Some halting problems for abelian sandpiles are undecidable in dimension three

Hannah Cairns

August 4, 2015

Abstract

The abelian sandpile model is a simple combinatorial model for critical behaviour, with the abelian property that the order in which we make moves does not change the final outcome of the game. This might seem to restrict the model’s computational ability, but we will show that, given three dimensions to work with, the sandpile is able to simulate a Turing machine. We then prove the undecidability of three halting problems.

Contents

1 Introduction 2
1.1 What is a sandpile? . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Three halting problems for sandpiles in \( \mathbb{Z}^3 \) . . . . . . . 3
1.3 Periodic sandpiles. . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Periodic+finite sandpiles. . . . . . . . . . . . . . . . . . . . . . 4
1.5 History of halting problems on sandpiles . . . . . . . . . . . . . 5

2 Sandpiles give you one-shot circuits 6
2.1 One-shot circuits. . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Sandpile wires. . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Toppling is like turning on. . . . . . . . . . . . . . . . . . . . . 6
2.4 Wait gates and AND and OR gates. . . . . . . . . . . . . . . . 7
2.5 Goles and Margenstern’s construction. . . . . . . . . . . . . . 9
2.6 Three vs. two dimensions and a conjecture based on results from complexity theory. . . . . . . . . . . . . . . . . . . . . . . . 9
2.6.1 What about more complicated planar networks? . . . . 10
2.6.2 A note: An easier proof for general abelian networks? . . 11
2.7 The course of the rest of the paper . . . . . . . . . . . . . . . 12

3 Building Turing machines from one-shots 12
3.1 Turing machines. . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Cellular automata. . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Simulating the cellular automaton with a sandpile. . . . . . . . 15
3.3.1 Storing the state. ........................................ 15
3.3.2 The evolution. ........................................ 15
3.4 Setting up the states at time zero. ......................... 16

4 Vertex prediction is undecidable ........................................ 17

5 Global halting: The lazy automaton ......................... 18
5.1 Lazy cellular automata. ..................................... 18
5.2 Implementation as a one-shot circuit. ..................... 19
  5.2.1 Storage ................................................. 19
  5.2.2 AND gates implement the partial rules ............... 19
  5.2.3 Relationship to the automaton ....................... 20
  5.2.4 Halting ................................................ 20

5.2.1 Storage ................................................. 19
5.2.2 AND gates implement the partial rules ............... 19
5.2.3 Relationship to the automaton ....................... 20
5.2.4 Halting ................................................ 20

5.2.1 Storage ................................................. 19
5.2.2 AND gates implement the partial rules ............... 19
5.2.3 Relationship to the automaton ....................... 20
5.2.4 Halting ................................................ 20

5.2.1 Storage ................................................. 19
5.2.2 AND gates implement the partial rules ............... 19
5.2.3 Relationship to the automaton ....................... 20
5.2.4 Halting ................................................ 20

6 A lazy automaton that simulates a Turing machine ........... 20
6.1 The set of states. .......................................... 20
6.2 Partial rules. ............................................. 21
  6.2.1 Rules out of sight of the wavefront ................. 21
  6.2.2 The wavefronts add blank tape ...................... 22
  6.2.3 Wave of laziness ....................................... 23
  6.2.4 The initial state ....................................... 23
6.3 This is a simulation of a Turing machine. ............... 23

7 Local halting is undecidable. ........................................ 24

A The process does not depend on choices ...................... 26

1 Introduction

1.1 What is a sandpile?

A sandpile is a discrete process on a graph \((V,E)\). We start with a certain number of chips on each vertex in \(V\). A vertex that has at least as many chips as its out-degree is unstable and we are allowed to topple it: we take that many chips off of that vertex and put one of them on each out-adjacent vertex.

We start with some initial configuration, topple some vertex that is unstable, and repeat. If at some point no vertex is unstable, the process is said to halt. We put no restriction on the choice of vertex to topple at each step, except that we do not allow an unstable vertex to be ignored forever.

Under that restriction, our choices of vertices do not affect whether or not the process halts, or even the number of times a particular vertex topples. This so-called ‘abelian property’ of the system is proved in Lemma A.1 in the appendix, following Björner, Lovász and Shor [3].

Does this abelian property put any restrictions on the computational power of the sandpile? To ask a more concrete question, is there a terminating algorithm that can tell whether a sandpile in \(Z^3\) will halt?
1.2 Three halting problems for sandpiles in $\mathbb{Z}^3$.

Can the behavior of a sandpile in $\mathbb{Z}^3$ be predicted by a Turing machine that always halts? We consider three such problems in this paper:

- **Vertex prediction.** Given a certain infinite sandpile in $\mathbb{Z}^3$, does the vertex at the origin ever topple?
- **Global halting prediction.** Are there infinitely many topplings overall?
- **Local halting prediction.** Are there infinitely many topplings at the origin?

If we are not careful how we encode the sandpile, these problems will be unsolvable because of a technical issue. A terminating algorithm can only read a finite amount of data before it decides to terminate.

Suppose we use a naive encoding where each square of the tape tells us the number of chips on one vertex. If we have a Turing machine which solves one of the problems, then run it on the sandpile where every vertex contains five chips. No vertex ever has six chips in it, so no vertex ever topples. So for this sandpile the answer to all of the problems is ‘no,’ and our algorithm must eventually halt at some finite time with a positive result.

However, the Turing machine only ran for a finite amount of time before halting, so it only looked at finitely many squares of its tape. That means it only looked at the number of chips on finitely many vertices.

If we add an extra chip to one of the unseen vertices, so that it now has six chips and is unstable, *every* vertex of the resulting sandpile topples infinitely often.\(^1\) So for this new sandpile, the answer to all three problems is ‘yes.’

Run the machine on this new sandpile. It must halt with the same answer, which is now wrong. So our algorithm must be defective. No terminating algorithm can solve the problem, so it is undecidable.

But the question was unfair: terminating Turing machines are not suited to solving problems with infinite input data. How can we modify this problem to make the input data finite and give the Turing machine a chance?

1.3 Periodic sandpiles.

One way to make the input data finite is to require the sandpile to be periodic. We say a sandpile is periodic if there are three periods $n_x, n_y, n_z \geq 1$ so that two vertices $(x, y, z)$ and $(x', y', z') \in \mathbb{Z}^3$ have the same number of chips whenever

$$x \equiv x' \mod n_x, \; y \equiv y' \mod n_y, \; z \equiv z' \mod n_z.$$ 

The number of chips on every other vertex is determined by periodicity.

If we give the periods and the number of chips on every vertex in the set

$$\{0 \leq x < n_x, 0 \leq y < n_y, 0 \leq z < n_z\} \subset \mathbb{Z}^3,$$

that determines the number of chips on every other vertex. That’s a finite amount of data, so it gives us hope that the problem is reasonable:

\(^1\)This follows easily from Lemma 7.3 by the method of proof of Lemma 7.1.
Periodic vertex prediction. Given a certain infinite periodic sandpile in \( \mathbb{Z}^3 \), does the vertex at the origin ever topple?

And the same question for global and local halting. Unfortunately, now the problems are all solvable, again for a simple reason. A periodic sandpile configuration with periods \( n_x, n_y, n_z \) will have the same behaviour as a sandpile on the finite torus \( (\mathbb{Z}/n_x\mathbb{Z}) \times (\mathbb{Z}/n_y\mathbb{Z}) \times (\mathbb{Z}/n_z\mathbb{Z}) \). There are only finitely many chips on the torus, so the system can be in only finitely many states.

Now we can solve the problems. First, if the sandpile is stable to start with, no topplings ever occur and the answer to all three problems is no. Otherwise, there are infinitely many unstable squares, and there are therefore infinitely many topplings overall. To solve the other two problems, we simulate the sandpile on the torus, keeping track of our history, until we either reach a stable state or come back to a state we were already in.

If we reach a stable state, the origin can only have toppled finitely many times, so the answer to the third question is no. If it hasn’t toppled yet, it will never topple, so we can answer the vertex prediction question.

