A MULTI-FACETED ATTACK ON THE BUSY BEAVER PROBLEM

By

Owen Kellett

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF COMPUTER SCIENCE

Approved:

Selmer Bringsjord
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

List of Algorithms

ACKNOWLEDGMENT

First of all, I would like to thank my advisor, Selmer Bringsjord, for his support during my time as a Master's student as well as an undergraduate here at Rensselaer. This research could not have been fully realized without his faith in my work as well as his bold ideas about the powers of the human mind.

Also, I would like to extend special acknowledgement to Bram van Heuveln. Not only did my work on the Busy Beaver problem begin with him as my undergraduate research advisor, but his tireless enthusiasm as the professor of several computability theory and logic courses has proved invaluable to the betterment of my knowledge and maturity in these disciplines.

Additionally, I am indebted to Kyle Ross for his original toils on the Busy Beaver problem. Not only is this thesis inspired by his work, but it is his original computer program that I extend in my research.

Finally, I would like to thank all of my family and friends that have been a part of my life during my time at Rensselear. It is them that have ultimately made this work possible.

ABSTRACT

Consider a binary alphabet Turing Machine which is given an infinite, blank tape as input. If this machine halts, we define its productivity as the number of 1's left on the tape after the machine is run to completion. If it does not halt, the machine is given a productivity value of zero. Now consider all of the binary alphabet Turing Machines that have n states. The machine in this set which has the highest productivity is called a Busy Beaver, and its productivity is the result of the Busy Beaver function $\Sigma(n)$.

This function, originally formulated by Tibor Rado (1963), has become a classical focus in computer science theory as well as research in the nature of computability. Rado (1963) proves that this function is not "computable" in the context that Turing (1936) defines the word. In other words, no Turing machine exists that can definitively solve $\Sigma(n)$ for any arbitrary input n. Regardless, the problem has been attacked on many levels by many different groups in attempts to determine the value $\Sigma(n)$ for small values of n.

Thus in this thesis, we present our own assault on the Busy Beaver function. However, not only do we employ computational techniques as they are defined by Turing in our attack, but we also appeal to certain abilities of the human mind which are grounded in *hyper*computational processes of the physical world. Specifically, we breathe substance into the claim that the human visual reasoning system possesses computational powers beyond the Turing limit, and these powers can be employed in efforts to solve $\Sigma(n)$. As a result, while the Busy Beaver function is proven to be non-*Turing*-computable, our efforts suggest hope that it is still computable by the fantastic capabilities of the human mind.

Our multi-layered assault is thus structured as follows: The core of the attack is built off of ingenuous previous efforts by Kyle Ross (2003). In his work, he incorporates tree normalization optimization techniques to greatly reduce the search space of the set of n-state machines in the $\Sigma(n)$ problem. With this core in place, we incorporate automated non-halt detection mechanisms to certify whether or not machines in this search space halt. The final layer in the attack requires a direct appeal to the human visual reasoning system mentioned above to confirm that the final set of machines do not halt.

Thus the result is that $\Sigma(1)$ through $\Sigma(5)$ are confirmed and a foundation is in place for a continued march upwards. Most importantly, though, is that we break the bounds of Turing-computation, and place the assault on $\Sigma(n)$ into the hypercomputable realm.

CHAPTER 1
Introduction

1.1 Turing, Rado, and the Busy Beaver Problem

Alan Turing is considered by many to be the founder of computer science and computability theory. His Turing machines (Turing 1936)[1] are accepted by many as the basis of all computation. In fact, he suggests a form of this notion in (Turing 1969):

> It is found in practice that L.C.Ms [Turing referred to his Turing machines as "Logical Computing Machines" or L.C.Ms. Present day reference to them as Turing machines suitably use his namesake] can do anything that could be described as 'rule of thumb' or 'purely mechanical'. This is sufficiently well established that it is now agreed amongst logicians that 'calculable by means of an L.C.M.' is the correct accurate rendering of such phrases.

What he refers to here as "rule of thumb" or "purely mechanical" is generally accepted to mean anything that is algorithmic. This concept is solidified in (Copeland 2000):

> A method, or procedure, M, for achieving some desired result is called 'effective' or 'mechanical' just in case
>
> 1. M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
> 2. M will, if carried out without error, always produce the desired result in a finite number of steps;
> 3. M can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;
> 4. M demands no insight or ingenuity on the part of the human being carrying it out.

Thus Turing's claim above suggests that anything that is algorithmic or "mechanical" in this sense can be simulated by some Turing machine.

Incidentally, Alonzo Church, working independently from Turing at the time, makes a similar claim in (Church 1936) regarding his lambda calculus. While he does not reference Turing machines in his work, it is later shown that the lambda calculus is equivalent in expressive power to Turing machines. Thus their combined generalized claim has since become known as the Church-Turing Thesis. What is interesting about this thesis is that it allows Turing to establish a concrete definition of what is "computable."[2] In short, a

[1] Briefly, a Turing machine is a model of computation that operates on an infinite tape of symbols. There is a read head that moves back and forth across the tape, transforming the symbols that it encounters according to a set of rules defined for that particular machine. We shall describe the structure and operation of Turing machines in more intimate detail in section 2.1.

[2] Clearly the thesis allows Church to make the same conclusions but it is Turing that pursues this notion in conjunction with his Turing machines.

"computable" number is one whose digits can be generated by some Turing machine[3] and any computable process is one that, again, can be simulated by some Turing machine.

Armed with this concept of "computability," Turing introduces this well defined and intuitive-to-understand procedure: Given a Turing machine M, and a corresponding input tape t, determine whether or not M halts when it is run on t.[4] This is famously referred to as the Halting Problem. Turing (1936) proves that such a procedure is not "computable" by showing that it is not possible for a Turing machine to exist that effectively gives an answer for any arbitrary input of M and t.

It is this initial work by Turing and his concept of "computable" that has served as a springboard for the basis of computability theory in computer science. To expand upon this notion further, this definition of "computable" can be extrapolated to encapsulate the nature of a "computable" function:

- A function f can be defined as a mapping such that for every whole number n, there is a corresponding value $f(n)$ associated with it.
- A "computable" function is one for which there is some algorithmic or "mechanical" procedure such that given a number n as input, it outputs $f(n)$ for any n.
- As we have already seen, any algorithmic or mechanical procedure can be simulated by some Turing machine.
- Therefore, a "computable" function is one for which there is a Turing machine such that given some representation of the input n on its tape, it performs some set of operations and then halts with some representation of $f(n)$ on its tape.

As can be seen, the halting problem as described by Turing does not easily fit into this paradigm since it does not take a number n as input but instead a representation of a Turing machine and input tape. Thus, Rado (1963) presents a true function in the sense that we have defined that is directly related to the halting problem. He calls it the Busy Beaver function and we summarize it as follows:

[3]Note that this definition does not exclude real numbers with an infinite sequence of digits after a decimal point such as π. A Turing machine can be constructed to simply "churn out" the digits of such numbers indefinitely.

[4]It should be noted here that Turing (1936) also introduces the concept of a Universal Turing Machine. Essentially, he shows that there exists a Turing machine that can be given a representational encoding of any Turing machine along with some representation of an input tape and it will simulate the operation of the given machine when run on the given input tape. Such a machine is called a Universal Turing Machine.

> Consider a binary alphabet Turing Machine which is given an infinite, blank tape as input. If this machine halts, we define its productivity as the number of 1's left on the tape after the machine is run to completion. If it does not halt, the machine is given a productivity value of zero. Now consider all of the binary alphabet Turing Machines that have n states. The machine in this set which has the highest productivity is called a Busy Beaver, and its productivity is the result of the Busy Beaver function $\Sigma(n)$.

Thus the Busy Beaver function takes as input a number n and returns the productivity of the most productive n-state machine. This function is very clearly defined but it is also relatively easy to show that it is not "computable" in the sense that Turing has defined it. In other words, no Turing machine exists that can take as input n and output what we shall refer to as $\Sigma(n)$.[5]

1.2 Computable vs. Turing-computable

Up to this point, the reader may be somewhat fascinated by the history of Turing machines, "computability," and how Rado's $\Sigma(n)$ function grew out of these concepts. However, there may be some confusion as to where this story is heading. Notice, then, the perhaps curious notation that we have been using thus far for each appearance of the word computable, computability, or some other related derivative. We have been including these words in quotation marks, perhaps suggestive that we are apprehensive about accepting the notion that Turing machines encapsulate all things that are truly computable.

It turns out that this is the case. Interestingly enough, however, it is pointed out in (Wegner & Goldin 2003) that Turing, *himself*, may have held these same concerns:

> Turing implied in his 1936 paper that Turing machines (which he called automatic machines, or a-machines) could not provide a complete model for all forms of computation, just as they could not provide a model for all forms of mathematics. He defined c-machines (choice machines) as an alternative model of computation, which added interactive choice as a form of computation; later, he also defined u-machines (unorganized machines) as another alternative that modeled the brain.

Thus, even though Turing defines "computable" as a concept that is directly connected to the power of Turing machines, he appears to concede the possibility that Turing machines do not encapsulate everything that is *truly* computable in the physical world. In other words, all things "computable," as defined by Turing, can be simulated by some Turing

[5] We shall explore the specifics and variations of the Busy Beaver function in greater depths in section 2.2 as well as reference a proof of its "uncomputability" in section 2.3.1.

machine. However, it is not clear whether all processes realizable in the physical world are in fact "computable" in the sense that they too can be simulated by some Turing machine.

It is therefore with this question in mind that we launch our assault on an already mentioned non-*Turing*-computable[6] function: Rado's Busy Beaver function. Specifically, we illuminate evidence that the physical world and consequently the human mind is, in fact, grounded in physical processes that are not Turing-computable. Therefore while the Busy Beaver function is not Turing-computable, it may yet be computable by even more powerful computational machines: humans. The outline of our attack is thus described in the next section.

1.3 Outline of the Attack

Before we flesh out our strategy, it is important to recognize the connection between the Busy Beaver function and the Halting problem mentioned above. Consider the set of n-state Turing machines that $\Sigma(n)$ directly inquires. This set is a finite, enumerable set.[7] Therefore, if the Halting problem was computable, we could easily construct a solution to $\Sigma(n)$:

(1) For each machine in the set of n-state machines, determine whether or not it halts. [This is the application of the halting problem and the crucial step]

(2) If the machine does not halt, discard it since all non-halters have a productivity of 0 according to the definition of $\Sigma(n)$.

(3) If the machine halts, run it to completion and note the number of 1's that are left on the tape. This value is the machine's productivity.[8]

(4) Return the productivity of the most productive machine.

While it is a brute fact that the Halting problem is not Turing-computable, it is trivial to show to that it can be partially solved by Turing machines. In other words, we

[6] Herein we shall refer to Turing's concept of "computable" as Turing-computable. We shall reserve the word computable to mean anything that is computable in the physical world. There are hypothetical computational machines that compute functions above the Turing-limit [i.e. Zeus machines (Boolos & Jeffrey 1989) to name one of many]. However, these machines defy physical possibility and thus are not included in our definition of computable.

[7] We shall present an effective procedure for enumerating this set in section 4.1

[8] It should be noted that there are several variations of the Busy Beaver function that specify certain requirements for the arrangement and pattern of the 1's on the tape after the machine has halted. We shall illuminate these variations in section 2.2. At this point, however, we need not be concerned with such details.

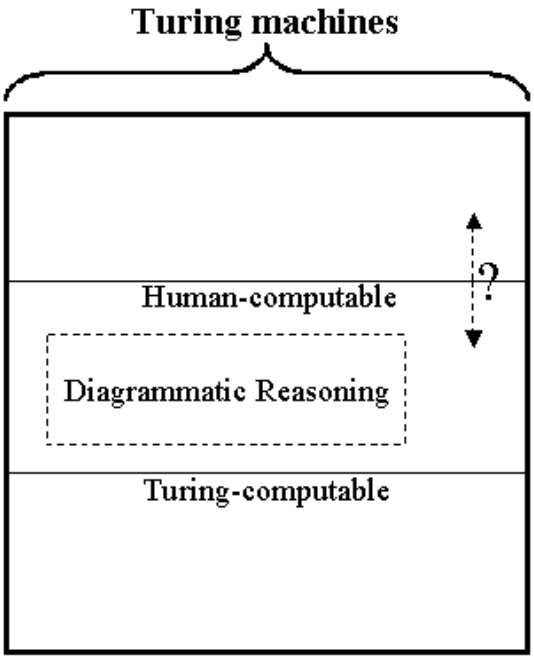


Figure 1.1: Computability Conceptualization

can construct a Turing machine that either definitively says "Yes the machine will not halt" or "I am not sure if the machine will halt or not."[9] Now let us consider the diagram shown in figure 1.1. Assume, for argument's sake, that the outer box represents the set of all Turing machines.[10] Now let us also consider a Turing machine which represents the *best* partial solution to the Halting problem. The area of the outer box below the line labeled "Turing-computable" is the set of all Turing machines that this Turing machine can decide. One of the core claims that we defend in this thesis is that the line labeled "Human-computable" is above the Turing-computable line. In other words, no Turing machine can encapsulate all methods of reasoning available to humans to reason about whether Turing machines do not halt. Therefore, there are some Turing machines that a human can prove will never halt, but some effective, Turing-computable procedure cannot.

Thus with this foundational argument in place, we are now ready to present our far reaching assault on the Busy Beaver function:

$\mathcal{A}1$ While much of the groundwork has already been presented thus far, there are still

[9] See section 2.3.3 for a trivial proof of concept partial solution to the Halting problem.

[10] We realize that this is an infinite set and thus our figure may not be graphically accurate but this is not important to the context of our argument.

significant details about Turing machines and the Busy Beaver function that must be illuminated. Therefore, in chapter 2, we give a more thorough description of Turing machines in general and the specifics of the Busy Beaver function, its variations, and previous assaults that have been made. We also elaborate on our proposed attack procedure that we have already briefly described above.

$\mathcal{A}2$ Building off of ideas originally pioneered by Roger Penrose (1989, 1994), the core of our argument is that humans possess capabilities beyond that of Turing machines and can specifically harness these capabilities to reason about the non-haltingness of Turing machines. Thus in chapter 3, we defend Penrose's claims in direct support of our Busy Beaver assault. Additionally, refer once again to figure 1.1. Notice that we annotate the segment above Turing-computable and below Human-computable as being the result of diagrammatic reasoning. An additional intent in this chapter is to ground humans' ability to reason above the Turing-limit in visual cognitive processes. The anchoring argument here is that visual reasoning and symbolic reasoning are independent processes that cannot be reduced to each other (Bringsjord & Bringsjord 1996, Barwise & Etchemendy 1995). Therefore, since Turing machines are purely symbolic, visual reasoning processes are non-Turing-computable.

$\mathcal{A}3$ Chapter 4 encapsulates our direct attack on the Busy Beaver function itself. The core strategy involves iteratively generating a partial solution to the Halting problem as mentioned above. While this partial solution is a Turing-computable procedure, it only allows us to confirm the value of $\Sigma(n)$ up through $n = 4$. For $n = 5$, we are left with a set of 98 machines that our partial solution answers "I am not sure if the machine will halt or not." It is for these 98 machines that we must appeal to human diagrammatic reasoning processes mentioned above to confirm that they are in fact non-halters. Thus our bold claim is that we have "cracked" $\Sigma(5)$ by human-computable means that lie above the Turing limit.

$\mathcal{A}4$ In a speculative segment, chapter 5, we address further lines of research that our work has laid a foundation for. Most notable, of course, is the upward push to confirm values of $\Sigma(n)$ for $n = 6, 7$, and beyond. However, since our assault directly references diagrammatic reasoning processes employed by humans, our research also provides a test bed for further studies in the complexities of the human visual reasoning system. Additionally, our work has important considerations for studies in

discovering truly hypercomputational processes in nature. Perhaps most intriguing, though, is the quest to push the "Human-computable" line even further up the diagram in figure 1.1. While our work only places it somewhere above the Turing-computable line, if it can be established to lie above all Turing machines, then the possibility that the Busy Beaver function is in fact *computable* still remains.

It should be noted that while we present convincing evidence for human hypercomputational powers, we realize that some readers may still be resistant to accepting our claims. It should appease those readers, though, that it is undisputable from our presentation that further progress on the Busy Beaver function has been and will continue to be made via Turing-computational mechanisms. So without further delay, we now proceed with our multi-faceted attack on the Busy Beaver problem.

CHAPTER 2
Laying the Groundwork

2.1 Turing Machine Formalization

As a precursor to our attack, let us first present a more rigorous definition of the makeup and operation of Turing machines that are briefly described in the preceding chapter.[11] As it is defined by Turing, a Turing machine contains a two-way infinite tape. The tape consists of a sequence of symbols and at any one point in time, one of the symbols is considered to be under the "read-write head" of the machine. Thus consider the tape shown in figure 2.1. In this figure, the read-write head is denoted with a darkened box around the symbol as well as an arrow above the box for clarity. Thus the machine associated with this tape is said to be reading a 0 (the current symbol under the read-write head) on the tape.

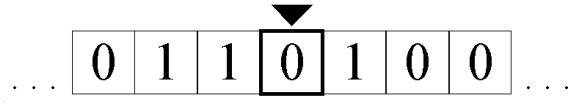


Figure 2.1: Example Turing Machine Tape

In addition to a tape, a machine is also comprised of a set of rules that govern the movement and operation of the read-write head along the tape. Conceptually, these rules can be broken down into two distinct components:

1. A Turing machine consists of a finite set of states. At any one point in time during its execution, a Turing machine is considered to be "in" one of these states. It is important to note that a state can be designated a halting state. If a Turing machine ever enters a state designated as a halting state during its execution, then it intuitively halts.

2. A set of transitions define specific actions to be taken according to the current state of the Turing machine as well as the current symbol under the read-write head of the tape. Each action consists of three parts:

   (a) Write a new symbol at the current read-write head of the tape.

[11] See (Révész 1983, Linz 1997, Fischer 1965, Ross 2003) for a considerably more thorough discussion of Turing machine formalizations.

(b) Move the read-write head one position either to the left or to the right.

(c) Designate a new state that the Turing machine is currently "in."

Thus a Turing machine, governed by the rules of its state and transition sets, manipulates its tape with a read-write head until it either enters a halting state or it enters a state for which there is no transition defined for the current symbol under the read-write head.

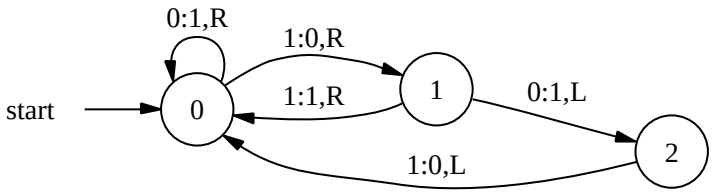


Figure 2.2: Turing Machine Example

Let us now further conceptualize these machines with an example. Consider the flow diagram shown in figure 2.2. The circles in this diagram represent the states of the machine. Each arrow that connects two states represents a transition where the notation of the transition is $[r] : [w], [m]$ where r is the current symbol under the read-write head, w is the new symbol to be written to the tape, and m is the direction to move the read-write head (one of $\{L, R\}$ for left and right respectively). Thus consider the situation where this machine is in state 0 and is operating on the tape shown in figure 2.1. Since the read-write head is positioned over a 0 on this tape, the machine will act according to the transition arrow that exits state 0 and has a 0 as the read symbol r in the transition. Therefore, it will write a 1 on the tape and then move the read-write head one position to the right, resulting in the new tape configuration shown in figure 2.3. It will subsequently transition "into" the state in which this transition arrow terminates (which happens to be the same state 0). Operation continues in this manner until the machine reaches a halt state or a {state, symbol} pair for which there is no transition defined.[12]

With a clear picture of how Turing machines function, we can now present a more formalized definition: For our purposes, a Turing machine is defined by a quadruple

[12] In the context of our example machine, there are no halt states defined. Therefore, the machine only halts if it reaches state 2 while reading a 0 under the read-write head since that is the only {state, symbol} pair for which there is no transition.

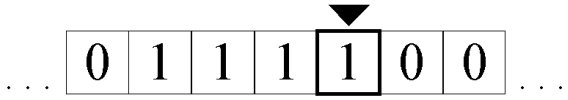


Figure 2.3: Example Tape After Transition

(Q, δ, q_0, F).[13] Here Q is a finite set of states, δ is a set of transitions, $q_0 \in Q$ is the starting state and $F \subset Q$ is the set of states that are designated halting states.

For the definition of Turing machines as we have defined them above, each transition in δ consists of a quintuple:[14] (q_c, r, w, m, q_n). Here q_c represents the current state of the machine, r represents the current symbol under the read-write head, w is the new symbol to be written to the tape, m is the direction that the read-write head is to be moved (as explained above), and q_n is the new state that the machine is to transition to.

For our purposes, however, we wish to use a slight modification of δ where in this case, each transition is a quadruple: (q_c, r, a, q_n). In this modified definition, the action a that the machine takes for each transition is one of $\{0, 1, L, R\}$ where 0 and 1 represent writing a new symbol under the read-write head and L and R represent moving the read-write head one position to the left or right respectively. Thus in the quadruple formulation of Turing machines, each transition can either modify the tape under the read-write head, or move the read-write head one position, but not both as in the quintuple formulation.

The quadruple formulation formulation first appeared in (Post 1947) and it can be shown that quadruple formulation Turing machines are equivalent in expressive power to quintuple formulation machines.[15] Herein we only concern ourselves with the quadruple formulation.

2.2 Busy Beaver problem definition

With a clearer definition of Turing machines established, we can now more formally define the Busy Beaver function or $\Sigma(n)$ as it is described in section 1.1. As has already been mentioned, Rado defines the problem in (Rado 1963). We appeal to a subsequent paper (Lin & Rado 1964) published in conjunction with one of his students for the definition of the problem as they define it:

[13] Many formal definitions of Turing machines include the specification of the alphabet that can be used for characters on the tape. For our purposes, we only consider machines with a binary alphabet of $\{0, 1\}$ and thus do not include it in the definition.

[14] The quintuple formulation of Turing machines follows the original specification as it is described by Turing.

[15] See (Ross 2003) for a proof of this concept.

> 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 [q_0 in our definition], 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 $\sigma(M)$ is the number of 1's remaining on the tape at the time it stops ... the scores of these valid entries constitute a nonempty finite set of non-negative integers, and thus this set has a (unique) largest element which we denote by $\Sigma(n)$ ... It is practically trivial that this function $\Sigma(n)$ is not general recursive [i.e. is non-Turing-computable]... On the other hand, it may be possible to determine the value of $\Sigma(n)$ for particular values of n.

This function is proven to be non-Turing-computable in (Rado 1963). However, this does not deter us in our assault as we have already suggested in the preceding chapter. It should also be noted that Lin and Rado define an additional non-Turing-computable function $SH(n)$:

> ... the determination of the function $SH(n)$ [is] defined as follows. Each valid entry M in the BB-n contest performs a certain number $s(M)$ of shifts by the time it stops; the function $SH(n)$ is the maximum of $s(M)$ for all valid entries in the BB-n contest.

As we shall see, this function has important ramifications on the computable nature of $\Sigma(n)$.

2.2.1 Variants of the Busy Beaver Problem

With the formal specification of Turing machines as we have described them in section 1.1, it easy to see how many variants of the Busy Beaver function could be established by simply modifying the Turing machine formalization used. Recall that we have already restricted our Turing machine focus on those that deal explicitly with binary alphabets of $\{0, 1\}$. Considering this restriction, we can further establish eight different formulations via three different parameters: transition type, halting type, and output restriction type.

2.2.1.1 Transition Type

As is previously discussed in section 2.1, Turing machines are commonly defined via the quintuple formulation where each construction contains five pieces of information (current state, current symbol, new state, new symbol, move direction).[16] Our attack focuses instead only on the quadruple formulation of Turing machines.

[16] Rado's (1963) original definition of the problem deals with the quintuple formulation of Turing machines.

2.2.1.2 Halting Type

Recall from our formalization of Turing machines in section 2.1 that a Turing machine halts execution when it reaches one of two conditions:

1. It enters a state $q \in F$
2. It enters a state q such that there is no transition in δ for the state q and the current symbol under the read-write head r.

For the purposes of defining the Busy Beaver function, we split these two halting mechanisms into two distinct categories.

For the explicit halting mechanism, the set of halting states F is required to have a cardinality of exactly one state. Thus, the machines considered when calculating the value of $\Sigma(n)$ for the explicit halt formulation of the problem contain n states *plus* the one halt state in F.

Conversely, the implicit halt formulation of the problem requires that F have a cardinality of exactly zero states. Therefore, no halt states are defined and the machine can only halt if it encounters a {state, symbol} pair for which there is no transition defined in δ.

2.2.1.3 Output Restriction Type

One final variable that we concern ourselves with is a certain restriction on the final configuration of the tape for those machines that have halted. According to Lin and Rado's original definition, there is no restriction on this configuration and the 1's can appear anywhere on the tape with any pattern. However, typical conventions for interpreting the output tape of a Turing machine require that the read-write head be positioned at the left-most of a continuous sequence of 1's (with no other 1's on the tape) in order for it to be a valid output configuration. Therefore, we can redefine the Busy Beaver function to only consider machines which halt in this standard conventional configuration. All other halting configurations are thus assigned a productivity of 0. For the purposes of this paper, we shall refer to the latter formulation as the standard configuration formulation of the Busy Beaver problem. The original unrestricted version as it is described in Lin and Rado's original definition is herein considered the non-standard formulation. (Ross 2003)

2.2.2 Previous Efforts of Busy Beaver Attacks

It should be noted at this point that our efforts to attack the Busy Beaver function have been partially motivated by the brilliant work done by Kyle Ross (2003). In fact, it is his original attack strategy and research in optimization techniques for the $\Sigma(n)$ search space that form a foundation for the present research for this paper.[17] Thus our focus for this problem extends his work on the four variants of the quadruple formulation of the problem as defined above and we cite his definition of these formulations as they are described in (Ross 2003):

> 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, Schmidt-Gottsch & Todt 1988) 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.

Please note that while we do distinguish between the four variants described above, in situations where the particular variant is not of particular concern, we may still refer to the function in general terms such as the Busy Beaver function or $\Sigma(n)$.

2.3 Difficulty of Problem

2.3.1 Proof of Uncomputability

Before divulging into our proposed search based optimization attack on the Busy Beaver problem, we wish to first discuss an important property that reveals itself in a certain variation of a proof of the uncomputability of the function. Let us first consider the set of all Turing-computable functions that operate on one parameter input value (that

[17] We should also not that we are indebted to Kyle for allowing us use of his original computer programs to form the basis of our assault on $\Sigma(n)$.

is functions that are computable by some well defined Turing machine which takes as input some finite sequence of consecutive 1's on its input tape with the read head positioned at the leftmost position of these contiguous 1's). For arguments sake we shall define this set as set T. In (Miller 2003), Miller shows that the following holds:

$$\forall_{t \in T}(\exists_n(\forall_{j>n}(\Sigma(j) > t(j))))$$

In other words, the Busy Beaver function dominates every existing Turing computable function. We revisit this facet of knowledge later in section 3.1 but at present we leave the reader to ponder what this ultimately means: *No Turing machine can generate a sequence of numbers for which each number in the sequence is larger than the corresponding value in an ordered sequence of Busy Beaver values.*

2.3.2 Mechanisms of Attack

As is already mentioned, the Busy Beaver problem is non-Turing-computable. Thus, a generalized Turing-computational mechanism to compute the value of $\Sigma(n)$ for any arbitrary n is unattainable. However, it is still possible to determine the value of $\Sigma(n)$ for small values of n. This requires a monumental task: a complete and exhaustive search based attack on the complete state space of Turing machines for each value of n. For explicit formulations of the problem ($P(n)$ and $R(n)$), this amounts to a search space size of $(4n+4)^{2n}$.[18]

Even for small values of n, examining a set of $(4n+4)^{2n}$ machines requires a massive amount of computational power even by today's standards. We shall see in section 4.1 that this search space can be dramatically reduced by incorporating a specialized machine enumeration strategy. This technique incorporates tree normalization filters that effectively eliminates machines from the search space by proving that they are equivalent to some other machine in the tree.

```
function: Σ(n)
1  Using a tree normalized approach, enumerate a set S of n-state Turing
   machines that behaviorally represents the entire set of n-state machines
2  foreach machine t in S do
3      Classify t as either a non-halter, or a halter
4      if t is a halter then
5          Prepare input tape x as an infinite bi-directional tape of all 0's
6          Run t to completion on x resulting in the output tape x'
7          if x' satisfies the conditions specified in section 2.2.1.3 then
8              Add t to our candidate set C
9          end
10     end
11 end
12 Return the productivity of the most productive machine in C (i.e. the machine
   that produces the most contiguous 1's on the tape)
```

Algorithm 1: Proposed solution to $\Sigma(n)$

2.3.3 Proposed Algorithm

We are now ready to present the early workings of our proposed solution to the Busy Beaver problem. Consider the algorithm outlined in Algorithm 1. We would like to suggest that this qualifies as a valid procedure for calculating values of $\Sigma(n)$. We cannot truthfully hope for this to be a Turing-computable procedure, however, considering that we have already referenced multiple proofs of the non-Turing-computability of the function (Rado 1963, Miller 2003). Clearly, the suggested procedure in line 3 of our algorithm is the problem. While we cannot hope to define a Turing-computational algorithm to compute this task (as this would be a solution to the halting problem below the Turing limit), we can still strive to prove that a particular machine does not halt on an individual basis by demonstrating that its behavior follows an infinite, repeatable pattern.

