The Busy Beaver Problem

Ling (Esther) Fu and Sarah Pan

May 21, 2022

Contents

1 Introduction

In 1962, mathematician Tibor Radó [7] introduced the *busy beaver* problem. The objective of the problem is to find an upper bound on the number of operations done by a *halting Turing machine* of a given size. (Its name comes from the fact that it writes so quickly and abundantly, reminiscent of a beaver running back and forth.) Since there exist Turing machines of known sizes that could decide various long-standing conjectures (including Goldbach's conjecture and the Riemann hypothesis), knowing certain values of the busy beaver function would reduce determining the truth or falsity of these conjectures to finite-step computations. Unfortunately, the rapidly growing busy beaver function is not computable by algorithm, as we will show in Section 6.

In this expository paper, we will start by introducing Turing machines as a model of computation and computability (Section 2). We will then develop the machinery used to show that the busy beaver function is uncomputable: in particular, we will use the *recursion theorem* (Section 3) to show that the *halting problem* for Turing

machines is *undecidable* (Section 5) by a *mapping reduction* from A_{TM} (Section 4). Finally, we will show that the busy beaver function is uncomputable and briefly discuss the problem's implications (Section 6).

2 Turing Machines

The *Turing machine* (TM), first introduced in 1936 by mathematician Alan Turing, models a modern computer in that it can implement any computer algorithm. It is commonly represented with a finite state control and an infinite tape as its unlimited memory. Upon taking an input, the machine may either accept, reject, or loop (i.e., never halt).

In this section, we will introduce TMs, primarily following the exposition in Chapter 3 of Michael Sipser's *Introduction to the Theory of Computation* [8]. To define a TM more formally, we start with some definitions to describe the input and tape format.

Definition 2.1. An **alphabet** is an nonempty finite set. We call members of the set **symbols** of the alphabet.

Definition 2.2. A **string** is a finite combination of symbols of an alphabet. A **language** is a set of strings.

In future sections, we will use angle bracket notation $\langle \cdot \rangle$ as a way to denote string representations of other objects. Now a TM may be defined as follows:

Definition 2.3. A **Turing machine** (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are finite sets and

1. Q is a set of **states**,
2. Σ is the **input alphabet**, excluding the blank character $\sqcup$,
3. Γ is the **tape alphabet**, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the **transition function**,
5. $q_0 \in Q$ is the **start state**,
6. $q_{\text{accept}} \in Q$ is the **accept state**,
7. $q_{\text{reject}} \in Q$ is the **reject state**, where $q_{\text{accept}} \neq q_{\text{reject}}$.

Definition 2.4. The **language** of a TM is the set of strings that the machine accepts.

To compute, the TM starts with its input string on the tape, and it uses a tape head to read and edit the tape contents according to its transition function. If $\delta(q, a) = (q', b, X)$, then when the TM is in state q and the tape head reads a symbol a from the tape, the machine moves into state q', replaces a with b, and moves its tape head in the direction X.

We will be interested in what problems are computable by Turing machine, or more formally, what languages are decidable and recognizable.

Definition 2.5. A language A is **recognizable** if some TM accepts every $s \in A$ and either rejects or enters an infinite loop for every $s \notin A$.

Definition 2.6. A language A is **decidable** if some TM accepts every $s \in A$ and rejects every $s \notin A$.

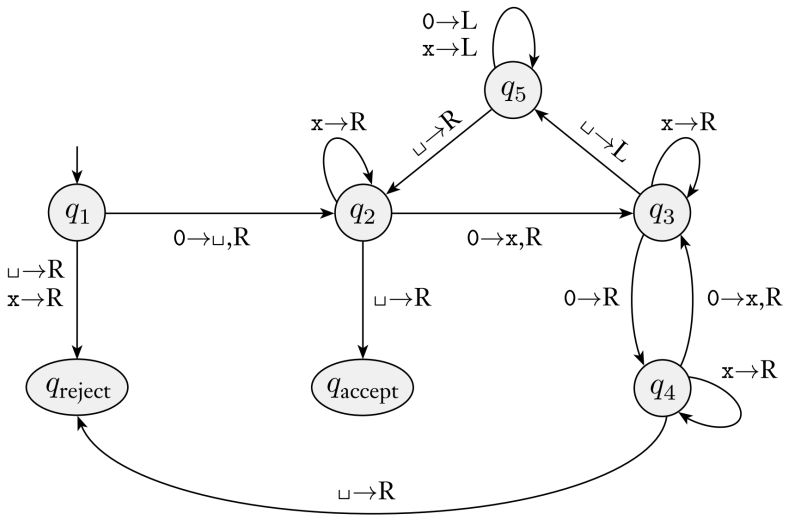
Note that all decidable languages are recognizable.