If we reach a loop, at least one vertex in the torus has to topple infinitely many times, so by Lemma 7.2 the origin topples infinitely many times also.

To summarize, we run the sandpile on the torus until it stabilizes or loops, and then look at the following table:

|               | Vertex | Global | Local |
|---------------|--------|--------|-------|
| Stable to start with | N      | N      | N     |
| Unstable but stabilizes | If the origin fired | Y      | N     |
| Unstable and loops    | Y      | Y      | Y     |

and all three problems are decidable.

1.4 Periodic+finite sandpiles.

A periodic sandpile is too simple. A sandpile with a finite number of chips is also too simple: any such sandpile on \( \mathbb{Z}^3 \) will always terminate.

What if we start with a periodic sandpile and add a finite number of chips? This new system will not be periodic, so we can’t determine its behaviour by looking at a finite torus. And both the periodic sandpile and the locations of the finite number of chips can be specified with a finite amount of data, so we avoid the difficulty in section 1.2.

We call a periodic sandpile which has had finitely many chips added to it a periodic+finite sandpile, or p+f sandpile. We propose the same three problems.

\( P+F \) vertex prediction. Given a certain infinite periodic sandpile with finitely many chips added to it, does the vertex at the origin ever topple?

And the same for halting and local halting:

\( P+F \) global halting prediction. Given a p+f sandpile, are there infinitely many topplings overall?

\( P+F \) local halting prediction. Given a p+f sandpile, are there infinitely many topplings at the origin?
All three of these are again undecidable, but the reason is much more interesting. That is the main result of this paper.

A construction due to Moore and Nilsson [10] gives a way to build wires, AND gates, and OR gates in sandpiles. We combine these to build logical circuits, and use them to build a cellular automaton that simulates a Turing machine. Finally, we will take the automaton and adapt it in three different ways to address the three problems. The construction of a Turing machine out of logical circuits is long since well-known, and the main novelty of this paper is in the adaptations.

First, we will see how these sandpile circuits work, and then come back to the problems of vertex, halting, and local halting prediction.

1.5 History of halting problems on sandpiles

This seems to be the first paper that considers the undecidability of sandpile halting problems on an infinite graph, but the question of how to tell whether a sandpile on a finite graph halts, and how many topplings it takes for it to halt, has been widely considered.

The first step was a paper by Tardos in 1988 [12] showing that if we have a sandpile on a finite simple undirected graph with $n$ vertices, and it halts, then it topples only $O(n^4)$ times. In fact, a bound on the number of topplings is $n^4$, which is sharp up to a constant.\(^2\)

Björner, Lovasz, and Shor [3] gave a different polynomial bound on the number of topplings in 1991 based on the smallest eigenvalue of the adjacency matrix. In the same year, Eriksson [6] gave an example of a directed multigraph which halts after an exponential number of topplings. It may be worth mentioning that, if we know that a halting sandpile must halt in $N$ steps, then there is a simple $O(N)$ algorithm to check whether it halts:

**Step 1.** Go through all $n$ vertices and list the unstable ones.

**Step 2.** If the list is empty, then the answer is Yes, the sandpile halts. Otherwise, take an unstable vertex off the list and topple it. This might make some of the adjacent vertices unstable; put all the vertices that become unstable on the list.

**Step 3.** If Step 2 has been repeated $N + 1$ times, then the answer is No, the sandpile doesn’t halt. Otherwise return to Step 2.

So we know already that the halting problem is polynomial for simple undirected graphs. Björner and Lovasz [2] gave a more sophisticated algorithm in 1992, but that algorithm still takes exponential time in general.

That was the state of the art for some time, but a recent paper by Farrell and Levine [7] has shown that the sandpile halting problem is NP-complete for general finite directed multigraphs, and on the other hand that it can be solved in linear time for the special case of ‘coEulerian’ directed graphs.

---

\(^2\)More precisely, the bound is $2mnmd$, where $m \leq n^2/2$ is the number of edges and $d \leq n$ is the diameter of the graph.
2 Sandpiles give you one-shot circuits

2.1 One-shot circuits.

A monotone circuit is a logical circuit that uses only AND and OR gates, with no NOT gates. In this paper we are interested in monotone circuits with the extra property that wires cannot be turned off once they have been turned on. We call these one-shot circuits.

We will describe a method for building one-shot circuits in a sandpile on $\mathbb{Z}^3$, due to Moore and Nilsson [10]. We need to build wires, AND gates, and OR gates. We start with the wire.

2.2 Sandpile wires.

The sandpile ‘wire’ is built by taking a connected path of vertices in $\mathbb{Z}^3$ and putting five chips on each vertex.

A long straight wire looks like this:

\[
5 \begin{array}{c} \cdots \\
5 \end{array} 5 \begin{array}{c} \cdots \\
5 \end{array} 5 \begin{array}{c} \cdots \\
5 \end{array} 5 \begin{array}{c} \cdots \\
5 \end{array}
\]

If we put an extra chip on any vertex in the wire, it topples and puts a chip on all its neighbours, including the ones in the wire. Those already had five chips, so they topple also, and eventually the whole wire topples.

When we put a chip on the right-hand end of a wire, the sandpile evolves as shown in Figure 1.

![Figure 1: A wire is set off.](image)

This page is only two-dimensional, so we are only showing a single horizontal slice of $\mathbb{Z}^3$, but there are two more lines of ‘1’s above and below the page.

The toppling of one wire will not affect another wire if they aren’t adjacent, and we can add bends and three-way branches. We will later have reason to be careful about how many chips we add to vertices that aren’t in the wire, but as long as we leave some space between the bends and branches, the toppling of the wire will add at most two chips to any vertex.

2.3 Toppling is like turning on.

This is supposed to be a wire in a one-shot circuit, so we need a wire that turns on and never turns off. Sandpile wires don’t turn on forever, they fall over once.
But the wire in the one-shot circuit can only turn on once, and the sandpile wire can only topple once. We will treat toppling the wire like turning it on.

In a one-shot circuit, the AND gate waits for all its input wires to turn on, and then turns on the output wire. So, in the sandpile, a two-input AND gate should wait for both its input wires to topple, and then topple its output wire.

Similarly, the OR gate should wait for one of the input wires to topple, and topple its output wire. We will build such gates in the next two sections.

2.4 Wait gates and AND and OR gates.

We begin with the so-called wait-2 and wait-1 gates shown in Figure 2.

\[
\begin{array}{c|c}
\cdot \cdot \cdot 5 5 5 4 5 5 \cdot \cdot \cdot & \cdot \cdot \cdot 5 5 5 5 5 5 5 \cdot \\
5 & 5 \\
5 & 5 \\
\end{array}
\]

Figure 2: wait-2 and wait-1 gates.

The wait-2 gate waits for two wires to topple, and then topples the third wire. The wait-1 gate waits for one wire to topple, and then topples the other two. This isn’t what we want: the logical AND gate waits for its input wires to topple and then topples the output wire, without affecting the other input wire. Similarly, the OR gate waits for one input wire and then topples the output wire. We need some way to keep the gate from affecting the input wires.

This is accomplished by the diode below.

\[
\begin{array}{c}
5 5 \\
\cdot \cdot \cdot 5 5 5 4 5 5 \cdot \cdot \cdot \\
\end{array}
\quad
\begin{array}{c}
11 \\
1 2 1 2 1 \\
\cdot \cdot \cdot 1 1 2 1 1 1 1 \cdot \\
1 1 1 1 1 \\
\end{array}
\]

If the left wire topples \( \rightarrow \)

\[
\begin{array}{c}
5 5 1 1 \\
\cdot \cdot \cdot 5 5 5 5 0 1 \cdot \\
\end{array} \quad
\begin{array}{c}
1 1 \\
\cdot \cdot \cdot 1 1 1 1 \\
\end{array}
\]

but if the right wire topples \( \rightarrow \)

Figure 3: Diode

If the left part of the diode topples, the vertex with 4 chips will receive two more chips and will topple. That will topple the right part of the diode.

