CDM

Turing Computability

Klaus Sutner

Carnegie Mellon University

Fall 2025

This is Fraud

The pictures are stunningly beautiful, but we have not really explained what they are a picture of.

More precisely, we have a definition of a Turing machine, but we appeal to intuition when it comes to computations of these machines. As far as intuition is concerned, this is fine, but to prove anything we also need clear, formal definitions.

The Approach

Given any type of machine $\mathcal{M}$, to define computations of $\mathcal{M}$, it is a good idea to think about interrupting a computation.

What information is minimally needed to resume the computation later? This is called a configuration or instantaneous description (ID).

Then we explain how $\mathcal{M}$ moves from one configuration to the next, the one-step relation. Chaining together single steps by induction, we get a many-step relation that formalizes computations of arbitrary length.

We have to add input/output conventions, and then we're done.

Formalizing TM Computation

Configurations here consist of:

- the current state
- the current head position
- the current tape inscription

We can think of the head position as an integer, and the tape inscription as a map $\mathbb{Z} \to \Gamma$. So the space of configurations is

$$Q \times \mathbb{Z} \times (\mathbb{Z} \to \Gamma)$$

For us only finite tape inscriptions matter, only finitely many cells carry a non-blank symbol.

As it turns out, the following, less obvious, representation for finite inscriptions is often better in applications.

String Representation

We may safely assume that $Q \cap \Gamma = \emptyset$. Use $Q \cup \Gamma$ as an alphabet and strings of the form $\Gamma^\star \, Q \, \Gamma^+$ to encode configurations.

Definition

A configuration or instantaneous description (ID) is a word $y \, p \, x$ where $x \in \Gamma^+$, $y \in \Gamma^\star$ and $p \in Q$:

$$y_m y_{m-1} \dots y_1 \, p \, x_1 x_2 \dots x_n$$

means that the read/write head is positioned at x_1 and the tape inscription is $y_m \dots y_1 x_1 \dots x_n$.

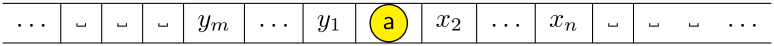
In this representation one can use the theory of finite state machines to explain what it means to perform one step in a computation (a finite state transducer is enough).

Details

The last description really captures one particular type of Turing machine: a

TM with a single, two-way-infinite tape

with finite inscriptions of the form



where the head is at $a = x_1$ and the machine is in state p.

Alas, this is just the tip of the iceberg: it is also useful to consider machines that have multiple tapes (each with a separate read/write head) and/or one-way-infinite tapes. In addition, later we will use separate read-only input tapes, and write-only output-tapes (these are always one-way-infinite).

More Details

We may safely assume that $n \geq 1$ since we can always let x_1 be the blank symbol.

It also makes sense to choose n and m to be minimal such that $\boldsymbol{yx}$ captures all the non-blank symbols on the tape. We won't bother to make this part of the definition, though, it really does not matter much.

This is subtly different from the model where a tape inscription is a map $\mathbb{Z} \to \Gamma$: our approach is coordinate free. Usually that is better, but sometimes it is preferable to keep track of the absolute position on the tape.

E.g., this is used in the proof of the Cook-Levin theorem (hardness of Satisfiability).

One Step

Next we need to explain a single step in a computation:

$$ypx \mathrel{\vdash^{1}_{\mathcal{M}}} y'qx'$$

Recall that we assume x to be non-empty and let $\delta(p, x_1) = (q, a, \varDelta)$. Then the next configuration is defined by

$$\begin{array}{ll}
y_m \dots y_1\, p\, x_1 x_2 \dots x_n & \vdash^{1}_{\mathcal{M}} \\
y_m \dots y_2\, q\, y_1 a x_2 \dots x_n & \varDelta = -1 \\
y_m \dots y_1\, q\, a\, x_2 \dots x_n & \varDelta = 0 \\
y_m \dots y_1 a\, q\, x_2\ \dots x_n & \varDelta = +1
\end{array}$$

Annoying Details

Strictly speaking, the last definition is wrong: on the one hand, we insist that $\boldsymbol{x}$ is not empty.

On the other hand, the definition can break this condition: think about $\Delta = +1$ and $n = 1$. Similarly, for $\Delta = -1$, we need y_1 to be defined. Of course, there is an easy fix ...