To further clarify this notion, consider the following simple routine $R(M, x)$. R is given as input some formally defined description of a Turing machine M as well as some representation of a corresponding tape x which can be fed to machine M as input. Our routine R outputs either a YES or NO response which are to be interpreted as follows:

[18] The derivation of this formula from (Ross 2003) 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.

- YES: The computation $M(x)$ does not halt
- NO: The computation $M(x)$ may or may not halt

Thus R encapsulates the capability to prove that some behaviorally similar subset of Turing machines do not halt.

As a trivial example, consider any machine that is given an infinitely blank tape as its input (analogous to *all* of the machines with which our definition of the Busy Beaver problem is concerned). Now consider all of such machines which contain the transition as denoted by the flow diagram in figure 2.4. When a machine contains this transition, any

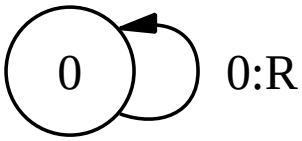


Figure 2.4: Trivial Infinite Loop Structure

time that it is in state 0, while reading a 0 on the input tape, the read head is moved to the right and the machine transitions back to state 0. Thus, if any machine begins with an infinite tape of all 0's and the the first state (state 0) contains this transition, then the machine will clearly never halt. Thus we can very easily embed this detection mechanism into a routine of the form described for R above. Let us call this routine NH_0 that we consequently describe in Algorithm 2.

```
   function: NH_0(M, x)
1  if x is a non-blank tape then
2      return NO
3  end
4  let t be the transition defined in M for state 0, symbol 0 in
5      if t is the transition to move right and go to state 0 then
6          return YES
7      else return NO
8  endlet
```

Algorithm 2: Non-Halt Detection algorithm for trivial rightward behavior

Thus we now can begin to see how a formulation of the required routine in line 3 of Algorithm 1 might be developed. Consider some set of non-halt detection routines such as NH_0 as is already described. The dream properties of such a set would be two-fold:

1. The set is a recursively enumerable set and thus there exists a machine that generates the sequence:

$$NH_0, NH_1, NH_2, NH_3, \ldots$$

2. Every Turing machine that does not halt can be demonstrated so by one of the routines in the set.

While we do not see how this dream could come true[19], it still lays a foundation for our multi-faceted assault on this problem.

[19] A set that exhibits such properties could easily be used to construct a Turing-computable solution to the halting problem

CHAPTER 3
Fronts of Attack: Solvability Claims

Before we present our actual attack on the Busy Beaver problem, we wish to suggest the possibility that the Busy Beaver function could potentially be solved by some mechanisms within the grasp of human capability. We have already seen in section 2.3.1 that there does not exist any Turing machine capable of producing a solution to the function for any arbitrary input n. Thus if there is truth to our suggestion, this would immediately induce the conclusion that the human mind possesses powers beyond that of a Turing machine. The goal for this chapter is to breathe substance into this claim.

We begin with an exercise in intuition in section 3.1 by presenting hope for the possibility that humans possess the ability to enumerate sets that cannot be enumerated by Turing computational mechanisms.[20] This exercise serves as a springboard for a presentation on our stance in the much debated "strong" vs. "weak" AI spectrum (Searle 1980, Penrose 1994, Bringsjord & Zenzen 2003). In the next section, we prop up a position taken by Roger Penrose (1989, 1994) which suggests that humans possess reasoning capabilities beyond that of Turing machines and this capability is *specifically* connected to humans' ability to reason about the non-haltingness of Turing machines. The third section is a presentation of our own argument that injects this specific human ability to ascertain whether or not a Turing machine halts into the realm of spatial reasoning and human diagrammatic reasoning processes. Finally, we encapsulate our arguments into our computability conjectures about the Busy Beaver problem in section 3.4.

3.1 Human Enumeration Capabilities

3.1.1 Pondering Large Numbers

Let us first revisit an important, proven fact that we have brought to attention in section 2.3.1. In particular, we are referring to the truth that the Busy Beaver function dominates *every* Turing computable function that exists. An easily derived corollary of this is that $SH(n)$ also dominates every Turing computable function.[21] It is easy to see that this must be so for if it was not, then we could very easily construct a solution to

[20] Such a result would be an important consideration for the set $\{NH_0, NH_1, NH_2, \ldots\}$ just described at the end of chapter 2.

[21] Recall that $SH(n)$ refers to the number of finite steps that the machine takes to halt of the n-state machine that takes the greatest number of finite steps to halt.

the Busy Beaver problem similar to that outlined in section 2.3.2.[22] The immediate direct conclusion that we have already stated in section 2.3.1 is as such: No Turing machine can generate a sequence of numbers for which each number in the sequence is larger than the corresponding value in an ordered sequence of Busy Beaver values.

The question that we immediately now pose is: how could this be? A skeptic might suggest that one could construct a Turing machine that randomly generates numbers of increasing size and by sheer luck, it enumerates a sequence of values that dominate the sequence of Busy Beaver values. We know, however, that pure randomness cannot be fully captured by mechanisms of Turing-computation and thus this suggestion is unfounded. Clearly we are dealing with an undeniably proven fact so it can truly be considered bulletproof. However, what of humans? If humans are bounded by the capabilities of Turing machines, then this would mean that there are sequences of numbers (i.e. Busy Beaver numbers, sequence of values of *any* function that dominates the Busy Beaver function, etc.) that are unattainable by any means whatsoever!

Consider the following proposition. Let us define a function $h(n)$ where $h(0)$ is set equal to the greatest number conceptualized in some representational form by any human to date. We can set $h(1)$ to be the greatest such number 1000 years from now, $h(2)$, 2000 years from now, and so on.[23] Clearly this function is certifiable and computable by humans (as it is a direct product of human thought).[24] If it is in fact true that the computational powers of the human mind lie below the Turing limit, then are we to believe that our

[22] Suppose that there is a Turing computable function $f(n)$ that dominates $SH(n)$. This would mean the following is true:

$$\exists_k \forall_{n>k} (f(n) > SH(n))$$

If this is the case, however, then we can easily define another Turing computable function $f'(n) = f(n) + f(k)$. Thus $f'(n) > SH(n)$ for all $n > 0$. We can now use $f'(n)$ as part of a very straightforward solution to $\Sigma(n)$:

- Enumerate the set T of all Turing machines with n-states
- Run each machine $t \in T$ until it has either halted or $f'(n)$ steps have been executed
- If $f'(n)$ steps have been executed, the machine is a non halter and can be discarded
- If the machine has halted, verify its candidacy and add to the candidate set C
- Return the maximal productivity of the most productive Turing machine in C

A similar procedure could be used to produce solutions to $SH(n)$, the halting problem, and other similar problems.

[23] We realize that due to the infinitary nature of numbers it is quite easy to conceptualize a greater number by simply say adding 1 or raising it to the power of *googool*. We will assume for our purposes that we can choose a greatest conceptualized number to date and any additions or modifications to this should be considered as candidates for the next value in the sequence.

[24] Clearly in order to calculate say $h(100)$, it will take $100,000$ years. This does not, however, affect the computable nature of the function.

function $h(n)$ *cannot possibly* dominate $\Sigma(n)$?! We find this very difficult to believe and thus present a more grounded case for this notion in the following sections.

3.1.2 Weak A.I. vs. Strong A.I.

It is at this point that we are truly entering the realm of a knotty philosophical debate about the scope of conventional artificial intelligence. This debate of course being whether or not it is possible to create a machine that is equivalent to a human in every aspect of intelligence (including the likes of cognitive states, thinking, understanding, etc.) Before we advocate our position (although it has probably already become increasingly clear at least what that approximate position might be), we must first specify the spectrum of views that we are pulling from. Specifically, artificial intelligence can be broken down into what are called "strong" and "weak" paradigms and consequently further classified as we shall see.

Perhaps the most well known opponent of "Strong" A.I. and pioneer of the distinction between "Weak" and "Strong" A.I. is John Searle. In (Searle 1980), he outlines this distinction:

> In answering this question I find it useful to distinguish what I call "strong" AI from "weak" or "cautious" AI. According to weak AI, the principal value of the computer in the study of the mind is that it gives us a very powerful tool. For example, it enables us to formulate and test hypotheses in a more rigorous and precise fashion than before. But according to strong AI the computer is not merely a tool in the study of the mind; rather the appropriately programmed computer really is a mind in a sense that computers given the right programs can be literally said to *understand* and have other cognitive states. (Searle 1980)

He then proceeds to suggest via his famous Chinese room experiment[25] that weak AI is perfectly reasonable yet at the same time strong AI is not. Without an involved discussion on the Chinese room experiment itself, we must note that the version of "weak" AI that Searle appears to claim seems to be self contradictory. We elaborate further shortly.

[25] A very brief summarization of this thought experiment is as follows: Suppose we lock a person in a room and provide him with a set of symbols and corresponding set of rules that indicate how to correlate sequences and/or subsets of such symbols into some other corresponding set of symbols. We then provide this person several batches of symbols and request that he use the given rules to generate appropriate response sets of symbols. Unbeknownst to the man in the room, the sets of symbols and rules that we are giving him are actually representative of a story written in Chinese with a corresponding set of questions about the story. To the outside observer, it appears as though the man in the room understands and is conversing in Chinese when in fact, he is simply mindlessly interpreting symbols according to a set of rules. Thus Searle's conclusion is that since this process is conceivably what an artificially intelligent machine would be doing, it is possible to simulate the *appearance* of a machine understanding Chinese (weak AI), but it is not possible to actually build a machine that understands in the way that a person does (strong AI).

First, however, it is clear that the concepts of strong and weak AI are still somewhat vague in definition. Thus we present a more comprehensive breakdown of common viewpoints given by Penrose (1994):

> $\mathcal{A}$ All thinking is [Turing-]computation; in particular, feelings of conscious awareness are evoked merely by the carrying out of appropriate [Turing-]computations.
>
> $\mathcal{B}$ Awareness is a feature of the brain's physical action; and whereas any physical action can be simulated [Turing-]computationally, [Turing-]computational simulation cannot by itself evoke awareness.
>
> $\mathcal{C}$ Appropriate physical action of the brain evokes awareness, but this physical action cannot even be properly simulated [Turing-]computationally.
>
> $\mathcal{D}$ Awareness cannot be explained by physical, [Turing-]computational, or any other scientific terms.
>
> (Penrose 1994, pg. 12)

We can see that Penrose lays out a framework where $\mathcal{A}$ is the "strongest" consideration for AI and $\mathcal{D}$ is consequently the "weakest." Clearly we can consider viewpoint $\mathcal{A}$ as corresponding to "Strong" AI as it is described by Searle. If we were to accept this claim, then our suggestion that humans encapsulate the ability to solve the Busy Beaver problem would be immediately rebuked. Thus moving down the list, $\mathcal{B}$ is the viewpoint that Searle appears to advocate but which we have already suggested might be self contradicting.[26] As Penrose points out himself, $\mathcal{D}$ suggests that human cognitive processes are detached from the physical world and are mystical in nature. We neither support this view nor do we attempt to scientifically address it. Thus we are left with position $\mathcal{C}$, incidentally the one which Penrose also supports.

What is it that we are suggesting then? We have chosen to align ourselves closely with the views purported by Penrose (1989, 1994) which is that there are certain elements of physical laws that are not Turing-computational in nature. Therefore, since the human mind is subject to these physical laws, it cannot be appropriately simulated via the instantiation of some Turing machine. Not only this, however, but we also wish to go one step further. As has already been stated, we wish to vouch for the possibility that the Busy Beaver function is solvable by mechanisms available in the human mind. This would

[26] Searle repeatedly claims that humans are in fact "thinking machines" and that all of their cognitive processes can be simulated Turing-computationally. However, he still contends that a program (Turing machine) cannot be constructed that "thinks" and "understand" like a human. Again, we do not dive into this debate too deeply for it is addressed in countless other papers and not intimately related to our focus. Regardless, we do point out that we feel $\mathcal{B}$ does not appear to be a true possibility. If the brain is a result of physical actions that can *all* be simulated by Turing-computable mechanisms, then it appears obvious, at least to us, that a properly configured Turing machine could possess consciousness and "awareness" just as a human does. Bringsjord & Zenzen (2003) raise this issue as well noting that $\mathcal{B}$ should be subdivided into distinct claims that are individually addressed.

mean that the human mind must encapsulate computational powers beyond that of the Turing limit. Thus, we contend that the physical laws suggested by Penrose[27] are not only non-Turing-computational, but they are in fact *hyper*computational and encapsulate methods of computation that are not available at or below the Turing limit. We thus present a concrete physical phenomena to support this view in the following section.

3.1.3 Hypercomputing the Steiner Tree Problem

In an as yet unpublished note, Bringsjord & Taylor (2005) present an exceedingly intriguing argument in which they claim to have proven the long standing P=?NP problem.[28] We do admit that the suggested proof is the result of very clever ingenuity, but it is subject to a fatal flaw if one unproven proposition (which they admit to themselves) is true. Consider, first, their proposed proof:

The proof is based on a known NP-complete problem that can be solved via a simple physical process. Bringsjord & Taylor (2005) describe this process:

> For example, the Steiner Tree problem (STP) is known to be NP-complete (see e.g. (Garey & Johnson 1979, pp. 208-209)). Nonetheless, a simple process (termed an **analog computation**) can apparently solve it quickly. STP is the problem of connecting n points on a plane with a graph of minimal overall length, using junction points if necessary. The physical process in question can be described in English as a straightforward algorithm: Make two parallel glass plates, and insert n pins between the plates to represent the points; dip the structure into a soap solution, and remove it; record the answer. The soap film will connect the n pins in the minimum Steiner-tree graph (Iwamura, Akazawa & Amemiya 1998).

The brunt of their argument can be summarized as follows: Since all physical processes can be simulated by Turing-computable mechanisms, then the physical process that connects the pins together with the soap film can be modeled via some Turing machine M_c. Thus since this process achieves a solution to the NP-complete STP problem in linear time, P=NP must be true.[29]

Clearly, the crucial assumption is that all physical processes can be simulated via some Turing machine. Bringsjord and Taylor concede this notion: "P≠NP thus immediately implies, courtesy of our arguments, that hypercomputational processes exist in some physical universes." They then attempt to remedy this problem by suggesting:

[27] We must note that Penrose's arguments are deeply rooted in quantum theory and other aspects of physics with which we have minimal familiarity. Regardless, we wish to make no claims about the physics aspect of the physical laws that we speak of, only of their computational qualities.

[28] See (Sipser 1992) for a description and discussion of the history of this problem.

[29] The formalization of their argument is slightly more complex, but the basics of the core can be summarized as above.

> While many are perhaps right to point out, *contra* Wolfram and company, that some physical phenomena (e.g., those associated with quantum mechanics) are so bizarre and complicated that they resist formalization in TM-level computational models, the fact of the matter is that the analog process we exploit is a painfully simple macroscopic phenomenon - as we say, a "normal" process.

Why should we accept such an explanation? Bringsjord and Taylor concede the possibility that at perhaps the microscopic level, the constituents of the soapy solution might follow hypercomputational physical laws. Why, then, does the soapy solution lose its hypercomputational properties when it is being viewed as a whole? If the very building blocks of the physical laws that govern the organization of the soapy solution are grounded in hypercomputational processes, then it seems nearly obvious that the process as a whole could harness this same power.

Thus we reject Bringsjord and Taylor's claim that P=NP. Since we align ourselves with the general consensus among the mathematical community (that P≠NP) (Kupchik 2004, Feinstein 2004, Ionescu 2005, Moscu 2005, Grover 2005, Ivanov 2005), as Bringsjord states plainly himself, we must immediately conclude that hypercomputational processes do in fact exist within the realm of physical possibility. Thus not only do we contend that hypercomputational processes exist within the human mind, considering these conclusions we must also affirm that hypercomputation exists within physical processes *outside* of the human mind.

3.1.4 HyperEnumeration

At last we now return to our original thought experiment that we describe in section 3.1.1. In this experiment, we suggest a function $h(n)$ that is the direct result of human thought (compounded over 1000 year intervals of time). Our intuition sees no conceptual reason why the possibility that this function dominates $\Sigma(n)$ should not exist. However, the "strong" AI view and even the "weak" AI view as it is supported by Searle (1980) would suggest that our intuitions are incorrect. Armed with the claims presented in sections 3.1.2 and 3.1.3, we have now at least opened up the possibility that humans possess the power to hypercompute and thus perhaps the ability to "hyperenumerate."

3.2 Defending Penrose: The Non-Halt Detection Capabilities of Humans

In light of the questions raised in the preceding section 3.1, we now turn to a more concrete argument that lends itself both to the potential solvability of the Busy Beaver

problem as well as the aforementioned claim that some physical processes can only be explained by hypercomputational means. As is previously mentioned, Roger Penrose (1994) presents a multi-faceted case against the view that a properly programmed Turing machine can encapsulate all of the abilities and qualities of a human. In particular, Penrose suggests a very concise argument that parallels Gödel's incompleteness theorems with the non-haltingness of Turing machines. His proposed conclusion is that a Turing machine cannot be programmed to encapsulate all elements of human understanding.

At this point, before we proceed, we must point out that Penrose's argument as it is presented, is part of a greater argument designed to refute not only the "Strong A.I." standpoint, but also the "Weak A.I." standpoint as well (see section 3.1.2). In a far-reaching presentation of a case against "Strong A.I." (but while at the same time supporting "Weak A.I.") Bringsjord & Zenzen (2003) present a formalized attack against Penrose's mentioned Gödelian argument. Our goal in this section is therefore threefold. We first wish to defend Penrose's line of reasoning against the critical refutation outlined in (Bringsjord & Zenzen 2003). Secondly, we wish to suggest that Penrose's core reasoning lends itself to the claim outlined in section 3.1 (namely that not all physical phenomena can be simulated by Turing-computable mechanisms). Finally, we wish to connect Penrose's argument to our attack on the Busy Beaver problem and examine the possibilities about the computability of the function that it suggests.

3.2.1 Penrose's Argument

The core of Penrose's argument as it is outlined in (Penrose 1994) makes an important connection between Gödel's incompleteness theorems and humans' ability to ascertain that particular Turing machines do not halt. Without delving too deeply into Gödel's theorems, the core of his ideas is that given any formal system of mathematical rules, there always exists statements within that system that cannot be proven solely by the rules set forth in that particular system. However, this does not mean that these statements cannot be proven via mechanisms that exist outside this formal system. Thus there exist statements that we, *as humans*, can see to be true by our own mechanisms of logical reasoning but this truth cannot be ascertained by the formal rules of the system in which the statement is presented.[30]

Armed with this knowledge, Penrose suggests the nonexistence of some Turing ma-

[30] See chapter 4 of (Penrose 1989) where Penrose presents a very comprehensive prose on the meaning of Gödel's ideas and theorems

chine that he calls A that "encapsulates *all* the procedures available to human mathematicians for convincingly demonstrating that computations do not stop." (Penrose 1994, pg. 73) Additionally, in this context he is concerned with demonstrating the non-haltingness of computations (and thus Turing machines) that are exclusively functions of one input value. Thus he outlines the enumeration of all Turing machines (an easily provable Turing-computational process) as $C_0, C_1, C_2, C_3, ...$ and the computation of the q^{th} Turing machine in this sequence on the input value n as $C_q(n)$. As is done in (Bringsjord & Zenzen 2003), we duplicate his argument here for clarity:

> A is just *any sound* set of computational rules for ascertaining that some computations $C_q(n)$ do not ever halt. Being dependent upon the two numbers q and n, the computation that A performs can be written $A(q, n)$, and we have:
>
> **(H)** If $A(q, n)$ stops, then $C_q(n)$ does not stop.
>
> Now let us consider the particular statement **(H)** for which q is put equal to n ... we now have:
>
> **(I)** If $A(n, n)$ stops, then $C_n(n)$ does not stop.
>
> We now notice that $A(n, n)$ depends upon just *one* number n, not two, so it must be one of the computations $C_0, C_1, C_2, C_3, ...$(as applied to n), since this was supposed to be a listing of *all* computations that can be performed on a single natural number n. Let us suppose that it is in fact C_k, so we have:
>
> **(J)** $A(n, n) = C_k(n)$.
>
> Now examine the particular value $n = k$. (This is the second part of Cantor's diagonal slash!) We have, from **(J)**,
>
> **(K)** $A(k, k) = C_k(k)$.
>
> and, from **(I)**, with $n = k$:
>
> **(L)** If $A(k, k)$ stops, then $C_k(k)$ does not stop.
>
> Substituting **(K)** in **(L)**, we find:
>
> **(M)** If $C_k(k)$ stops, then $C_k(k)$ does not stop.
>
> From this, we must deduce that the computation $C_k(k)$ does *not* in fact stop. (For if it did then it does not, according to **(M)**!) But $A(k, k)$ cannot stop either, since by **(K)**, it is the *same* as $C_k(k)$. Thus, our procedure A is incapable of ascertaining that this particular computation $C_k(k)$ does not stop even though it does not. Moreover, if we *know* that A is sound, then we *know* that $C_k(k)$ does not stop. Thus, we know something that A is unable to ascertain. It follows that A *cannot* encapsulate our understanding. (Penrose 1994, pg. 74-75)

3.2.2 Bringsjord and Zenzen's Refutation

There is a clear-cut flaw in Penrose's argument as it has just been presented and Bringsjord & Zenzen (2003) quite convincingly demonstrate this flaw. Before we present this demonstration, we first must indicate the notational scheme that they use and henceforth will be used by us and in our following argument in section 3.2.3:

> For our formalization we follow the notation of (Ebbinghaus, Flum & Thomas 1994), and hence deploy atomic formulas
>
> $$M_t : u \rightarrow v$$
>
> to denote the fact that TM M_t, starting with u as input on its tape, halts and leaves v as output. Similarly
>
> $$M_t : u \rightarrow halt$$
>
> and
>
> $$M_t : u \rightarrow \infty$$
>
> denote, respectively, that the TM in question halts and doesn't halt (on input u). Next, assume that the alphabet with which our TMs work is of the standard sort, specifically $\{|, \bullet\}$, where a natural number n is coded as a string of n |'s, and $\bullet$ is used solely for punctuation. Finally, fix some enumeration of all Turing machines and a corresponding Gödel numbering scheme allowing us to reference these machines via their corresponding natural numbers. (Bringsjord & Zenzen 2003, pg. 60-61)

Bringsjord and Zenzen reveal the aforementioned flaw in Penrose's reasoning by utilizing this formal notational scheme to formalize Penrose's argument as it is presented in the preceding section (3.2.1). Thus they begin by attempting to encode Penrose's "proof" into a first order logic style proof and the beginning of the proof (encapsulating Penrose's line of reasoning up through statement **(I)**) can be achieved without issue:[31]

> The initial part of the formalization is straightforward. Penrose begins by assuming that there is some set A of computational rules (we use 'M_a' to refer to A as TM; this is an identification Penrose himself, following standard mathematical practice, explicitly sanctions in Appendix A of *SOTM*) such that: if A yields a verdict that some TM M fails to halt on input n, then M *does* fail to halt on n. He then moves, via quantifier manipulation, through **(H)** to **(I)**. Here's how the initial reasoning runs:
>
> | | | |
> |---|---|---|
> | $1'$ | $\exists m \forall q \forall n[M_m : q \bullet n \rightarrow halt \Rightarrow M_q : n \rightarrow \infty]$ | supposition |
> | $2'$ | $\quad \forall q \forall n[M_a : q \bullet n \rightarrow halt \Rightarrow M_q : n \rightarrow \infty] =$ **(H)** | supposition |
> | $3'$ | $\quad \forall n[M_a : b \bullet n \rightarrow halt \Rightarrow M_b : n \rightarrow \infty]$ | $2'\forall$E |
> | $4'$ | $\quad M_a : b \bullet b \rightarrow halt \Rightarrow M_b : b \rightarrow \infty$ | $3'\forall$E |
> | $5'$ | $\quad \forall n[M_a : n \bullet n \rightarrow halt \Rightarrow M_n : n \rightarrow \infty] =$ **(I)** | $4'\forall$I |
>
> (Bringsjord & Zenzen 2003, pg. 62)

It is at this point that the flaw becomes obviously apparent. Recall that after this portion of Penrose's reasoning (**(I)**), he proceeds to suggest that the computation $A(n, n)$ is identical to some computation $C_k(n)$ in the corresponding enumeration of all possible Turing machines. In Bringsjord and Zenzen's formalized model, therefore, this means that the machine M_a when operating on the input $n \bullet n$ is identical to that of some Turing machine M_k when operating on input n. Clearly this cannot be the case.

[31] We assume that the reader is trained in the rules of formal first order logic in order to follow the formalization of Penrose's arguments. See (Barwise & Etchemendy 1999) for an excellent resource in this discipline.

What follows from this point forward in (Bringsjord & Zenzen 2003) is an attempt by Bringsjord and Zenzen to salvage Penrose's line of reasoning such that perhaps Penrose's "proof" can be fixed in a formal sense that produces the resulting conclusion that Penrose ultimately sets out to show. They reason about the possibility that while the machine M_a operating on the input $n \bullet n$ cannot possibly be identical to some machine M_k operating on the input n, it is still quite easy to show that there does in fact exist some machine M_k such that $M_a : n \bullet n$ and $M_k : n$ result in identical *behavior*:

> Given a TM M_1 that operates on input $n \bullet n$ and eventually halts, it's easy to build a TM M_3 which starts with just n on its tape, calls a TM M_2 which copies n so that $n \bullet n$ is written on the tape, and then proceeds to simulate M_1 step for step. (Bringsjord & Zenzen 2003, pg. 63)

Thus they concede that in this case M_1 and M_3 are *"approximately identical"* and thus write this as $M_1 \approx M_3$.

As a result of this concession, they propose the following Lemma as the next line of proof to allow Penrose's argument to continue:

$$6' \qquad \forall n[(M_n : n \bullet n \rightarrow halt \Rightarrow M_n : n \rightarrow \infty) \Rightarrow \exists q(M_q \approx M_n \wedge (M_q : q \rightarrow halt \Rightarrow M_q : q \rightarrow \infty))] \quad \text{Lemma}$$

Consider, however, a prose interpretation of what this Lemma actually suggests: Given any TM M_n, if the fact that it halts on input $n \bullet n$ implies that it does not halt on input n, then there exists a TM M_q which is "approximately identical" to M_n such that if M_q halts on input q then it does not halt on input q. The problem with attempting to fix Penrose's problem by introducing such a Lemma is that it forces the instantiation of the constant k in Penrose's original argument to be equal to the constant a if there is any hope of establishing his goals. Why, however, should this be the case? Even though Penrose does claim that his machine A is equal to one of the enumerated machines C_k, might it still be possible to conclude the proof via some intermediary machine M_k that is not the same machine as M_a? We believe that this is the case and present our argument in the following section.

3.2.3 Defense of Penrose's Reasoning

Before we dive into our own attempted rescue of Penrose's reasoning via formalization, let us first back up a step. Recall from the previous section 3.2.2 the beginnings of a formalization of Penrose's argument by Bringsjord and Zenzen as steps $1'$ - $5'$. In order to encapsulate the totality of Penrose's argument, we must ultimately refute the claim that "A encapsulates *all* the procedures available to human mathematicians for convincingly

demonstrating that computations do not stop." (Penrose 1994, pg. 73) Thus if this is the case, then not only must we assume the proposition outlined in line $1'$ of Bringsjord and Zenzen's formalization, but we also must suitably encode the fact that for any pair of values q and n, if $C_q(n)$ can be shown to never stop by some conventions of human reasoning, then $A(q,n)$ will also stop. Thus we define $\psi(q,n)$ to be equivalent to the statement: $M_q : n \rightarrow \infty$ can be shown in some way by the procedures available to human mathematicians for convincingly demonstrating that a Turing machine does not halt. We include this in a modified formalization of Penrose's argument that corresponds to lines $1'$ - $5'$ of Bringsjord and Zenzen's proof above:

$1''$	$\exists m \forall q \forall n[(M_m : q \bullet n \rightarrow halt \Rightarrow M_q : n \rightarrow \infty) \wedge$	
	$(\psi(q,n) \Rightarrow M_m : q \bullet n \rightarrow halt]$	supposition
$2''$	$\quad \forall q \forall n[(M_a : q \bullet n \rightarrow halt \Rightarrow M_q : n \rightarrow \infty) \wedge$	
	$\quad (\psi(q,n) \Rightarrow M_a : q \bullet n \rightarrow halt)]$	$[a]$ assumption
$3''$	$\quad\quad [b]$ assumption	
$4''$	$\quad\quad \forall n[(M_a : b \bullet n \rightarrow halt \Rightarrow M_b : n \rightarrow \infty) \wedge$	
	$\quad\quad (\psi(b,n) \Rightarrow M_a : b \bullet n \rightarrow halt)]$	$2''\forall$E
$5''$	$\quad\quad [(M_a : b \bullet b \rightarrow halt \Rightarrow M_b : b \rightarrow \infty) \wedge$	
	$\quad\quad (\psi(b,b) \Rightarrow M_a : b \bullet b \rightarrow halt)]$	$4''\forall$E
$6''$	$\quad\quad M_a : b \bullet b \rightarrow halt \Rightarrow M_b : b \rightarrow \infty$	$5''\wedge$E
$7''$	$\quad\quad \psi(b,b) \Rightarrow M_a : b \bullet b \rightarrow halt$	$5''\wedge$E
$8''$	$\quad \forall n[M_a : n \bullet n \rightarrow halt \Rightarrow M_n : n \rightarrow \infty] =$ **(I)**	$3'' - 6''\forall$I[32]
$9''$	$\quad \forall n[\psi(n,n) \Rightarrow M_a : n \bullet n \rightarrow halt]$	$3'' - 7''\forall$I

At this point, we can now proceed to attempt to rectify Penrose's flaw as is described above. Recall that Bringsjord and Zenzen introduce a proposed line of reasoning that could potentially help Penrose's case. Specifically, it can be easily shown that for any given TM M_1 that operates on the input $n \bullet n$, we can construct a TM M_3 that is "approximately identical" to M_1 in that it operates on input in the form of n but its ultimate behavior is the same as that of M_1 when the two machines are given inputs in their respective required formats. Bringsjord and Zenzen then proceed to introduce a Lemma which does not directly encapsulate the meaning of this statement. Instead, it makes a fatal assumption about the instantiation of the machine that is to make use of this property in Penrose's case. Thus we wish to strip all assumptions and directly encapsulate the meaning of this notion of "approximately identical" as follows:

[32] Here we use a somewhat liberal interpretation of the $\forall$I rule since line $6''$ is not in the last line of the subproof. The reasoning is still perfectly valid, however.