On the other hand, if the right part of the diode topples, the vertex with 4 chips will only receive one chip and will not topple, leaving the left side of the diode untouched.

So signals pass from left to right through the diode, but not the other way around. If we attach two diodes to the input wires of a wait-2 gate, as in Figure 4, it will turn it into an AND gate.
If the left or right input wire topples, then one chip will be added to the underlined 4 vertex. If both input wires topple, the vertex with 4 chips in the middle will get two chips and topple the bottom output wire.

On the other hand, if two of the other wires topple — say, if the left and bottom wires topple — the central area will topple, but the resulting chain reaction will be absorbed by the business end of the right-hand diode and will not set off the right wire. This is shown in Figure 5.

So this gate operates like a two-input AND gate. We could put a diode on the output wire too, but that is unnecessary, since it would only prevent the output wire from setting off the input wires, and the input diodes do that already.

Similarly, if we attach two diodes to a WAIT-1 gate, it will behave like an OR gate. See Figure 6.

The right side of the figure shows what happens after we topple the left wire. The two 4 vertices are underlined. The toppling traveled through the left diode and set off the bottom wire. The 4 square in the diode on the right received one chip, but that wasn’t enough to make it topple.
This gate waits for one of its two input wires to topple, and then topples its output wire. It can’t topple either input wire because of the diodes. That makes it an OR gate.

Having built two-input gates, we can construct AND and OR gates with arbitrarily many inputs by chaining together two-input gates as usual.

### 2.5 Goles and Margenstern’s construction.

We note here that Bitar, Goles, and Margenstern used a different design for their circuits in [1, 9].

Aside from the fact that their circuits are defined on trees rather than on a lattice, their design has a serious flaw from our point of view. All of the problems we propose in this paper are questions about which vertices are toppled and how many times it happens.

Their design encodes bits in the relative timing of two signals traveling through two parallel wires, but by the abelian property, we can delay any toppling for as long as we want without changing that set, so any bit can be flipped by delaying the toppling of the leading signal until the trailing signal passes it.

In other words, the answers to our problems cannot depend on the bits traveling through the computer! This makes it very difficult to use this design to show undecidability.

### 2.6 Three vs. two dimensions and a conjecture based on results from complexity theory.

We are in three dimensions, so there is no planarity obstruction: given sufficient space, we can connect gates together in any arrangement we like. This makes it possible to construct a sandpile corresponding to any one-shot circuit, in the sense that a sandpile ‘wire’ topples if and only if the corresponding wire in the circuit turns on.

In two dimensions, there is no obvious way to run sandpile wires through each other without interference, and this seems to impose a restriction of planarity.

Gajardo and Goles [8] prove some version of this planarity restriction. They show that there is no stable configuration with four wires coming in, say from the north, east, south, and west, with the property that the north wire can topple
the south wire without toppling the east one and the west wire can topple the east wire without topple the south one.

Planarity is known to change the complexity behaviour of monotone circuits, assuming some plausible conjectures about complexity classes. The evaluation of finite monotone circuits is P-complete, while on the other hand fast parallel algorithms are known for the evaluation of finite planar monotone circuits.

Yang [13] shows that the evaluation of planar monotone circuits is in NC³, and on the practical side, there is an algorithm of Ramachandran and Yang [11] which runs in O(\log³ n) time on a linear number of parallel processors. So, under the typical assumption that P ≠ NC, there is a qualitative difference between three and two dimensions.

We conjecture that there is also a difference in kind for infinite sandpiles:

**Conjecture.** The halting problem for periodic planar sandpiles with finitely many chips added or removed is decidable.

We want to generalize this a little. Following Bond and Levine [4], we say that a unary network is a message-passing network with only one type of message, where each node is a finite state machine. The sandpile is a special case of a unary network: each node waits to get four messages and then sends one message to each of its neighbours. A unary network on a planar graph seems to be subject to the same kind of planarity restriction as a sandpile, so we propose the more general conjecture:

**Conjecture.** The halting problem for a periodic unary network on \( \mathbb{Z}^2 \) with finitely many messages added is decidable.

That is, we conjecture that the planarity of the network restricts the system enough that its behaviour becomes predictable.

### 2.6.1 What about more complicated planar networks?

Can we make this conjecture in more generality? For example, what if we give the message-passing network two different types of messages instead of just one?

No. All our gates can be constructed in two dimensions, so the only obstacle is getting wires to cross each other. It turns out that two messages let us build wires that can cross and get around the planarity. This is true even when the network is restricted to the class of abelian networks (as in [4]).

Start with the two-dimensional sandpile on \( \mathbb{Z}^2 \) and change some of the vertices to crossovers. They have two counters, one for chips that come in vertically and one for chips that come in horizontally. When the vertical counter reaches two, it sends one chip north and one chip south. Similarly, when the horizontal counter reaches two, it sends one chip west and one chip east. The remaining vertices behave as normal: they have one counter, and when it reaches four, they send one chip in each direction. This is a simple system, and it has the abelian property too.

With this setup, signals can cross over each other without interfering, as in Figure 7. This allows us to run wires wherever we like without a planarity
obstruction. The gates described in section 2 all work fine in two dimensions$^3$, so our Turing machine constructions in later sections carry over to this situation, and vertex and global halting are undecidable for general two-message abelian networks on $\mathbb{Z}^2$, let alone more complicated ones.$^4$

Here is another way to reach this conclusion. In our construction, the graph $\mathbb{Z}^3$ is broken up into a lattice of large $N \times N \times N$ cubes, and the only connections are horizontal: no cube is ever connected to a cube above or below. The sandpile will behave the same if we delete all the vertices outside $\mathbb{Z}^2 \times [0, N]$.

So we can think of each stack $\{x\} \times \{y\} \times [0, N]$ as a single node in a message-passing network on the graph $\mathbb{Z}^2$. The abelian property is not injured by this grouping procedure, so the resulting network is abelian, and it’s periodic except for a state change at finitely many vertices.

This network behaves identically to the sandpile. An algorithm to predict halting on this class of networks would allow us to predict the halting of the sandpile, so there is no such algorithm, and again we see that vertex and global halting are undecidable for general abelian networks on $\mathbb{Z}^2$.$^5$

### 2.6.2 A note: An easier proof for general abelian networks?

In the previous section, we see that the halting problem for a general periodic+finite abelian networks on $\mathbb{Z}^2$ is undecidable. But this conclusion relies on the multilayered construction in the rest of the paper. Isn’t there some more simple construction that we can use in the more general case?

Well, yes. Not all of the details are necessary. But there is a hidden simplicity in our method of proof: a sandpile is automatically an abelian network! It is easy to write down a message-passing network that simulates a cellular automaton with just one processor per square per unit of time, but arranging things so

---

$^3$Although the numbers have to be changed: 5 and 4 become 3 and 2.

$^4$The proof for local halting doesn’t carry over, since the ‘bomb’ gadget doesn’t work in two dimensions: it is either not sensitive enough to explode, or so sensitive that a wire with a bend in it will set it off. Instead, we can notice that the normal sites are not using the second message and give them some explosive behaviour when they receive it: say, they send one copy of the second message in all directions.

$^5$I am not sure what happens on a line or half-line.
The sandpile operates like

\[ \text{sec. 2.1} \]

a one-shot circuit, which simulates

\[ \text{sec. 3} \]

a cellular automaton
(vertically and locally halting)
which simulates

\[ \text{sec. 3, 5} \]

a lazy cellular automaton
(global halting)
which simulates

\[ \text{sec. 3, 6} \]

a Turing machine

Figure 8: The layers of the paper.

that the processors in that network are locally abelian (in the sense of [4]) is a significant technical inconvenience that is not all that enlightening.

2.7 The course of the rest of the paper

Having described Moore and Nilsson’s construction of sandpile circuits and made some grand conjectures based on the work of Yang, we now get around to the actual content of this paper, three relatively prosaic proofs that the three problems earlier are undecidable.