In polite company, this is a non-issue. Alas, any theorem prover would strenuously disagree.

Multiple Steps

Now we extend the "one-step" relation[†] to multiple steps by induction:

- one step

  $C \mathrel{\Big|\!\frac{1}{\mathcal{M}}} C' \iff$ as on the last slide

- exactly t steps

  $C \mathrel{\Big|\!\frac{t}{\mathcal{M}}} C' \iff \exists\, D\, (C \mathrel{\Big|\!\frac{t-1}{\mathcal{M}}} D \wedge D \mathrel{\Big|\!\frac{1}{\mathcal{M}}} C')$

- any finite number of steps

  $C \mathrel{\Big|\!\frac{*}{\mathcal{M}}} C' \iff \exists\, t\, (C \mathrel{\Big|\!\frac{t}{\mathcal{M}}} C')$

[†]For us, the relation is actually a function, but in complexity theory one often consider nondeterministic TMs where there are several possible next configurations.

Complexity

Lemma

The relation $\vdash^{t}_{\mathcal{M}}$ *is primitive recursive, uniformly in* t*.*

Meaning that there is a primitive recursive relation $R \subseteq \mathbb{N}^3$ such that

$$R(t, \langle C \rangle, \langle C' \rangle) \iff C \vdash^{t}_{\mathcal{M}} C'$$

Here $\langle C \rangle$ is supposed to be a code number representing the configuration C. Say, let $\Gamma = [n]$ and $Q = [n{+}1, m]$ and set

$$\langle C \rangle = \langle y_k, \ldots, y_1, q, x_1, \ldots, x_\ell \rangle$$

Warning

From the perspective of abstract computation, the relation $\vdash^{t}_{\mathcal{M}}$ is easy. But the relation $\vdash^{*}_{\mathcal{M}}$ most emphatically is not.

If there is a p.r. bound on the length of the computations of a TM, then the machine computes a p.r. function: we just iterate the one-step operation an appropriate number of steps. Otherwise we just have to run the machine without any idea if and when it might stop.

So the difference between primitive recursive and Turing computable is just one unbounded search, one existential quantifier. This is the critical difference.

Input and Output

Given any input $x = x_1 x_2 \dots x_n \in \Gamma^\star$, the initial configuration for x is

$$C_x^{\mathsf{init}} = q^{\mathrm{init}} \,\llcorner\!\lrcorner\, x_1 x_2 \dots x_n$$

A halting configuration is any configuration where the state is q^{halt} and the current symbol is a blank. In this case, we are only interested in the non-blank symbols immediately to the right of the head.

$$C_y^{\mathsf{halt}} = \dots q^{\mathrm{halt}} \,\llcorner\!\lrcorner\, y_1 y_2 \dots y_m \llcorner\!\lrcorner \dots$$

This makes sequential composition of TMs very easy.

Conventions

Our conventions are fairly natural, but there are lots of alternatives:

- Initial configurations could look like $q^{\mathrm{init}} x_1 \dots x_n$.
- In a halting configuration, we could require the machines to erase the tape except for the output block.
- We are using the blank symbol $\textvisiblespace$ to terminate input and output; we could use a special endmarker like $\#$ instead.

It is tedious but straightforward to check that none of this makes any difference: we get the same clone of computable functions.

Exercises

Exercise

Explain in detail how to deal with blanks in the input.

Exercise

Come up with a way to associate output with arbitrary halting configurations.

Exercise

Can your machines be simulated by machines conforming to our definitions?

Halting and Output

A configuration C is mortal if $C \vdash^{*}_{\mathcal{M}} C'$ where C' is halting.

Machine $\mathcal{M}$ halts on input x if C_x^{init} is mortal.

We say that $y \in \Gamma^\star$ is the output of the computation of machine $\mathcal{M}$ on input $x \in \Sigma^\star$ if

$$C_x^{\mathsf{init}} \vdash^{*}_{\mathcal{M}} C_y^{\mathsf{halt}}$$

So if we wind up in configuration $abbb\, q^{\mathrm{halt}} \sqcup abc \sqcup accc$ we consider abc to be the output.

Computing a Function

We want to use Turing machines to define computable functions.

Recall from our discussion of primitive recursive functions that we will have to deal with partial functions if we want evaluation to be computable.