$10''$	$\forall m \exists k \forall n[(M_m : n \bullet n \rightarrow halt \Leftrightarrow M_k : n \rightarrow halt) \wedge$	
	$(M_m : n \bullet n \rightarrow \infty \Leftrightarrow M_k : n \rightarrow \infty) \wedge$	
	$\forall o(M_m : n \bullet n \rightarrow o \Leftrightarrow M_k : n \rightarrow o)]$	Lemma

Essentially what this statement formalizes is the notion that for any TM M_m, there exists some TM M_k such that for all inputs of the form $n \bullet n$ run on M_m, machine M_k will produce identical output when run on input n. By identical output we mean that whether or not the machine halts will be the same in both respects and if the machines halt, the output tapes will be the same. The truth of this Lemma should be undisputed given the line of reasoning already presented concerning TM's M_1 and M_3.

We are thus now armed with an appropriate formalized property that allows us to continue with Penrose's line of reasoning. The remainder of the proof is as follows:

$11''$	$\exists k \forall n[(M_a : n \bullet n \rightarrow halt \Leftrightarrow M_k : n \rightarrow halt) \wedge$	
	$(M_a : n \bullet n \rightarrow \infty \Leftrightarrow M_k : n \rightarrow \infty) \wedge$	
	$\forall o(M_a : n \bullet n \rightarrow o \Leftrightarrow M_k : n \rightarrow o)]$	$10''\forall$E
$12''$	$\quad \forall n[(M_a : n \bullet n \rightarrow halt \Leftrightarrow M_c : n \rightarrow halt) \wedge$	
	$\quad (M_a : n \bullet n \rightarrow \infty \Leftrightarrow M_c : n \rightarrow \infty) \wedge$	
	$\quad \forall o(M_a : n \bullet n \rightarrow o \Leftrightarrow M_c : n \rightarrow o)]$	$[c]$ assumption
$13''$	$\quad (M_a : c \bullet c \rightarrow halt \Leftrightarrow M_c : c \rightarrow halt) \wedge$	
	$\quad (M_a : c \bullet c \rightarrow \infty \Leftrightarrow M_c : c \rightarrow \infty) \wedge$	
	$\quad \forall o(M_a : c \bullet c \rightarrow o \Leftrightarrow M_c : c \rightarrow o)$	$12''\forall$E
$14''$	$\quad M_a : c \bullet c \rightarrow halt \Leftrightarrow M_c : c \rightarrow halt$	$13''\wedge$E
$15''$	$\quad M_a : c \bullet c \rightarrow halt \Rightarrow M_c : c \rightarrow \infty$	$8''\forall$E
$16''$	$\quad\quad M_c : c \rightarrow halt$	assumption
$17''$	$\quad\quad M_c : c \rightarrow \infty$	$14'', 16'' \Leftrightarrow$E
$18''$	$\quad\quad \bot$	$16'', 17'' \bot$I[33]
$19''$	$\quad M_c : c \rightarrow \infty$	$16'' - 18'' \neg$I[34]
$20''$	$\quad \psi(c, c)$	$19''$ (definition)[35]
$21''$	$\quad \psi(c, c) \Rightarrow M_a : c \bullet c \rightarrow halt$	$9''\forall$E
$22''$	$\quad M_a : c \bullet c \rightarrow halt$	$20'', 21'' \Rightarrow$E
$23''$	$\quad M_a : c \bullet c \rightarrow \infty \Leftrightarrow M_c : c \rightarrow \infty$	$13''\wedge$E
$24''$	$\quad M_a : c \bullet c \rightarrow \infty$	$19'', 23'' \Leftrightarrow$E
$25''$	$\quad \bot$	$22'', 24'' \bot$I
$26''$	$\bot$	$11'' - 25''\exists$E
$27''$	$\bot$	$1'' - 26''\exists$E

Thus we see that Penrose's original assumption specified in line $1''$ has yielded a contradiction and by *reductio ad absurdum*, we conclude that it must therefore be false. That is

[33] Here, we again use a liberal interpretation of $\bot$ introduction by assuming that statements such as $M_c : c \rightarrow halt$ and $M_c : c \rightarrow \infty$ are negations of each other. We see no reason why this should be disputed.

[34] Again we take a shortcut without formally defining the negation of Turing machine behaviors.

[35] Recall that $\psi(q, n)$ indicates that $M_q : n \rightarrow \infty$ can be shown to be true by some method of human reasoning. In this case, we have shown that $M_c : c \rightarrow \infty$ is true using simple logic (which is clearly a method available to human mathematicians) and therefore we can conclude $\psi(c, c)$

in its informal form: there is no Turing machine A that encapsulates all available mechanisms to human mathematicians to ascertain that a Turing machine does not halt. As a result of this conclusion, we can immediately deduce that the mechanisms with which human mathematicians determine that a Turing machine does not halt must contain some non-Turing-computational properties.[36] If they do not, then we could easily create the suggested machine A.

3.2.4 Revisiting the Dream

Recall now our dream as outlined at the end of section 2.3.3. We can see now how our dream, while not possible in a Turing-computational sense, is gaining momentum as a still human-computable possibility. Drawing from Penrose's line of reasoning in (Penrose 1994) and coupling it with our presentation in section 3.1, we have breathed life into the notion that human reasoning mechanisms used to deduce that Turing machines do not halt must tap into hypercomputational properties of the physical makeup of the mind. Thus as it relates to our dream, we see two not necessarily exclusive possibilities that could contribute to our hope:

1. The set $\{NH_0, NH_1, NH_2, NH_3, \ldots\}$ is hyperenumerable by human capabilities.

2. There exist some items $\{NH_x, NH_y, \ldots\}$ in this set that are not Turing-computable but they are hypercomputable by human capabilities.

We have already briefly addressed the hyperenumeration possibility in section 3.1. In the next section, we explore a concept that has important ramifications for both possibilities.

3.3 Diagrammatic Reasoning Potential

At this point it has become increasingly clear what our position is on human computational abilities and their specific tie-in to the computability of the Busy Beaver problem. To reiterate briefly, we believe that hypercomputational processes exist naturally in nature, and as is evidenced by our defense of Penrose's argument in section 3.2, humans are able to tap into these processes *specifically* via their abilities to determine whether or not particular Turing machines halt. Thus it should already be clear to some observers that humans possess the power to make "more progress" on the Busy Beaver problem than

[36] As has already been presented in section 3.1, these non-Turing-computational properties we contend are hypercomputational in nature.

any properly programmed Turing machine.[37] However, an intelligent and skeptical reader might offer one or more of the following rebuttals to what we have claimed thus far:

$\mathcal{R}1$ You have only provided evidence for hypercomputational processes that exist in nature (and not in humans). Even if there is some merit to this bold claim, why are we to believe that humans maintain the ability to tap into this power? Even though you have shown that Penrose's suggested procedure is non-Turing-computational, how can you immediately make the claim that it is *hyper*computational?

$\mathcal{R}2$ We are not convinced that humans can hypercompute. Despite your arguments, you have still not provided any hard evidence of situations where humans clearly engage in hypercomputational methods.

In response to $\mathcal{R}1$, let us reconsider the machine A presented in Penrose's argument of section 3.2.1. We have revised Penrose's proof and brought formal validity to the ultimate claim: That Turing machine A, which purportedly encapsulates all methods available to humans for ascertaining that a Turing machine does not halt, does not and cannot exist. Clearly, though, humans can perform the procedure that is defined for machine A (since by definition, the procedure directly cites human capabilities). Thus the ability of humans to process information about non-halting Turing machines is beyond that of standard computational mechanisms and this should intuitively lead to the conclusion that humans can hypercompute. The evidence provided in section 3.1.3 that hypercomputational processes exist in nature simply provides further explanation as to how the human mind could possibly be capable of hypercomputation. Since some physical processes can only be explained by hypercomputational means, then clearly the human brain could be grounded in these fundamental physical processes thus lending itself to hypercomputational capabilities.

Regardless, in spite of our arguments, we realize that many skeptics may still raise doubts and offer statements such as $\mathcal{R}2$ above. In response to these statements, we continue to solidify our stance in the following sections by offering certain human reasoning

[37] We make this statement under the assumption that Algorithm 1 in general is the only reasonable algorithm that could potentially be used to solve $\Sigma(n)$. If this is indeed the case, then clearly "more progress" can be made by humans since their ability to determine whether or not Turing machines halt cannot be encapsulated in some Turing-computable process. Thus line 3 of this algorithm can be achieved with greater success by humans than by a machine. We concede the possibility that some other reasonable algorithm could potentially exist, but the mere fact that $\Sigma(n)$ is proven to be uncomputable suggests that any other reasonable algorithm would be subjected to the same line of reasoning concerning hypercomputational processes.

methods about diagrammatic information as a grounding example of hypercomputation.

3.3.1 Efficiency of Diagrams

It should be immediately apparent that diagrams[38] can provide information in a very compact, succinct, and readily available form. These diagrams could include a vast range of contexts from direct spatial representations of physical objects or processes (i.e. maps, sketches of physics problems, etc.) to conceptual mappings of non-physical things (such as trends in stock prices or perhaps the execution of a Turing machine over time). Larkin & Simon (1987) contend that at least from an external standpoint, human processing of diagrams is often considerably more efficient than processing of corresponding symbolic representations of the same information:

> The advantage of diagrams, in our view, are computational. That is diagrams can be better representations not because they contain more information, but because the indexing of this information can support extremely useful and efficient computational processes.

What we wish to support is that the computational processes mentioned are in fact hypercomputational processes and thus humans can exploit their previously claimed hypercomputational powers through diagrammatic reasoning.

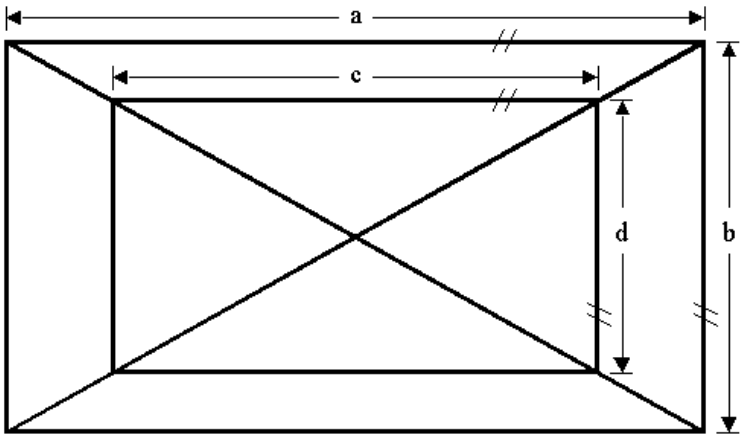


Figure 3.1: Diagram for Reasoning Test

Let us consider first the diagram depicted in figure 3.1. To the observer trained in basic geometry, it should be immediately apparent that with respect to the diagram, $\frac{a}{b} = \frac{c}{d}$ is a true statement. Not only this, but also consider how this conclusion is actually

[38] We refer to diagrams here as any type of picture, image, or diagram that represents information in some spatial way and not exclusively with some formal symbolic language. See (Chandrasekaran 2005) for a more elaborate discussion of the difference between diagrammatic and symbolic representations.

ascertained. Indisputably, a logical proof using basic theorems of geometry can easily be constructed to verify the truth of this claim. However, does the trained observer actually construct a complete mental logical proof before being self-convinced of the above statement? Consider the following hypothetical dialog between two people observing the diagram in question:

Observer A :	In figure 3.1, is $\frac{a}{b} = \frac{c}{d}$ a true statement?
Observer B :	Yes.
Observer A :	Are you sure?
Observer B :	Absolutely.
Observer A :	Prove it to me.
Observer B :	[re-examines diagram and thinks for a minute] [constructs a logical proof]

First of all, we are not likely to receive many objections regarding the hypothetical nature of this dialog. Clearly it is descriptive of an event that is very conceivable. The question that we maintain, however, is why does Observer B need to re-examine the diagram and re-think the problem before providing a logical proof? He has already claimed with absolute certainty that $\frac{a}{b} = \frac{c}{d}$ is an undeniable fact. Therefore, where does this certainty come from? If it is from his provided logical proof, than why could he not immediately provide the proof without additional processing of the diagram? It is in this original reasoning process of Observer B that we contend hypercomputational powers are at work.

We realize that up to this point in this section we have simply been begging the question on our position about hypercomputational powers in human diagrammatic reasoning processes. While we contend that we have been begging a good question, we now turn to a pair of arguments to support our claim.

3.3.2 The Searle-ian Argument

In our first line of defense, consider first what a Turing-equivalent, machine processing of the diagram in figure 3.1 and the corresponding question raised in the preceding section might entail. Clearly we would need to first transform the diagram into some symbolic representational form that can be entered as input on the tape of such a machine. Thus we can think of a diagram as an array of pixels. Each pixel is assigned a corresponding RGB (Red/Green/Blue) set of color values and the entire array of pixels is transformed into a suitable one line format that can be entered on our tape. Additionally, we must program the machine so that it is capable of transforming this array of pixels into suitable statements representing the geometric elements displayed in the figure. In order to respond to the question raised (is $\frac{a}{b} = \frac{c}{d}$ a true statement?), two lines of processing

could be followed:

1. The machine could directly calculate the length of each of the required line segments (a, b, c, d) and then perform a direct comparison of the necessary ratios.

2. The machine could be encoded with the necessary geometric theorems to construct a logical proof to show that the equality in question is true.

We realize that it is possible that there are other methods of machine processing that could yield the same result. The important point that we are making, however, is not the exact process involved, but rather that any process we present requires the transformation of the visual components of any diagram into some digitized symbolic format.

With this issue clarified, we now return to John Searle's Chinese room argument (Searle 1980) previously mentioned in section 3.1.2. Recall that the core of this argument is that it is possible for a *human* to process stories written in Chinese and then answer questions about them without having the slightest bit of *understanding* of the Chinese language at all. In a similar light, let us consider the following possibility:

A man, we'll call him Diagram, is put in a room with no windows and the only contact that he receives with the outside world is through a small slot in the door where messages can be passed back and forth. Diagram has no training in any geometric principles nor does he have any understanding of formal logic syntax and structure. He is, however, extremely skilled at following directions. He has at his disposal the following:

$\mathcal{A}$ A set of transformations that allow him to transform a certain format of sequences of numbers into a new sequence of alphanumeric, geometric, and logical symbols.

$\mathcal{B}$ A set of transformations that, given two sequences of alphanumeric / geometric / logical symbols, allows him to generate a result of either True or False.

Little does Diagram know that the transformations in $\mathcal{A}$ are actually the rules necessary for transforming a sequence of pixels into a set of statements representing geometric truths about a diagram. Additionally, the rules outlined in $\mathcal{B}$ actually encapsulate the inner workings of some automated deduction system grounded in formal logic. Given two sets of symbols for processing with $\mathcal{B}$, the first set should represent the set of premises extracted from the diagram and the second set represents the goal conclusion.

Thus, through the small slot in the door, we could feed Diagram a sequence of pixels that represents the diagram in figure 3.1. We could also give him a representative form of

the statement $\frac{a}{b} = \frac{c}{d}$. We could then ask him to use the rules that he has available in $\mathcal{A}$ to transform the first set of symbols we have given him into a new set. Finally, we could ask that he use the rules in $\mathcal{B}$ to transform this new set along with the second set of symbols that we have given him. The result is that Diagram can answer the same question about our diagram as Observer B does in the preceding section without ever seeing it, visualizing it, or even understanding what he is doing. It is all, instead, the result of him mindlessly transforming sets of symbols into other sets of symbols, which at the base level, is exactly what a properly programmed computer does.

The core of this argument, the reasoning of which is borrowed directly from John Searle's Chinese room, is that clearly it is possible for a computer to process an image and arrive at the same conclusion as a human. However, it can do all of this without ever actually *seeing* the image or *understanding* how the conclusion is realized. Diagram is a direct example of this for he processes a diagram in a manner identical to that of a computer without even realizing what this process results in as a whole.

We would anticipate that at this point, the alert reader might offer the following question: "But how does this show that humans can hypercompute? Even if a computer does not truly *understand* what it is doing nor *see* the image that it is processing that does not mean that human processing of images is beyond the Turing limit." In response to this query we ask the reader to consider, once again, the hypothetical conversation between Observers A and B outlined in the preceding section 3.3.1. Recall that Observer B proclaims with absolute certainty that $\frac{a}{b} = \frac{c}{d}$ is true almost immediately upon hearing the question. A knowledgeable reader can confirm that this fact can be "read right off the diagram" as it is quickly an obvious fact. However, now recall that when asked to prove this claim, an increased level of processing is required from Observer B. It is as if the conclusion is instantly discovered, yet the Turing-equivalent reasoning to arrive at this conclusion must be reverse-engineered! This would suggest that the conclusion is ascertained by some other means (and Observer B would surely affirm that his reasoning process does not follow that of Diagram in order to arrive at the ultimate claim).

To put this in another light, let us appeal to a similar line of reasoning presented by Bringsjord & Bringsjord (1996). In our specific example, we present a diagram which can clearly be processed by both humans and machines (albeit in a suggested different manner) to arrive at the same conclusions. In Bringsjord and Bringsjord's terminology, we have presented an example of a "simple diagram" or S-D:

> ...D is an S-D if and only if D and the diagrammatic reasoning thereon can clearly be

> fully represented in some logical system $\mathcal{L}$. (Bringsjord & Bringsjord 1996, pg. 387)

While our presentation of an S-D may convince some readers that human diagrammatic reasoning processes are hypercomputational in nature, we realize that the evidence is somewhat circumstantial. Thus Bringsjord and Bringsjord push to reveal images that are *not* realizable in some logical system. They call these TEMIs (Temporally Extended Mental Image):

> A TEMI is the sort of image which those who are expert in imagistic fields routinely process. That is, TEMIs are not simply 'visual aids', like drawings [and whether or not S-D's are used as drawings, they do appear to be always *usable* as such; consider, in this connection, Euclid's (1956; trans (Euclid 1956)) own reasoning]; they are much more robust. For example, consider a screenwriter. Many mature screenwriters are able to write complete drafts of movies 'in their heads'. That is, they are able to watch a movie play out before their mind's eye, from beginning to end, replete with dialogue, props, intonation, scene changes, camera angles, and so on. (Bringsjord & Bringsjord 1996, pg. 387)

We do not duplicate the full argument here, but appealing to lines of reasoning from (Block 1981, Jackson 1982, Nagel 1974, Searle 1992), Bringsjord and Bringsjord suggest that TEMIs cannot be fully reduced to some symbolic form like our toy diagram or any S-D.[39] Thus if this is the case, then human diagrammatic reasoning processes cannot be fully encapsulated by some symbol manipulating Turing machine which is suggestive that hypercomputation is at work.

3.3.3 The Penrose-ian Argument

In section 3.2, we pull out all the stops to revive Penrose's Godëlian case from (Penrose 1994, Penrose 1989) against Bringsjord and Zenzen's attack in (Bringsjord & Zenzen 2003). We invite the reader to revisit that defense as we again directly borrow from Penrose's line of reasoning to support our claim that hypercomputation lies within human diagrammatic reasoning processes. Recall that Penrose's original argument (section 3.2.1) supports the view that it is not possible to encapsulate all methods of reasoning available to humans for determining whether Turing machines do not halt into some defined Turing machine A. We later claim that this is a result of human hypercomputational ability. Our goal for this section is to provide a parallel to Penrose's argument to directly inject human diagrammatic reasoning processes into the hypercomputation debate.

For the purposes of this discussion, consider the nature of diagrams as we have described them. In order to invoke machine reasoning on diagrams, we must digitize their

[39] See (Bringsjord & Bringsjord 1996) for the full defense

representations in some way. Therefore, assume that given some diagram N, we can refer to it and perform computations on it through some digitized representational form n.[40] Additionally, we wish to make statements about diagrams and encode these statements in some representational format as well. Since these statements are already symbolic, it is trivial to represent them in some notational symbolic form.

With this groundwork established, let us now consider a Turing machine D that can reason about diagrams. Not only does it reason about diagrams, but encapsulates the same capabilities as humans for determining true statements about diagrams. By definition, the machine takes as input a diagram n and a statement k about n and operates as follows:

$$D(n,k) = \begin{cases} 1 & \text{if } k \text{ is true with respect to } n \\ \infty & \text{if } k \text{ is false with respect to } n \text{ or unknown}[41] \end{cases}$$

Now as of yet, we have not discussed representing the execution of Turing machines over time in diagrammatic form.[42] Therefore, consider the very simple Turing machine shown in figure 3.2. We can represent a portion of its execution on an infinitely blank tape over time as a diagram shown in figure 3.3. It is important to note that while this diagram consists entirely of symbolic symbols, what makes it diagrammatic is the structural layout of these symbols. Thus notice that the bold symbol represents the current read head on the tape and each of the first 20 steps of the machine are graphically aligned to visually demonstrate the movement of the read head on the tape. Additionally, 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. Incidentally, it should be easy to deduce from this diagram that the Turing machine in figure 3.2 will never halt.

With this diagrammatic representation of the execution of Turing machines, we can now proceed to the crux of our argument. Consider a diagram S and its symbolic description s. We can describe S as a diagram that depicts the operation of Turing machine D over some finite number of steps when run on the input $s \bullet t$. In this particular case, t is a representational form of the statement: "S is a diagram of a non-halting Turing

[40] It is easy to see how this digitization could take place. Similar to the process outlined in section 3.3.2, think of a diagram as an image which is an array of pixels. Each pixel can be transformed into its corresponding RGB (Red/Green/Blue) color value and encoded as such. This array of pixels can then be transformed into a continuous sequence of symbols which we denote as n above. Clearly this, or some similar mechanism, is how digital computers treat images at their base level.

[41] We use ∞ to indicate that the machine does not halt.

[42] We do allude to this possibility in section 3.3.1

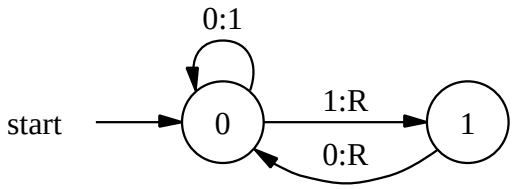


Figure 3.2: Simple Turing Machine Example

```
0              State 0
1              State 0
10             State 1
100            State 0
101            State 0
1010           State 1
10100          State 0
10101          State 0
101010         State 1
1010100        State 0
1010101        State 0
10101010       State 1
101010100      State 0
101010101      State 0
1010101010     State 1
10101010100    State 0
10101010101    State 0
101010101010   State 1
1010101010100  State 0
1010101010101  State 0
```

Figure 3.3: Simple Turing Machine Example Execution

machine execution." Thus in a classical self-referencing case, Turing machine D is asked to determine whether it does not halt given a representation of itself as input.

Consider the possible responses of D in this case. If $D(s,t)$ returns 1 then it means that t is true. However, this would mean that S is a diagram of a non-halting Turing machine and S is a diagram of the execution of $D(s,t)$ itself. Therefore, if $D(s,t)$ returns 1, it is saying that it does not halt even though it does. The only other possible behavior for $D(s,t)$ is that it does not halt and therefore we can immediately conclude that $D(s,t)$ does not halt (since the other possibility leads to a contradiction). However, this means that we have just determined a truth about a particular diagram S that D cannot itself determine. Thus we cannot possibly construct such a machine D that encapsulates *all* methods available to humans to reason about diagrams.

Clearly this argument is extremely similar to Penrose's argument that we defend in section 3.2. In this defense we attribute the human methods about determining whether a Turing machine does not halt to the hypercomputational processes that exist in nature (and thus that the human brain is fundamentally subjected to). By a similar line of reasoning, our new Penrose-ian case that we outline here allows us to now directly attribute this hypercomputational power to diagrammatic reasoning processes used by humans.

3.3.4 Anchoring Example

As a demonstrative example to illuminate the human diagrammatic reasoning processes that we speak of in the preceding sections, consider the Turing machine depicted in figure 3.4. Clearly the representation of the machine in this form yields no immediately apparent conclusions about the behavior of this machine in terms of its non-haltingness. However, similar to the Turing machine execution shown in figure 3.3, we can represent the execution of the initial steps of this machine (when run on an infinitely blank tape) in diagrammatic form as shown in figure 3.5. In this format, it is immediately apparent to the trained observer that this machine sweeps the read head back and forth across the tape in a repetitive manner, continually pushing the right and left boundaries of each successive sweep outwards.

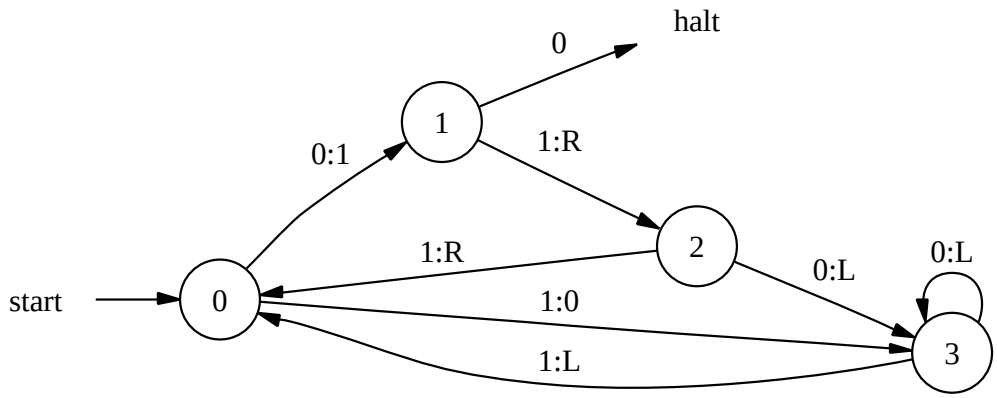


Figure 3.4: Example Turing machine to demonstrate diagrammatic reasoning

In the context of our assault, the machine that we have just described exhibits a

proven non-halting behavior known as a "Christmas Tree" pattern.[43] As we shall see in section 4.2.4, the automated process to establish that Christmas tree machines do not halt requires extracting a certain set of partial tape components as well as confirming that a series of specific transformations on those components hold true in the context of the specific machine.[44] However, consider once again, the diagrammatic representation of the machine depicted in figure 3.5. It should be easy to convince any observer with proper knowledge of Turing machines that this machine does not halt. Even more importantly, though, this persuasion should be possible to do *without* appealing to the formal definition of Christmas trees as they are defined in section 4.2.4. In fact, the repetitive behavior is so apparent, that it can seemingly be simply "read off of the diagram."

What does this process of "reading" entail though? Is it simply some embedded brain process that is analogous to the automated reasoning process for Christmas trees already mentioned? Perhaps one could attempt to argue that this is so. However, consider the reasoning experiment outlined in section 3.3.1. In this experiment, Observer B was absolutely convinced of the truth of a particular statement concerning figure 3.1 *before* he was able to formulate a symbolic logical proof. By the same vein, it should be nearly indisputable that an intelligent observer of figure 3.5 could produce the following chronological line of reasoning:

1. The observer is positive with *absolute conviction* that the Turing machine in question does not halt.

2. *After* the observer has assured himself of this fact, it takes him several minutes, hours, or even *days* to construct a certifiable symbolic proof that parallels the lines of reasoning about Christmas trees in section 4.2.4.

Thus the question we ask is: How can an intelligent observer be convinced that a Turing machine does not halt and yet still require a significant additional amount of time to formulate a symbolic explanation if, at the base level, humans' thought processes can be reduced to some symbolic form? We once again appeal to the Searle-ian argument presented in section 3.3.2. Here we note that, while a computer might be capable of processing images in ways that externally resemble human reasoning, computers can never have any concept of what images "look like," nor can they "visualize" diagrams in the

[43] See section 4.2.4 for a complete description of the "Christmas Tree" non-halting behavior.

[44] Essentially, the Christmas tree behavior (and many other behaviors that we shall describe in chapter 4) is analogous to a grammar that defines a set of transformations that can be shown to repeat in an infinite manner.

same manner that humans do. Thus, the concept of "seeing" and "visualizing" is not something that can be explained in a Turing-equivalent way.

Barwise & Etchemendy (1995) allude to this notion:

> People who accept the argument against a universal diagrammatic system, and so accept the idea of a heterogeneous reasoning system, often suppose that in order to have a rigorous heterogeneous system, there must be *some* system of conventions into which all the others can be embedded and compared, some sort of "interlingua" to mediate between the various systems of representation. But this is not correct. Whether it is *useful* to have an interlingua is debatable, but there is certainly no logical necessity to employ one.

Here they verbalize the notion that it is not necessary to have an interlingua between a diagrammatic and symbolic reasoning system that are combined together in a heterogeneous reasoning system. While their arguments are fleshed out in the context of their own heterogeneous system *Hyperproof*, the argument, when coupled with Bringsjord & Bringsjord's (1996) claims about TEMI's,[45] allows us to parallel the human reasoning system as also being some heterogeneous flavor. Thus, the human diagrammatic reasoning process can be considered completely independent and *irreducible* to the human symbolic reasoning system. In the context of our example, therefore, the observer is able to ascertain that the Turing machine does not halt through hypercomputational diagrammatic reasoning mechanisms, and then independently formulates the symbolic line of reasoning.