The structure of the paper is multi-layered, and is summarized in Figure 8. We have already seen how to make one-shot circuits in sandpiles in section 2.1, which covers the top-left arrow. For vertex and local halting, we need the other two arrows on the left: a one-shot circuit simulates a cellular automaton, and a cellular automaton simulates a Turing machine.

For global halting, the approach with normal cellular automata doesn’t work, and we introduce the idea of a lazy automaton in section 5 and connect it to the Turing machine in section 6. This gives us the diagonal arrows.

Since the lazy automaton is a variation on the usual cellular automaton, we will explain the left-hand chain of arrows next, in section 3.

3 Building Turing machines from one-shots

For all three problems, we will show that the problem is undecidable by reducing it to the halting problem for Turing machines using some sort of special gadget.

The first step in all these problems is building a periodic one-shot circuit that simulates a Turing machine. There is no novelty in this part: a computer scientist could do this blindfolded. But there is no harm in a detailed description.

We start by recalling what a Turing machine is.
3.1 Turing machines.

A Turing machine is a six-tuple \((Q, q_0, F, \Gamma, \beta, \delta)\).

- It has a set of states \(Q\), one of which is the starting state \(q_0\). Some subset of these states are final states \(F \subseteq Q\): if the machine enters any of these states, it halts.
- It has an infinite one-dimensional tape with one square for every \(n \in \mathbb{Z}\), and each square has one letter written on it from the tape alphabet \(\Gamma\). One of the letters is the blank letter, \(\beta\).
- It has a tape read/write head that starts on square zero.
- Finally, it has a transition function \(\delta: (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{\text{left}, \text{right}\}\).

The machine takes as input a tape, which starts in some initial state \(T: \mathbb{Z} \to \Gamma\) with all but finitely many squares blank.

At every time step, the machine looks at its state and the letter on the square of the tape that the head is on. If the state is final, the machine halts.

Otherwise, let \(q\) be the current state, and let \(\gamma\) be the letter on the tape at the current position of the head. Apply the transition function \(\delta\) to \((q, \gamma)\). This gives us a three-tuple \((q', \gamma', d)\) \(\in Q \times \Gamma \times \{\text{left}, \text{right}\}\). The machine changes to the new state \(q'\), writes the letter \(\gamma'\) on the tape at the current position, and moves one square left or right depending on the direction \(d\). On the new square, it looks at the new state and letter, moves again, and so on.

Since the head moves exactly one square per time step, to find out what is happening on a square at the current time step, we only need to look at the three adjacent squares at the previous time step.

The head might be moving left or right onto the square we are looking at, or it might be changing the letter on our square and moving off to the left or right; on the other hand, it might not be visible at all, or it might be on an adjacent square but moving away from us.

We can simulate this behaviour with a cellular automaton.

3.2 Cellular automata.

A cellular automaton on \(\mathbb{Z}\) has a set of states \(S\), a neighbourhood size \(r\), and a transition function \(f: S^{2r+1} \to S\). At a given time \(t\), each square \(x \in \mathbb{Z}\) is in some state \(s(x, t) \in S\). The state of a square at time \(t > 0\) is determined by the states of the neighbouring squares at time \(t - 1\):

\[
s(x, t) = f(s(x - r, t - 1), s(x - r + 1, t - 1), \ldots, s(x + r, t - 1)).
\]

We give the automaton some initial state \(S: \mathbb{Z} \to S\) at \(t = 0\), and the states at every other time are determined inductively by the transition function.

\[
s(x, 0) = S(x) \quad s(x, 1) = f(S(x - r), \ldots, S(x + r)) \quad \ldots
\]

To simulate a Turing machine \((Q, q_0, F, \Gamma, \beta, \delta)\), we will use a cellular automaton with neighbourhood size 1, and three kinds of states:
• One state \( t(\gamma) \) for every \( \gamma \in \Gamma \), which means:

The head of the Turing machine is not on this square, and the tape has the letter \( \gamma \) written on it.

• One state \( h(q, \gamma) \) for every pair \( q \in Q, \gamma \in \Gamma \), which means:

The head of the Turing machine is on this square, in state \( q \). The head is in state \( q \), and the tape has \( \gamma \) written on it.

• \( e \): error state.

If \( T : \mathbb{Z} \rightarrow \Gamma \) is the initial state of the Turing machine, then the initial state of the cellular automaton will be

\[
S(x) = t(T(x)) \quad \text{if} \quad x \neq 0 \quad S(0) = h(q_0, T(0)).
\]

That is, we copy the tape letters from the Turing machine’s initial state \( T \), but then we put a head on square zero and start it in state \( q_0 \).

\[
S(x) = \ldots \ t(T(-1)) \ h(q_0, T(0)) \ t(T(1)) \ t(T(2)) \ \ldots
\]

We choose the transition function of the cellular automaton as follows.

If the current square is in one of the states \( h(q, \gamma) \), so that the head is on that square, it is going to write a letter to it and move off at the next step. Let the output of the transition function be \( \delta(q, \gamma) = (q', \gamma', d) \). As the head left the square, it wrote the letter \( \gamma' \) on the tape, so the new state should be \( t(\gamma') \):}

\[
f(t(\gamma_1), h(q, \gamma), t(\gamma_3)) = t(\gamma')
\]

for all \( q \in Q, \gamma_1, \gamma, \gamma_3 \in \Gamma \).

If a square is in state \( t(\gamma) \), then the head is not on that square. In that case, there are three possibilities. The head could be on a square to the immediate right or left, or it could be nowhere in sight.

First, the head might be on the square to the right, threatening to move left. If the square to the right is in state \( h(q, \gamma) \), and the output of the transition function is \( \delta(q, \gamma) = (q', \gamma', d) \) with \( d = L \), the head is going to move left onto our square, and change to state \( q' \). If the direction is \( d = R \), the head will move right, and nothing will happen to our square at all. So:

\[
f(t(\gamma_1), t(\gamma_2), h(q, \gamma)) = \begin{cases} 
  h(q', \gamma_2) & \text{if } d = L \\
  t(\gamma_2) & \text{if } d = R
\end{cases}
\]

for all \( q \in Q, \gamma, \gamma_1, \gamma_2 \in \Gamma \).

Second, the head might be on the square to the left. If \( d = R \), the head will move right onto our square; if \( d = L \), nothing will change.

\[
f(h(q, \gamma), t(\gamma_2), t(\gamma_3)) = \begin{cases} 
  t(\gamma_2) & \text{if } d = L \\
  h(q', \gamma_2) & \text{if } d = R
\end{cases}
\]
for all \( q \in Q, \gamma, \gamma_2, \gamma_3 \in \Gamma \).

Third, if the head is nowhere to be seen, nothing changes:

\[
f(t(\gamma_1), t(\gamma_2), t(\gamma_3)) = t(\gamma_2)
\]

for \( \gamma_1, \gamma_2, \gamma_3 \in \Gamma \).

This covers every possibility with zero or one heads visible. The process doesn’t add extra heads, so as long as we start with no errors and with just one head, we will never need to know any other values of the function. But we have to give the automaton some behaviour in the other cases also. If there are two or three heads in our neighbourhood, or any square is in the error state, we set the new state to \( e \). We will now explain how to use the one-shot circuits we built in section 2.1 to simulate this cellular automaton.

3.3 Simulating the cellular automaton with a sandpile.

We construct a sandpile that simulates the cellular automaton.

The idea is essentially due to Goles and Margenstern [9] except that we are using Moore and Nilsson’s circuit designs. Choose some large unspecified period \( n \), and use it to slice \( \mathbb{Z}^3 \) up into cubes \( c(x, y, t) \) labeled by their position:

\[
c(x, y, t) = [nx, (n + 1)x - 1] \times [ny, (n + 1)y - 1] \times [nt, (n + 1)t - 1] \subset \mathbb{Z}^3
\]

Here \( t \) represents time. The cube at \( c(x, 0, t) \) will hold the state of the automaton on square \( x \) at time \( t \).

We will only be considering the cubes with \( y = 0 \), so we fix \( y = 0 \) and drop it from the notation: \( c(x, t) = c(x, 0, t) \).