Turing machine $\mathcal{M}$ computes the partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$ if, for all $x \in \Sigma^\star$, we have:

- If f is defined on x, then $C_x^{\text{init}} \vdash_{\mathcal{M}}^{*} C_y^{\text{halt}}$ and $f(x) = y$.
- If f is undefined on x, the computation of $\mathcal{M}$ on x does not halt.

Computability Defined

Definition

A partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$ is (Turing) computable if there is a Turing machine $\mathcal{M}$ that computes f.

There is a mountain of evidence that, for any reasonable model of computation $\mathfrak{M}$, it turns out that $\mathfrak{M}$-computable is equivalent to Turing computable, so it makes sense to simply say computable, without reference to any particular model.

Arities

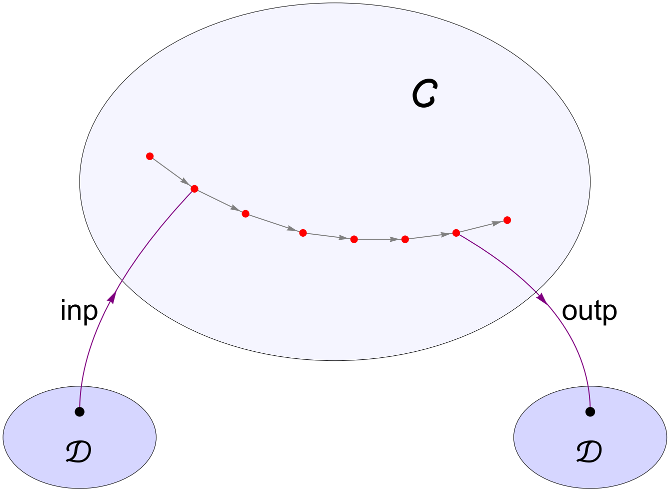
By our definition, Turing computable functions are all unary, they take exactly one string as input.

That's not a problem if we want to compute, say, binary arithmetic functions like addition or multiplication.

$$q^{\text{init}} \sqcup \#1001\#11000\# \sqcup$$

We use $\#$ as a separator and write the numbers in binary.

So we will write things like $\mathcal{M}(x, y)$ to indicate TM $\mathcal{M}$ is running on inputs x and y.



$\mathcal{C}$
inp
outp
$\mathcal{D}$
$\mathcal{D}$

Turing Decidability

We use characteristic functions to lift Turing computability to relations (or sets).

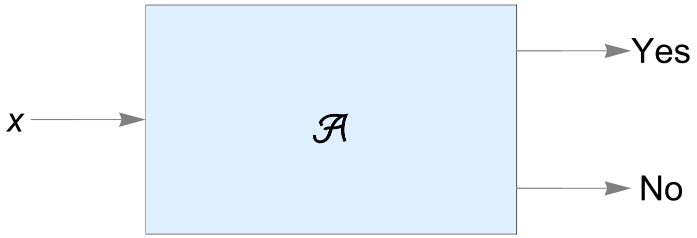
$$\text{char}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise.} \end{cases}$$

Definition

A relation $A \subseteq \Sigma^\star$ is (Turing) decidable if its characteristic function is Turing computable.

We use the same terminology for $A \subseteq \mathbb{N}^k$ via encoding.

Decidability



Note that these machines $\mathcal{A}$ always halts.

Acceptors versus Transducers

Informally, a problem is decidable if there is a decision algorithm $\mathcal{A}$ that returns Yes or No depending on whether the input has the property in question.

Again, these Turing machines must always halt, and return a one-bit output. They are often called acceptors.

By contrast, a Turing machine that computes some arbitrary (and possibly partial function) is called a transducer[‡].

[‡]The same terminology is also used for finite state machines.

Closure Properties

Lemma

The decidable sets are closed under intersection, union and complement. In other words, the decidable sets form a Boolean algebra.

Proof.

Consider two decidable sets $A, B \subseteq \Sigma^\star$. We have two TMs $\mathcal{M}_A$ and $\mathcal{M}_B$ that decide membership.

We can simply run both $\mathcal{M}_A$ and $\mathcal{M}_B$ on input x sequentially, producing a 2-bit result.

It is straightforward to process the output of the two runs and return the correct yes/no answer.