To illustrate how a TM works, we consider the following example.

Example 2.7. Consider the language $A = \{0^{2^n} | n \geq 0\}$ of strings of 0s whose lengths are of powers of 2. To see that A is decidable, we will describe a Turing machine M that decides A.

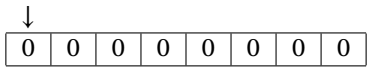
The formal description of M is $(Q, \Sigma, \Gamma, \delta, q_1\ q_{\text{accept}}, q_{\text{reject}})$, where

1. $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$;
2. $\Sigma = \{0\}$;
3. $\Gamma = \{0, \mathtt{x}, \sqcup\}$;
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is illustrated by the arrows in the state diagram below. In the diagram, an arrow from state q_i to q_j labeled by an expression the form $a \to b, X$ indicates that $\delta(q_i, a) = (q_j, b, X)$, and an arrow from q_i to q_j labeled by an expression of the form $a \to X$ indicates that $\delta(q_i, a) = (q_j, a, X)$;
5. q_1 is the start state;
6. q_{accept} is the accept state; and
7. q_{reject} is the reject state.

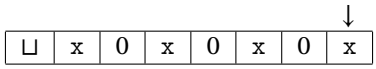


State diagram for the language $A = \{0^{2^n} | n \geq 0\}$ [8]

To see how M computes, we consider how it operates when given the following input:

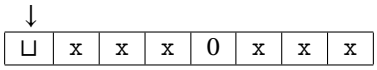


The TM M will start by replacing the first 0 with the ⊔ symbol to mark the left end of the input tape. It will then read from left to left, crossing off every other zero, as shown in the state diagram, as M moves back and forth between states q_3 and q_4.

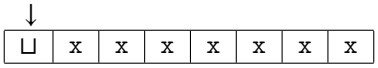


The TM M will then move left, back to the beginning of the tape. At this point, the TM has returned to state q_2.

Until the machine accepts or rejects, the TM M will repeat the above process, reading from left to right, crossing off every other zero, and moving back to the start of the tape. In the next several steps of our example, M crosses out every other uncrossed zero and starts back at the leftmost character:



In the final round, M crosses out the second uncrossed 0, leaving only the ⊔ symbol uncrossed on the tape, as shown below. After returning to state q_2, M will accept and halt.



Ultimately, in this scenario, M cuts the number of 0s in half until the number reaches one (which has been replaced by a ⊔ symbol). In particular, this process checks whether the length is a power of 2, consistent with the language of the machine. In our example, the length of the input is 8, a power of 2; therefore, the machine will accept upon running on input 00000000.

The **Church-Turing thesis** states that the intuitive notion of algorithms, proposed by Alonzo Church, is essentially equivalent to Turing machine algorithms. Moving forward, our descriptions of TMs use algorithm pseudocode rather than the formal notation of Definition 2.3.

3 The Recursion Theorem

In this section we introduce the recursion theorem, roughly following the discussion in Section 6.1 of Sipser's *Introduction to the Theory of Computation* [8]. This theorem allows a Turing machine to procure its own description and to run itself as a subroutine. Evidently, implications of this theory serve as a tenet to self-replicating

programs such as computer viruses. However, the recursion theorem's applications are not limited to malware, as we will see in the next section.

An important part of the proof is to see that we can produce a Turing machine that prints out any given input string. To do this, we use a particular **computable function** q. Vaguely, we say a function is *computable* if we can describe an algorithm that can carry out the function. We define the function q to take an input string w and output the description of Turing machine P_w that prints out w. The following TM Q computes $q(w)$:

Q = "On input string w :

1. Construct the Turing machine P_w as follows:

   P_w = "On any input:

   (a) Write w on the tape after the input.

   (b) Halt."

