem for simplicity's sake. In this case BB(n) is representative of the particular formulation for which the statistics apply. In general, however, if a result is confirmed for one particular formulation for the value n, it is likely confirmed for all four.

[25] For those who are unaware, the permutations function xPy is defined as $x!/(x-y)!$ where the result is equal to the number of sets of size y elements that can be created from the elements in a set of size x

ber of used states in the machine.[26] Thus this machine has $(6-1)P(3-1) = 20$ equivalent isomorphic machines.

In addition to isomorphic machines, we can also classify each child machine of fig. 21 as an empty state machine as well. To calculate this, for implicit formulations we use $Children(M) = (4n+1)^{2n-m}$, and for explicit machines we use $Children(M) = (4n+4)^{2n-m}$, where n is again the number of states in machine M and m is the number of defined transitions in the machine.[27] Our example, therefore, has $(4 \times 6 + 1)^{2\times 6-4} \simeq 1.53 \times 10^11$ children machines.

Each of the isomorphs also has the same number of children as well; thus the total number of machines which a particular machine pruned from the tree represents is equal to the number of isomorphs times the number of children machines:

$$Represents(M) = (n-1)P(r-1) \times [(4n+1)^{2n-m}]$$

The machine illustrated in fig. 21 is therefore representative of $\simeq 3.05 \times 10^12$ machines.

Tables 4 and 5 respectively represent the extrapolated distribution data for the implicit and explicit formulations of the problem using the above representative calculation statistic. The data is compiled in the same manner as the observed statistics in sect. 10.2, except instead of adding 1 to the necessary category when classifying a machine, its representative statistic using the formula specified above is added. As expected, the sum of the invidual categories for each value of n is equivalent to the size of the entire search space for that n. Our assertion that the normalized data set is a complete and fully representative sample is confirmed.

10.4 Records Set by the Present Effort

At last, we turn to the results of the present effort. Table 6 displays the results of our efforts in addition to those previously known. The records set by the present effort have been marked with asterisks. As mentioned earlier, the values up through $n = 4$ are confirmed as truths due to the classification of every machine in the search space into an identifiable category.

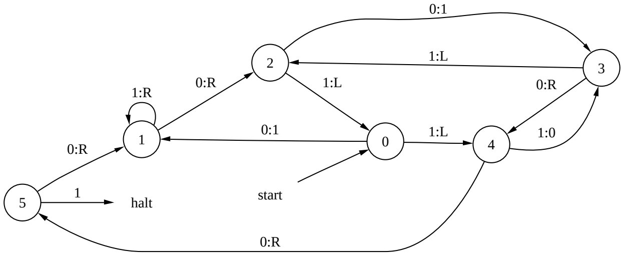


Figure 22: $B(6) - 25$

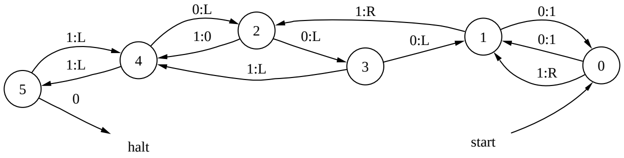


Figure 23: $P(6) - 41$

We also present several Turing machines that are of interest. Several are champions in one of the Busy Beaver contests under consideration—either in terms of productivity or in terms of shifts:

- Fig. 22 is the current world champion for the $B(6)$ and $b(6)$ which our efforts have confirmed. This machine writes 25 contiguous 1's on its tape before halting in standard position after 255 shifts.
- Fig. 23 is a new champion for $P(6)$ found by the present effort; it halts after 841 transitions at which point it is scanning the left-most of 41 consecutive 1's on its tape.
- Fig. 24 is the current $R(6)$ champion revealed by the

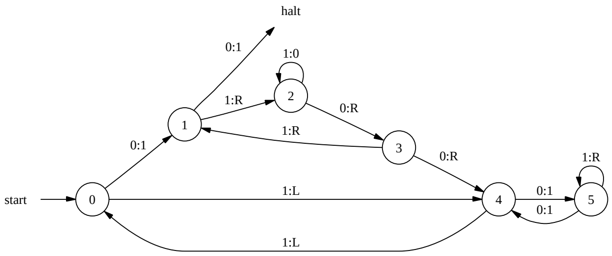


Figure 24: $R(6) - 163$

[26] We subtract 1 from n and r because the starting state is always state 0 and is thus never renamed

[27] These formulas require a similar derivation to those illustrated in sect. 5. For an n-state machine, there are a total of $2n$ possible transitions that can be defined (one for each of the two symbols 0 and 1 on each of the n states). Each of these transitions has 4 possible choices times $n+1$ possible end states. Since m transitions are already defined, we eliminate these possibilities from consideration rendering the formula $(4n+4)^{2n-m}$ for explicit machines and $(4n+1)^{2n}$ for implicit machines.