3.4 Busy Beaver Solvability Claim

At this point, we hope that the discussions from the preceding sections have convinced the reader that there is truth to the following claims:

A. Hypercomputational processes exist in nature and hypercomputation is the only explanation for some physical laws.[46]

B. The human mind is grounded in these hypercomputational physical processes and thus possesses the power to hypercompute.[47]

C. Humans can utilize their ability to hypercompute to *specifically* reason about whether Turing machines do not halt.[48]

[45] Refer to section 3.3.2 for an analysis of Bringsjord and Bringsjord's claims in the context of our arguments

[46] See section 3.1.3

[47] Here we draw from the reasoning set forth in section 3.2 with explicit connection to the facts presented in section 3.1.3

[48] See our defense of Penrose's core argument in section 3.2

D. Human diagrammatic reasoning is a *specific* case of humans' ability to hypercompute about whether Turing machines do not halt.[49]

Thus from this knowledge, we have directly opened the possibility that the Busy Beaver problem is computable (not by some Turing-equivalent machine, of course, but by a human hypercomputer). Appealing to our original proposed algorithm (Algorithm 1) as a solution to the problem, we can see that our original hurdle (determining whether a machine is a halter or non-halter) is now potentially possible[50] within the context of human capabilities. In the following chapter, we present a first-hand assault on the Busy Beaver function as the initial beginnings of our solution to the problem.

[49] See section 3.3

[50] While we have not explicitly proven that humans can provide a full solution to the halting problem, we have shown that there exists at least one non-halting Turing machine that a human can deduce as such while a Turing machine cannot. If this were not the case, then machine A from Penrose's original argument could exist. We have shown in section 3.2 that this is not the case.

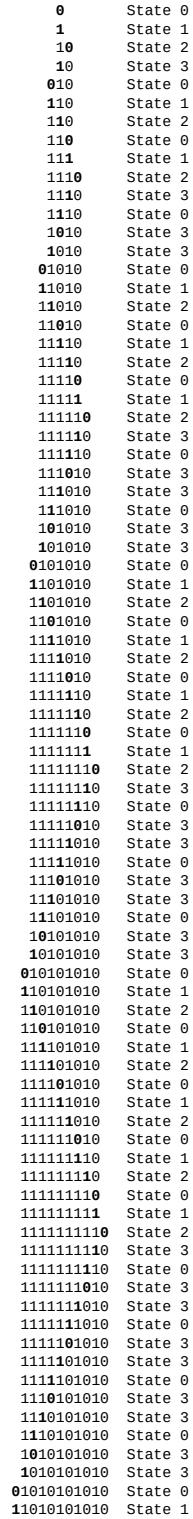


Figure 3.5: Diagrammatic Representation of Turing machine execution

CHAPTER 4
The Busy Beaver Assault

Now that we have hardened our position on hypercomputational abilities of humans in the preceding chapter, we now turn to the core of our thesis: that these capabilities can be harnessed to compute values of the Busy Beaver function. Our attack can be considered three layers deep. We first present a Turing machine enumeration strategy that significantly reduces our search space in the spirit of the initial steps outlined in Algorithm 1. Secondly, we outline an array of explicitly proven non-halting behaviors that we have embedded into specific detection routines that empowers us to classify certain classes of machines as non-halters.

As it turns out, our initial attack consists of constructing a computer program that enumerates the set of Turing machines corresponding to some n representing the desired number of states in the machines.[51] We then construct additional programs which encapsulate the detection strategies for the said set of non-halting behaviors and use these programs to test whether each of the machines in our enumeration is a non-halter. Thus if we continue indefinitely with this process, we are doomed to ultimately fail because we are attempting to do two things which we have already deemed to be impossible:

1. Construct a Turing machine that solves $\Sigma(n)$ for any arbitrary n.

2. Encapsulate all mechanisms available to humans to reason about whether Turing machines do not halt into one procedure.

Thus our goal in this endeavor is to dramatically reduce the machines that we must address hypercomputationally by automating the decision process for those that can be handled in some Turing-equivalent manner.

The result is that we are left with some set of machines that we have neither shown to halt nor not halt. The third layer in our attack involves the specific classification of the behavior of these machines, directly appealing to human diagrammatic reasoning in this case. Thus without further delay, we present our attack below:

[51] This initial program was constructed by Kyle Ross (2003) whom we are indebted to for allowing us use of his work. Our end results build off of his original code.

4.1 Turing Machine Enumeration Strategy

As it is described by Ross in (Kellett, Ross, Bringsjord & van Heuveln forthcoming), the search space associated with determining values of $\Sigma(n)$ via our approach is intimidating:

> 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 ... 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.

Thus the following adapts elements of discussions in (Ross 2003, Kellett et al. forthcoming) which encapsulate Ross's exclusive work. In this work, he employs certain optimization techniques to enumerate a minimal subset of this search space that eliminates redundancies and yet still maintains the integrity of a complete search.

4.1.1 Search Space Redundancies

There are four core types of redundancy that Ross (2003) originally presents as part of the $\Sigma(n)$ search space. We thus outline them briefly:

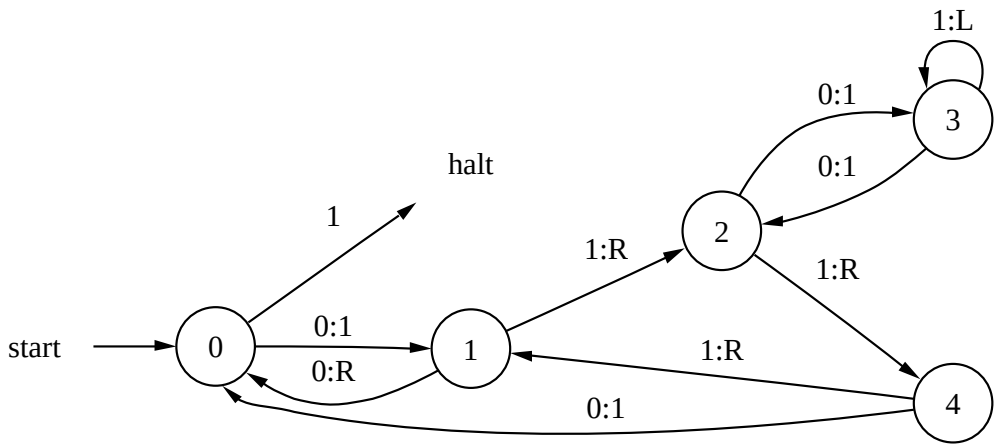


Figure 4.1: B(5) Champion

Isomorphic machines are machines that are identical in structure, but have different arrangements in the configuration of their states. Thus consider the machines il-

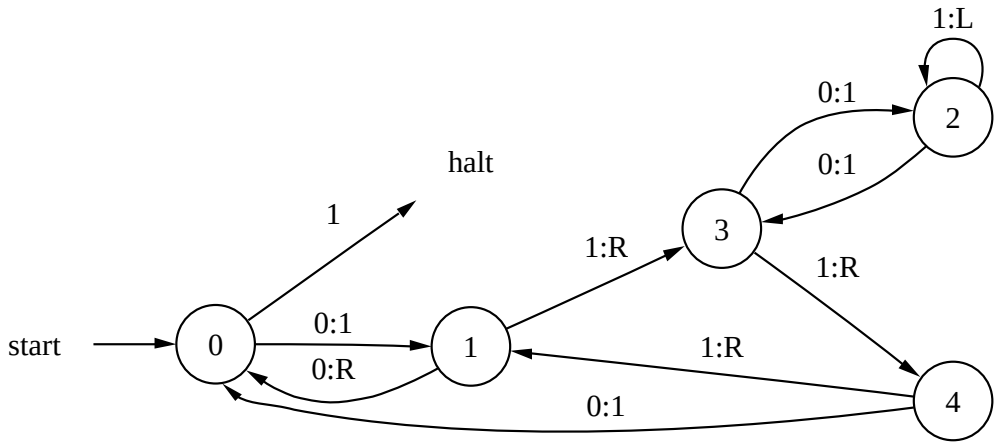


Figure 4.2: B(5) Champion Isomorph

lustrated in figures 4.1 and 4.2.[52] These machines are clearly identical in structure and behavior but states 2 and 3 have been swapped. Ross points out that: "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 behavior will be identical."

Unused state machines such as the one illustrated in figure 4.3 are Turing machines which contain states that cannot be reached (notice that there are no transitions defined that terminate in states 4 or 5 for this machine). Again, Ross explains: "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."

Empty tape machines are Turing machines whose tape is re-established as an infinite sequence of 0's after some set of executed transitions. It can be shown that for such a machine, there is always another machine in the search space with equivalent behavior.[53] Thus we need not consider these either.

Mirror machines are intuitively Turing machines that exhibit the exact mirror behavior

[52] Note: All figures referenced in this Turing machine enumeration strategy section (section 4.1) are taken from (Ross 2003).

[53] Again, we direct the reader to (Ross 2003) for an explanation of this claim.

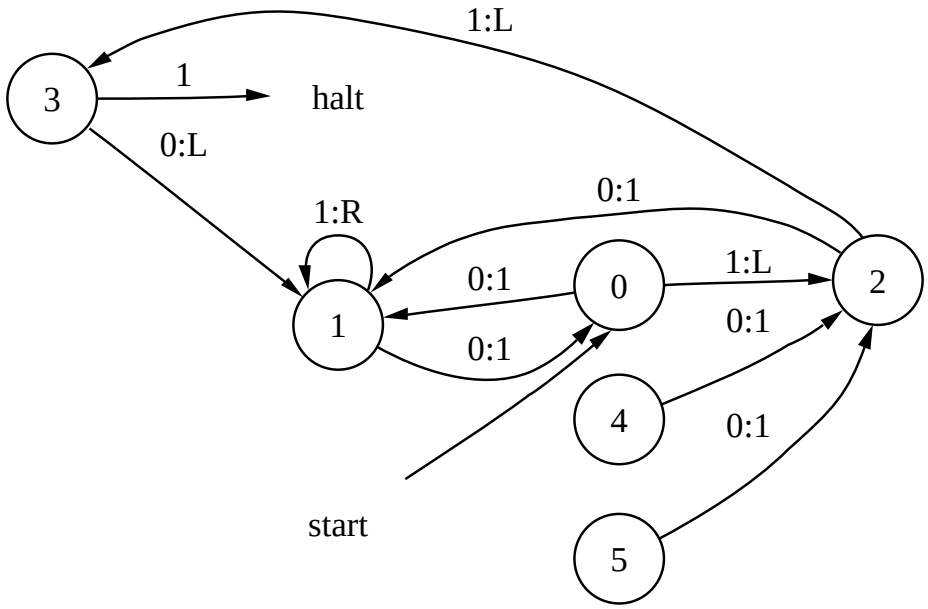


Figure 4.3: A 6-state Turing Machine with 2 unused States

of some other machine in the search space. Thus the mirror machine M^c of Turing machine M would have the identical specification of M except all transitions that define an action of L would be replaced by an action of R and subsequently all R actions would be replaced with L's. Therefore, since their behavior is equal, only one of these machines needs to be considered.

4.1.2 Solution: Tree Normalization

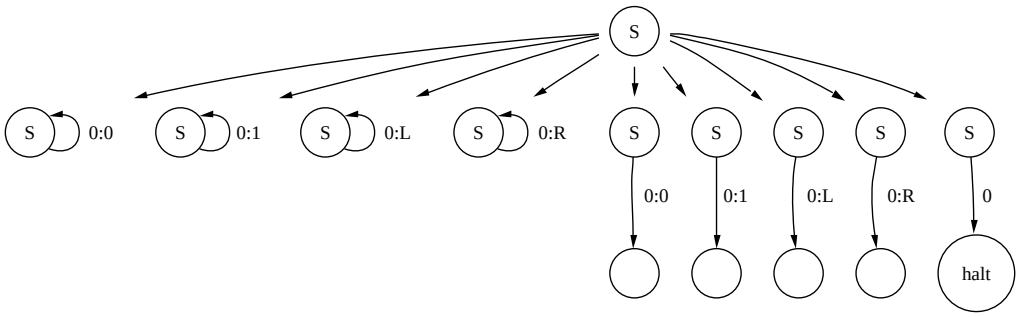


Figure 4.4: 0th and 1st Levels of the Normalized Tree

As a solution to eliminate the redundancies just described in the previous section, Ross (2003) suggests a tree normalization enumeration technique. Consider the beginnings of this tree illustrated in figure 4.4. At the root of this tree is a basic Turing machine with one state and no transitions defined. This machine is consequently run on an infinite, blank tape. Clearly, it will do nothing, as no transitions are defined. However, Ross considers this a "partial machine" since the machine halts, but is not in a halting state.[54] The conditions under which it halts, therefore, are not halting conditions, but rather opportunities to create additional transitions. In the case of the root node, it halts in state 0, reading a 0 on the tape. Therefore, we create child machines, each of which contains a transition for the case in state 0, reading a 0. We enumerate all possible transitions for this case to all possible existing states, as well as to one of the remaining unused states (if the number of currently used transitions is less than n for which we are calculating $\Sigma(n)$). The child machines are then pushed onto a stack, and successfully popped off, with the above general procedure applied to each until all possible machines have been defined.

It is easy to see that no machine which qualifies as an unused state machine specified above will be generated with such a schema. This is because a particular transition is only defined if a machine is guaranteed to reach the particular state and read symbol for this particular transition. Therefore, no transition can ever be defined from a state in which there are no terminating transitions. Also, it can be proven that this technique never generates a pair of isomorph machines as specified above. (Ross 2003)

Additionally, Ross explains in (Kellett et al. forthcoming) how this technique can be leveraged to account for empty tape machines:

> 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, as shown in fig. 4.5; compare this against fig. 4.4. ... We also implement the generalized 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.

In a similar vein, we can eliminate the problem of mirror machines by forcing the first

[54] It is clearly valid by mechanisms of implicit halt machines to halt due to simply a lack of a defined transition. However, in the context of the tree normalization schema, we use this lack of a defined transition to *define a new transition* and subsequently a sub machine in the tree. A consequence of this is that for one of the sub machines, we define *no transition* to denote an implicit halt machine, or as it is denoted in figure 4.4, a transition to a halt state with no action defined (which behaviorally is identical).

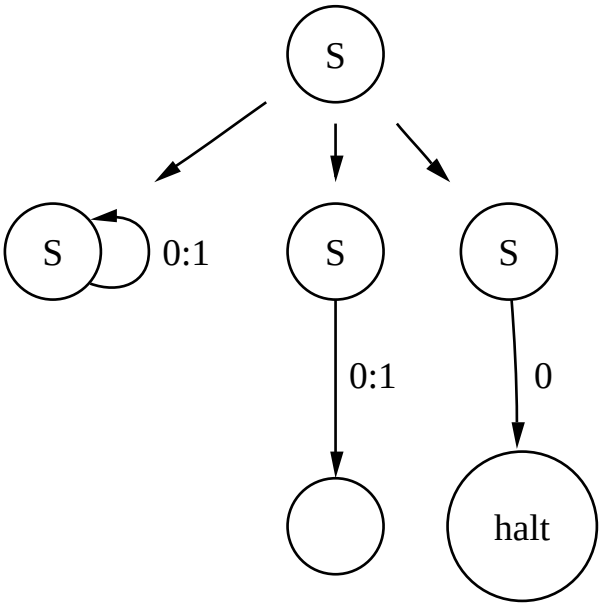


Figure 4.5: Normalization Tree Pruned Based on Forcing First Write

move of any Turing machine to be to the right.[55] The end result is the finalized initial tree shown in figure 4.6 Thus by incorporating this tree normalization mechanism, we are able to eliminate the redundancies presented by isomorphic machines, unused state machines, empty tape machines, and mirror machines from the Busy Beaver search space.

4.1.3 Additional Redundancies

Ross (2003) also encodes detection mechanisms for two additional redundancies in the search space that cannot be directly encoded in the tree normalization strategy but instead must be implemented as a separate component:

Simple non-productive transitions are transitions that read a symbol and then write the same symbol onto the tape. Ross explains in (Ross 2003) that machines with these transitions can be pruned from the search space.

Complex non-productive transitions are similar to their simple counterpart except they are stretched out into two sequential transitions. Thus, as it is explained by Ross (2003): "effectively, the first transition replaces the symbol s with the symbol

[55] This could easily have also been solved by forcing first move to the left. This was simply an arbitrary choice made by Ross (2003).

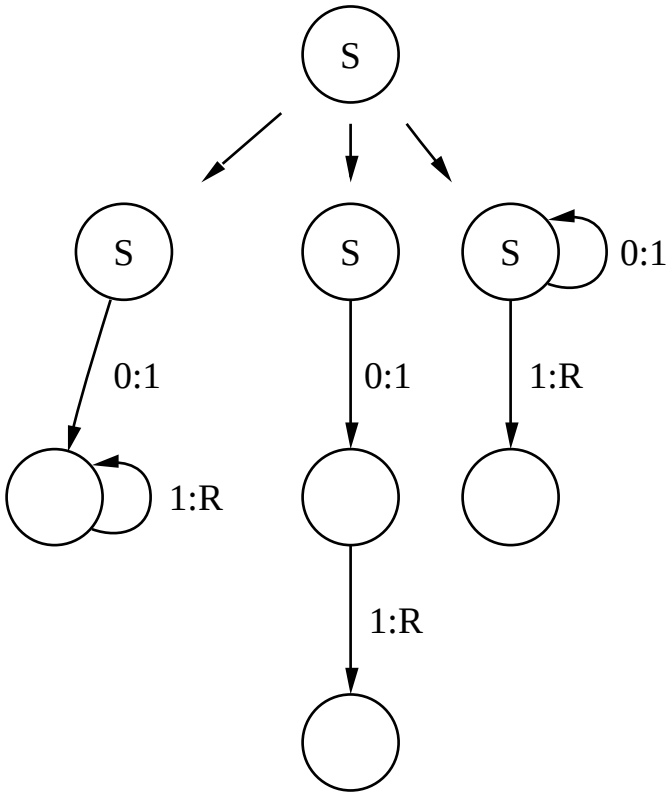


Figure 4.6: The 3 Start Machines

s' and the second transition reverses this action." Machines with such properties are similarly redundant.[56]

Ross explains in (Kellett et al. forthcoming) that these optimization techniques described including the tree normalization schema as well as the non-productive transition redundancy detection mechanisms produce an effectively complete and optimal search space that is a greater than 99.9999999% reduction in the search space for $\Sigma(6)$! Thus the foundation is in place for us to press forward in our assault.

4.2 Non-Halt Detection Mechanisms

As has already been discussed in section 2.3.3, clearly the greatest barrier in calculating the value of the Busy Beaver function for a particular value of n using our approach is

[56] We invite the reader to see (Ross 2003) for a more thorough explanation of these redundancies as well as the tree normalization schema described in section 4.1.2.

the determination of whether a machine halts. Despite the hopelessness of our "dream" described at the end of said section (2.3.3), we have breathed new hope into the enumeration possibilities of the set of non-halt detection routines $\{NH_0, NH_1, NH_2, \ldots\}$ in chapter 3. Thus, inspired by the brilliant work presented in (Lin & Rado 1964, Brady 1983, Machlin & Stout 1990),[57] we begin our barrage on enumerating this set:

The following adapts elements of discussions in (Brady 1983, Machlin & Stout 1990).[58] We use a modified notation 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. 4.1.1) respectively. A *word* or *tape component* is defined as an arbitrary length continuous sequence of characters on the tape and is represented by $[X]$[59]. 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 concatenated together.

4.2.1 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}

[57] It should be noted that the three works mentioned here all deal with the quintuple formulations of the Busy Beaver problem. The corresponding non-halt behaviors described, however, are still relevant to the quadruple formulations.

[58] The initial non-halt behaviors presented in this section are directly derived from Brady's (1983) original work as well as Machlin & Stout's (1990) derivative efforts. Specifically, our base definitions of backtrackers, simple loops, Christmas trees, and counters are adapted from the corresponding presentation of these behaviors in (Machlin & Stout 1990). Additional modifications of these behaviors as well as other unique non-halt patterns presented later are the result of our original work. (It should be noted, though, that Brady (1983) makes brief reference to some of these additional behaviors without describing them in detail.)

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

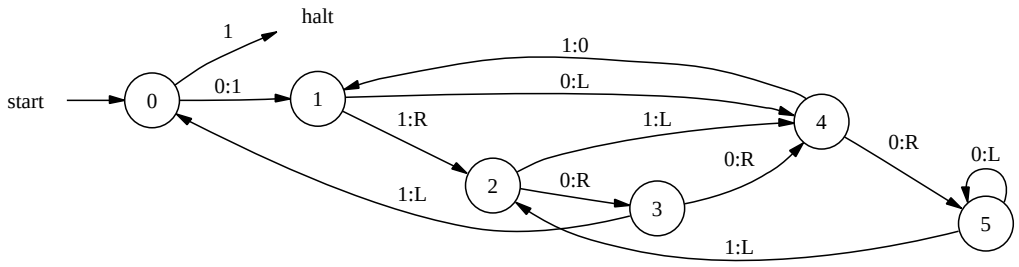


Figure 4.7: Back Tracking example

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.[60] 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.

4.2.1.1 Back Tracking Example

Consider the machine in fig. 4.7. 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

[60] 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

time: 11_3[61]

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.

4.2.1.2 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. This algorithm, specified in Algorithm 3 is adapted from that put forth by Machlin & Stout (1990) and as they so gracefully put it: "While backtracking can be useful, it cannot be guaranteed to always stop since otherwise it would supply a solution to the halting problem." Intuitively, therefore, some non-halting Turing machines cannot be proven as such by this procedure. Attempts to apply this procedure to these machines causes the algorithm to "infinitely backtrack." 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.

4.2.2 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.

[61] 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

```
   function: Backtracker(M)
 1 foreach {state, symbol} pair a in the set of pairs as specified in sect. 4.2.1 do
 2     Construct a local tape configuration x which must exist in order for
       machine M to perform the transition defined for a
 3     Backtrack(a,x)
 4 end
 5 if every {state, symbol} pair in the above set produces a false return value
   when backtracking then
 6     The machine cannot possibly halt and is classified as a non-halter
 7 else
 8     The proof fails and the results are inconclusive
 9 end

   function: Backtrack(b, y)
10 let b be a {state, symbol} pair parameter and y be a local tape configuration
   parameter in
11     foreach {state, symbol} pair c for which there is a transition defined that
       terminates in the state of b do
12         Construct a local tape configuration z such that:
13           (a) the transition defined for c will be performed on this tape
14           (b) performing this transition results in a local tape that matches y
15         if such a tape cannot possibly be created then
16             return false
17         else
18             return Backtrack(c,z)
19         end
20     end
21 endlet
```

Algorithm 3: Backtracking non-halt detection algorithm

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. 4.8 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:

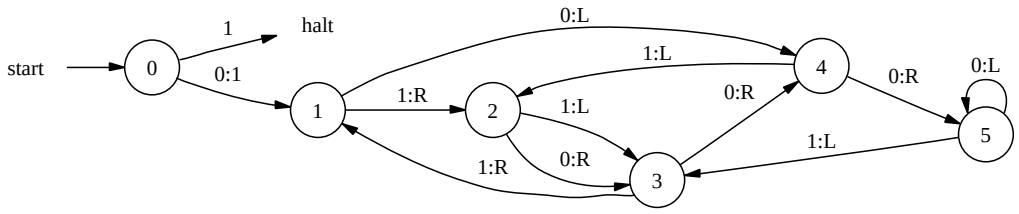


Figure 4.8: Subset Loop machine

Recall from sect. 4.1.2 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.[62] 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. 4.1.2 for additional clarification.

4.2.3 Simple Loops

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

   (a) The same tape configuration is reached at some later point.

   (b) The tape configuration $0^*[C][V][X_s][Y]0^*$ is reached at some later point b *and* between points a and b, the read head never moves past the left edge of the

[62] The tree normalization machine enumeration strategy generates machines by adding transitions to partially run machines. When a partially run machine reaches a {state, symbol} configuration where no transition is defined, it branches off, creating a machine for each possible transition that would be defined for this state and symbol. Therefore, every transition that is created in this manner is guaranteed to be used at some point during the execution of the machine.

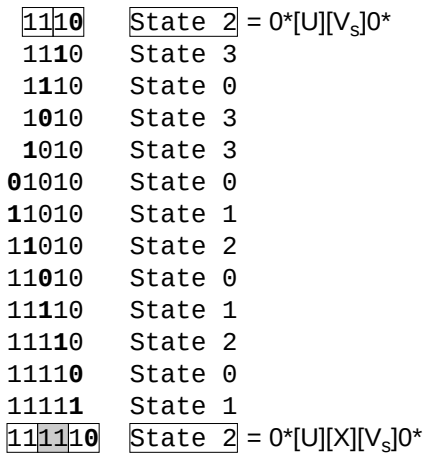


Figure 4.9: Christmas Tree execution

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.

4.2.4 Christmas Trees

"Christmas Tree" machines, studied rigorously by both Brady (1983) and Machlin & Stout (1990), are classified as non-halters due to a repeatable back and forth sweeping motion which they exhibit on the tape. A behavior that we have already touched upon in section 3.3.4, 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. 4.9. 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,

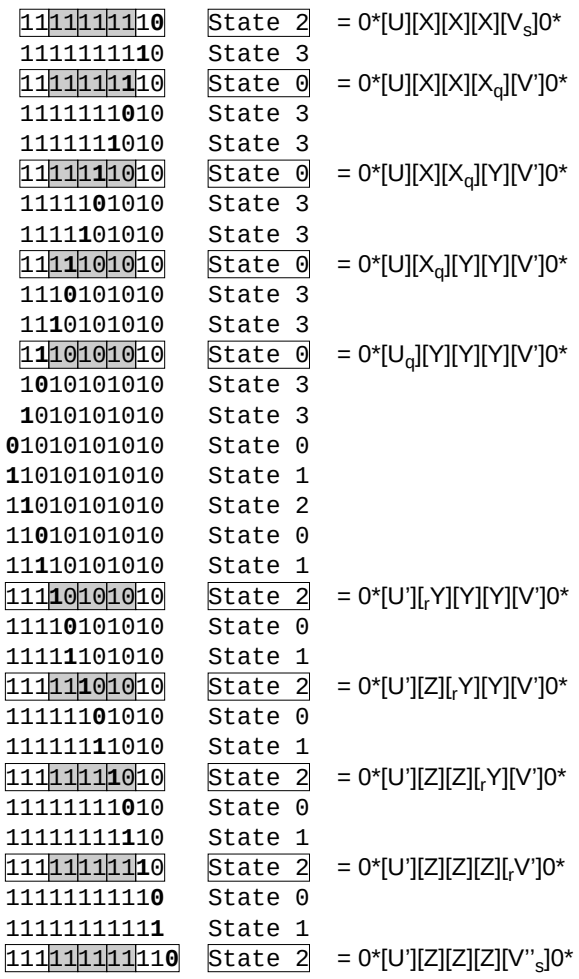


Figure 4.10: Christmas Tree execution2

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:

4.2.4.1 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. 4.10. 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 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. 4.10. 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.

4.2.4.2 Christmas Tree Formalization

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

> Formally, a Turing Machine M is a *Christmas tree* if either M or M^c satisfy the following conditions for some ... state s:
>
> 1. There are ... 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 ... words and q and r are ... 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 transformation from

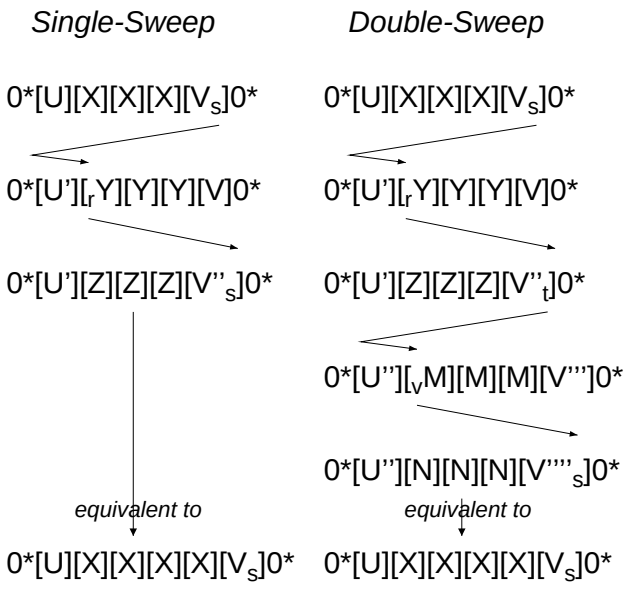


Figure 4.11: Comparison of cycles between single-sweep and double-sweep Christmas trees

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

4.2.5 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. 4.11 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 ma-

chine 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.

4.2.5.1 Double-sweep Christmas Tree Formalization

Brady (1983) pioneers the concept of a double-sweep Christmas tree referring to them as "alternating Christmas trees." He does not, however, formalize their behavior, only making quick reference to their similarities to Christmas trees. 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. 4.2.4.2 concatenated together. Thus we define these rules:

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

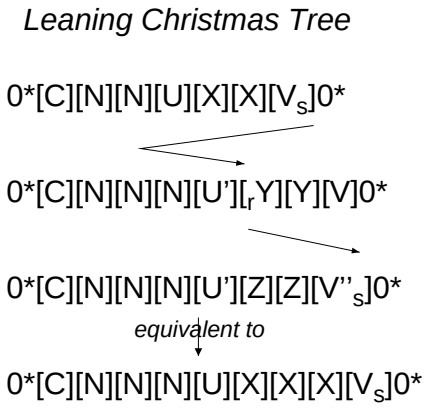


Figure 4.12: Representative cycle of a leaning Christmas tree

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.[63]

4.2.6 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 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 (or rightmost for the corresponding mirror specification) 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.