2. Output $\langle P_w \rangle$."

Theorem 3.1 (Recursion theorem)**.** *Suppose the Turing machine T computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. Then there exists a Turing machine R that computes the function $r : \Sigma^* \to \Sigma^*$ defined by*

$$r(w) = t(\langle R \rangle, w).$$

Our original goal was to allow a Turing machine to compute its own description, which it now can! To give more insight into the problem, say that we have a TM U, which we want to obtain its own description. The recursion theorem comes into play when we create a TM T to behave just like U but with an extra first input. The machine T essentially simulates U on the second input, string w, but it also records the first input, which we interpret as the description of a machine. The recursion theorem guarantees the existence of a machine R such that when R receives the input w, it simulates T on the second input w but receives its own description $\langle R \rangle$ as the first input. The construction of TM T implies that R simulates U on the input w and receives its own description $\langle R \rangle$ as an extra input.

Proof of the recursion theorem. We construct the TM R by combining three Turing machines A, B, and T. The role of A is to produce a description of the TMs B and T, and the role of B is to produce a description of A. Then the descriptions of A, B, and T (or, in other words, the description of the TM R) are fed to the TM T, where they are then used to compute on input w.

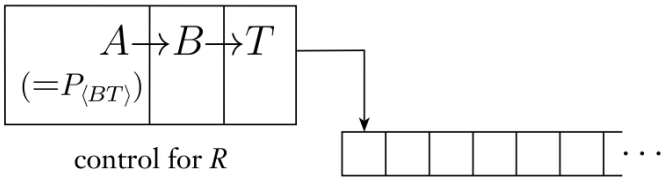


Figure 1: Schematic for R (see [8], p. 249)

In order to do this, we utilize $q(w)$. Constructing TM A is simple as we can define it as the TM $P_{\langle BT \rangle}$ given by $q(\langle BT \rangle)$. We can use the output of this TM A to create B's description. In particular, the TM B applies q to the tape contents (produced by A) to get $\langle A \rangle$ since A is defined to be the printer that prints a description of B. Then B combines all three parts A, B, and T to obtain the description $\langle ABT \rangle = \langle R \rangle$. It passes control to T after recording $\langle R, w \rangle$ on the tape. □

4 The Undecidability of A_{TM}

In this section, we will use the recursion theorem to prove the undecidability of the acceptance problem for TMs. More formally,

Definition 4.1. The language A_{TM} consists of all TM Ms and strings ws for which M accepts w:

$$A_{TM} = \{\langle M, w \rangle | \ M \text{ is a TM and } M \text{ accepts } w\}.$$

Theorem 4.2. *The language A_{TM} is undecidable.*

Proof. Assume that the language A_{TM} is decidable, and let some TM H decide it. The machine H takes the description of a machine and a string as its input. We will construct another TM B that runs H as a subroutine. The machine B only takes in a string w. By the recursion theorem, B is able to obtain its own description $\langle B \rangle$. It then feeds this description and its input string w to H. After the TM H runs on $\langle B, w \rangle$, control is passed back to B where the opposite is returned: if H rejects w, B accepts, and vice versa.

Evidently, this is a contradiction because H should work on $\langle B, w \rangle$ exactly as B works on w. The opposite is true, however, which contradicts our original assumption that A_{TM} is decidable. □

5 Mapping Reductions

Now that we have shown the undecidability of A_{TM}, we naturally wonder about other languages that are undecidable. Instead of fleshing out an entirely new proof to show undecidability, however, we can simply demonstrate that certain other languages reduce to A_{TM}.

Formally, we define mapping reducibility as follows:

Definition 5.1. Language A is *mapping* reducible to language B (written as $A \leq_m B$) if there is a function f such that for every string w, we have

$$w \in A \iff f(w) \in B.$$

Theorem 5.2. *If $A \leq_m B$ and B is decidable, then A is decidable.*

Proof. We can show this by letting M be the Turing machine that decides B. We then construct a TM N that decides A as follows.

N = "On input w :

1. Compute $f(w)$.
2. Run M on whatever $f(w)$ produced and output as M does." □

In proving undecidability, we will mainly use the contrapositive: if $A \leq_m B$ and A is undecidable, then B is undecidable. In particular, to show a language is undecidable, it suffices to reduce to that language from A_{TM}.

One language that will be useful in Section 6 is the following, which is broader than A_{TM} but similar in essence:

Definition 5.3. The language $HALT_{TM}$ contains the descriptions of all machines M and w such that M halts on w:

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ halts on input } w\}.$$

This language is very similar to A_{TM}, but broader in the sense that the TM can also reject its input. Intuition about this problem comes from the fact that we'll never know whether a Turing machine is taking a long time to run or if it is looping indefinitely. As a result, we propose that $HALT_{TM}$ is also undecidable.

