Algorithmica (2024) 86:2069–2091
https://doi.org/10.1007/s00453-024-01222-7

Embedding Arbitrary Boolean Circuits into Fungal Automata

Augusto Modanese[1] · **Thomas Worsch**[2]

Received: 24 February 2023 / Accepted: 23 February 2024 / Published online: 21 March 2024

Abstract
Fungal automata are a variation of the two-dimensional sandpile automaton of Bak et al. (Phys Rev Lett 59(4):381–384, 1987. https://doi.org/10.1103/PhysRevLett.59.381). In each step toppling cells emit grains only to *some* of their neighbors chosen according to a specific update sequence. We show how to embed any Boolean circuit into the initial configuration of a fungal automaton with update sequence HV. In particular we give a constructor that, given the description B of a circuit, computes the states of all cells in the finite support of the embedding configuration in $O(\log |B|)$ space. As a consequence the prediction problem for fungal automata with update sequence HV is P-complete. This solves an open problem of Goles et al. (Phys Lett A 384(22):126541, 2020. https://doi.org/10.1016/j.physleta.2020.126541).

Keywords Fungal automata · Prediction problem · P-completeness · Two-dimensional cellular automata

1 Introduction

The two-dimensional sandpile automaton by Bak et al. [1] has been investigated from different points of view. Because of the simple local rule, it is easily generalized to the d-dimensional case for any integer $d \geq 1$.

Several *prediction problems* for these cellular automata (CA) have been considered in the literature. Their difficulty varies with the dimensionality. The recent survey by Formenti and Perrot [2] gives a good overview. For one-dimensional sandpile CA the problems are known to be easy (see, e.g., [3]). For d-dimensional sandpile CA where

✉ Augusto Modanese
augusto.modanese@aalto.fi

✉ Thomas Worsch
worsch@kit.edu

1 Aalto University, Espoo, Finland

2 Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

$d \geq 3$, they are known to be P-complete [4]. In the two-dimensional case the situation is unclear; analogous results are not known.

Fungal automata (FA) as introduced by Goles et al. [5] are a variation of the two-dimensional sandpile automaton where a toppling cell (i.e., a cell with state ≥ 4) emits 2 excess grains of sand either to its two horizontal ("H") or to its two vertical neighbors ("V"). These two modes of operation may alternate depending on an *update sequence* specifying in which steps grains are moved horizontally and in which steps vertically.

The construction in [5] shows that some natural prediction problem is P-complete for two-dimensional fungal automata with update sequence H^4V^4 (i.e., grains are first transferred horizontally for 4 steps and then vertically for 4 steps, alternatingly). The paper leaves open whether the same holds for shorter update sequences. The shortest non-trivial sequence is HV (and its complement VH); at the same time this appears to be the most difficult to use. By a reduction from the well-known *circuit value problem* (CVP), which is P-complete, we will show:

Theorem 1 *The following prediction problem for FA with update sequence HV is P-complete:*

***Given** as inputs initial states for a finite rectangle R of cells, a cell index y (encoded in binary), and an upper bound T (encoded in unary) on the number of steps of the FA,*

***decide** whether cell y is in a state $\neq 0$ or not at some time $t \leq T$ when the FA is started with R surounded by cells all in state 0.*

We assume readers are familiar with cellular automata (see Sect. 2 for the definition). We also assume knowledge of basic facts about Boolean circuits and complexity theory, some of which we recall next.

This paper is an extended and revised version of a preliminary paper that previously appeared in LATIN 2022 [6].

1.1 Boolean Circuits and the CVP

A Boolean circuit is a directed acyclic graph of *gates*: NOT gates (with one input), AND and OR gates with two inputs, a number of INPUT gates and one OUTPUT gate. The output of a gate may be used by an arbitrary number of other gates. Throughout the paper n will always denote the number of inputs and m will denote the number of gates.

Since a circuit is a dag and each gate obtains its inputs from gates in previous layers, ultimately the output of each gate can be computed from a subset of the input gates in a straightforward way.