[63] 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.

Refer to fig. 4.12 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 $[N][U_q][X'] \Rightarrow [N][N][U'][Y']_r$.

4.2.7 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 (1983) 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 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 (or its corresponding mirror 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:

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

Figure 4.13: Execution of a counter Turing machine

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. 4.13 for an illustration of this process.[64]

4.2.7.1 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 correct 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.

[64] 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.

4.3 Revised Attack Strategy, Results, and Records

Recall from section 2.3.3 our proposed solution to $\Sigma(n)$ specified in Algorithm 1. In light of the behavioral patterns that can be explicitly shown to exhibit non-halting behavior just described in section 4.2, we can now modify our basic algorithm designed to solve $\Sigma(n)$. Refer to Algorithm 4 for this presentation. The core of this algorithm is the set

$$\{NH_0, NH_1, NH_2, NH_3, \ldots\}$$

which we have already introduced in section 2.3.3. This set, referred to in abbreviated form as NH, is proposed to represent an ever increasing set of non-halting behaviors. Once again, we are faced with issues of Turing-computational impossibility. As has already been addressed in our defense of Penrose's argument in section 3.2, we cannot hope to encapsulate all mechanisms of human understanding available to demonstrate that Turing machines do not halt into one well defined procedure. If Algorithm 4 has any hope of being a solution to the Busy Beaver problem, it would appear to be the case that we are attempting to attain just that in NH.

This is not so, however. Consider what Algorithm 4 actually entails. Clearly, on line 6 we suggest an ever increasing set of well defined, proven non-halt behaviors that are used to question the non-haltingness of some Turing machine t. Consider, however, how this set is achieved. It is in fact continually built by the statement in line 20 of the algorithm. What is important to note is that line 20 implies direct intervention by *humans*. The intent of the algorithm is for humans to continually add non-halt behaviors to NH by examining and analyzing unproven holdout machines. Thus, at any particular point in time, NH encapsulates some set of non-halt behaviors that can be ascertained by the mechanisms of human thought. However, it can *never* encapsulate all of them because it is an infinite set being enumerated by humans and not some Turing-computational machine.[65] At any particular point in time, therefore, it is simply a finite subset of this infinite set.

4.3.1 Implementation Methodology

As we have mentioned previously, our attack builds off of a previous program provided by Ross (2003). This original program is written in C++ and encapsulates the Turing machine enumeration strategy described in section 4.1 using the tree normaliza-

[65] We have already addressed the potential for human enumeration capability exceeding the Turing limit in sections 3.1 and 3.2. Thus while we do not claim that our revised proposed algorithm for $\Sigma(n)$ is a cut and dry solution, we do claim and defend the possibility.

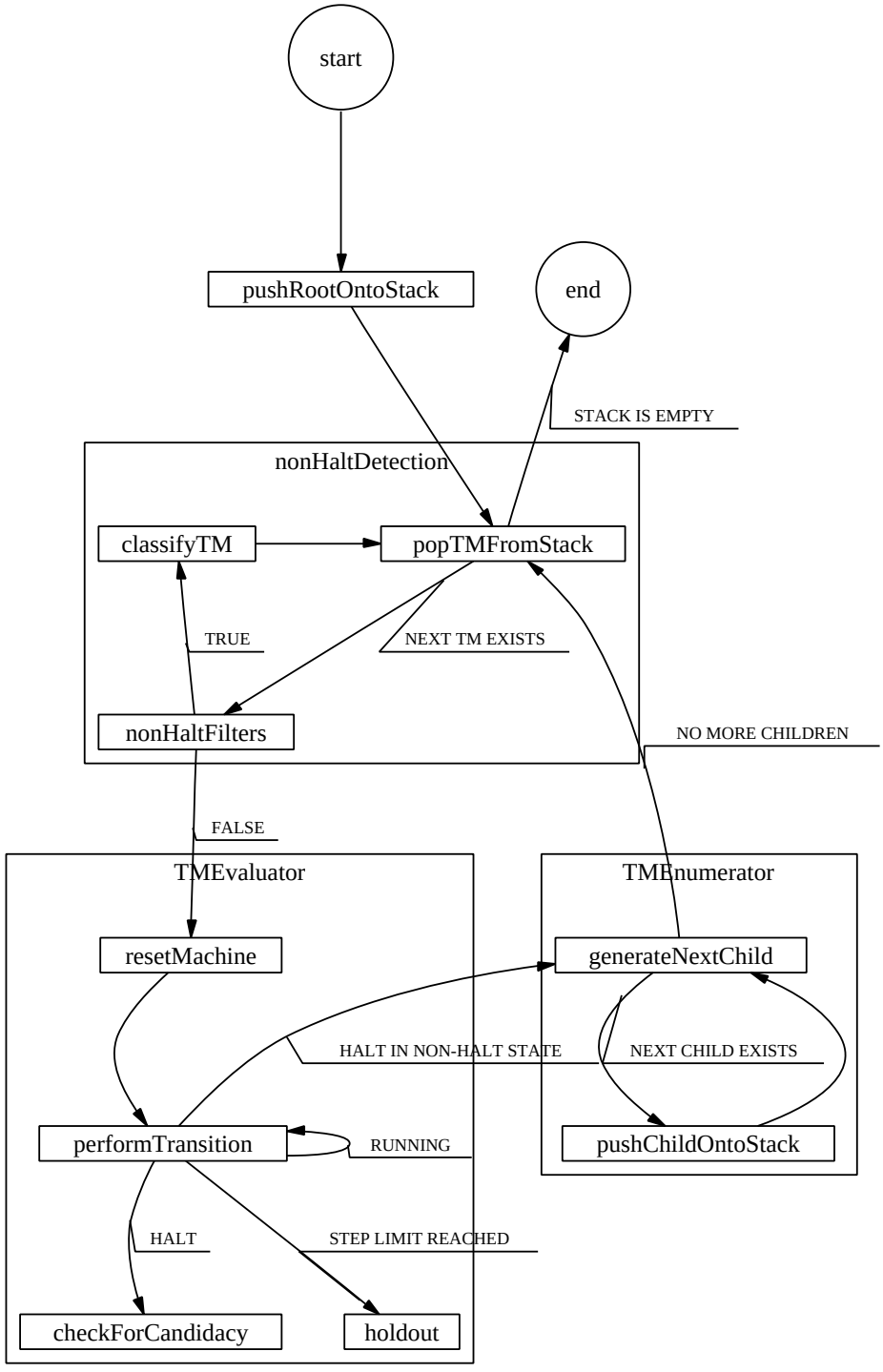


Figure 4.14: Representation of the program control sequence used in our implementation

function: $\Sigma(n)$

```
1  repeat
2      H = ∅
3      C = ∅
4      Using a tree normalized approach, enumerate a set S of n-state Turing
       machines that behaviorally represents the entire set of n-state machines
5      foreach machine t in S do
6          Attempt to show that t exhibits one of the defined non-halting behaviors
           in NH
7          if The attempt to prove t a non-halter fails then
8              Prepare input tape x as an infinite bi-directional tape of all 0's
9              Run t on x for a predetermined limit of steps
10             if t halts before this step limit is reached then
11                 Consider the resulting output tape x'
12                 if x' satisfies the conditions specified in section 2.2.1.3 then
13                     Add t to our candidate set C
14                 end
15             else
16                 Add t to our holdout set H
17             end
18         end
19     end
20     Examine each machine in H and define new non-halting behaviors and
       detection techniques
21     Add these new behaviors to our non-halting behaviors NH
22 until |H| = 0
23 Return the productivity of the most productive machine in C (i.e. the machine
   that produces the most contiguous 1's on the tape)
```

Algorithm 4: Revised proposed solution to $\Sigma(n)$

tion techniques. Thus, Ross's initial attack incorporates an algorithm similar to that described in Algorithm 1, except in place of detecting non-halters in line 3 of this algorithm, he simply specifies a step limit for execution of each Turing machine. Thus if the machine has not halted after this arbitrary number of steps is made, it is considered a non-halter and discarded from consideration.

As a remedy for this problem, we have encoded the non-halt detection mechanisms described in section 4.2 as C++ routines that mesh into Ross's original program. Refer to fig. 4.14 for a graphical representation of the overarching approach. This figure adapts our proposed Algorithm 4 presented above. We use a stack-based approach, beginning the stack with the three machines enumerated in fig. 4.6. 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 applied is based largely on simulation tests to determine which routines are most efficient and which behaviors are most prominent 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[66]
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 which is described succinctly in (Ross 2003):

> 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.

What should be noted is that the inclusion of the non-halt detection routines into Ross's original program (as opposed to the basic step limit approach) allows us to increase

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

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

the subsequent step limit of the execution stage as it has just been described.[67] The result is that we are able to establish much more productive records for any one particular n even if we cannot completely confirm $\Sigma(n)$ for this n.

4.3.2 Results

Given our defined strategy and implementation mechanisms, 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. 4.2), and machines pruned from the tree (according to the rules outlined in sect. 4.1). Tables 4.1 and 4.2 respectively outline the

[67] The reason for this is purely due to the extremely time consuming nature of running each machine for a large number of steps. If the step limit approach is used with a massive step limit, then *every* non-halter will be run for this number of steps. However, by incorporating the non-halt detection mechanisms into the algorithm, only the "holdouts" are run all the way to the step limit. Thus since the non-halt routines eliminate most of the non-halters (and in the cases of $n = 1, 2, 3, 4$ *all* of them), then only a few machines must be run to this limit.

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

overall statistics for the implicit formulations of the problem (B and O) and the explicit formulations (P and R).

Recall from sect. 4.3.1 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 4.1 and 4.2. 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 classifications 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

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	≥ 70*	≥ 3985*	≥ 41*	≥ 1256*	≥ 71*	≥ 3986*

Table 4.3: Status of $\Sigma(n)$ records from (Ross 2003)

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*

Table 4.4: Updated $\Sigma(n)$ records

also been run to the predefined 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 $\Sigma(n)$.[68] As is illustrated in Tables 4.1 and 4.2, the present effort has confirmed $\Sigma(n)$ for $n = 2, 3, 4$, and very nearly 5.

4.3.3 Records

We now turn to the established values and records for $\Sigma(n)$. Table 4.3 displays Ross's (2003) results up through $n = 6$ using his original program that includes only a step limit mechanism for conjecturing that a machine is a non-halter. The records marked with an asterisk are those that had been newly established by his efforts. Table 4.4 indicates our updated version of this table using the previously described program with the non-halt detection routines embedded in it as well. In this case, the marked records are those that have been established by our combined Ross-Kellett effort. 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.

[68] We generalize the four variants of the problem for simplicity's sake. In this case $\Sigma(n)$ 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.

4.4 Holdout Patterns: Human Perceptual Power at Work

Most of the behaviors described above in section 4.2 are defined more specifically as sets of concrete transitions that transform predefined localized portions of the tape into certain configurations. If it can be demonstrated that the set of transitions will repeat infinitely, and that a particular machine transforms the tape in a manner identical to each of the transitions in the set, than the machine can be deemed a non-halter without question. However, the immediately clear difficulty with this approach lies in identifying the components that are defined in the set of transitions. As is shown in Tables 4.1 and 4.2, we have confirmed the results of $\Sigma(n)$ up through $n = 4$ but not yet 5. Algorithm 4 suggests that we examine the holdouts that remain in search of new non-halt behaviors.

We therefore begin our analysis of the 98 holdouts[69] with a subset that can be identified as exhibiting one of the behaviors defined above in section 4.2: Leaning Christmas trees. These machines escaped the current detection routine because of certain difficulties with the problem just described.

4.4.1 Leaning Christmas Trees

We refer the reader to section 4.2.6 which explicitly describes the behavior of leaning Christmas trees. Since we do not originally formally describe the specification there, we do so here for clarity:

> A Turing Machine M is a *Leaning Christmas tree* if either M or its mirror M^c satisfy the following conditions for some state s:
>
> 1. There are words C, N, U, V, and X such that the tape configuration at some time is $0^*[C][U][V_s]0^*$, and at some later time is $0^*[C][N][U][X][V_s]0^*$.
> 2. The following conversions hold, where X, X', Y, Y', Z, V, V', V'', U, and U' are words and q and r are states (the symbol $\Rightarrow$ means that M

[69] Note that for both the implicit formulations (B and O) and explicit formulations (P and R) of the problem, there are the same number of holdouts defined in Tables 4.1 and 4.2. This is due to the tree normalization mechanisms used to enumerate the relevant set of Turing machines described in section 4.1. Since, in both cases, the holdout machines are those machines for which is has not yet been determined whether or not they halt, no halting transitions/states have been defined for them. Thus while the implicit and explicit formulations both have different halting requirements, the holdout machines are actually *partial* machines that can be easily modified to contain the necessary halting specifications for each of the respective formulations of the problem. The overall result of this discussion is that the set of holdouts for both of the types of formulations is the same, and thus addressing the one set allows us to confirm the results of all four formulations of the problem that we consider.

transforms the left-hand side into the right-hand side after some number of steps):

- $[X][V_s]0^* \Rightarrow_q [X'][V']0^*$
- $[X_q][X'] \Rightarrow_q [X'][Y]$
- $[N][U_q][X'] \Rightarrow [N][N][U'][Y']_r$
- $[Y'][_r Y] \Rightarrow [Z][Y']_r$
- $[Y'][_r V'] \Rightarrow [Z][V''_s]$

3. $[N]^{i+1}[U'][Z]^i[V''] = [N]^{i+1}[U][X]^{i+1}[V]$ for all $i \geq 1$.

This definition, while complex, is also relatively straight-forward. With an accurate identification of the required words, it is easy to demonstrate whether or not a machine transforms the tape exactly according to the transitions defined above. However, given just an arbitrary Turing machine, how does one go about identifying these components? The general strategy used in the current implementation is as follows:

1. Run the machine for an arbitrary number of steps to establish a sweeping motion and account for any startup effects that may be out of line with the transitions defined.

2. Determine the step counters of the points in execution where the machine reaches its right extremum, or in other words when the configuration is $0^*[C][N]^i[U][X]^i[V_s]0^*$ for some i. Find these step counters for the next five extrema following the initial startup of the machine.

3. Use the tape configurations at each extremum and compare them to extract the components. For example comparing the first two extrema reveals the C, N, X, U, and V components by examining the differences between the two configurations.

4. The rest of the tape components can be derived by attempting to perform the above transitions at the corresponding points where they should appear, and extracting the additional components from these procedures. If at any point this fails, then the entire proof fails and the machine cannot be classified as a leaning Christmas tree.

The main problem in this approach lies in the determination of the right extrema in step 2. Because the left extremum of each sweep is not continually pushed outward like in a standard Christmas tree, and instead indeterminately lies to the *right* of the

previous left extremum, it is extremely difficult to determine when these right extrema occur. This is especially true in machines that exhibit both leftward and rightward minor motions during one major sweep in one direction across the tape. As a result, the current implementation does not always properly identify these step counters in some machines that are in fact leaning Christmas trees.

While the automated process is clearly difficult, identifying the components by hand is time consuming but possible. After a thorough examination of the 98 holdouts, it was determined that 10 of them (0, 1, 3, 9, 12, 13, 14, 32, 88, 95)[70] are leaning Christmas trees. A description of each of them can be found in Appendix A.[71]

4.4.2 Base 3 Counters

Intuitively, base 3 counters are extremely similar to the already established counter behavior except for the one obvious difference. Instead of counting in representative binary notation, they count in base 3. For completeness, we outline the full set of requirements for a machine to be considered a base 3 counter. Unsurprisingly, it is extremely similar to the original counter specification shown in section 4.2.7.

1. For machine M or its mirror M^c, there are words E, A, B, C, T, and Z. In the context of a base 3 counter, E represents the end cell that is used as the checkpoint. A, B, and C are used to represent the values 0, 1, and 2 respectively in terms of a base 3 number. T is a transitory word that occurs during the incrementing of the representative number in between checkpoints. Finally, Z is a blank word which consists of all 0's and is of the same length as the A, B, C, and T words.

2. At some point during execution, the machine reaches the following configuration where c is some state: $0^*[E][_cZ][Z]^*$.

3. The following transitions hold where r is also some state:

   - $[_cA] \Rightarrow_r [B]$

[70] Note that each holdout is assigned a numeric value in Appendices A and B. These values are assigned according to the order in which each of the machines are generated in the enumeration strategy and subsequently popped off of the stack described in section 4.3.1

[71] Explicit confirmations that each of these machines do in fact follow the leaning Christmas tree specification are not included for brevity. Such confirmations require the correct identification of the specified words and states as well as explicit demonstration that each of the transitions and rules are followed exactly to specification. Appendices A and B do, however, contain information about each of the 98 holdouts including flow diagrams and their individual non-halt behavior classifications. For more information about the explicit proofs of select machines, please contact the author of this paper.

- $[_cB] \Rightarrow_r [C]$
- $[_cC] \Rightarrow [T]_c$
- $[_cZ] \Rightarrow_r [B]$
- $[T_r] \Rightarrow_r [A]$
- $[E_r] \Rightarrow [E]_c$

This definition can clearly be extended for base 4, base 5, etc. counters by adding additional components and transitions similar to those involved in the transformation of the base 2 specification to that for base 3.

4.4.3 Alternating Counters

Alternating counters deviate from the behavior of ordinary counters by the behavior of the transition that occurs when the carry signal c hits the blank word Z. In alternating counters, the size of the Z word is smaller than the size of the words for the representative one, two, and transitory words. Therefore, when the transformation occurs, the resulting tape configuration is in an inconsistent state. As a result, each time this transition occurs, the representative structure of the tape is modified in a similar fashion as the last step that occurs in the Christmas tree behavior. Let us look at the specification of this behavior more closely to clarify:

1. For machine M or its mirror M^c, there are words E, E', A, B, B', T, and Z. The A, B, and T words again respectively represent the values of 0, 1, and a transitory word. The additional words E', and B' are included to account for the abnormally sized blank Z.

2. At some point during execution, the machine reaches the following configuration where c is some state: $0^*[E][T][_cZ][Z]^*$.

3. The following transitions hold for some states c, r, and r':

   - $[_cA] \Rightarrow_r [B]$
   - $[_cB] \Rightarrow [T]_c$
   - $[_cZ] \Rightarrow [B'_{r'}]$
   - $[B_{r'}] \Rightarrow_r [B]$
   - $[T_r] \Rightarrow_r [A]$

- $[E_r] \Rightarrow [E]_c$
- $[E'_r] \Rightarrow [E']_c$

4. $[E][T]^i[B'] = [E'][T]^{i+1}[B]$ for all $i \geq 1$.

The last requirement allows one to redefine the makeup of the tape in order for the correct transitions to be applied. After the carry signal c reaches a blank Z, the tape will be redefined from having an E component to having an E'. Then the next time this occurs, it will be redefined again to an E. It will "alternate" like this forever and thus machines following this format are non-halters.

4.4.4 Resetting Counters

Intuitively, resetting counters periodically "reset" themselves back to zero and then start counting up again. In fact, resetting counters differ from ordinary counters in only one minor modification to one transition. In plain counters, when the carry signal c reaches the blank word Z, the Z is transformed into a B word which is representative of the number 1. When resetting counters encounter this scenario, the Z is instead transformed into an A which is the 0 representative. Thus a resetting counter will count up to 2^0, reset, 2^1, reset, 2^2, reset, and so on. The formal specification for a resetting counter is trivially derived from the original counter definition in sect. 4.2.7. We therefore do not include it here.

4.4.5 Complex Counters

Complex counters follow a modified specification to that outlined in sect. 4.2.7 that increases the scope of the behavior while still maintaining the guaranteed non-haltingness of the machines that follow it. First of all, the original description specifies two single states c and r that must remain consistent throughout the transisitions in order for the proof to hold. A simple extension can redefine c and r as sets of states instead of one single state. If this is done, than all transitions which specify c as a state must be split into n different transitions where n is the size of the set c and each transition uses a different element of the set c. The same would obviously apply to set r. Similarly, an extension can be made for each of the words A, B, T, etc. included in the specification.

Clearly, automated detection of such an extension can become a significant burden. Even relatively small sets used in the specification would give rise to a significantly greater number of defined transitions. Additionally, larger sets of words would give rise to an even

greater overall set of words that need to be identified, which is one of the main obstacles of the automated routines in the first place. Regardless, with a relatively small number of holdouts to examine, machines of this nature can be identified by manual analysis.

4.4.6 Combination Counters

All of the above counter behaviors have a very clear, provably infinite specification which a machine must follow in order to be considered in that particular class of non-halt behavior. However, many of the 98 holdout machines do not fall into one explicit category of counters. Instead, some of them exhibit characteristics from some, or even all of the behaviors shown above. The specifications for any of these combination classes of machines can be derived by meshing the necessary specifications together. Going through each of the possible combinations here would be largely redundant, so we do not include them for brevity.

After manually examining the holdouts, 30 of them have been identified as some variation (or combined variation) of the counter behaviors described above.[72] While some of them have been explicitly proven as their identified behaviors by manually identifying each of the components that make up the specification, others remain as conclusive counters based on a close visual examination of their behavior and diagrammatic recognition of features common to each particular category. The real perceptual bridge of human diagrammatic reasoning begins to be seen here. While we have not established a symbolic logical proof of the non-haltingness of each of these machines, we are nonetheless completely convinced of their behaviors.

4.4.7 Nested Christmas Trees

We invite the reader to refer to section 4.2.4 to once again become familiar with the described Christmas tree specification. We have already seen several variations of these Christmas trees, including the already analyzed leaning Christmas trees, as well as multi-sweep Christmas trees. Let us now examine an even more bizarre variation dubbed nested Christmas trees.

Nested Christmas trees exhibit the same repeatable sweeping motion across the tape as do regular Christmas trees. However, the one very distinct difference, is that during

[72] Holdouts numbered 2, 5, 6, 7, 8, 10, 11, 16, 17, 22, 23, 27, 28, 29, 34, 37, 40, 42, 43, 44, 53, 54, 55, 56, 57, 61, 63, 71, 96, 97 have been established as some type of counter variation. See Appendix A for more information on these machines.

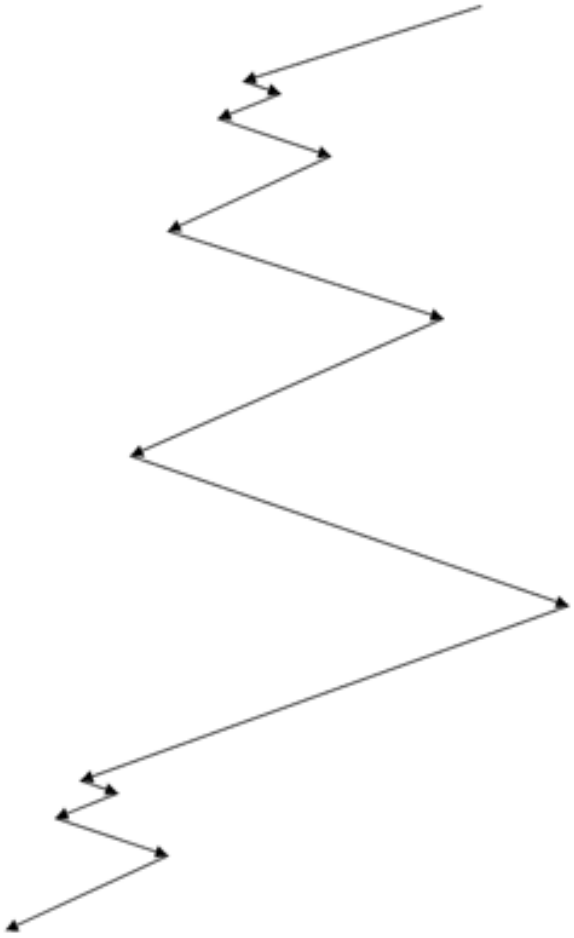


Figure 4.15: Nested Christmas tree conceptual behavior

each return sweep of the tree, the machine undergoes a miniature, nested set of sweeps resembling a nested Christmas tree until it reaches the previous extremum point of the read head. Consider fig. 4.15 for a more intuitive understanding of how nested Christmas trees work. Imagine that the top of the figure represents some particular point of execution of the machine. Specifically, the read head is located at the right most extremum point of the end of some sweep. The end of the arrow thus represents the location of the read head on the tape (which is not shown for simplicity). Now as one progresses down the figure, imagine that the tape at successively later points in time are shown, and that the read head is located at the intersection of the arrows and the invisible tape.

Thus we have a conceptual visual representation of how the machine behaves over time. As we can see by the figure, when the tape reaches the end of the sweep moving to the left, it undergoes a nested set of sweeps until it reaches the previous right extremum

point of the read head. It then sweeps back to the left again, and starts up another nested tree. The pattern continues forever and thus the machine is a non-halter.

With a generalized understanding of how nested Christmas trees work, let us formalize the behavior:

1. For machine M or its mirror M^c, there are words U, V, X, Y, Z, U^n, V', and V''. U and V are end components that cap the left and the right sides of the overall tree. X, Y, and Z represent the repeating interior components during different points of the execution of the machine. V' and V'' are intermediate components that the right end component is in while the machine is in between sweeps. The U^n component is used as the right end component of the nested sweeps.

2. At some point during execution, the machine reaches the following configuration where s is some state: $0^*[U][X][V_s]0^*$.

3. The following transitions hold for some states q, q^n, and r^n:

   - $[V_s] \Rightarrow_q [V']$
   - $[X_q] \Rightarrow_q [Y]$
   - $0^*[U_q] \Rightarrow 0^*[U^n]_{r^n}$
   - $[_{r^n} Y] \Rightarrow_{q^n} [Z]$
   - One of the following two holds:
     - $0^*[U^n_{q^n}] \Rightarrow 0^*[U^n]_{r_n}$
     - $0^*[U^n_{q^n}] \Rightarrow 0^*[U^n][Z]_{r_n}$
   - $[_{r^n} Z] \Rightarrow [Z]_{r^n}$
   - $[_{r^n} V'] \Rightarrow [V''_s]$ if $V' \neq Y$
   - $[_{r^n} 0^*] \Rightarrow [V''_s]0^*$ if $V' = Y$
   - $[Z_{q^n}] \Rightarrow_{q^n} [Z]$

4. $\forall i(0^*[U^n][Z]^i[V'']0^* \Rightarrow \exists j(0^*[U][X]^j[V]0^*))$

This specification is slightly complicated to verbally explain, but it is essentially adapted from a simplified version of the original Christmas tree specification with the inclusion of additional transitions to encapsulate the nested Christmas tree behavior of the return sweep. Again, after a manual analysis of the holdout machines, it was inferred

that 27 of the remaining machines can be classified as nested Christmas trees.[73] Similar to the case with the counters, not all of these machines have been explicitly confirmed as being nested Christmas trees. In fact, it is likely that some of the 27 machines do not actually follow the above specification but require some modification of it in order to be proven a non-halter. Nevertheless, a visual inspection of the machines mentioned leaves no doubt that they are at least some variation of a nested Christmas tree.

4.4.8 Uneven multi-sweep Christmas trees

Another 18 of the holdouts in question exhibit a behavior that is extremely similar to that of multi-sweep Christmas trees.[74] In fact, for each of these machines in question, the specification that can prove its non-haltingness is identical to the particular multi-sweep class that it is a part of (2 sweeps, 3 sweeps, etc.). Now of course the obvious question becomes why are they not flagged by the automated detection routine as such.

The details concerning how these machines escape the automated routine are very specific to the implementation and therefore need not be fully discussed here. However, refer to fig. 4.16 for a general idea as to why these machines are not automatically detected. As one can see, on certain sweeps, the read head does not reach the extremum of the previous sweep. Therefore, complications arise when attempting to automatically extract the necessary components outlined in the specification.

Attempts to symbolically prove the non-haltingness of these machines by manually identifying the components and verifying the transitions of the specifications have not been made. Not only is this process incredibly time consuming like those for the nested Christmas trees and the counter variations, but the hope is that the necessary changes can be incorporated into the automated detection routine so that the proofs can be done automatically. Regardless, once again visual inspection of the machine behaviors allows us to classify these machines as uneven multi-sweep Christmas trees.

4.5 Solidifying $\Sigma(5)$ by Visual Inspection

After classifying the holdouts for $\Sigma(5)$ into the already mentioned categories, 13 machines remain that do not follow any of the aforementioned behaviors. Formal behaviors for these machines have not yet been defined; however, careful visual analysis of their

[73] The machines referred to here are the holdouts numbered 18, 24, 25, 30, 31, 33, 35, 36, 38, 39, 45, 46, 47, 48, 62, 64, 65, 70, 75, 80, 81, 82, 85, 86, 89, 90, 91.

[74] Holdouts 26, 41, 50, 51, 52, 66, 67, 72, 73, 76, 77, 78, 79, 83, 84, 92, 93, 94.