□

Evaluation

Again there is an evaluation operation or a simulator that takes as input a Turing machine and a string and runs the machine on the string. The type of eval is

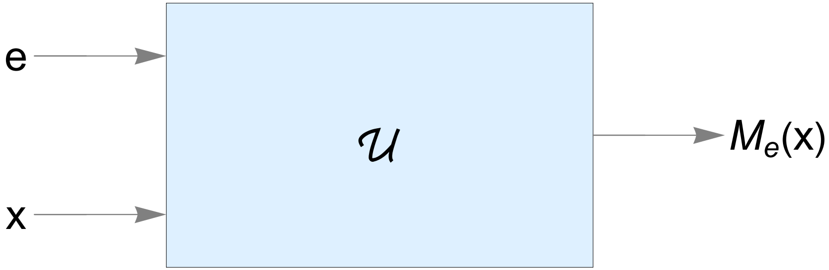
$$\text{eval} : \text{TMs} \times \Sigma^\star \longrightarrow \Sigma^\star$$

Encoding a TM as a string is trivial. But note, some care is needed to handle the tape alphabet of the machine, we need a little coding to express the arbitrary alphabet in the simulated machine in terms of the fixed alphabet Σ.

At any rate, we can think of a map $\text{eval} : \Sigma^\star \times \Sigma^\star \to \Sigma^\star$.

So the next question is whether evaluation can be handled by a Turing machine, a so-called universal Turing machine.

UTM



e is the index for the machine and x the input string.

They Exist

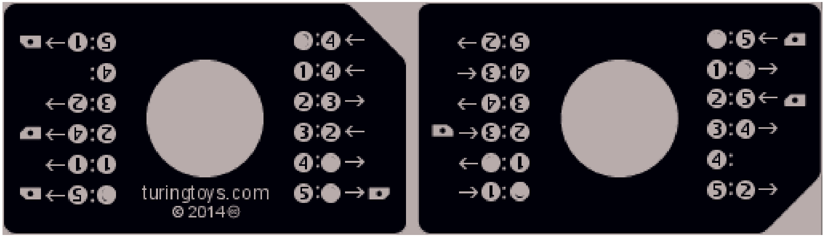
Theorem (Turing 1936)

There is a universal Turing machine.

It immediately follows that the clone of Turing computable functions must contain partial functions.

Finding small universal TMs is still an active research area.

Small Universal



A universal machine (4 states, 6 tape symbols) due to Yurii Roghozin.

Halting Problem(s)

Problem:	**Halting**
Instance:	Index $e \in \Sigma^\star$, an argument $x \in \Sigma^\star$.
Question:	Does Turing machine $\mathcal{M}_e$ halt on input x?

Problem:	**Pure Halting**
Instance:	Index $e \in \Sigma^\star$.
Question:	Does Turing machine $\mathcal{M}_e$ halt on empty tape?

Undecidability

Theorem

The Halting Problem is undecidable.

Proof. Assume a TM $\mathcal{H}$ can decide Halting. Construct the following machine:

```
// machine $\mathcal{M}$
// on input $x$
if $\mathrm{halts}(x, x)$                         // use $\mathcal{H}$ here
then halt with output $0 \frown \mathcal{M}_x(x)$
else halt with output $0$
```

Then $\mathcal{M}$ has some index e and running $\mathcal{M}$ on e produces a contradiction.

$\square$

A Reduction

Claim: It follows that Pure Halting is also undecidable.

To see why, note that there is a primitive recursive function f such that, given index e and a string x, $e' = f(e, x)$ is the index of a machine that, starting on empty tape,

- first writes x on the tape, and
- simulates machine $\mathcal{M}_e$ on that tape.

It is important there that f is computable, existence of e' alone would not help.

Turing Computability and Robustness

So, we have a notion of a function being computable by a Turing machine that seems to conform very well to our intuition about computability..

This notion does **not change** if we modify our definitions slightly:

- one-way infinite tapes
- multiple tapes
- different head movements
- multiple heads
- different input/output conventions
- different coding conventions

Note that without this kind of robustness our model would be of rather dubious value: each variant would produce a different notion of computability.

Dire Warning

We have a perfectly good way to code a Turing machine $\mathcal{M}$ as, say, a sequence number $\langle\mathcal{M}\rangle$.