Category	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Standard Halt	166	90207	103718908	206260328949	631450578198330
Non-standard Halt	18	15032	21405426	48677326680	163683442664210
s:s-transition	1826	1429596	2141852028	5246088511050	19075918871435310
s:s'-s':s-transition	330	266585	405653368	1002627681897	3667246872283790
Write-1	3645	2599051	3693048057	8737080512391	30994415283203125
Move-R	324	257049	386201104	945571484025	3433227539062500
Empty-tape	0	2704	7859610	26709723432	123179236700000
Back-tracker	135	66745	72352399	138322654089	230363405036890
Subset loop	0	0	67116	248518872	13058421488500
Simple loop	117	99788	143484879	327997316304	1270875034985250
Christmas tree	0	52	112200	287787024	1176689708500
Double sweep Christmas Tree	0	0	2346	8485344	374554887500
Leaning Christmas Tree	0	0	0	95256	1552083000
3-Sweep Christmas Tree	0	0	0	236880	1562700000
4-Sweep Christmas Tree	0	0	0	38304	261123000
5-Sweep Christmas Tree	0	0	0	0	6756000
6-Sweep Christmas Tree	0	0	0	0	4056000
7-Sweep Christmas Tree	0	0	0	0	1035000
8-Sweep Christmas Tree	0	0	0	0	195000
Counter	0	0	0	208152	7838415000
Holdout	0	0	0	69552	719346000
Total	6561	4826809	6975757441	16679880978201	59604644775390625
Expected Total $(4n+1)^{2n}$	6561	4826809	6975757441	16679880978201	59604644775390625

Table 4: Distribution of Normalized Machines for implicit formulations (B and O)

Category	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Standard Halt	2070	1305996	1637990190	3449265173568	10998281857655016
Non-standard Halt	342	264574	378013468	873071200752	2984875787381704
s:s-transition	4088	3837006	6380815032	16730013296160	63831699096659136
s:s'-s':s-transition	872	803730	1322214470	3441456618024	13060513915153128
Write-1	12096	9437184	14080000000	34343498022912	124402642012078080
Move-R	864	786432	1280000000	3302259425280	12440264201207808
Empty-tape	2	7256	20959920	74931910032	360142150219680
Back-tracker	210	129214	160885480	338954514360	629351707467304
Subset loop	0	0	90480	396497088	27109195802080
Simple loop	192	205760	338861280	849030572736	3481118924095680
Christmas tree	0	64	166920	491091264	2187363411520
Double sweep Christmas Tree	0	0	2760	11862720	58408052640
Leaning Christmas Tree	0	0	0	119232	2255084160
3-Sweep Christmas Tree	0	0	0	270720	2065355040
4-Sweep Christmas Tree	0	0	0	43776	342414240
5-Sweep Christmas Tree	0	0	0	0	7607040
6-Sweep Christmas Tree	0	0	0	0	4056000
7-Sweep Christmas Tree	0	0	0	0	1035000
8-Sweep Christmas Tree	0	0	0	0	195000
Counter	0	0	0	263808	
Holdout	0	0	0	82944	2250309960
Total	20736	16777216	25600000000	63403380965376	232218265089212416
Expected Total $(4n + 4)^{2n}$	20736	16777216	25600000000	63403380965376	232218265089212416

Table 5: Distribution of Normalized Machines for explicit formulations (P and R)

n	$B(n)$	$b(n)$	$O(n)$	$o(n)$	$P(n)$	$p(n)$	$R(n)$	r(n)
1	1		1		1		1*	
2	2	3	2	3	2	4	2	4
3	3	13*	3	13*	4*	14*	4*	14*
4	5	31*	8	37*	7*	32*	8*	38*
5	≥ 11	≥ 57*	≥ 15	≥ 111	≥ 16*	≥ 112*	≥ 16*	≥ 112*
6	≥ 25	≥ 255	≥ 239*	≥ 41606*	≥ 163*	≥ 27174*	≥ 240*	≥ 41607*
7	≥ 192	≥ 13682						
8	≥ 672	≥ 198339						

Table 6: Our Results

present effort. It halts after 33527 transitions with 163 1's on its tape in a very curious pattern: a single 1 followd by 81 repetitions of 011; the machine halts on the right-most 1. This machine is also the longest running known halting 6-state quadruple Turing machine.

11 Current Runs/Attacks

- currently running ...
- farmer/worker(salsa)
- what's running now with NSF hardware
- $\vdots$

12 The Future of the Busy Beaver Problem

- hypercomputation and the mind. Would we be entitled to infer that the human mind is hypercomputational from continued success? (Selmer)
- Truly chaotic machines, isolating them, etc.
- finding new non-halting patterns? (Bram/Owen/Selmer)
- visual reasoning. Are there better ways to represent the behavior of TMs? Instead of tape records, etc.? (Bram)

References

Boolos, G. S. & Jeffrey, R. C. (1989), *Computability and Logic*, Cambridge University Press, Cambridge, UK.

Brady, A. (1983), 'The determination of the value of rado's noncomputable function $\Sigma(k)$ for four-state turing machines', *Mathematics of Computation* **40**(162), 647–665.

Church, A. (1936), An unsolvable problem of elementary number theory, *in* M. Dave, ed., 'The Undecidable', Raven Press, New York, NY, pp. 89–100.

Lin, S. & Rado, T. (1964), 'Computer studies of turing machine problems', *Journal of the Association for Computing Machinery* **12**(2), 196–212.

Linz, P. (1997), *An Introduction to Formal Languages and Automata*, Jones and Bartlett, Sudbury, MA.

Machlin, R. & Stout, Q. (1990), 'The complex behavior of simple machines', *Physica D* **42**, 85–98.

Pereira, F., Machado, P., Costa, E. & Cardoso, A. (n.d.), Busy beaver: An evolutionary approach. citeseer.nj.nec.com/pereira99busy.html.

Rado, T. (1963), 'On non-computable functions', *Bell System Technical Journal* **41**, 877–884.

Révész, G. (1983), *Introduction to Formal Languages*, McGraw-Hill, New York, NY.

Ross, K. (2003), Master's thesis, Rensselaer Polytechnic Institute.

Russell, S. & Norvig, P. (1994), *Artificial Intelligence: A Modern Approach*, Prentice Hall, Saddle River, NJ.

van Heuveln et al, B. (n.d.), Attacking the busy beaver problem by incorporating the tree normalization method into a farmer/worker scheme.