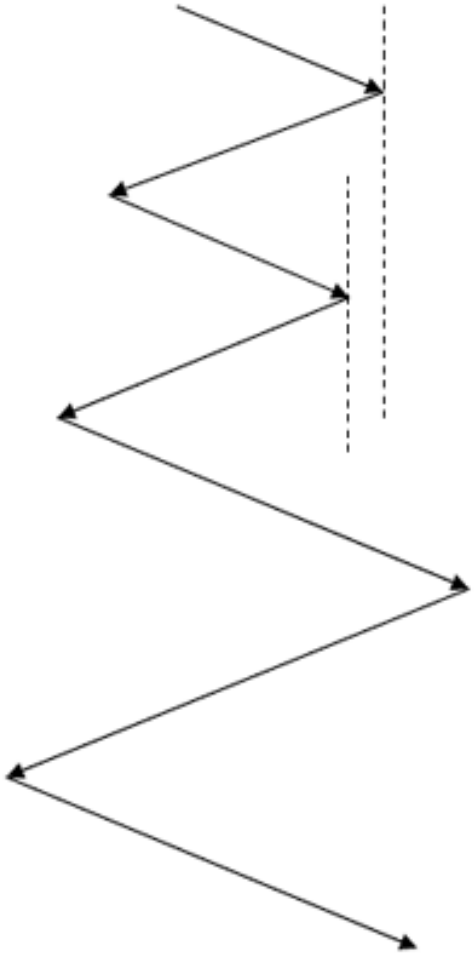


Figure 4.16: Uneven multi-sweep conceptual behavior

executions have led to the following conclusions:

- 1 of these machines (holdout 69) is most definitely what we call a "startup effects Christmas tree." The current Christmas tree detection routine runs the machine for a specified number of steps before beginning to attempt to identify components and transitions of the Christmas tree behavior. This accounts for any "startup effects" that the machine may undergo before exhibiting the infinite repeatable behavior. This 1 machine in question has an unusually long period where it exhibits startup behavior and therefore overruns the arbitrarily chosen "startup effects" threshold. A simple increase of this threshold should pickup this machine.

- 5 machines (holdouts 15, 58, 59, 60, and 87) are somewhat similar to nested Christ-

mas trees. However, in these cases, instead of performing a miniature nested Christmas tree behavior on its return sweep, the machines incorporate a nested *counter* into their return sweeps. We call these machines nested Counter Christmas trees.

- 1 machine (holdout 4) is the corresponding equivalent of a multi-sweep Christmas tree for leaning Christmas trees. The behavior is a double sweep leaning Christmas tree which would be an intuitive extension of leaning Christmas trees.

- 3 machines (holdouts 19, 20, and 21) are very bizarre and can be best described as 1.5 sweep Christmas trees. On every other sweep, the read head reaches only halfway to the previous extremum point before returning.

- The final 3 machines (holdouts 49, 68, and 74) are most easily named asymmetric Christmas trees. They exhibit a similar back and forth sweeping motion of regular Christmas trees except their interior components are not identical from end to end. Instead, one component repeats until halfway to the other extremum, and then another, different component repeats until the end. Thus it is asymmetric.

Our contention once again is that all of these machines are "diagrammatically" confirmed as non-halters by our own visual deductive reasoning system.[75] However, we realize that the lack of concrete, formal symbolic proofs about the non-haltingness of these machines may concern some readers. Let us consider, then, the diagrammatic representation of the execution of one of these machines.[76]

Figure 4.17 displays the extended diagrammatic representation of the execution of the first 300 steps of holdout 21.[77] As is indicated, this holdout follows the behavior that we describe as a 1.5 sweep Christmas tree. Notice from the diagram that each time the read-write head reaches a right extremum point on the tape, the tape is in a configuration

[75] It should be noted that Brady (1983) makes a similar assertion when he is faced with his own set of holdouts in his attack on the *quintuple* formulation of $\Sigma(4)$ (the same formulation of the problem that Lin & Rado (1964) attack):

> The general behavior of all of the 218 holdout machines has already been described. ... All of the remaining holdouts were examined by means of voluminous printouts of their histories along with some program extraced features. It was determined to the author's satisfaction that none of these machines will ever stop.

Therefore, our claims, while bold, are not unprecedented.

[76] Note that we include the flow charts as well as the diagrammatic executions of the first 75 steps of all 13 of the machines in question in Appendix B.

[77] Note that the diagram is split into four separate columns. Each column represents 75 steps of the execution. Also, the word "State" which has appeared in previous diagrammatic representations has been shown as simply an "S" in this figure due to space constraints.

such that half of the relevant tape (the lefthand portion) is a contiguous sequence of 1's. The right hand portion, on the other hand, is a sequence of 01 components. The machine then sweeps leftward, following a jagged pattern through this sequence of 01's until it reaches halfway across the relevant portion of the tape and into the contiguous 1 pattern. At this point, it continues directly in a leftward motion until it establishes a new left extremum point. During the corresponding rightward sweep, the machine first sweeps halfway across the tape[78] (to the right edge of the contiguous 1's), and then sweeps back before executing a full rightward sweep to establish a new right extremum point.

It should be abundantly clear from the diagram and this verbal description of the machine's behavior that this pattern will continue indefinitely, pushing the left and right extrema successively outward while continuing to add components to the relevant center portion of the tape. The significance of this discussion, however, is the fact that this knowledge is obtained without any construction or reference to a formal specification of the behavior or symbolic logical proof. As is discussed in sections 3.3.1 and 3.3.4, we are simply able to "read" the infinite behavior off of the diagrammatic representation of the machine's execution and subsequently deduce by our visual cognitive abilities that the machine is a non-halter.

As a final result, we thus claim that the values for the quadruple formulations of $\Sigma(1)$ through $\Sigma(5)$ from Table 4.4 are confirmed as truths. Every Turing machine in the search space of these problems have been classified as either a halter (and the corresponding productivity relevant to the Busy Beaver formulation recorded) or a non-halter in one of the defined categories of behaviors that we have described. All of the machines for the $n = 1$ through 4 search spaces have been categorized via formalized automatic behavior detection mechanisms while the final set of 98 holdouts for $\Sigma(5)$ require an appeal to human diagrammatic reasoning processes to finish the job.

[78] This halfway sweep is what constitutes the ".5" portion of the behavior name.

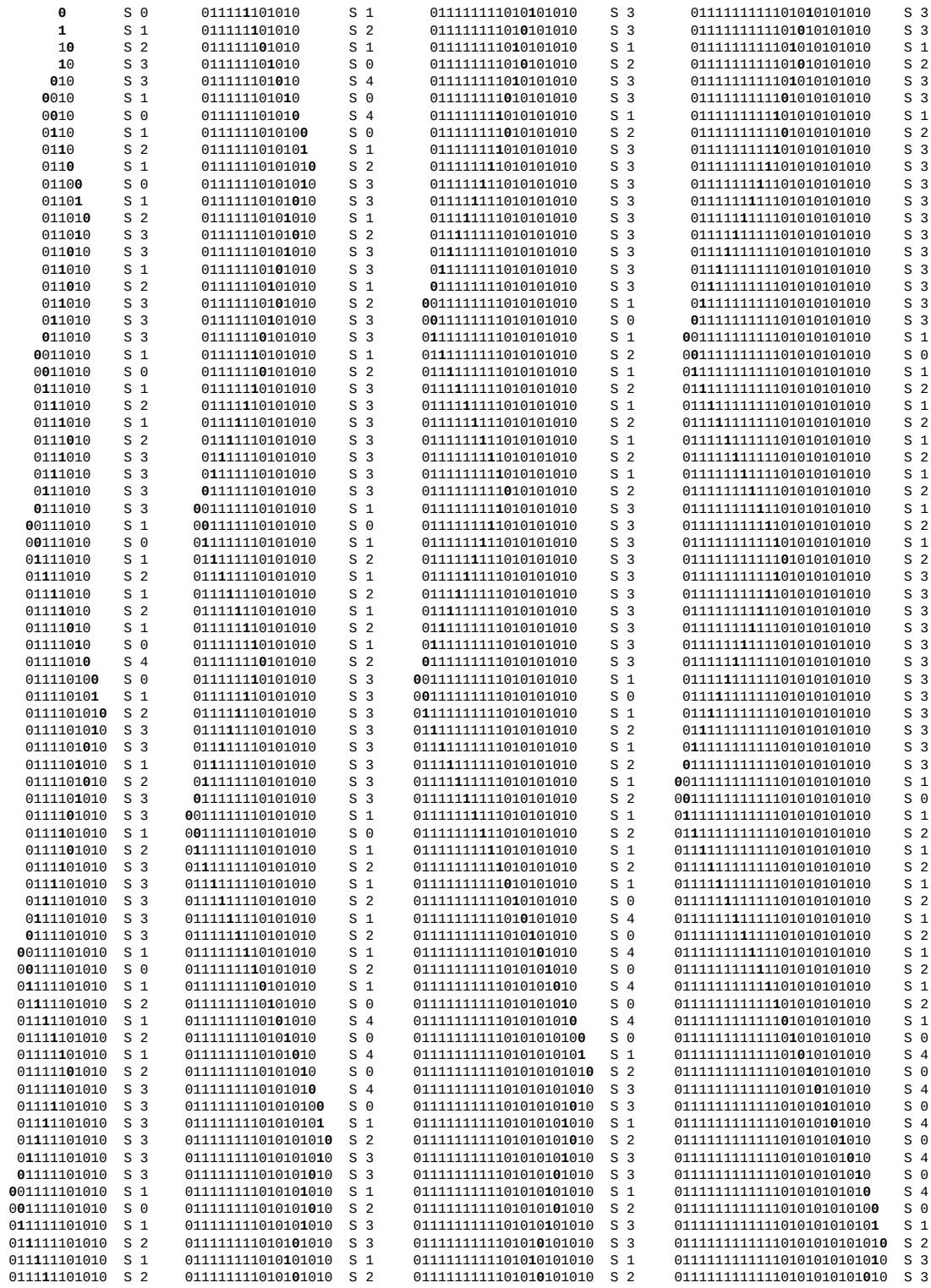


Figure 4.17: Holdout machine 21: 1.5 Sweep Christmas Tree

CHAPTER 5
Future Work

5.1 Press Onward

An immediate and obvious extension to our research presented in this thesis is the continued assault on the Busy Beaver function in an attempt to establish new records, new non-halt behavior specifications, and ultimately solidified values for $\Sigma(6)$, $\Sigma(7)$, and beyond. We have already seen by our presentation in chapter 3 that significant evidence exists that humans are capable of hypercomputation that has direct relevance to the attack on the Busy Beaver problem.[79] Additionally, our assault in chapter 4 anchors this possibility with a rigorous establishment of $\Sigma(n)$ up through $n = 5$. Thus a continued push upward to nail down $\Sigma(n)$ for successively higher values of n by incorporating the mechanisms described in this thesis would undoubtedly further the evidence of hypercomputation that we suggest.

Considering this, let us revisit our dream described at the end of section 2.3.3 one last time. In our dream, we suggest a set of non-halt detection mechanisms

$$\{NH_0, NH_1, NH_2, NH_3, \ldots\}$$

that encapsulate the behaviors of all non-halting Turing machines. The existence of such a set, as is already described, would clearly provide a solution to the Halting problem as well as the Busy Beaver function. We have fleshed out the early beginnings of this dream set in chapter 4 and our finalized contention is encapsulated as two overlapping possibilities:

1. Every member of this set is Turing-computable and thus every Turing machine that does not halt can be proven so in some Turing-computable way. The reason that the Halting problem as a whole is non-Turing-computable is because this set is not a recursively enumerable set. Humans, however, can enumerate this set by hypercomputational means. We have seen from the presentation in section 3.3 as well as the concrete examples in section 4.5 that human diagrammatic reasoning processes

[79] Specifically, humans are capable of hypercomputable processes that are directly related to determining whether Turing machines do not halt. This capability is also directly tied to diagrammatic reasoning processes employed by the mind.

hold the key to extracting these non-halt behaviors. Specifically, we have seen by our analysis of the $\Sigma(5)$ holdouts that clear non-halt behaviors can be recognized quickly by visual analysis and then later stamped out in formal, symbolic patterns. It should be noted that this overall view is one that also appears to be directly supported in (Bringsjord, Kellett, Shilliday, Taylor, van Heuveln, Baumes & Ross forthcoming):[80]

> We declare with extreme confidence that all 98 of these holdouts [the same 98 $\Sigma(5)$ holdouts that we reference in our work] will soon be *automatically* proven non-halters as well: the case of 5, and then 6, will soon enough be determined by ... the ever ascending human mind. Thus the search continues. By incorporating formal behaviors found from the [$\Sigma(5)$] holdouts, we hope to use these behaviors to significantly reduce the [$\Sigma(6)$] holdout set and continue to adapt, combine, and formulate new behaviors for the remaining holdouts. ... Considering this, we see no reason why the above algorithm [essentially the same algorithm presented in Algorithm 4] cannot be sustained, and applied to [$\Sigma(7)$], [$\Sigma(8)$], and beyond.

Thus this "hyperenumeration" process is one that is possible via the computational powers of the human mind.

2. The other possibility that is supported by our work is that not every member in our dream set is a Turing-computable procedure. Therefore, there must exist some non-halting Turing machine that cannot be proven as such by some Turing computable mechanism. If this is the case, then again, we have suggested that human diagrammatic reasoning processes are capable of surpassing the Turing limit for this purpose.

Thus while our research still leaves open this question, we have established human visual reasoning capabilities as being within the hypercomputable spectrum, and therefore maintain the dream that the Busy Beaver function is computable by tapping into this power.

5.2 Barwise-ian Reasoning System

As an additional tangent line of research, our work has provided a springboard for development of diagrammatic reasoning systems that are not anchored in symbolic mechanisms and cannot be reduced to a Turing equivalent form. As we have already mentioned in section 3.3.4, Barwise & Etchemendy's (1995) *Hyperproof* system parallels this concept by incorporating into a proof system separate diagrammatic and symbolic

[80] Note that this cited paper pulls directly from the same Busy Beaver research associated with this thesis.

representation schemes that are not completely reducible to each other. While their work makes no reference to hypercomputational powers of the human mind, it still serves as a foundation for the development of reasoning systems that more closely resemble true human cognitive processes.

Tall (1995) also addresses this concept in a commentary on visual proof systems and proof systems in general:

> Difficulties occur when the enactive or visual form of the proof does not suggest an obvious sequence of deductions to use for a formal proof, so that the individual seems to "know" that the theorem is true and yet has no method of proving it. There are numerous examples in topology where an "obvious" visual property fails to have a correspondingly simple proof (such as the Jordan curve theorem that every closed path in the plane that does not cross itself divides the plane into two regions, the inside and the outside). ... Educators and mathematicians need to rethink the nature of mathematical proof and give appropriate consideration to the different types of proof related to the cognitive development of the individual.

Here we can see that Tall references a very similar situation to our discussion in section 3.3.1 as well as our analysis of the final $\Sigma(5)$ holdouts in section 4.5. Thus our work provides stepping stones for additional research in the formalization of visual cognitive processes.

5.3 Hypercomputational Processes in Nature

We have suggested at numerous points throughout this thesis that human hypercomputational abilities are the direct result of natural, physical hypercomputational processes of which the human brain is comprised. Additionally, we cite specific examples of naturally occurring hypercomputational processes that exist *outside* of the mind (see section 3.1.3). Thus an additional obvious line of research that could be inspired by this work is the quest for artificial hypercomputing machines that are realizable within the realms of natural physics.

As we have already noted, Penrose (1989, 1994) explores the physical makeup of the brain via quantum mechanics as possessing non-Turing-computable processes. Additionally, Hava Siegelmann's (1995, 1994) work on analog computation has suggested that analog neural networks, which possess powers beyond the Turing limit, hold the key to the limits of physically realizable computation. Also, Kieu (2003) has proposed a solution to the halting problem which is grounded in theoretical quantum computing.

Thus despite Turing's brilliant work, we can see that the key to the true limits of computation may not lie exclusively in Turing machines, but instead require a deeper

understanding of quantum mechanics and the totality of the laws that govern the physical universe.

5.4 The Human-Computable Gap

As a final, and perhaps most crucially relevant line of research that our work induces we must first revisit a diagram that we present way back in figure 1.1. Recall that this illustration as a whole represents the set of all Turing machines. The Turing-computable and Human-computable lines respectively denote the capability of Turing-computation and Human-computation in determining whether Turing machines do not halt. Thus our work presented in this thesis has anchored the Human-computable line as clearly above the Turing-computable line and therefore humans are more capable of determining whether Turing machines do not halt than any Turing equivalent machine.

The question still remains, however, regarding the precise location of this line. Specifically, we leave open the problem of determining whether this line lies above the set of all Turing machines and thus humans possess the power to solve the generalized form of the Halting problem (and consequently $\Sigma(n)$). Let us briefly consider the possibility where this is not the case and the Human-computable line is positioned such that humans cannot solve $\Sigma(n)$: We have seen by our work presented thus far, that $\Sigma(1)$ through $\Sigma(5)$ have been established with absolute certainty. Also, there are no foreseeable barriers to the confirmation of $\Sigma(6)$, $\Sigma(7)$, and beyond. However, if humans cannot solve $\Sigma(n)$, then there must exist some smallest k for which $\Sigma(k)$ cannot be confirmed by mechanisms of human reasoning. Thus consider the set of all k-state Turing machines. If humans cannot confirm $\Sigma(k)$, then there must exist at least one of these machines that cannot be confirmed as a non-halter by humans.

What might the behavior of such a machine look like? Such a "chaotic" machine might provide crucial insights about the cusp of the Turing barrier. However, what we suggest is more likely is that humans *can* in fact solve $\Sigma(n)$ and our research is the beginning of that solution. This exact notion is presented in (Bringsjord et al. forthcoming) by appealing to a Gödelian type line of reasoning:

> The idea is really quite simple: If humans are smart enough to determine $[\Sigma(n)]$, they will eventually (perhaps after 100 years, perhaps after 1000, perhaps after $1,000,000,000$, ...) be smart enough to determine $[\Sigma(n+1)]$: they will invent some new technique for economically representing the behavior of $n+1$ machines, and for detecting in that representation when activity will cease, and when it will not: there will remain no holdouts.

Thus our dream continues to gain momentum. While we have not provided a conclusive argument that $\Sigma(n)$ is human-computable, we have cemented the human-computable line above the Turing-limit and set the human driven assault on $\Sigma(n)$ into high gear.

CHAPTER 6
Conclusion

Turing's (1936) original work established the generally accepted basis of all computation. Not only have his Turing machines formed the theoretical foundation of digital computers, but their computational and expressive power have also been widely acknowledged as the decisive measuring stick for the limits of computability. In fact, many have argued (e.g. Dennett 1991, Kurzweil 2000, Hofstadter & Dennett 1981) that all processes in the physical world can be simulated computationally by Turing machines and thus a human mind can be simulated with a properly programmed Turing machine that mimics the behavior of its physical parts.

It has been our contention in this thesis that this suggestion is bogus. While Turing machines hold their place in conventional computability theory, the laws of the physical world are beyond the computational expressive power of Turing machines. Therefore, humans, a part of this physical universe, can harness this power to perform computations beyond the Turing limit. Leveraging this concept in the context of a proven, non-Turing-computable problem (the Busy Beaver function), our accomplishments in this thesis have been four-fold:

1. Despite a harsh refutation by Bringsjord & Zenzen (2003), we have propped up a Gödelian argument presented by Penrose (1994) that anchors the powers of human reasoning as being above the Turing limit. More specifically, we have formalized his proof by contradiction that there cannot exist a Turing machine that possesses the same capabilities of the human mind to ascertain that Turing machines do not halt. Thus not only do we solidify the hypercomputational power of humans, but via Penrose's line of reasoning, we are able to directly associate this power to determining the non-haltingness of Turing machines.

2. Appealing to Searle-ian (1980) as well as Penrose-ian (1989, 1994) lines of reasoning, we are able to further isolate human hypercomputational powers as a direct product of visual and diagrammatic reasoning.

3. Building off of Ross's (2003) original assault on the Busy Beaver function, we have enhanced his search-based optimization strategy with specific non-halt detection behaviors. The result is that we have confirmed the values of $\Sigma(1)$ through $\Sigma(5)$.

Perhaps more importantly, however, is that the final machines in the $\Sigma(5)$ search space are categorized as non-halters not by our automated detection routines, but instead by the visual reasoning processes of the human mind mentioned above. Thus our attack on the Busy Beaver function has pushed the limits of the human mind and tapped into its awesome hypercomputational power.

4. Finally, our research has raised important considerations for the fields of visual reasoning, quantum mechanics, and of course the continued upward quest to confirm values of $\Sigma(n)$.

While the Busy Beaver problem is unassailable by mechanisms of Turing computation, we have established new hope that the dream of a human computable solution is a realizable possibility.

References

Barwise, J. & Etchemendy, J. (1995), Heterogeneous logic, *in* J. Glasgow, N. Narayanan & B. Chandrasekaran, eds, 'Diagrammatic Reasoning: Cognitive and Computational Perspectives', MIT Press, Cambridge, UK, pp. 211–234.

Barwise, J. & Etchemendy, J. (1999), *Language, Proof, and Logic*, Seven Bridges, New York, NY.

Block, N. (1981), *Imagery*, MIT Press, Cambridge, MA.

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.

Bringsjord, S. & Bringsjord, E. (1996), 'The case against AI from imagistic expertise', *Journal of Experimental and Theoretical Artificial Intelligence* **8**, 383–397.

Bringsjord, S., Kellett, O., Shilliday, A., Taylor, J., van Heuveln, B., Baumes, J. & Ross, K. (forthcoming), 'A new gödelian argument for hypercomputing minds based on the busy beaver problem', *Journal of Applied Mathematics and Computation* . preprint available at http://kryten.mm.rpi.edu/laihyper6.pdf.

Bringsjord, S. & Taylor, J. (2005), An argument for p=np. available at http://kryten.mm.rpi.edu/scb_pnp_solved21.pdf.

Bringsjord, S. & Zenzen, M. (2003), *Superminds: People Harness Hypercomputation, and More*, Kluwer Academic Publishers, Dordrecht, The Netherlands.

Chandrasekaran, B. (2005), What makes a bunch of marks a diagrammatic representation, and another bunch a sentential representation?, *in* 'Proc. AAAI Symposium Spring 2005, Reasoning with Mental and External Diagrams: Computational Modeling and Spatial Assistance.'.

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

Copeland, B. (2000), The church-turing thesis. available at http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html.

Dennett, D. (1991), *Consciousness Explained*, Little, Brown, Boston, MA.

Dreyfus, H. & Dreyfus, S. (1986), *Mind Over Machine*, Free Press, New York.

Ebbinghaus, H. D., Flum, J. & Thomas, W. (1994), *Mathematical Logic (second edition)*, Springer-Verlag, New York, NY.

Euclid (1956), *The Thirteen Books of Euclid's Elements*, Thomas Heath, New York: Dover.

Feinstein, C. (2004), Evidence that p $\neq$ np. available at http://arxiv.org/abs/cs/0310060.

Fischer, P. (1965), 'On formalisms for turing machines', *J. ACM* **12**(4), 570–580.

Garey, M. R. & Johnson, D. S. (1979), *Computer and Intractability: A Guide to the Theory of NP-Completeness.*, W. H. Freeman.

Grover, R. (2005), Np != p and co-np != p. available at http://arxiv.org/abs/cs/0502030.

Harnad, S. (1998), The symbol grounding problem, *in* A. Clark & J. Toribio, eds, 'Machine Intelligence: Perspectives on the Computational Model', Garland Publishing, Inc., New York and London, pp. 89–100. original paper published in 1990.

Hofstadter, D. & Dennett, D., eds (1981), *The Mind's I: Fantasies and Reflections on Self and Soul*, Basic Books.

Ionescu, M. (2005), P is not np. available at http://arxiv.org/abs/cs/0409039 or http://1wayfx.com./.

Ivanov, V. (2005), A proof of p $\neq$ np. available at http://www.math.usf.edu/ eclark/NPvsP.pdf.

Iwamura, H., Akazawa, M. & Amemiya, Y. (1998), 'Single-electron majority logic circuits', *IEICE Trans. Electronics* pp. 42–48.

Jackson, F. (1982), 'Epiphenomenal qualia', *Philosophical Quarterly* **32**, 127–136.

Kellett, O., Ross, K., Bringsjord, S. & van Heuveln, B. (forthcoming), A new millenium attack on the busy beaver problem. available at http://www.cs.rpi.edu/ kelleo/busybeaver.

Kieu, T. (2003), Quantum algorithm for hilbert's tenth problem. available at http://arxiv.org/abs/quant-ph/0110136.

Kupchik, M. (2004), P versus np problem solution. available at http://users.i.com.ua/$\sim$zkup/pvsnp_en_002.pdf.

Kurzweil, R. (2000), *The Age of Spiritual Machines: When Computers Exceed Human Intelligence*, Penguin USA, New York, NY.

Larkin, J. & Simon, H. A. (1987), 'Why a diagram is (sometimes) worth ten thousand words', *Cognitive Science* **10**, 65–100.

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

Lindsay, R. (1995), Imagery and inference, *in* J. Glasgow, N. Narayanan & B. Chandrasekaran, eds, 'Diagrammatic Reasoning: Cognitive and Computational Perspectives', MIT Press, Cambridge, UK, pp. 111–136.

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.

Miller, D. (2003), Solution to a generalization of the busy beaver problem. available at http://www2.warwick.ac.uk/fac/soc/philosophy/staff/miller/.

Moscu, M. (2005), On invariance and convergence in time complexity theory. available at http://arxiv.org/abs/cs.CC/0411033.

Myers, K. & Konolige, K. (1995), Reasoning with analogical representations, *in* J. Glasgow, N. Narayanan & B. Chandrasekaran, eds, 'Diagrammatic Reasoning: Cognitive and Computational Perspectives', MIT Press, Cambridge, UK, pp. 273–302.

Nagel, T. (1974), 'What is it like to be a bat?', *Philosophical Review* **LXXXIII**, 435–450.

Oberschelp, A., Schmidt-Gottsch, K. & Todt, G. (1988), 'Castor quadruplorum', *Archive for Mathematical Logic* **27**, 35–44.

Penrose, R. (1989), *The Emperor's New Mind*, Oxford, Oxford, UK.

Penrose, R. (1994), *Shadows of the Mind*, Oxford, Oxford, UK.

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

Post, E. (1947), 'Recursive unsolvability of a problem of thue.', *J. Symb. Log.* **12**(1), 1–11.

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.

Rogers, E. (1995), A cognitive theory of visual interaction, *in* J. Glasgow, N. Narayanan & B. Chandrasekaran, eds, 'Diagrammatic Reasoning: Cognitive and Computational Perspectives', MIT Press, Cambridge, UK, pp. 481–500.

Ross, K. (2003), Use of optimisation techniques in determining values for the quadruplorum variants of rado's busy beaver function, Master's thesis, Rensselaer Polytechnic Institute.

Searle, J. (1980), 'Minds, brains and programs', *Behavioral and Brain Sciences* **3**, 417–424. This paper is available online at http://members.aol.com/NeoNoetics/MindsBrainsPrograms.html.

Searle, J. (1992), *The Rediscovery of the Mind*, MIT Press, Cambridge, MA.

Siegelmann, H. (1995), 'Computation beyond the Turing limit', *Science* **268**, 545–548.

Siegelmann, H. & Sontag, E. (1994), 'Analog computation via neural nets', *Theoretical Computer Science* **131**, 331–360.

Sipser, M. (1992), The history and status of the p versus np question, *in* 'STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing', ACM Press, New York, NY, USA, pp. 603–618.

Tall, D. (1995), 'Cognitive development, representations, and proof', *Justifying and Proving in School Mathematics* pp. 27–38.

Turing, A. (1969), Intelligent machinery, *in* B. Meltzer & D. Michie, eds, 'Machine Intelligence', Edinburgh University Press, Edinburgh, UK, pp. 1–24.

Turing, A. (1998), Computing machinery and intelligence, *in* A. Clark & J. Toribio, eds, 'Machine Intelligence: Perspectives on the Computational Model', Garland Publishing, Inc., New York and London, pp. 1–28. original paper published in 1950.

Turing, A. M. (1936), 'On computable numbers with applications to the entscheidung-problem', *Proceedings of the London Mathematical Society* **42**, 230–265.

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

Wegner, P. & Goldin, D. (2003), 'Computation beyond the turing limit', *Communications of the ACM* .

APPENDIX A
Initial Categorized $\Sigma(5)$ Holdouts

In this appendix, we present the 5-state Turing machines that can be classified in one of the non-halt behavior patterns described in section 4.4. While only a small subset of these machines have been symbolically confirmed as belonging to their respective suggested behavior categories (such machines are marked with a star),[81] the remainder of the machines we contend have been "diagrammatically" confirmed by human reasoning processes that cannot be symbolically described.

A.1 Leaning Christmas Trees

The leaning Christmas tree Turing machines are depicted in figures A.1 through A.10.

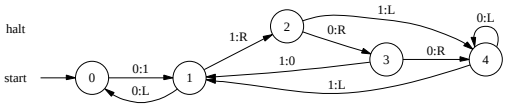


Figure A.1: Holdout machine 0 (*)

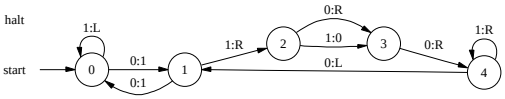


Figure A.2: Holdout machine 1 (*)

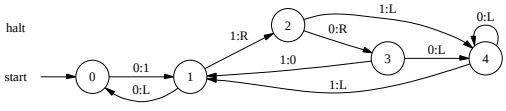


Figure A.3: Holdout machine 3 (*)

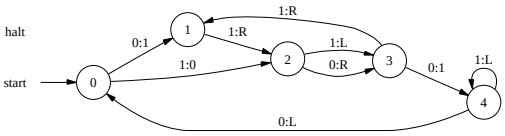


Figure A.4: Holdout machine 9 (*)

[81] We ask the reader to contact the author for a complete description of the extracted components that constitute symbolic confirmation of the said machines.