3.3.1 Storing the state.

We need to store the state of the cube. We can code the states in \( S \) as binary numbers using \( b = \lceil \log_2(|S|) \rceil \) bits, so from now on we will think of a state in \( S \) as a \( b \)-bit binary number. Now all we need is a way to store a single bit.

Let a cell be a pair of two wires, a zero wire and a one wire. If the zero wire is toppled the cell is storing zero, and if the 1 wire is toppled the cell is storing one. If neither is toppled, the cell is not storing anything: it’s in a neutral state.

We can set a neutral cell to zero or one by toppling the corresponding wire. Add \( b \) cells to the cube \( c(x, t) \). We’ll use them to store \( s(x, t) \).

3.3.2 The evolution.

Second, we need to simulate the evolution of the automaton. There is a certain transition function \( f : S^{2r+1} \to S \). The state of a square is determined by the states of the squares in a neighbourhood at the previous time:

\[
s(x, t) = f(s(x - r, t - 1), s(x - r + 1, t - 1), \ldots, s(x + r, t - 1))
\]

We need to replicate this behaviour with a circuit. Here is how we do that.
Put one AND gate in the cube \( c(x, t) \) for each tuple \( s \in S^{2r+1} \). Call it \( g_s(x, t) \). It will have \( b \times (2r + 1) \) inputs, which are set up as follows:

for \( j = 1, \ldots, b, i = -r, \ldots, r \):
- if the \( j \)-th bit of \( s_i \) is 1:
  - connect an input to the 1 wire of the \( j \)-th cell in \( c(x + i, t - 1) \);
- otherwise, connect an input to the 0 wire of that cell.

It has \( b \) outputs, which are connected to the \( b \) storage cells in \( c(x, t) \):

- if the \( j \)-th bit of the new state \( f(s) \) is 1:
  - connect the output of the gate to the 1 wire of the \( j \)-th cell in \( c(x, t) \);
- otherwise, connect it to the 0 wire of that cell.

The AND gate \( g_s(x, t) \) topples if the cells in \( c(x - r, t - 1), \ldots, c(x + r, t - 1) \) have been set to \( s = (s_{-r}, \ldots, s_r) \) respectively. Once it does, it sets the cells in \( c(x, t) \) to \( f(s) \). And there’s one AND gate for each possible tuple. So, once the cells at the previous time are all set, exactly one AND gate will topple and set the cells in \( c(x, t) \) to the appropriate new state.

If the cells at time \( t - 1 \) are set to the state of the automaton at time \( t - 1 \), the AND gates will compute the state of the automaton at time \( t \) and store it in the cells at time \( t \). This will simulate the cellular automaton process, as long as the states are set correctly at time zero. That presents its own problems.

### 3.4 Setting up the states at time zero.

The storage cells all start unset. We are allowed to add a finite number of chips, with which we can set the states of finitely many cubes, but we can’t set all infinitely many cubes \( c(x, 0) \) this way.

The initial state of the Turing machine has finitely many non-blank squares, which we can set with finitely many chips. All the other squares are blank.

We need a way to set infinitely many squares we don’t set directly to the state \( t(\beta) \) corresponding to blank tape. We add a wire along the \( x \)-axis which is broken up by diodes, as in Figure 9.

![Figure 9: An infinite wire broken up by diodes. The vertical lines show the divisions between cubes.](image)

Putting a chip on the underlined square in one cube topples the wire in that cube and every wire to its right, without affecting any of the cubes to its left. Each cube has a wire going downward. We connect that to the storage cells in that cube via diodes, so that it will set the cells to the bit pattern \( t(b) \) when it topples.
Putting a chip on this wire in the cube $c(x, 0)$ sets its state and the states of all the cubes to its right to $t(b)$ without affecting any of the other cubes. This lets us set a half-infinite set of cubes to the blank state $t(\beta)$ by placing just one chip on the wire. This, and another similar circuit going to the left instead of to the right, lets us set up any initial condition with only finitely many chips.

It is worth pointing out here that this circuit alone will topple infinitely many times overall, so it won’t be very useful for the global halting problem. We will have to do something else for that.

### 4 Vertex prediction is undecidable

The first problem in our list is not a halting problem:

P+F vertex prediction. Given a certain infinite periodic sandpile with finitely many chips added to it, does the vertex at the origin ever topple?

**Theorem 4.1.** The p+f vertex prediction problem is undecidable.

**Proof.** We will show that the halting problem for Turing machines can be reduced to vertex prediction: if we are able to solve this problem for any input, we can determine whether an arbitrary Turing machine halts.

In the previous section, we managed to build a periodic+finite circuit that simulates a Turing machine given a certain input. All we need to do is add a little extra circuitry to make the origin topple when the Turing machine reaches a final state.

Put an alarm wire in every cube $c(x, t)$. Connect it to the alarm wires in all adjacent cubes. Put one AND gate in every cube for every $f \in F$ and $g \in \Gamma$. Connect the inputs to the storage cells so that the gate will fire if and only if the cube is in state $h(f, \gamma)$. Connect all the outputs to the alarm wire in that cube. Finally, translate the whole sandpile so that one alarm wire goes through the origin.

Suppose the Turing machine being simulated enters a final state $f \in F$. It does so at some time step $t \geq 0$ with the head in some position $x \in \mathbb{Z}$.

The circuit will simulate the Turing machine, and eventually the storage cells in the cube $c(x, t)$ will be set to a bit pattern corresponding to some state $h(f, \gamma)$.

If the Turing machine being simulated ever enters a final state $f \in F$, one of the cubes in the automaton will enter a state $h(f, \gamma)$, and it will set off the alarm in some cube. That will topple the neighbouring alarm wires and so on. Eventually the wire connected to the origin will topple, and the answer will be yes. If not, the alarm wires will never go off, and the answer will be no.

So the vertex prediction problem “does the origin ever topple?” for this sandpile is equivalent to asking whether the Turing machine being simulated will halts.

The input Turing machine is arbitrary, and it is well known that the question of whether an arbitrary Turing machine halts is undecidable: there is no terminating algorithm that can solve the problem in all cases. So p+f vertex prediction is also undecidable.
5 Global halting: The lazy automaton

The second problem in our list is:

P+F GLOBAL HALTING PREDICTION. Given a periodic sandpile with finitely many chips added, are there infinitely many topplings in total?

To reduce Turing halting to global halting prediction, we need a circuit that only does a finite amount of toppling when the Turing machine halts. Again, the strategy is to hook up an infinite periodic circuit that simulates a cellular automaton. Unfortunately, our earlier circuit doesn’t work here, because it has to topple infinitely many vertices to move one step forward in time.

We will introduce the idea of a lazy cellular automaton. We will describe how to implement a lazy automaton as a circuit, and then write down a lazy automaton that works like a Turing machine and halts when it halts.

5.1 Lazy cellular automata.

A lazy cellular automaton on the vertex set $\mathbb{Z}$ with radius $r$ has a set of states $S$ and an extra lazy state $\lambda$ which is not in $S$.

The automaton has an initial state $\Sigma_0 : \mathbb{Z} \to S \cup \{\lambda\}$ and evolves with time. It is possible for it to malfunction, but if it hasn’t malfunctioned yet, every vertex is either in one of the states in $S$ or in the lazy state.

For a regular cellular automaton, the state of a vertex at time $t$ is uniquely determined by the states of neighbouring vertices at time $t-1$. This is not true for a lazy cellular automaton; instead it is equipped with a set of partial rules $R$, which may not cover every case or may even be contradictory.

A partial rule is a pair $\pi \to s$, where $s$ is a state in $S$ and $\pi$ is a pattern: a string of length $2r+1$ with letters in $S \cup \{*\}$, where $*$ is a special wildcard symbol. There is one exception: $(*, \ldots, *)$ is not a legal pattern.

A pattern $\pi = (\pi_{-r}, \ldots, \pi_r)$ matches at time $t$ at $x$ if, for all $k = -r, \ldots, r$:

- either $\pi_k$ is the wildcard,
- or the state of the vertex $x$ at $t$ is $\pi_k$.