Theorem 5.4. *The language $HALT_{TM}$ is undecidable.*

Proof. To prove the undecidability of $HALT_{TM}$, we will show that A_{TM} reduces to $HALT_{TM}$. By our definition of mapping reductions this also means that

$$\langle M, w\rangle \in A_{TM} \iff \langle M', w'\rangle \in HALT_{TM},$$

where M' and w' are given by the function f. We construct a TM F as follows:
F = "On input $\langle M, w\rangle$:

1. Construct the following machine M'.

   M' = "On input x:

   (a) Run M on x.

   (b) If M accepts, *accept*.

   (c) If M rejects, enter a loop."

2. Output $\langle M', w\rangle$."

Because we are able to reduce the problem of whether something is in A_{TM} to whether something is in $HALT_{TM}$, it follows that $HALT_{TM}$ is also undecidable. □

6 The Busy Beaver Problem

Proving the undecidability of $HALT_{TM}$ is among the most important ideas in computability theory. It is more commonly known as the *halting problem*, and its inability to be solved by algorithm has many intriguing applications, one of which is the busy beaver problem.

We start by introducing a variant of the Turing machine, which we will call the *halt-state Turing machine*. The only difference between the two machines lies in their terminating states. Instead of having accept or reject states, the halt-state Turing machine has an all-encompassing halt state. Everything else remains identical.

Definition 6.1. The **nth busy beaver number**, denoted $BB(n)$, is the maximum finite number of state shifts undergone by a halt-state Turing machine with n non-accept/reject states and the alphabet $\{0, 1\}$. Here, we require that each halt-state TM under consideration actually halts (i.e., does not loop indefinitely).

The first busy beaver number is $BB(1) = 1$. Trivially, a Turing machine with one non-halting state can only move from its start state to its halt state. The second busy beaver number is 6 (see [6]), which we discuss in the next example.

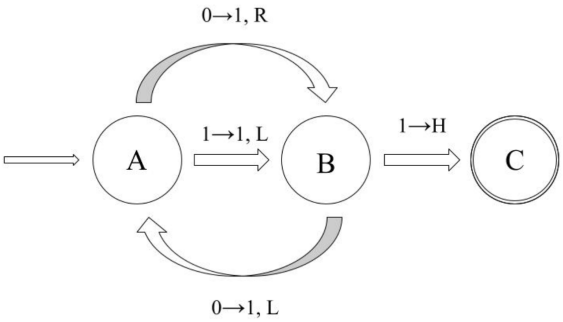


Figure 2: A state diagram for $BB(2)$

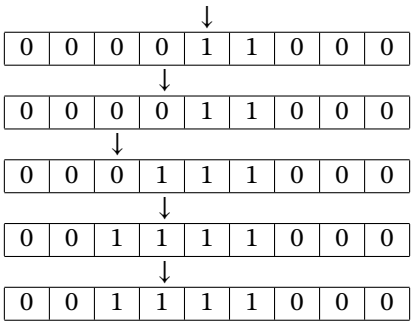
Example 6.2. The figure above gives one example of a "busiest" halt-state TM with two non-accept states: it goes through $BB(2) = 6$ state shifts when given an input tape of all zeros. To see how the TM operates, we illustrate its tape below. Initially it only contains zeros. We also imagine this tape to be infinite in both directions.

				↓				
0	0	0	0	0	0	0	0	0

First, the TM reads a zero, which prompts it to replace it with a one and move its tape reader to the right.

					↓			
0	0	0	0	1	0	0	0	0

Following the transition functions as described in the state diagram, we see that the next few computations proceed as so:



The reason the last two tapes are identical is due to our definition of $BB(n)$. Originally, this function is what Radó proposed to be his shift function $S(n)$. Essentially, $S(n)$ increments every time a state is visited. Although nothing new appears on the tape after the fifth operation, the state control finally visits state C, or the halting state, which counts as a shift.

As mentioned earlier, one can show that the above example is a "busiest" 2-state halt-state Turing machine, i.e., $BB(2) = 6$. The next value in the sequence is $BB(3) = 21$. Although this number seems small, it was not easy to show that this was the "busiest" 3-state Turing machine (see [6]). Intuitively, there are many Turing machines with n states. However, determining which machines halt, and among those that do, which one is the "busiest" requires a lot of work. Perhaps unsurprisingly, the busy beaver function is *uncomputable* by Turing machine (i.e., there exists no Turing machine that takes n as input and computes $BB(n)$).