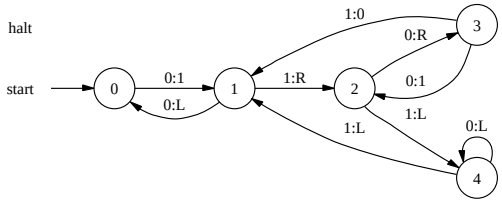


Figure A.5: Holdout machine 12 **(*)**

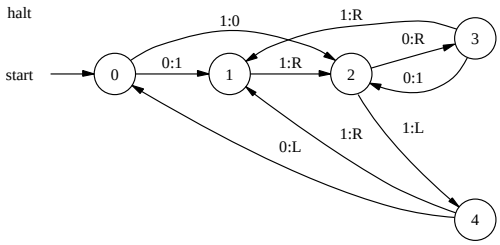


Figure A.6: Holdout machine 13 **(*)**

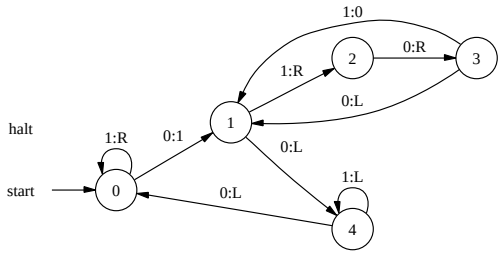


Figure A.7: Holdout machine 14 **(*)**

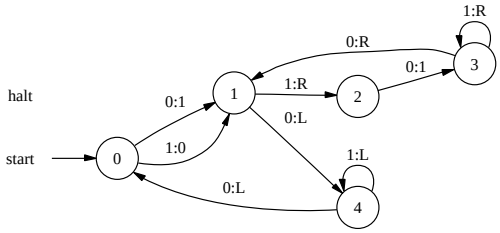


Figure A.8: Holdout machine 32 **(*)**

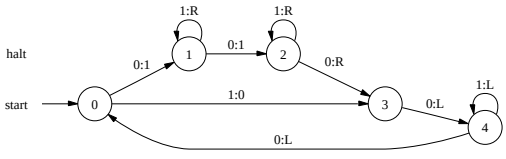


Figure A.9: Holdout machine 88 **(*)**

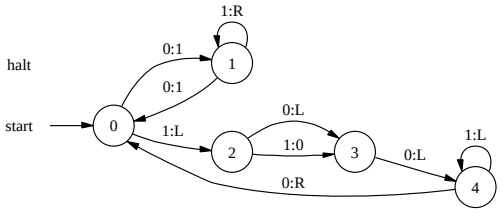


Figure A.10: Holdout machine 95 **(*)**

A.2 Nested Christmas Trees

The nested Christmas tree Turing machines are depicted in figures A.11 through A.37.

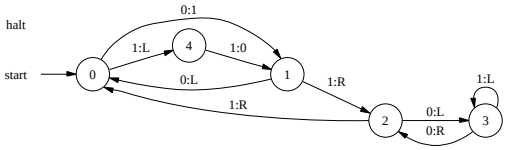


Figure A.11: Holdout machine 18

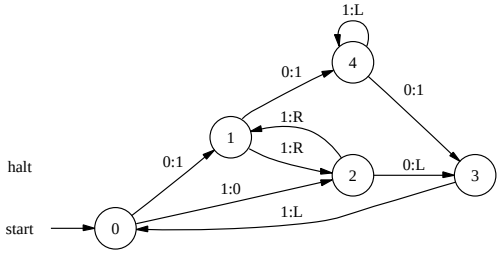


Figure A.12: Holdout machine 24

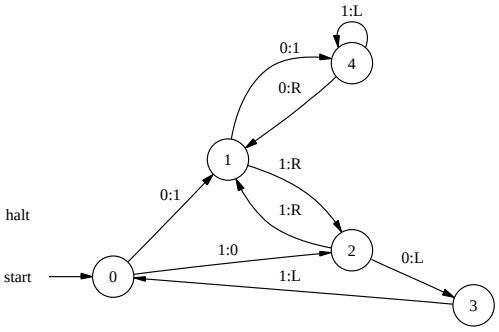


Figure A.13: Holdout machine 25

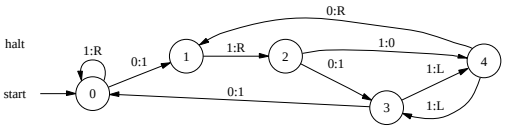


Figure A.14: Holdout machine 30

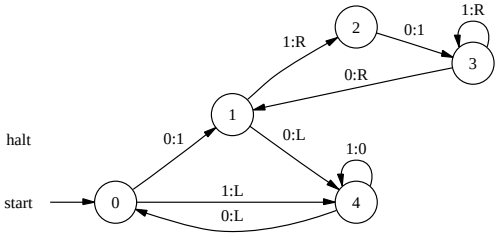


Figure A.15: Holdout machine 31

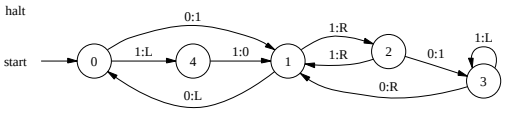


Figure A.16: Holdout machine 33

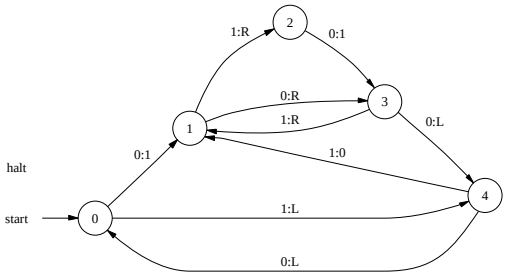


Figure A.17: Holdout machine 35

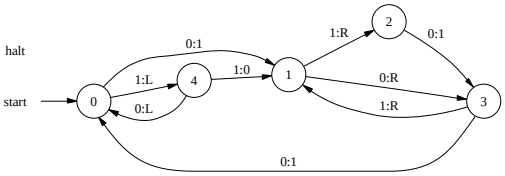


Figure A.18: Holdout machine 36

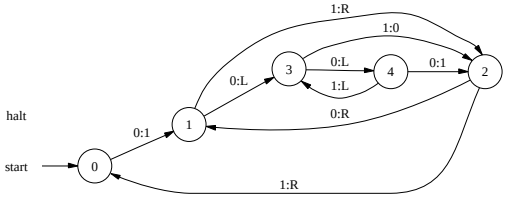


Figure A.19: Holdout machine 38

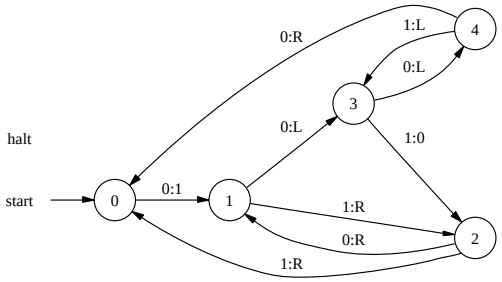


Figure A.20: Holdout machine 39

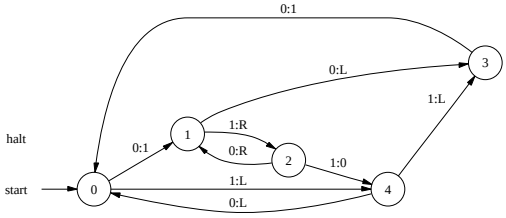


Figure A.21: Holdout machine 45

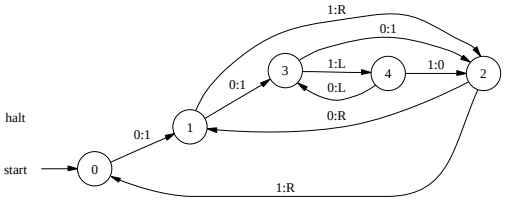


Figure A.22: Holdout machine 46

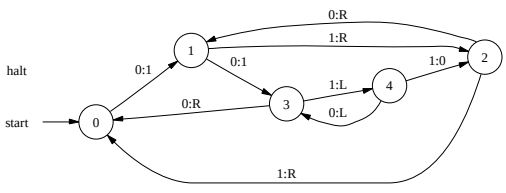


Figure A.23: Holdout machine 47

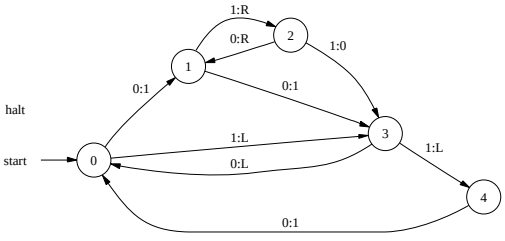


Figure A.24: Holdout machine 48

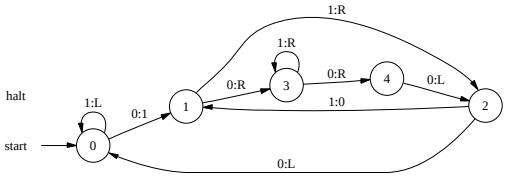


Figure A.25: Holdout machine 62

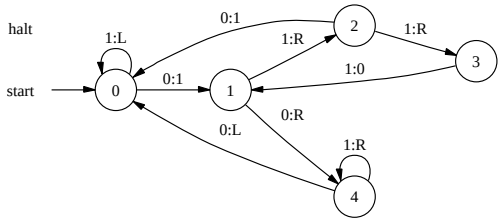


Figure A.26: Holdout machine 64

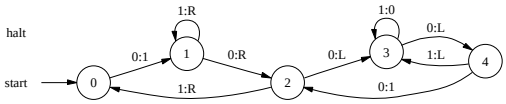


Figure A.27: Holdout machine 65

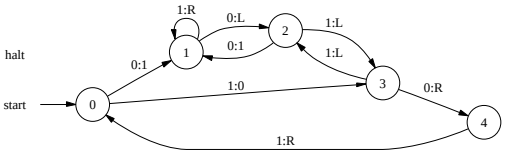


Figure A.28: Holdout machine 70

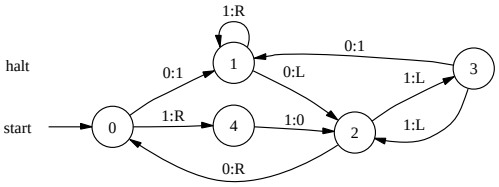


Figure A.29: Holdout machine 75

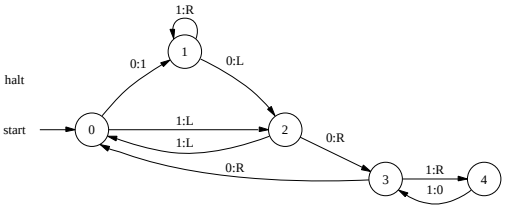


Figure A.30: Holdout machine 80

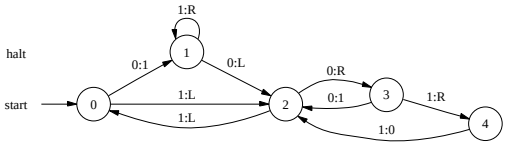


Figure A.31: Holdout machine 81

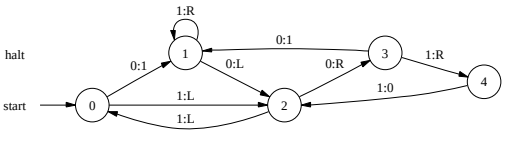


Figure A.32: Holdout machine 82

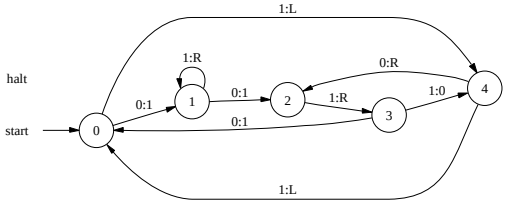


Figure A.33: Holdout machine 85

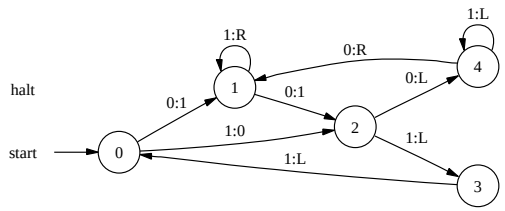


Figure A.34: Holdout machine 86

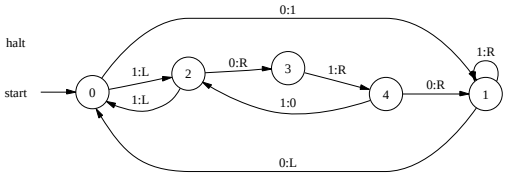


Figure A.35: Holdout machine 89

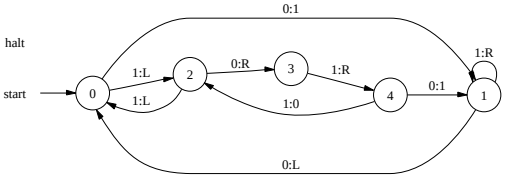


Figure A.36: Holdout machine 90

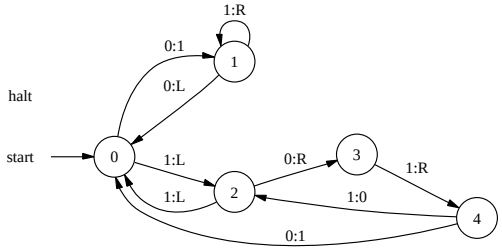


Figure A.37: Holdout machine 91

A.3 Uneven Multi-sweep Christmas Trees

The uneven multi-sweep Christmas tree Turing machines are depicted in figures A.38 through A.55.

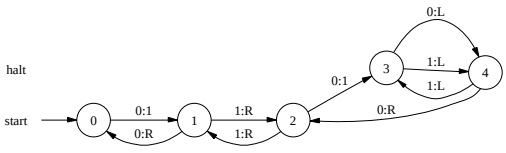


Figure A.38: Holdout machine 26

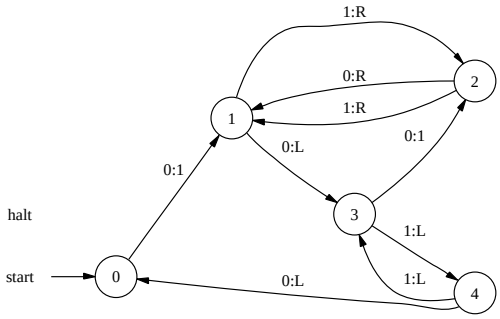


Figure A.39: Holdout machine 41

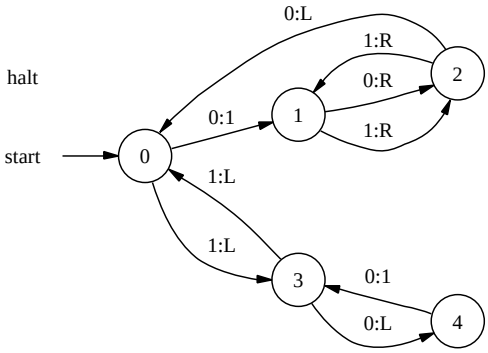


Figure A.40: Holdout machine 50

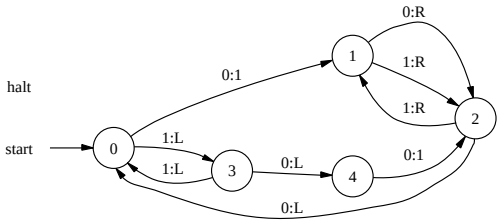


Figure A.41: Holdout machine 51

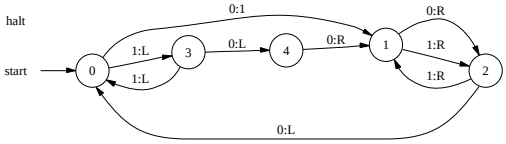


Figure A.42: Holdout machine 52

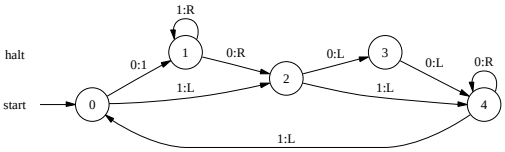


Figure A.43: Holdout machine 66

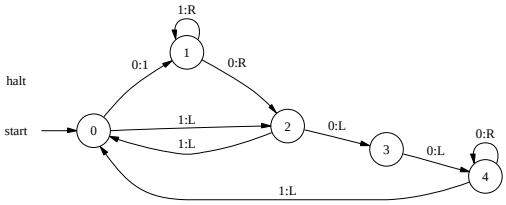


Figure A.44: Holdout machine 67

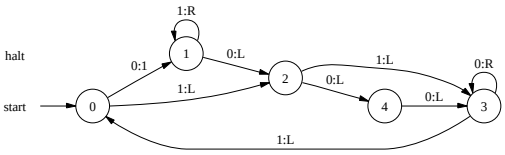


Figure A.45: Holdout machine 72

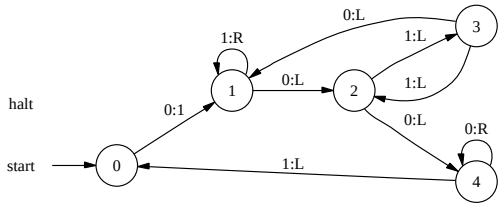


Figure A.46: Holdout machine 73

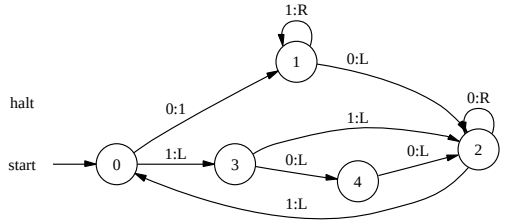


Figure A.47: Holdout machine 76

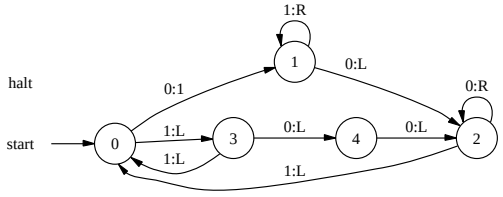


Figure A.48: Holdout machine 77

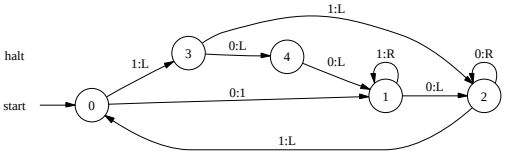


Figure A.49: Holdout machine 78

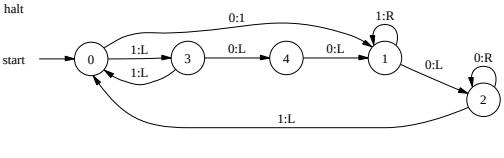


Figure A.50: Holdout machine 79

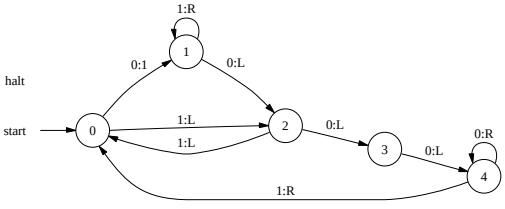


Figure A.51: Holdout machine 83

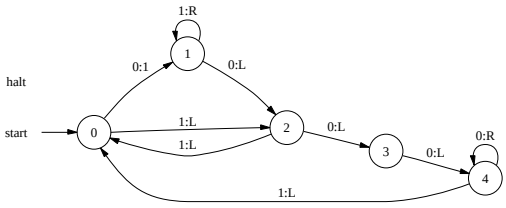


Figure A.52: Holdout machine 84

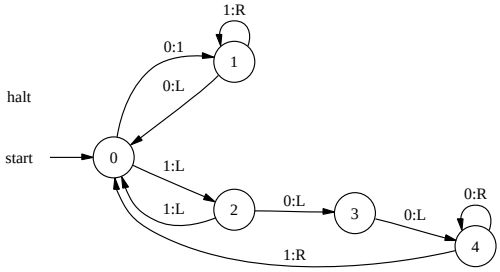


Figure A.53: Holdout machine 92

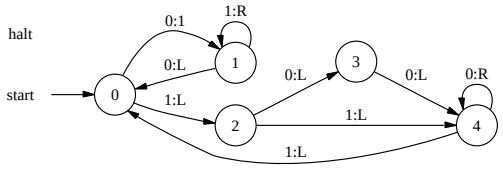


Figure A.54: Holdout machine 93

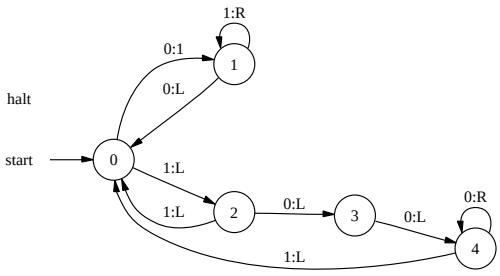


Figure A.55: Holdout machine 94

A.4 Standard Counters

The Turing machines in figures A.56 to A.58 exhibit standard counter behavior as specified in section 4.2.7 that escape the specific implementation of the automated detection routine.

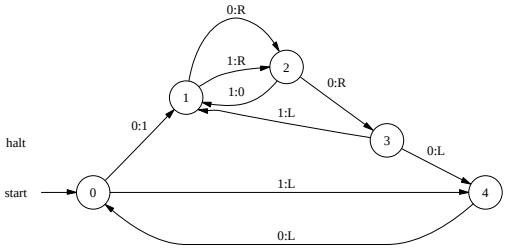


Figure A.56: Holdout machine 5 **(*)**

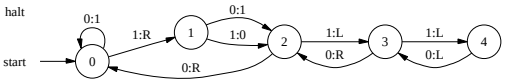


Figure A.57: Holdout machine 96 **(*)**

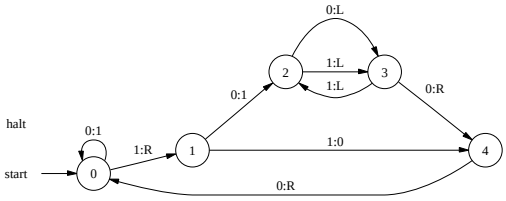


Figure A.58: Holdout machine 97 **(*)**

A.5 Base 3 Counters

The Turing machine in figure A.59 is the only holdout that has been symbolically confirmed to be a non-combination base 3 counter. However, some of the combination counters specified in section A.9 exhibit modifications of base 3 counter behavior. Also, the counters in section A.10 have not been explicitly classified into individual counter categories but it is likely that some of them are also non-combination base 3 counters as well.

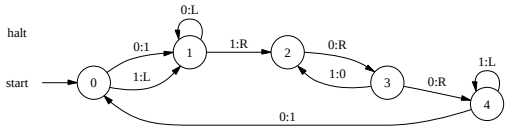


Figure A.59: Holdout machine 2 **(*)**

A.6 Base 4 Counters

While not confirmed, it has been diagrammatically deduced that the machine in figure A.60 is a base 4 counter.

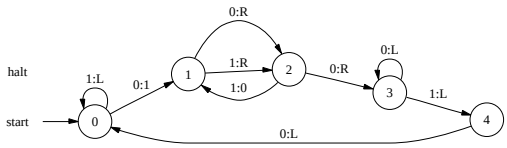


Figure A.60: Holdout machine 10

A.7 Alternating Counters

The machine in figure A.61 is a confirmed alternating counter.

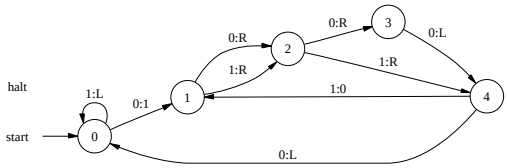


Figure A.61: Holdout machine 7 **(*)**

A.8 Resetting Counters

The machines in figures A.62 and A.63 are resetting counters.

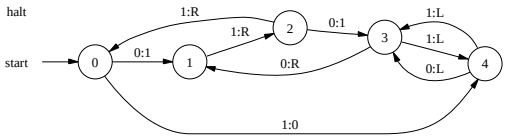


Figure A.62: Holdout machine 27

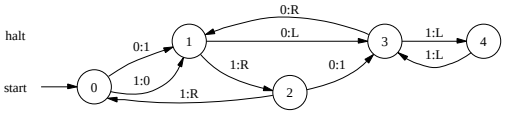


Figure A.63: Holdout machine 28

A.9 Combination Counters

The machine in figure A.64 is a confirmed combination, base 3, alternating, resetting, complex counter. Figure A.65 is a confirmed combination, alternating, resetting, complex counter. Figure A.66 is an unconfirmed base 3, alternating counter.

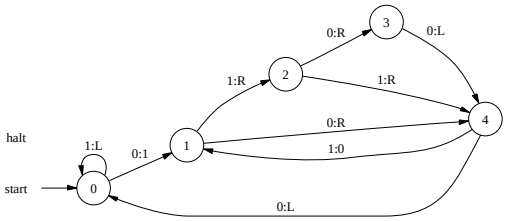


Figure A.64: Holdout machine 6 **(*)**

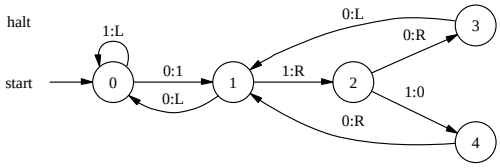


Figure A.65: Holdout machine 16 **(*)**

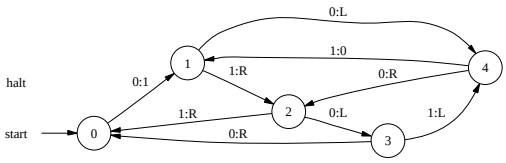


Figure A.66: Holdout machine 17

A.10 Uncategorized Counters

The machines shown in figures A.68 to A.85 have all been visually analyzed and determined to be some variation of a counter.