The automaton is lazy in two ways: first, if the partial rule has wildcards in it, the automaton won’t bother to look at the state of those vertices; second, sometimes no partial rule applies, in which case that vertex gives up and enters the lazy state. Notice that the lazy state $\lambda$ cannot appear in a pattern, so the only way that a pattern can match $\lambda$ is by having a wildcard in that position.

The evolution of the automaton works as follows. For $x \in \mathbb{Z}, t \geq 0$, let

$$R_{x,t} = \{ \text{rules } \pi \to s \in R \text{ for which the pattern } \pi \text{ matches at time } t-1 \text{ at } x \}.$$ 

If there is exactly one rule in $R_{x,t}$, then the new state at $x$ is $s$. If there are none, the new state is the lazy state $\lambda$. If there are two or more, the whole automaton malfunctions starting at time $t$ and we say nothing more about its behavior. The automaton is said to halt if it never malfunctions and there is some time $t \geq 0$ when every square is lazy. We will hook up a one-shot circuit that works like this.
5.2 Implementation as a one-shot circuit.

We start by cutting $\mathbb{Z}^3$ up into large periodic cubes, which we label by their position $(x, y, t)$. The circuit must be periodic in three dimensions, but we only need one dimension of space and one of time, so we will only add connections between cubes with the same $y$-value.

We will add extra chips only to the cubes on the line $t = y = 0$. Our circuit does not start out in a toppling configuration, and no other circuit will have chips added to it, so nothing will happen outside of the cubes in the $y = 0$ plane. From now on we will suppress $y$ in the notation, writing cube positions as $(x, t)$.

5.2.1 Storage

Each cube has to store its current state. Let $S$ be some set of states. Assign a distinct bit string $s_1 \cdots s_b$ to every element $s \in S$, where $b = \lceil \log_2(|S|) \rceil$ is the number of bits required to do that.

We want to store one of these bit strings in each square. To do this, we put $2b$ state wires in every square of the lattice. We will call them $w_m(x, t)$ and $\overline{w}_m(x, t)$.

Initially, neither wire is toppled. In this situation, we say the bit $m$ is lazy. If $w_m$ is toppled, it means that bit $m$ of the state is 1; if $\overline{w}_m$ is toppled it is zero. There is no way for both bits to topple unless the automaton has malfunctioned, as we will see once the description is over.

We can then read the bit string of the state of a square off the state wires, with the proviso that if any of the bits are lazy, we will say the state is lazy.

5.2.2 AND gates implement the partial rules

We put a multi-way AND gate $g_\rho(x, t)$ for every cube $(x, t)$ and every partial rule $\rho = (\pi_{-r}, \ldots, \pi_r) \rightarrow s$.

The inputs of the gate are connected to the state wires of neighbouring squares at the previous time in a way that depends on the pattern $\pi$. For $k = -r, \ldots, r$:

- If $\pi_k = \ast$, don’t connect anything.
- If $\pi_k \in S$, write $\pi_k$ as a bit vector $\pi_{k_1} \cdots \pi_{k_m}$, and then:
  
  For $m = 1, \ldots, b$,
  
  if $\pi_{km} = 1$, connect an input to $w_{m}(x + k, t - 1)$
  
  otherwise, connect an input to $\overline{w}_{m}(x + k, t - 1)$.

  The gate has one output, which is connected to one wire from each of the $b$ pairs of state wires in the current square. Write $s = s_1 \cdots s_b$. For $m = 1, \ldots, b$:

  connect the output to $w_m(x, t)$ if $s_m = 1$, or
connect the output to \( \pi_m(x,t) \) if \( s_m = 0 \).

The ultimate result of all this is that each AND gate \( g_\rho(x,t) \) fires if and only if \( \pi \) matches at time \( t - 1 \) at \( x \). When it fires, it sets one bit in each of the pairs of state wires so that they will say the current state is \( s \).

5.2.3 Relationship to the automaton

We claim that this circuit works just like a lazy cellular automaton, unless the automaton malfunctions. The AND gate \( g_\rho(x,t) \) fires if and only if the partial rule \( \rho \) has a pattern that matches the states encoded in the state wires. If none of the gates fire, none of the state bits will ever be set for that cube, so its state will be \( \lambda \).

So as long as at most one AND gate in every cube fires, the behavior of this circuit will match the behavior of the lazy cellular automaton. If two or more gates in the same cube fire, there is a good chance that both wires \( w_m, \bar{w}_m \) in a pair will be toppled and the circuit will stop working correctly. But this cannot happen unless the automaton malfunctions.

5.2.4 Halting

We now come to the reason why we are using a lazy automaton. Each activated rule sets the state of exactly one cube, so:

**Lemma.** If there is no malfunction, the automaton halts if and only if finitely many AND gates in the corresponding circuit fire.

When we implement the one-shot circuit as a sandpile, the sandpile halts if and only if finitely many gates in the circuit ever fire. Therefore, we have the corollary:

**Corollary.** If there is no malfunction, the lazy automaton halts if and only if the corresponding sandpile halts.

The next step is to come up with a lazy automaton that simulates a Turing machine, complete with identical halting behaviour.

6 A lazy automaton that simulates a Turing machine

6.1 The set of states.

The set of states \( S \) for the lazy automaton will be broken into four classes:

- tape states \( t(\gamma) \) for \( \gamma \in \Gamma \): the tape has \( \gamma \) written on it, and the head is not on this square;
- head states \( h(q,\gamma) \) for \( q \in Q, \gamma \in \Gamma \): the tape has \( \gamma \) written on it, and the head is on this square and is in state \( q \);
• ◀: the left wavefront of the tape;
• ▶: the right wavefront of the tape.

We set \( r \), the radius of the automaton’s visible neighbourhood, to 2. That is, the five squares \( x-2, \ldots, x+2 \) will be visible from \( x \).

We use the two special ‘wavefront’ states to create more blank tape squares to the left and right as the automaton operates.

### 6.2 Partial rules.

We give the partial rules of the lazy automaton in the rest of this section. Because \( r \) is 2, a partial rule is a pair in \((S \cup \{\ast\})^5 \times S\). We will write these as

\[(a, h, \underline{c}, d, e) \rightarrow f\]

where the underline is there to remind us which square is getting the new state.

We recall that the automaton evolves as follows: if there is a partial rule \( \pi \rightarrow s \) whose pattern matches the states around position \( x \) at a certain time step, then the new state at \( x \) will be \( s \). If no partial rule matches, the state will be \( \lambda \).

If two or more rules match, the automaton will malfunction, but this will not occur with the rules below and the given initial state.

#### 6.2.1 Rules out of sight of the wavefront

The following partial rules cover the cases where all five states in the visible neighbourhood are tape or head states.

**If all squares within distance one are tape squares, nothing changes.**

For all \( a, b \in S, \gamma_1, \gamma_2, \gamma_3 \in \Gamma \):

\[(a, t(\gamma_1), \underline{t(\gamma_2)}, t(\gamma_3), b) \rightarrow t(\gamma_2).\]

**If there is a head square within distance one, it moves left or right and changes state.** The details are as follows:

For every non-final state \( q \in Q \setminus F \) and every letter \( \gamma \in \Gamma \), the Turing machine changes state to \( q' \), writes a new letter \( \gamma' \) to the tape, and moves in a direction \( d \):

\[\delta(q, \gamma) = (q', \gamma', d).\]

Let \( q', \gamma', d \) be the state, letter, and direction so that \( \delta(q, \gamma) = (q', \gamma', d) \).

If the head was on the present square last time, it is no longer on this square, and the letter written on the tape is now \( \gamma' \).

For all \( a, b \in S, \gamma_1, \gamma_2 \in \Gamma \), add the rule:

\[(a, t(\gamma_1), \underline{h(q, \gamma)}, t(\gamma_2), b) \rightarrow t(\gamma').\]
If the head was right of the present square in state \( q, \gamma \) and moved left, it is now on the present square, with the new state \( q' \).