Theorem 6.3. *The busy beaver function is uncomputable.*

Proof. Let's say that TM A computes $BB(n)$. We define another TM M as follows:
M = "On input $\langle T \rangle$:

1. Find the number of states in T (this is included in the description of T), and assign this value to n.
2. Run A to find $BB(n)$.
3. Simulate an empty input on T.
   (a) If T halts within $BB(n)$ steps, accept.
   (b) If T has not halted within $BB(n)$ steps, reject.

As shown, if the TM A could compute $BB(n)$, we would have solved the halting problem. However, because the halting problem was shown to be undecidable in Theorem 5.4, we have arrived at a contradiction. Thus, $BB(n)$ is undecidable. □

6.1 Implications of the Busy Beaver

Currently, the best candidate for $BB(5)$ is 47,176,870 (see [4], as well as Section 5 of [1]), and it is known that $BB(6)$ is at least 7.4×10^{36543} (see [2]). Research about busy beaver numbers seems stagnant as these results were discovered using supercomputers some 30 years ago. However, insights about this fast-growing function could provide headway in other mathematical areas. For example, consider the following famous conjecture from number theory:

Conjecture 6.4 (Goldbach, 1742)**.** *Every even whole number greater than two can be expressed as the sum of two primes.*

In 2015, an anonymous Github user proposed a 27-state Turing machine[1] that halts if and only if the Goldbach conjecture is false. If so, then $BB(27)$ would give an upper bound on the number of operations this algorithm could perform. If the maximum is exceeded before the machine halts, then the conjecture would be proven true. On the other hand, if the algorithm halts within $BB(27)$ operations, the conjecture would be proven false.

Similar results for many longstanding mathematical results can also be shown. For example, Matiyasevich, O'Rear, and Aaronson[2] showed in 2016 that a 744-state Turing machine halts if and only if the Riemann Hypothesis, dubbed as the most important conjecture in pure mathematics, is false. As for the Zermelo-Fraenkel axioms, O'Rear[3] created a 748-state Turing machine that halts if and only if ZF is inconsistent. Using busy beaver numbers, we can reduce proving these conjectures to finite-step computations.

7 Acknowledgements

First and foremost, we would like to thank to our phenomenal, cool, intelligent mentor Alexandra Hoey!

Thanks to the PRIMES Circle program and the amazing coordinators Marisa Gaetz and Mary Stelow!

Thanks to Sarah's computer science teacher, Tom, Michael Sipser for writing such an informative book, and Scott Aaronson for being very knowledgeable about the busy beaver problem.

References

[1] S. Aaronson. *The Busy Beaver Frontier*, 2020.
https://www.scottaaronson.com/papers/bb.pdf

[1] See https://github.com/sorear/metamath-turing-machines/blob/master/zf2.nql

[2] The construction is given at https://github.com/sorear/metamath-turing-machines/blob/master/riemann-matiyasevich-aaronson.nql.

[3] See https://github.com/sorear/metamath-turing-machines/blob/master/zf2.nql

[2] P. Kropitz. Busy Beaver Problem. Bachelors thesis, Charles University in Prague, 2011. In Czech. https://is.cuni.cz/webapps/zzp/detail/49210/.

[3] J. Kun. *Busy Beavers, and Quest for Big Numbers*, 2012. https://jeremykun.com/2012/02/08/busy-beavers-and-the-quest-for-big-numbers/

[4] H. Marxen and J. Buntrock. *Attacking the Busy Beaver 5.* Bulletin of the EATCS, 40:247–251, 1990. http://turbotm.de/~heiner/BB/mabu90.html.

[5] R. Mullins. *What is a Turing machine?* [illustration], 2012. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html

[6] S. Lin and T. Radó. *Computer studies of Turing machine problems*. J. of the ACM, 12(2):196– 212, 1965

[7] T. Radó. *On non-computable functions.* Bell System Technical Journal, 41(3):877–884, 1962. https://archive.org/details/bstj41-3-877/mode/2up.

[8] M. Sipser. *Introduction to the Theory of Computation* (3rd ed.). *Cengage Learning*, 2013.