But the following "encoding" is strictly verboten.

$$\text{code}(\mathcal{M}) = \begin{cases} 2\langle\mathcal{M}\rangle & \text{if } \mathcal{M}(\mathbf{0})\downarrow \\ 2\langle\mathcal{M}\rangle + 1 & \text{otherwise.} \end{cases}$$

Exercises

Exercise

Show how to modify our definitions so that Turing machines have total transition functions, but produce the same class of computable functions.

Exercise

Prove that some of the modifications on the last slide similarly yield a type of machine that produces the same class of computable functions as our original Turing machines.

“Prove” here means: think about it for long enough so that you become convinced that an actual proof could be constructed if one really needed a detailed argument.

Rado's Problem

Tibor Radó

On Non-computable Functions

Bell System Technical Journal, 41(3):877-884, 1962.

Busy Beaver Problem

Radó described a deceptively simple problem in computability. Consider Turing machines on tape alphabet $\Gamma = \{0, 1\}$ and n states.

Question: What is the largest number of 1's any such machine can write on an initially blank tape, and then halt?

Incidentally, it is standard practice to ignore the halting state in the count, so n means "n ordinary states plus one halting state." Also, one insists that that the tape head always moves.

Other Variants

Rado's original question is somewhat arbitrary, here are two versions more firmly rooted in computability theory.

Time Complexity What is the largest number of moves a halting n-state machine can make?

Space Complexity What is the largest number of tape cells a halting n-state machine can use?

The "number-of-1s" question is below the space complexity, which is below the time complexity.

Notation

$\mathsf{BB}_{\mathrm{H}}(n)$	is the largest time complexity of any halting n-state machine and refer to BB_{H} as the Busy Beaver function.
$\mathsf{BB}_{\mathrm{W}}(n)$	is the largest number of 1's written on the tape when an n-state machine halts.

Clearly, $\mathsf{BB}_{\mathrm{H}}(n) \geq \mathsf{BB}_{\mathrm{W}}(n)$, but the former has the advantage of relating more directly to the Halting Problem, which one would suspect to be the central issue with busy beaver functions.

Busy Beaver $n = 1$

$$\mathsf{BB}_{\mathrm{H}}(1) = 1 \qquad \mathsf{BB}_{\mathrm{W}}(1) = 1$$

Make sure to give a precise proof.

Busy Beaver $n = 2$

$$\mathsf{BB}_{\mathrm{H}}(2) = 6 \qquad\qquad \mathsf{BB}_{\mathrm{W}}(2) = 4$$

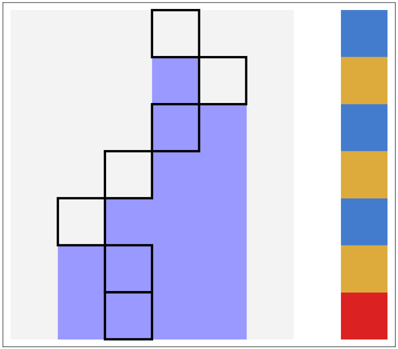
Amazingly, the answer is no longer obvious.

The champion for both measures is

	0	1
p	q,1,R	q,1,L
q	p,1,L	halt

$$p0 \vdash 1q0 \vdash p11 \vdash q011 \vdash p0111 \vdash 1q111$$

Orbit



Busy Beaver $n = 3$

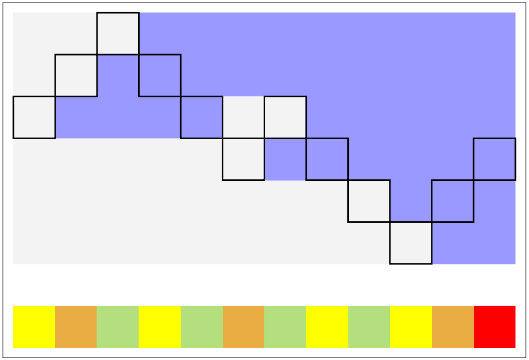
$$\mathsf{BB}_{\mathrm{H}}(3) = 21 \qquad\qquad \mathsf{BB}_{\mathrm{W}}(3) = 6$$

Here things start to get messy: there are $4\,826\,809$ Turing machines to consider. Exploiting isomorphisms, filtering out machines where all $3 + 1$ states are reachable we get down to $405\,072$.