For all \( a, b, c \in S \) and \( \gamma_1 \in \Gamma \), if \( d = R \), add the rule:

\[
(a, b, t(\gamma_1), h(q, \gamma), c) \rightarrow b(q', \gamma_1)
\]

For all \( a, b, c \in S \) and \( \gamma_2 \in \Gamma \), if \( d = L \), add the rule:

\[
(a, h(q, \gamma), t(\gamma_1), b, c) \rightarrow h(q', \gamma_1)
\]

There are two important things to notice: first, these rules will not match if any state within distance two is lazy. Second, we only added rules for non-final states \( q \in Q \setminus F \). No rules will match if the old state \( q \) was final.

### 6.2.2 The wavefronts add blank tape

A Turing machine has an infinite tape, but our machine must start with only finitely many non-blank squares. To make the tape look infinite, we use the left and right wavefront states to add more blank squares onto the tape at each time step.

The wavefronts propagate left and right and leave blank tape behind, as long as the state behind them is not lazy.

For all \( a, b \in S \), add the rules:

\[
(*, *, *, \downarrow, a) \rightarrow \downarrow \quad \quad \quad \quad \quad \quad \quad (a, \downarrow, *, *, *) \rightarrow \downarrow
\]

\[
(*, *, \uparrow, a, b) \rightarrow t(\beta) \quad \quad \quad \quad \quad \quad \quad (a, b, \uparrow, *, *) \rightarrow t(\beta)
\]

Any tape square near a wavefront goes blank.

For all \( a, b, c, d \in S \), add the rules:

\[
(*, \downarrow, a, b, c) \rightarrow t(\beta) \quad \quad \quad \quad \quad \quad \quad (a, b, \uparrow, *, *) \rightarrow t(\beta)
\]

\[
(\downarrow, a, b, c, d) \rightarrow t(\beta) \quad \quad \quad \quad \quad \quad \quad (a, b, \downarrow, d, \uparrow) \rightarrow t(\beta)
\]

This rule could in principle wipe out important data or destroy the head, but if we leave three squares of space between the wavefronts and the head and non-blank starting data, the wavefronts and head both move at speed one, so the head can never catch up or write non-blank data within distance two of the wavefronts.

The wavefront is creating new blank squares, but the head cannot catch up to see it. So, as far as the head is concerned, the left and right ends of the tape just look like an infinite string of blank squares, which is what we want.
6.2.3 Wave of laziness

We set up the rules above so that the states of the squares at distance two do not affect the choice of the new state at all, unless the state is $\lambda$.

The rules that don’t involve the wavefront have no wildcards in them. If a certain square can see a lazy square, and it can’t see a wavefront square, no rules will apply, so it will become lazy at the next time step.

The rules that do involve the wavefront have wildcards on the outside, but not the inside, so again if a lazy square appears on the wrong side of the wavefront, any square that sees it will become lazy at the next time step.

That creates a wave of laziness that propagates at speed two. It is faster than the wavefronts, which only move at speed one, so it will eventually catch up with it and destroy them. At this point the machine will halt.

That means that we can halt the machine just by making one square between the wavefronts lazy.

6.2.4 The initial state

We set up the initial state as follows: the left wavefront symbol, a few blank tape squares, tape and head squares representing the initial condition of the Turing machine, a few blank tape squares, and the right wavefront.

\[ \ldots, \lambda, \lambda, \lambda, \leftarrow, t(b), t(b), t(b), \]
\[ t(T(-B)), t(T(-B + 1)), \ldots, t(T(-1)), h(q_0, T(0)), t(T(1)), \ldots, t(T(B)), \]
\[ t(b), t(b), t(b), \rightarrow, \lambda, \lambda, \lambda, \ldots \]

As mentioned earlier, if we have those three blank spaces between the wavefronts and the data, the left and right wavefronts will expand the blank tape in both directions by one square at every time step without destroying any of the data.

6.3 This is a simulation of a Turing machine.

Theorem 6.1. The $p+f$ global halting problem is undecidable.

Proof. Given a Turing machine and the input for it, we construct a lazy automaton as above. The rules make sure that there is a single head square moving, changing state, and writing to the tape, just like the head of the real Turing machine, except when the Turing machine enters a halting state. We left out the partial rules that would have applied to the head of a Turing machine in a halting state, so at the next time step every square within distance one becomes lazy.

As discussed in section 6.2.3, a lazy state that appears between the two wavefronts will create a wave of laziness that will eventually catch up to the wavefronts and halt the automaton. On the other hand, if the Turing machine never enters a halting state, no lazy state can appear between the wavefronts, so they will travel to the left and right forever, and the lazy automaton will not...
halt. That is, the lazy automaton halts if and only if the Turing machine it is simulating halts.

We have reduced the halting problem for a Turing machine to the global halting problem for a p+f sandpile, so global halting must be undecidable. □

7 Local halting is undecidable.

The third problem is:

**P+f Local Halting Prediction.** Given a periodic sandpile with finitely many chips added, does the origin topple infinitely many times?

We start from the sandpile that we built in section 3 to simulate a Turing machine. Every vertex topples at most once, so the operation of the Turing machine itself will not make the origin topple infinitely many times.

We will set up a sandpile configuration that will topple every vertex infinitely often, and that can be set off by a wire. On the other hand, there will be a large region which is far enough away from criticality that the sandpile circuits we built in section 2 can function normally inside it without setting it off.

Consider a cubical periodic domain in $\mathbb{Z}^3$ with period $n$. We break vertices into four types. A vertex $(x, y, z)$ is a face vertex if exactly one coordinate is zero modulo $n$, an edge vertex if two are, and a corner vertex if all three are. All other vertices are body vertices.

We are about to put chips on every vertex that doesn’t have chips on it already. But before doing that, we need to rearrange the circuit so that it won’t cause unwanted topplings on the faces or edges.

Start with the vertex halting machine as in section 4, but disconnect the alarm wires from the origin. Now, move the gates and wires, possibly enlarging the periodic domain, so that, first, none of the wires or gates are within distance one of any edge or corner vertex, and second, everything is spaced out enough that the operation of the circuit puts:

- at most two chips on any body vertex without chips on it already, and
- at most one chip on any face vertex without chips on it already.

This is possible because the edges and corners together make a sparse cubic mesh, so there is plenty of room for gates and wires in the body of the cubes, and circuits in adjacent cubes can be connected by wires running through the faces, which can have as much space between them as one likes. It will be seen that none of the gates put more than two chips on adjacent vertices, so the above conditions can be met just by leaving enough space between parts.

Now we add the extra chips. Put five chips on every edge and corner vertex, four chips on every face vertex without chips already, and three chips on every body vertex without chips already. Every vertex that starts with chips on it has at least four, so after adding chips, every edge or corner vertex has five chips, every face vertex has four or five, and every body vertex has at least three.
Finally, ignoring all the restrictions above, connect the alarm wires back up to the origin with a wire.

If the conditions above are met, the operation of the gates and wires will add at most two chips to body vertices and at most one chip to face vertices, so this will not cause any extra topplings.

The Turing machine operates as in section 4. As before, when the alarm wire goes off, it topples the origin. By the following lemma, this makes every vertex topple at least once:

**Lemma 7.1.** If a sandpile on $\mathbb{Z}^3$ has at least:

- five chips on every vertex $(x, y, z)$ with at least two coordinates zero,
- four chips on every vertex with one coordinate zero, and
- three chips on every vertex,

and the origin ever topples, then every vertex topples at least once.

**Proof.** We will prove that all vertices $(x, y, z)$ with $x, y, z \geq 0$ topple. We prove this by induction on the sum $x + y + z$ of the coordinates. The origin topples, so that covers the base case $x + y + z = 0$.

Let $n \geq 1$. Suppose all the vertices $(x, y, z)$ with $x + y + z < n$ topple. Let $(x, y, z)$ be a vertex in $x, y, z \geq 0$ with $x + y + z = n$.