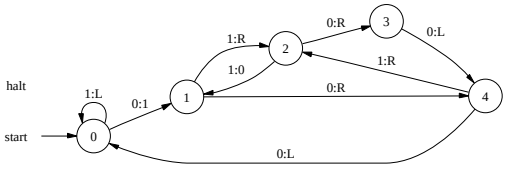


Figure A.67: Holdout machine 8

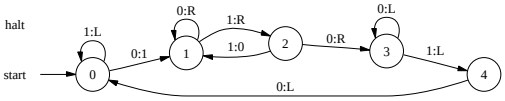


Figure A.68: Holdout machine 11

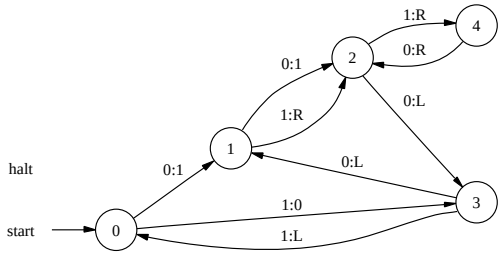


Figure A.69: Holdout machine 22

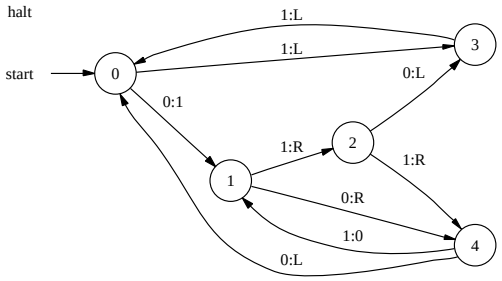


Figure A.70: Holdout machine 23

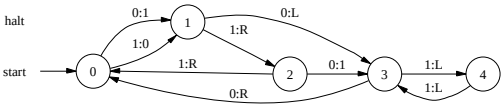


Figure A.71: Holdout machine 29

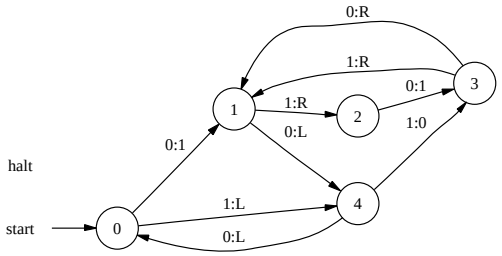


Figure A.72: Holdout machine 34

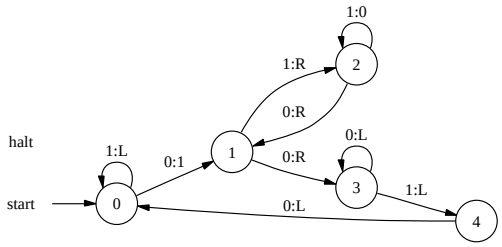


Figure A.73: Holdout machine 37

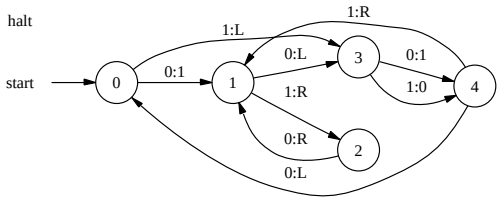


Figure A.74: Holdout machine 40

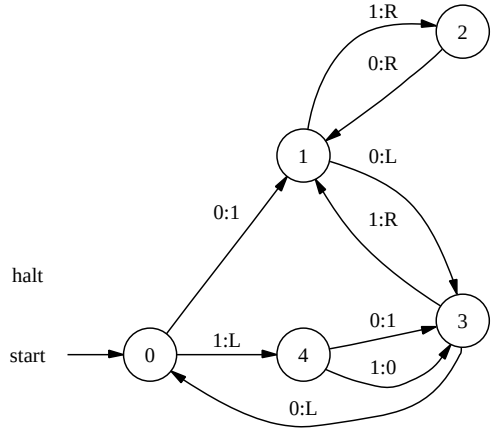


Figure A.75: Holdout machine 42

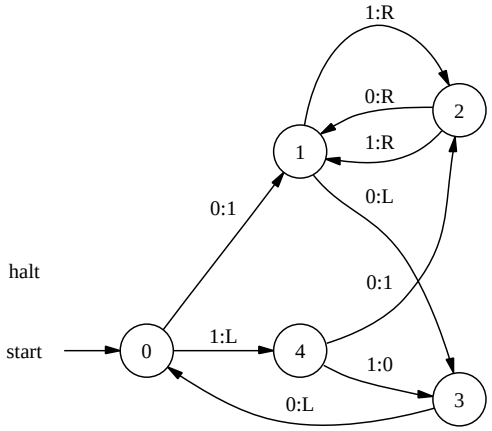


Figure A.76: Holdout machine 43

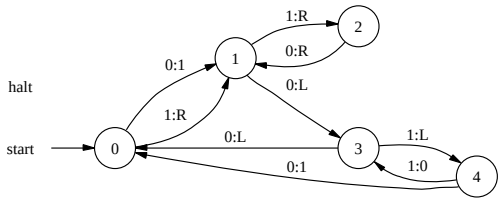


Figure A.77: Holdout machine 44

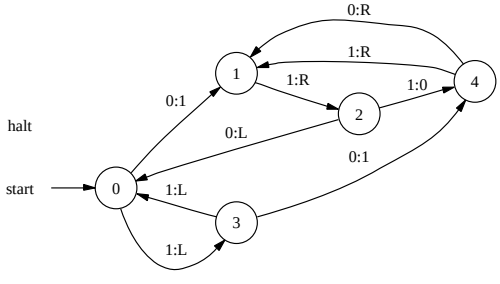


Figure A.78: Holdout machine 53

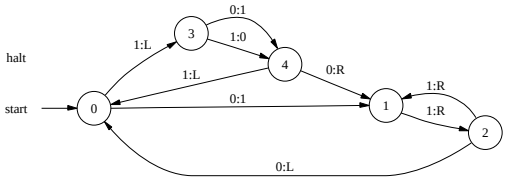


Figure A.79: Holdout machine 54

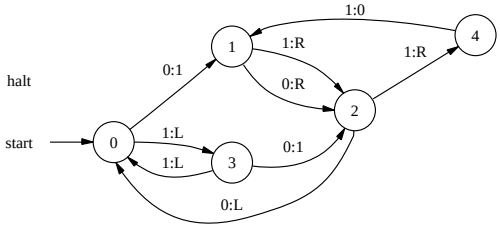


Figure A.80: Holdout machine 55

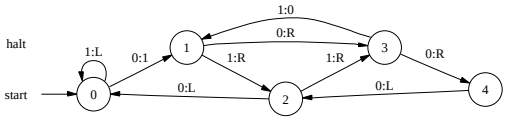


Figure A.81: Holdout machine 56

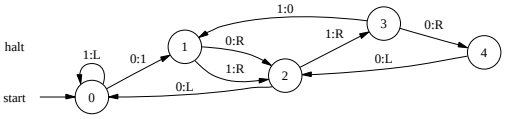


Figure A.82: Holdout machine 57

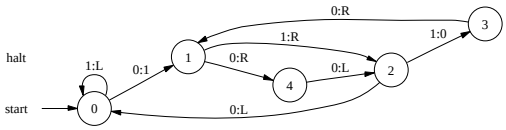


Figure A.83: Holdout machine 61

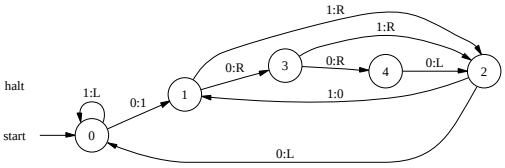


Figure A.84: Holdout machine 63

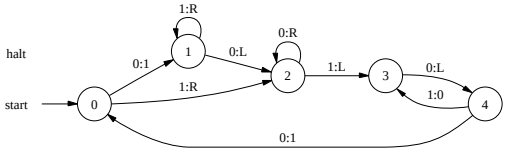
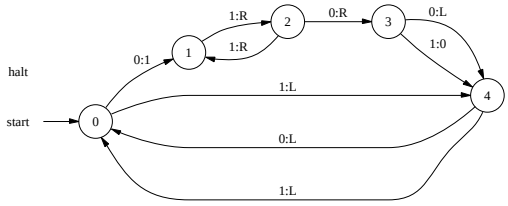


Figure A.85: Holdout machine 71

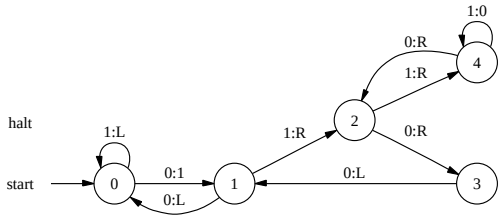
APPENDIX B
Final $\Sigma(5)$ Holdouts

The machines illustrated in this appendix have been classified into their respective non-halt categories via mechanisms of human diagrammatic reasoning. Both the flow charts and diagrammatic representations of the first 75 steps of their executions are included. For additional information on these machines as well as extended diagrams of their executions, contact the author of this thesis.



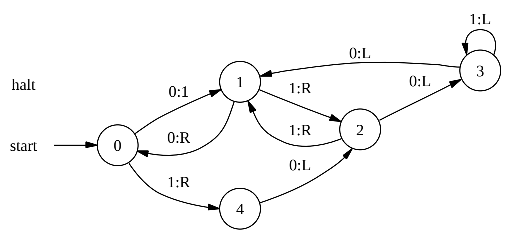

```
      0    State 0
      1    State 1
      10   State 2
      100  State 3
      100  State 4
      100  State 0
     0100  State 4
    00100  State 0
    10100  State 1
    10100  State 2
    10100  State 3
    10000  State 4
    10000  State 0
    11000  State 1
    11000  State 2
    11000  State 3
    11000  State 4
    11000  State 0
    11000  State 4
   011000  State 0
   111000  State 1
   111000  State 2
   111000  State 1
   111000  State 2
   111000  State 3
   111000  State 4
   111000  State 0
   111000  State 4
   111000  State 0
  0111000  State 4
 00111000  State 0
 10111000  State 1
 10111000  State 2
 10111000  State 3
 10011000  State 4
 10011000  State 0
 11011000  State 1
 11011000  State 2
 11011000  State 3
 11001000  State 4
 11001000  State 0
 11101000  State 1
 11101000  State 2
 11101000  State 3
 11100000  State 4
 11100000  State 0
 11110000  State 1
 11110000  State 2
 11110000  State 3
 11110000  State 4
 11110000  State 0
 11110000  State 4
 11110000  State 0
 11110000  State 4
 11110000  State 0
  0111110000  State 4
 00111110000  State 0
 10111110000  State 1
 10111110000  State 2
 10111110000  State 3
 10011110000  State 4
 10011110000  State 0
```

Figure B.1: Holdout machine 4: Multi-sweep leaning Christmas Tree



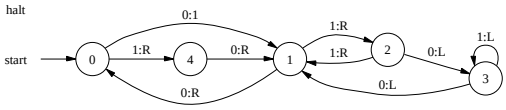

```
  0       State 0
  1       State 1
  10      State 2
  100     State 3
  10      State 1
  100     State 0
 0100     State 0
 1100     State 1
 1100     State 2
 1100     State 4
 1100     State 2
 11000    State 3
 11000    State 1
 11000    State 0
 11100    State 1
 11100    State 2
 11100    State 3
 11100    State 1
 11100    State 0
 11100    State 0
 11100    State 0
011100    State 0
111100    State 1
111100    State 2
111100    State 4
110100    State 4
110100    State 2
110100    State 4
110100    State 2
1101000   State 3
1101000   State 1
1101000   State 0
1101100   State 1
1101100   State 2
1101100   State 3
1101100   State 1
1101100   State 0
1101100   State 0
1101100   State 0
1111100   State 1
1111100   State 2
1111100   State 4
1111000   State 4
1111000   State 2
1111000   State 3
1111000   State 1
1111000   State 0
1111100   State 1
1111100   State 2
1111100   State 3
1111100   State 1
1111100   State 0
1111100   State 0
1111100   State 0
1111100   State 0
1111100   State 0
01111100  State 0
11111100  State 1
11111100  State 2
11111100  State 4
11011100  State 4
11011100  State 2
11011100  State 4
11010100  State 4
11010100  State 2
11010100  State 4
11010100  State 2
110101000 State 3
110101000 State 1
110101000 State 0
110101100 State 1
110101100 State 2
110101100 State 3
110101100 State 1
110101100 State 0
```

Figure B.2: Holdout machine 15: Christmas Tree Counter



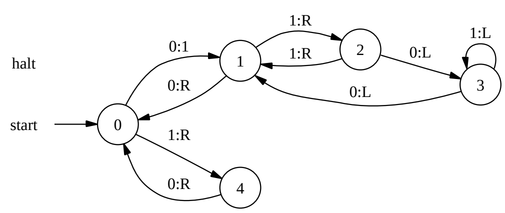
0	State 0
1	State 1
1**0**	State 2
10	State 3
010	State 3
0010	State 1
0**0**10	State 0
0**1**10	State 1
01**1**0	State 2
011**0**	State 1
0110**0**	State 0
0110**1**	State 1
01101**0**	State 2
0110**1**0	State 3
011**0**10	State 3
01**1**010	State 1
011**0**10	State 2
01**1**010	State 3
0**1**1010	State 3
011010	State 3
0011010	State 1
0**0**11010	State 0
0**1**11010	State 1
01**1**1010	State 2
011**1**010	State 1
0111**0**10	State 2
011**1**010	State 3
01**1**1010	State 3
0**1**11010	State 3
0111010	State 3
00111010	State 1
0**0**111010	State 0
0**1**111010	State 1
01**1**11010	State 2
011**1**1010	State 1
0111**1**010	State 2
01111**0**10	State 1
011110**1**0	State 0
0111101**0**	State 4
011110**1**0	State 2
0111101**0**	State 1
01111010**0**	State 0
01111010**1**	State 1
011110101**0**	State 2
01111010**1**0	State 3
0111101**0**10	State 3
011110**1**010	State 1
0111101**0**10	State 2
011110**1**010	State 3
01111**0**1010	State 3
0111**1**01010	State 1
01111**0**1010	State 2
0111**1**01010	State 3
011**1**101010	State 3
01**1**1101010	State 3
0**1**11101010	State 3
0111101010	State 3
00111101010	State 1
0**0**111101010	State 0
0**1**111101010	State 1
01**1**11101010	State 2
011**1**1101010	State 1
0111**1**101010	State 2
01111**1**01010	State 1
011111**0**1010	State 2
01111**1**01010	State 3
0111**1**101010	State 3
011**1**1101010	State 3
01**1**11101010	State 3
0**1**111101010	State 3
01111101010	State 3
001111101010	State 1
0**0**1111101010	State 0
0**1**1111101010	State 1
01**1**111101010	State 2

Figure B.3: Holdout machine 19: 1.5 Sweep Christmas Tree



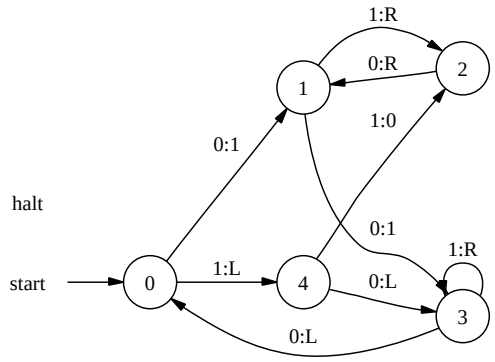

```
  0          State 0
  1          State 1
  10         State 2
  10         State 3
 010         State 3
0010         State 1
00010        State 0
01110        State 1
01110        State 2
01110        State 1
011000       State 0
011001       State 1
0110100      State 2
0110100      State 3
0111010      State 3
0111010      State 1
0111010      State 2
0111010      State 3
0111010      State 3
0111010      State 3
0111010      State 1
0111010      State 0
0111010      State 1
0111010      State 2
0111010      State 1
0111010      State 2
0111010      State 3
0111010      State 3
0111010      State 3
0111010      State 3
0111010      State 3
00111010     State 1
00111010     State 0
01111010     State 1
01111010     State 2
01111010     State 1
01111010     State 2
01111010     State 1
01111010     State 0
01111010     State 4
011110100    State 1
0111101000   State 0
0111101001   State 1
01111010010  State 2
01111010010  State 3
01111010010  State 3
01111010010  State 1
01111010010  State 0
01111010110  State 1
01111010110  State 2
01111010110  State 1
011110101100 State 0
011110101101 State 1
0111101011010 State 2
0111101011010 State 3
0111101011010 State 3
0111101011010 State 1
0111101011010 State 2
0111101011010 State 3
0111101011010 State 3
0111101011010 State 1
0111101011010 State 2
0111101011010 State 3
0111101011010 State 3
0111101011010 State 3
0111101011010 State 3
0111101011010 State 3
0111101011010 State 1
0111101011010 State 0
0111101011010 State 1
0111101011010 State 2
```

Figure B.4: Holdout machine 20: 1.5 Sweep Christmas Tree



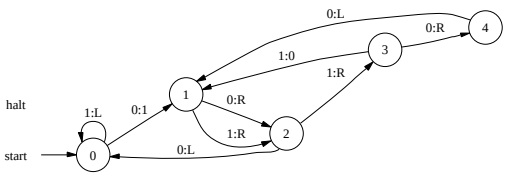
0	State 0
1	State 1
1**0**	State 2
10	State 3
010	State 3
0010	State 1
0**0**10	State 0
0**1**10	State 1
01**1**0	State 2
011**0**	State 1
0110**0**	State 0
0110**1**	State 1
01101**0**	State 2
0110**1**0	State 3
011**0**10	State 3
01**1**010	State 1
011**0**10	State 2
01**1**010	State 3
0**1**1010	State 3
011010	State 3
0011010	State 1
0**0**11010	State 0
0**1**11010	State 1
01**1**1010	State 2
011**1**010	State 1
0111**0**10	State 2
011**1**010	State 3
01**1**1010	State 3
0**1**11010	State 3
0111010	State 3
00111010	State 1
0**0**111010	State 0
0**1**111010	State 1
01**1**11010	State 2
011**1**1010	State 1
0111**1**010	State 2
01111**0**10	State 1
011110**1**0	State 0
0111101**0**	State 4
01111010**0**	State 0
01111010**1**	State 1
011110101**0**	State 2
01111010**1**0	State 3
0111101**0**10	State 3
011110**1**010	State 1
0111101**0**10	State 2
011110**1**010	State 3
01111**0**1010	State 3
0111**1**01010	State 1
01111**0**1010	State 2
0111**1**01010	State 3
011**1**101010	State 3
01**1**1101010	State 3
0**1**11101010	State 3
0111101010	State 3
00111101010	State 1
0**0**111101010	State 0
0**1**111101010	State 1
01**1**11101010	State 2
011**1**1101010	State 1
0111**1**101010	State 2
01111**1**01010	State 1
011111**0**1010	State 2
01111**1**01010	State 3
0111**1**101010	State 3
011**1**1101010	State 3
01**1**11101010	State 3
0**1**111101010	State 3
01111101010	State 3
001111101010	State 1
0**0**1111101010	State 0
0**1**1111101010	State 1
01**1**111101010	State 2
011**1**11101010	State 1
0111**1**1101010	State 2

Figure B.5: Holdout machine 21: 1.5 Sweep Christmas Tree



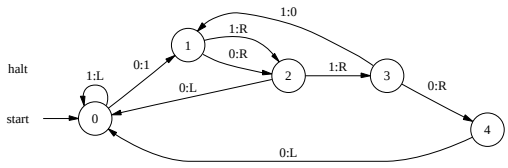

```
      0        State 0
      1        State 1
      10       State 2
      100      State 1
      101      State 3
      1010     State 3
      1010     State 0
      1010     State 4
      1010     State 3
      1010     State 3
      1010     State 0
     01010     State 4
    001010     State 3
   0001010     State 0
   1001010     State 1
   1001010     State 2
   1001010     State 1
   1011010     State 3
   1011010     State 3
   1011010     State 3
   1011010     State 0
   1011010     State 4
   1001010     State 2
   1001010     State 1
   1001010     State 2
   1001010     State 1
   1001010     State 2
   10010100    State 1
   10010101    State 3
   100101010   State 3
   100101010   State 0
   100101010   State 4
   100101010   State 3
   100101010   State 3
   100101010   State 0
   100101010   State 4
   100101010   State 3
   100101010   State 0
  0100101010   State 4
 00100101010   State 3
000100101010   State 0
100100101010   State 1
100100101010   State 2
100100101010   State 1
101100101010   State 3
101100101010   State 3
101100101010   State 3
101100101010   State 0
101100101010   State 4
100100101010   State 2
100100101010   State 1
100100101010   State 2
100100101010   State 1
100101101010   State 3
100101101010   State 3
100101101010   State 3
100101101010   State 0
100101101010   State 4
100100101010   State 2
100100101010   State 1
100100101010   State 2
100100101010   State 1
100100101010   State 2
100100101010   State 1
100100101010   State 2
1001001010100  State 1
1001001010101  State 3
10010010101010 State 3
10010010101010 State 0
10010010101010 State 4
10010010101010 State 3
```

Figure B.6: Holdout machine 49**: Asymmetric Christmas Tree**



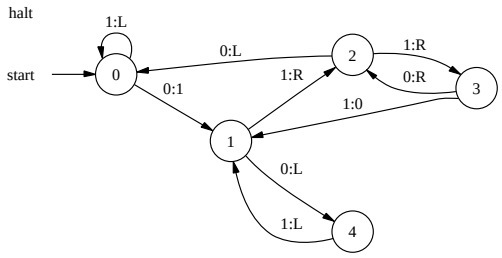

```
  0      State 0
  1      State 1
  10     State 2
  10     State 0
 010     State 0
 110     State 1
 110     State 2
 110     State 3
 1100    State 4
 1100    State 1
 1100    State 2
 1100    State 0
 1110    State 1
 1110    State 2
 1110    State 0
 1110    State 0
 1110    State 0
01110    State 0
11110    State 1
11110    State 2
11110    State 3
11010    State 1
11010    State 2
11010    State 3
110100   State 4
110100   State 1
110100   State 2
110100   State 0
110110   State 1
110110   State 2
110110   State 0
110110   State 0
110110   State 0
111110   State 1
111110   State 2
111110   State 3
111100   State 1
111100   State 2
111100   State 0
111110   State 1
111110   State 2
111110   State 0
111110   State 0
111110   State 0
 11110   State 0
 11110   State 0
01111110 State 0
11111110 State 1
11111110 State 2
11111110 State 3
11011110 State 1
11011110 State 2
11011110 State 3
11010110 State 1
11010110 State 2
11010110 State 3
110101000 State 4
110101000 State 1
110101000 State 2
110101000 State 0
110101100 State 1
110101100 State 2
110101100 State 0
110101100 State 0
110100110 State 0
110111100 State 1
110111100 State 2
110111100 State 0
110111100 State 1
110111100 State 2
110111100 State 0
110111110 State 1
110111110 State 2
110111110 State 0
110111110 State 0
```

Figure B.7: Holdout machine 58: Christmas Tree Counter



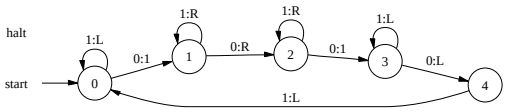

```
  0      State 0
  1      State 1
  10     State 2
  10     State 0
 010     State 0
 110     State 1
 110     State 2
 110     State 3
 1100    State 4
 1100    State 0
 1110    State 1
 1110    State 2
 1110    State 0
 1110    State 0
 1110    State 0
01110    State 0
11110    State 1
11110    State 2
11110    State 3
11010    State 1
11010    State 2
11010    State 3
110100   State 4
110100   State 0
110110   State 1
110110   State 2
110110   State 0
110110   State 0
110110   State 0
111110   State 1
111110   State 2
111110   State 3
111100   State 1
111100   State 2
111100   State 0
111110   State 1
111110   State 2
111110   State 0
111110   State 0
111110   State 0
111110   State 0
01111110 State 0
11111110 State 1
11111110 State 2
11111110 State 3
11011110 State 1
11011110 State 2
11011110 State 3
11010100 State 4
11010100 State 0
11010110 State 1
11010110 State 2
11010110 State 0
11010110 State 0
11010110 State 0
11011110 State 1
11011110 State 2
11011110 State 3
11011100 State 1
11011100 State 2
11011100 State 0
11011110 State 1
11011110 State 2
11011110 State 0
11011110 State 0
11011110 State 0
11011110 State 0
11011110 State 0
11111110 State 1
11111110 State 2
11111110 State 3
```

Figure B.8: Holdout machine 59: Christmas Tree Counter



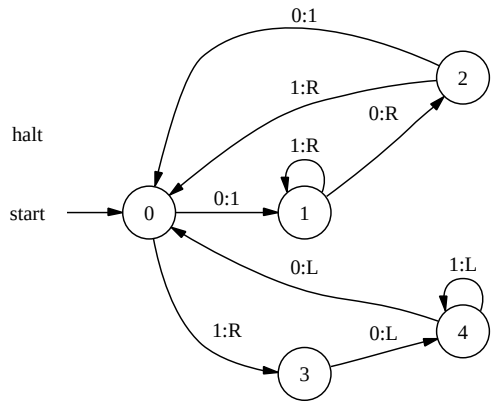

```
  0      State 0
  1      State 1
  10     State 2
  10     State 0
 010     State 0
 110     State 1
 110     State 2
 110     State 3
 1100    State 2
 1100    State 0
 1110    State 1
 1110    State 2
 1110    State 0
 1110    State 0
 1110    State 0
01110    State 0
11110    State 1
11110    State 2
11110    State 3
11010    State 1
11010    State 4
11010    State 1
11010    State 2
11010    State 3
11010    State 2
11010    State 3
110100   State 2
110100   State 0
110110   State 1
110110   State 2
110110   State 0
110110   State 0
110110   State 0
111110   State 1
111110   State 2
111110   State 3
111100   State 1
111100   State 4
111100   State 1
111100   State 2
111100   State 3
111100   State 2
111100   State 0
111110   State 1
111110   State 2
111110   State 0
111110   State 0
111110   State 0
111110   State 0
111110   State 0
0111110  State 0
1111110  State 1
1111110  State 2
1111110  State 3
1101110  State 1
1101110  State 4
1101110  State 1
1101110  State 2
1101110  State 3
1101110  State 2
1101110  State 3
1101010  State 1
1101010  State 4
1101010  State 1
1101010  State 4
1101010  State 1
1101010  State 2
1101010  State 3
1101010  State 2
1101010  State 3
1101010  State 2
1101010  State 3
11010100 State 2
11010100 State 0
11010110 State 1
```

Figure B.9: Holdout machine 60: Christmas Tree Counter



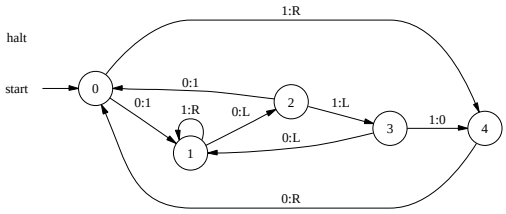

```
     0        State 0
     1        State 1
     10       State 1
     100      State 2
     101      State 3
     101      State 3
     101      State 4
    0101      State 0
    1101      State 1
    1101      State 1
    1101      State 1
    1101      State 2
    11010     State 2
    11011     State 3
    11011     State 3
    11011     State 3
    11011     State 4
    11011     State 0
   011011     State 0
   111011     State 1
   111011     State 1
   111011     State 1
   111011     State 1
   111011     State 2
   111011     State 2
   1110110    State 2
   1110111    State 3
   1110111    State 3
   1110111    State 3
   1110111    State 3
   1110111    State 4
   1110111    State 0
   1110111    State 0
  01110111    State 0
  11110111    State 1
  11110111    State 1
  11110111    State 1
  11110111    State 1
  11110111    State 1
  11110111    State 2
  11110111    State 2
  11110111    State 2
  111101110   State 2
  111101111   State 3
  111101111   State 3
  111101111   State 3
  111101111   State 3
  111101111   State 3
  111101111   State 4
  111101111   State 0
  111101111   State 0
  111101111   State 0
 0111101111   State 0
 1111101111   State 1
 1111101111   State 1
 1111101111   State 1
 1111101111   State 1
 1111101111   State 1
 1111101111   State 1
 1111101111   State 2
 1111101111   State 2
 1111101111   State 2
 1111101111   State 2
11111011110   State 2
11111011111   State 3
11111011111   State 3
11111011111   State 3
11111011111   State 3
11111011111   State 3
11111011111   State 4
11111011111   State 0
11111011111   State 0
11111011111   State 0
11111011111   State 0
```

Figure B.10: Holdout machine 68**: Asymmetric Christmas Tree**



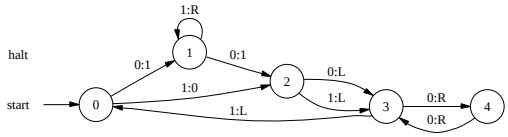

```
      0          State 0
      1          State 1
      10         State 1
      100        State 2
      101        State 0
      1010       State 3
      1010       State 4
      1010       State 4
      1010       State 0
      1010       State 3
      1010       State 4
     01010       State 4
    001010       State 0
    101010       State 1
    101010       State 1
    101010       State 2
    101010       State 0
    101110       State 1
    101110       State 1
    101110       State 1
    1011100      State 2
    1011101      State 0
    10111010     State 3
    10111010     State 4
    10111010     State 4
    10111010     State 0
    10111010     State 3
    10111010     State 4
    10111010     State 4
    10111010     State 4
    10111010     State 4
    10111010     State 0
    10111010     State 3
    10111010     State 4
   010111010     State 4
  0010111010     State 0
  1010111010     State 1
  1010111010     State 1
  1010111010     State 2
  1010111010     State 0
  1011111010     State 1
  1011111010     State 1
  1011111010     State 1
  1011111010     State 1
  10111110100    State 2
  10111110100    State 0
  10111110101    State 1
  10111110110    State 1
 101111101100    State 2
 101111101101    State 0
1011111011010    State 3
1011111011010    State 4
1011111011010    State 4
1011111011010    State 0
1011111011010    State 3
1011111011010    State 4
1011111011010    State 4
1011111011010    State 4
1011111011010    State 4
1011111011010    State 4
1011111011010    State 0
1011111011010    State 3
1011111011010    State 4
0101111011010    State 4
0010111111011010 State 0
1010111111011010 State 1
1010111111011010 State 1
1010111111011010 State 2
```

Figure B.11: Holdout machine 69: Startup effects Christmas Tree




```
          0       State 0
          1       State 1
          10      State 1
          10      State 2
         010      State 3
        0010      State 1
       00010      State 2
       10010      State 0
       10010      State 4
       10010      State 0
       10110      State 1
       10110      State 1
       10110      State 1
       10110      State 2
       10110      State 3
       10010      State 4
       10010      State 0
       10010      State 4
      100100      State 0
      100101      State 1
     1001010      State 1
     1001010      State 2
     1001010      State 3
     1001010      State 1
     1001010      State 1
     1001010      State 2
     1001010      State 3
     1001010      State 1
     1001010      State 2
    01001010      State 3
   001001010      State 1
  0001001010      State 2
  1001001010      State 0
  1001001010      State 4
  1001001010      State 0
  1011001010      State 1
  1011001010      State 1
  1011001010      State 1
  1011001010      State 2
  1011001010      State 3
  1001001010      State 4
  1001001010      State 0
  1001001010      State 4
  1001001010      State 0
  1001011010      State 1
  1001011010      State 1
  1001011010      State 1
  1001011010      State 2
  1001011010      State 3
  1001001010      State 4
  1001001010      State 0
  1001001010      State 4
  1001001010      State 0
  1001001010      State 4
 10010010100      State 0
 10010010101      State 1
100100101010      State 1
100100101010      State 2
100100101010      State 3
100100101010      State 1
100100101010      State 1
100100101010      State 2
100100101010      State 3
100100101010      State 1
100100101010      State 1
100100101010      State 2
100100101010      State 3
100100101010      State 1
100100101010      State 2
100100101010      State 3
100100101010      State 1
100100101010      State 2
 01001001010      State 3
 01001001010      State 1
 01001001010      State 2
```

Figure B.12: Holdout machine 74: Asymmetric Christmas Tree




```
  0     State 0
  1     State 1
  10    State 1
  11    State 2
  11    State 3
 011    State 0
 111    State 1
 111    State 1
 111    State 1
 1110   State 1
 1111   State 2
 1111   State 3
 1111   State 0
 1011   State 2
 1011   State 3
01011   State 0
11011   State 1
11011   State 1
11011   State 1
11111   State 2
11111   State 3
11111   State 0
01111   State 2
001111  State 3
001111  State 4
001111  State 3
001111  State 0
011111  State 1
011111  State 1
011111  State 1
011111  State 1
011111  State 1
0111110 State 1
0111111 State 2
0111111 State 3
0111111 State 0
0111011 State 2
0111011 State 3
0111011 State 0
0101011 State 2
0101011 State 3
0101011 State 0
1101011 State 1
1101011 State 1
1101011 State 1
1111011 State 2
1111011 State 3
1111011 State 0
0111011 State 2
0111011 State 3
0111011 State 4
0111011 State 3
0111011 State 0
1111011 State 1
1111011 State 1
1111011 State 1
1111011 State 1
1111011 State 1
1111111 State 2
1111111 State 3
1111111 State 0
1101111 State 2
1101111 State 3
1101111 State 0
0101111 State 2
0101111 State 3
0101111 State 4
0101111 State 3
0101111 State 0
1101111 State 1
1101111 State 1
1101111 State 1
1111111 State 2
1111111 State 3
1111111 State 0
```

Figure B.13: Holdout machine 87: Christmas Tree Counter