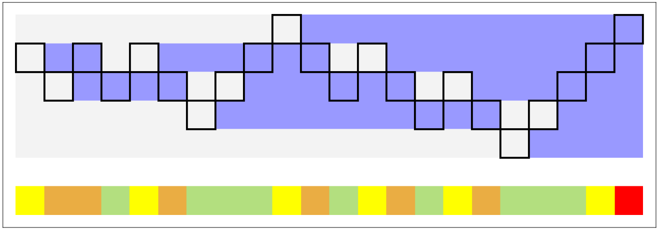
Now we just check those for halting and pick out the champion.

Minor Problem: Halting is undecidable and we have to handle each case by hand—somehow concoct an argument that shows that the machine must halt, or that it will not halt forever.

Write-Champion



Halt-Champion



How bad can it be?

The number of n-state machines grows exponentially:

$$(4n+1)^{2n}$$

This spins out of control very quickly:

n	#machines
1	25
2	6561
3	4,826,809
4	6,975,757,441
5	16,679,880,978,201
6	59,604,644,775,390,625

Isomorphs

These are the raw counts that ignore symmetries between the machines: permuting the states does not produce a “new” machine, nor does switching left and right moves.

Unfortunately, the real problem here is not isomorph-rejection (which requires constructing all machines first), but we need to find a way to only build non-isomorphic ones to begin with.

And, given these numbers, it won’t make much of dent no matter what.

Exercise

Try to count only non-isomorphic machines.
How would you go about generating non-isomorphic machines for $k = 5$*?*

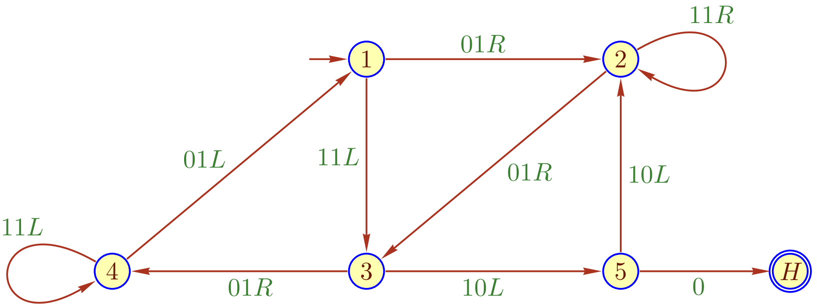
The Marxen-Buntrock Machine

The 5-state champion was found by Marxen and Buntrock in 1989, and its discovery is a small miracle. Here is the table of the machine. Clearly all 5 states plus the halt state are reachable in the diagram.

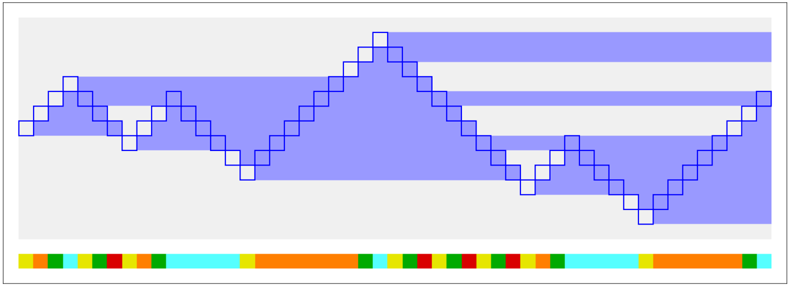
	0	1
1	2,1,R	3,1,L
2	3,1,R	2,1,R
3	4,1,R	5,0,L
4	1,1,L	4,1,L
5	halt	1,0,L

Of course, that's nowhere near enough: they need to appear in the computation on empty tape.