The vertex has at least one neighbour that topples for each of its three coordinates $x, y, z$ that is positive. It is easy to see that the number of chips it starts with, plus the number of neighbours that topple, is always at least six. For example, if all three are positive, the vertex starts with three chips and has three neighbours $(x - 1, y, z), (x, y - 1, z), (x, y, z - 1)$ that are guaranteed to topple. Once they topple, the vertex will have six chips, so it will topple too.

By induction, every vertex in $x, y, z \geq 0$ topples. The same argument works in every other octant.

The lemma below sets up for Lemma 7.3.

**Lemma 7.2.** If a vertex in a sandpile on a connected graph topples infinitely often, then all vertices topple infinitely often.

**Proof.** Let $w$ be a vertex that topples infinitely many times, and $v$ an adjacent vertex. Each time $w$ topples, it puts one chip on $v$, but $v$ has to topple at least once for every six chips that fall on it, so $v$ topples infinitely many times too.

If a vertex topples infinitely often, so does its neighbour. The graph is connected, so all vertices topple infinitely often.

The next lemma is a dichotomy similar to Lemma 4 from Tardos [12].

**Lemma 7.3.** If every vertex topples at least once, all vertices topple infinitely often.
Proof. By Lemma 7.2, either all vertices topple infinitely often, or all vertices topple finitely often. Suppose they topple only finitely often.

By the proof of Lemma A.2, the sandpile process produces a legal complete toppling sequence $a_1, a_2, \ldots$ in which each vertex shows up only finitely often.

Pick any vertex $x$. Once we have finished toppling at $x$ and all its neighbours, the number of chips on $x$ will stop changing. The final number of chips must be less than six.

Delete the first occurrence of each vertex from the list $a$ to produce a new toppling sequence $b$. This is possible since every vertex appears at least once. We topple one less time at $x$, but also one less time at each of its neighbours. Six fewer chips will go out, six fewer chips will come in, and we will wind up with the same final number of chips — less than six.

This means no vertex has six or more chips on it forever: if it does have six or more chips on it at some point, there must be at least one toppling afterward. So the shortened sequence $b$ is still complete. By Lemma A.1, if $a$ is legal and $b$ is complete, $N_a(x) \leq N_b(x)$ for every vertex $x$. But $N_b(x) = N_a(x) - 1$.

This is a contradiction, so it cannot be true that every vertex topples only finitely often. All vertices must topple infinitely often. \qed

A The process does not depend on choices

In this appendix we prove that the number of topplings in the sandpile process does not depend on the choices of vertices to topple.

This is because the process is “abelian,” in the sense that switching the order of two topplings does not affect the result. The terminology is due to Dhar [5], although for convenience our proof will use the fact that topplings on stable vertices also commute, which is not part of the original requirements.

We use this fact, together with the fact that the only thing that can reduce the number of chips on a square is a toppling on that square.

The following theorem is due to Björner, Lovász, and Shor [3]. The proof below is adapted from Lemma 4.5 in Bond and Levine [4].

Let $G = (V, E)$ be a graph. If $x \in V$, let $d(x)$ denote the out-degree of $x$. Each vertex has a certain number of chips on it. Recall that a vertex is unstable if it has $d(x)$ chips or more, and that we “topple” at a vertex by taking $d(x)$ chips off of it and putting one on every out-adjacent vertex. We allow ourselves to topple even at a stable vertex, resulting in a negative number of chips.

A toppling sequence $a_1, a_2, \ldots$ is a finite or infinite sequence of vertices in $V$. It is legal if every vertex that is toppled is unstable. It is complete if no vertex that can be toppled stays untoppled: any vertex that gets at least $d(x)$ chips on it at some time has to appear at least once afterward.

Let $N_a(x)$ be the number of times $x \in V$ appears in the toppling sequence $a$.

Lemma A.1. If $a$ is legal and $b$ is complete, $N_a \leq N_b$ for all vertices.

Proof. Match every term $a_j$ in $a$ with a corresponding term $b_{h(j)}$ in $b$, with no two indices the same. If this is possible for every $j$, then $N_a \leq N_b$. 26
Otherwise, stop at the first index $\ell$ where this is not possible. Let $x = a_\ell$ be the vertex where the problem occurs. We couldn’t find an unused copy of $x$ in $b$, so $a_1, \ldots, a_{\ell-1}$ must already contain $N_b(x)$ occurrences of $x$.

After we topple at $a_1, \ldots, a_{\ell-1}$, $x$ has at least $d(x)$ chips on it, because $a$ is a legal sequence. Let $m$ be the largest index in $h(1), \ldots, h(\ell - 1)$. How many chips does $x$ have after we topple at $b_1, \ldots, b_m$?

This second list must contain $a_1, \ldots, a_m$ as a multiset, and it can’t contain any more occurrences of $x$, because we have already exhausted them all.

Extra topplings on vertices other than $x$ only increase the number of chips on $x$, so $x$ must have at least six chips after toppling at $b_1, \ldots, b_m$. The topplings in the two lists may be in a different order, but we are assuming that rearranging the order of topplings doesn’t change the result.

By the fact that $b$ is complete, there must be another occurrence of $x$ after $b_m$. But $b_1, \ldots, b_m$ must exhaust all the occurrences of $x$ in $b$. This is a contradiction.

Corollary. If $a$ and $b$ are legal and complete, $N_a = N_b$ for all vertices.

Lemma A.2. If $V$ is finite or countable, the sandpile process produces a legal complete toppling sequence.

Proof. We just carry out the sandpile process and write down the list of vertices that we topple at. We never topple at a stable vertex, and if a vertex is unstable we will eventually check it again and topple it, so the result is a legal complete toppling sequence.

Corollary. Given any sandpile on a finite or countable graph $(V,E)$, there is a unique function $N : V \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ so that every legal complete toppling sequence topples exactly $N(x)$ times at $x$. 

27
References

[1] Bitar, J., and Goles, E. Parallel chip firing games on graphs. *Theoretical Computer Science* 92, 2 (1992), 291–300.

[2] Björner, A., and Lovász, L. Chip-firing games on directed graphs. *Journal of Algebraic Combinatorics* 1, 4 (1992), 305–328.

[3] Björner, A., Lovász, L., and Shor, P. W. Chip-firing games on graphs. *European Journal of Combinatorics* 12 (1991), 283–291.

[4] Bond, B., and Levine, L. Abelian networks: foundations and examples. http://arxiv.org/abs/1309.3445.

[5] Dhar, D. Self-organized critical state of sandpile automaton models. *Physical Review Letters* 64 (Apr 1990), 1613–1616.

[6] Eriksson, K. No polynomial bound for the chip firing game on directed graphs. *Proceedings of the American Mathematical Society* 112, 4 (1991), 1203–1205.

[7] Farrell, M., and Levine, L. CoEulerian graphs. http://arxiv.org/abs/1502.04690.

[8] Gajardo, A., and Goles, E. Crossing information in two-dimensional sandpiles. *Theoretical Computer Science* 369, 3 (2006), 463–469.

[9] Goles, E., and Margenstern, M. Sand pile as a universal computer. *International Journal of Modern Physics C* 07, 02 (1996), 113–122.

[10] Moore, C., and Nilsson, M. The computational complexity of sandpiles. *Journal of Statistical Physics* 96 (July 1999), 205–224.

[11] Ramachandran, V., and Yang, H. An efficient parallel algorithm for the general planar monotone circuit value problem. *SIAM Journal on Computing* 25 (Aug 1994), 312–339.

[12] Tardos, G. Polynomial bound for a chip firing game on graphs. *SIAM Journal on Discrete Mathematics* 1, 3 (1988), 397–398.

[13] Yang, H. An NC algorithm for the general planar monotone circuit value problem. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. (Dec 1991), pp. 196–203.

Acknowledgements

The majority of my thanks go to my advisor, Lionel Levine, who suggested that I put this idea in the form of a paper and made extensive suggestions. Matt Farrell and Swee Hong Chan both kindly read some of my drafts and gave valuable comments.