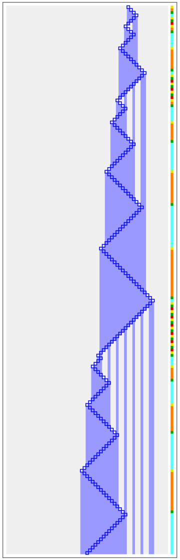
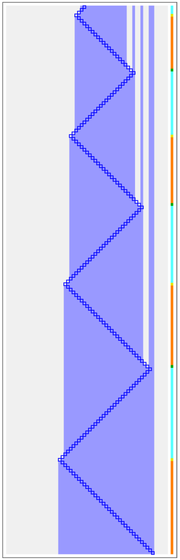
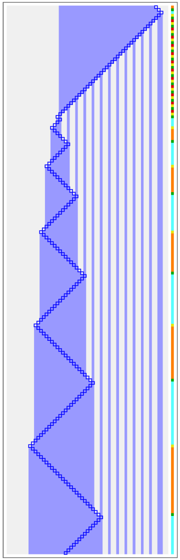
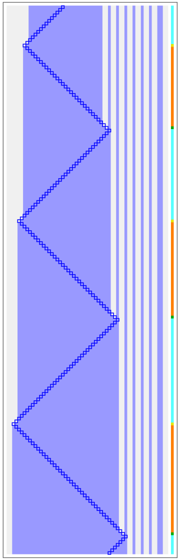
Diagram



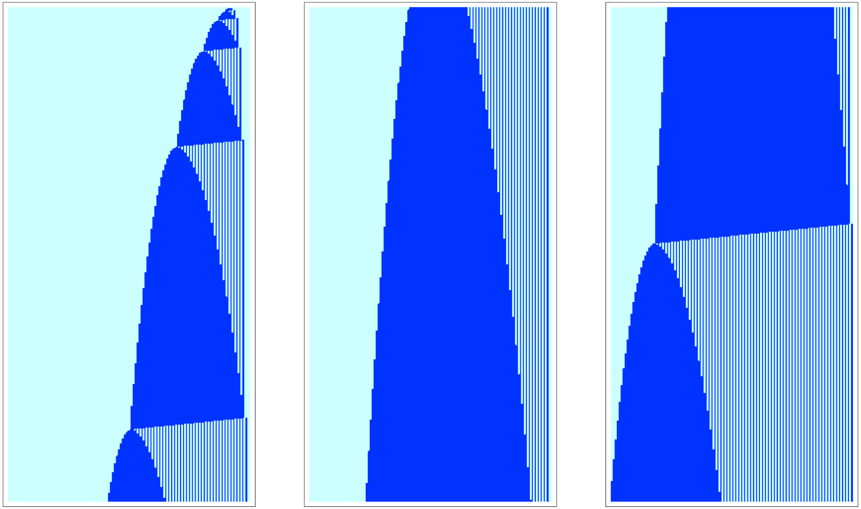
50 Steps



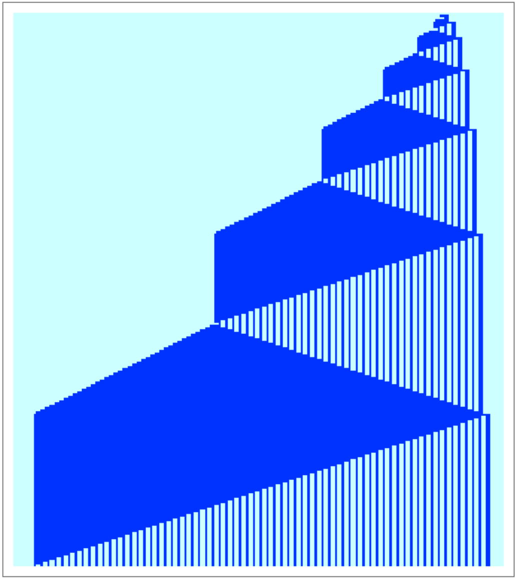
400 Steps

15000 Steps, Decimated by 10



State 1 Configurations, 18000 Steps



Misleading Pictures

Looking at a run of the Marxen-Buntrock machine for a few thousand steps one invariably becomes convinced that the machine never halts: the machine zig-zags back and forth, sometimes building solid blocks of 1's, sometimes a striped pattern.

Whatever the details, the machine seems to be in a “loop” (not a an easy concept to clarify for Turing machines). Bear in mind: there are only 5 states, there is no obvious method to code the instruction “do some zig-zag move 1 million times, then stop”.

And yet, this machine

stops after 47,176,870 steps

writes $10(100)^{4097}$ on the tape

Why on Earth?

With a bit of work, one can construe the computation of the MB machine as iterating an arithmetic function

$$f(x) = \begin{cases} (5x+18)/3 & \text{if } x \bmod 3 = 0 \\ (5x+22)/3 & \text{if } x \bmod 3 = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

Starting at 0, the orbit looks like so:

$$0, 6, 16, 34, 64, 114, 196, 334, 564, 946, 1584, 2646, 4416, 7366, 12284, \uparrow$$

Big Surprise:
The machine computes these values and then halts.

Quoi?

More precisely, we can filter out configurations of the form

$$C_n = \ldots 0 \boxed{1} 0 \underbrace{11\ldots11}_{n} 0 \ldots$$

n	steps	n	steps
0	15	564	180,307
6	73	946	504,027
16	277	1,584	1,403,967
34	907	2,646	3,906,393
64	2,757	4,416	10,861,903
114	7,957	7,366	30,196,527
196	22,777	12,284	24,576
334	64,407	halt	

For $n = 3k$, the step number is $5k^2 + 19k + 15$,
for $n = 3k + 1$ it is $5k^2 + 25k + 27$.

Why is this Hard?

- $\mathrm{BB_H}$ is not computable.
- $\mathrm{BB_H}$ has no computable upper bound.
- $\mathrm{BB_H}$ dominates all computable functions.

f dominates g if $\exists\, m\, \forall\, n \geq m\, (f(n) > g(n))$.

So busy beaver is all about insanely

The Obstructions

There are several fundamental difficulties in computing busy beaver numbers, even for annoyingly small numbers of states.

- Brute-force search quickly becomes infeasible, even for small k.
- The Halting Conundrum:
  Even if we could somehow deal with combinatorial explosion, we don't know if a machine will ever halt—it might just run forever.
- Reasoning about the behavior of Turing machines in a formal system like Dedekind-Peano arithmetic or even Zermelo-Fraenkel set theory is necessarily of limited use.

ForAll Statements

A Π_1 statement of arithmetic ϕ is a formula that can be written in the form

$$\phi \equiv \forall\, x\, R(x)$$

where $R(x)$ is primitive recursive [†]. The hard part in checking whether ϕ is true is that there are infinitely many $x \in \mathbb{N}$ to check.

There are quite few open problems in math that are Π_1.

- The Goldbach Conjecture is Π_1.
- The Riemann Hypothesis is also Π_1.

  This is much more surprising; unlike with Goldbach, it requires quite a bit of work to establish this claim.

[†]Written as a formula, it requires only bounded quantifiers and basic arithmetic.

Translation to Halting

Suppose we have some Π_1 statement $\phi \equiv \forall\, x\, R(x)$.

Given a specific natural number n, a Turing machine can easily verify that $R(n)$ holds.

Hence we can build a TM that loops through all values of n and checks $R(n)$. If it ever finds a counterexample, it halts; otherwise it runs forever.

So if we could check Halting, we could handle these Π_1 conjectures.

Worse

Allegedly, there is a Turing machine $\mathcal{G}$ on 27 states, binary alphabet, that checks the Goldbach conjecture in this way.

If we could figure out $\mathsf{BB}_{\mathrm{H}}(27)$ we could crack the Goldbach conjecture. In fact, even an upper bound would suffice.

This is a bit of a white lie, $\mathsf{BB}_{\mathrm{H}}(27)$ is a ridiculously large number. In this particular universe we could never even begin to run $\mathcal{G}$ for that many steps—but you get my drift.

And Reasoning?

Maybe there is a way to avoid all these pesky computational problems?

Instead of computing, we could try to find some clever proof in a sufficiently powerful formal system like Dedekind-Peano arithmetic or Zermelo-Fraenkel set theory[†].

This is most likely wishful thinking. It is currently known (more or less) that $\mathsf{BB}_{\mathrm{H}}(n)$ would provide answers to

- $n = 27$: Goldbach
- $n = 744$: Riemann
- $n = 748$: statements independent of ZFC

[†] Of course, a proof is just another computation.

State of the Art

n	$\mathsf{BB}_{\mathrm{H}}(n)$	$\mathsf{BB}_{\mathrm{W}}(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\mathbf{47\,176\,870}$	$\mathbf{4098}$
6	$> 10 \uparrow\uparrow 15$	$?$

Concrete values are only available for $n \leq 5$, beyond that, we only have bounds. And these bounds are ridiculously large, we have to use special notation to write down the bound even for $n = 6$ in a civilized manner.