It is straightforward to realize NOT, AND, and OR gates in terms of NAND gates with two inputs (with an only constant overhead in the number of gates). To simplify the construction later on, we assume that circuits consist exclusively of NAND gates. Furthermore it is without loss of generality to assume that $m \geq n$. If a circuit does not satisfy this restriction it can be adapted by computing $0 = x_i \wedge \neg x_i$ from each INPUT

x_i using constant size subcircuits c_i and then extending the circuit by using the result o of its original output gate and compute $(\cdots((o \vee c_1) \vee c_2) \cdots \vee c_n) = o$.

Each gate of a circuit is described by a 4-tuple (g, t, g_1, g_2) where g is the number of the gate, t describes the type of the gate, and g_1 and g_2 are the numbers of the gates (called sources of g) which produce the inputs for gate g; all numbers are represented in binary. If gate g has only one input, then $g_2 = g_1$ by convention. Without loss of generality the INPUT gates have numbers 1 to n and since their predecessors g_1 and g_2 will never be used, assume they are set to 0. All other gates have subsequent numbers starting at $n + 1$ such that the input bits for gate g are coming from gates with strictly smaller numbers. The NAND gates with number $n + i$ will occasionally be denoted as G_i. Following Ruzzo [7] the description B of a complete circuit is the concatenation of the descriptions of all of its gates, sorted by increasing gate numbers.

Problem instances of the *circuit value problem* (CVP) consist of the description B of a Boolean circuit C with n inputs and a list x of n input bits. The task is to decide whether $C(x) = 1$ holds or not. It is well known that the CVP is P-complete.

1.2 Challenges

Given an initial configuration for a rectangle of cells R of an FA, it is straightforward to simulate the evolution of R with a polynomial-time Turing machine. Hence it is clear that the prediction problem is in P and that the challenging aspect of Theorem 1 is obtaining the P-hardness of the problem. A standard strategy for showing the P-hardness of a problem Π that concerns predicting the behavior of a machine model $\mathcal{M}$ is by a reduction from the CVP to Π, which essentially amounts to describing how to "embed" circuits in $\mathcal{M}$. (This is the approach used in the previous work by Goles et al. [5].)

Formally what we need to show is that there is a *constructor* operating in logarithmic space that, given an instance (C, x) of the CVP, constructs an instance (R, y, T) of the prediction problem (such that one is a yes-instance if and only if the other is also a yes-instance of the respective problem). From a practical aspect, the greater challenge is showing how to implement basic circuit components (wires, gates, crossings, etc.) and then connect these to realize C in the initial configuration R (in such a way that can be constructed in logarithmic space). Once we have done so, we simply place the input x in the cells in R that represent the input of C and identify y with the one that represents the output of C. As for T, we choose a large enough value so that enough time steps have elapsed for the output to be observed at y.

In our setting of fungal automata with update sequence HV, although realizing wires and signals as in [5] is possible, there is no obvious implementation for negation nor for a reliable wire crossing. Hence it seems one can only directly construct circuits that are *both* planar and monotone. Although it is known that the CVP is P-complete for *either* planar or monotone circuits [8], it is unlikely that one can achieve the same under both constraints. This is because the CVP for circuits that are both monotone and planar lies in NC^2 (and is thus certainly not P-complete unless $P \subseteq NC^2$) [9].

We are able to overcome this barrier by exploiting features that are present in fungal automata but not in general circuits: *time* and *space*. Namely we deliberately *retard*

signals in the circuits we implement by extending the length of the wires that carry them. We show how this allows us to realize a primitive form of transistor. From this, in turn, we are able to construct a NAND gate, thus allowing both wire crossings and negations to be implemented.

Our construction is not subject to the limitations that apply to the two-dimensional case that were previously shown by Gajardo and Goles [10] since the FA starting configuration is not a fixed point. The resulting construction is also significantly more complex than that of [5].

1.3 Overview of the Construction

In the rest of the paper we describe how to embed any Boolean circuit with description B and an assignment of values to the inputs into a configuration c of a fungal automaton in such a way that the following holds:

- "Running" the FA for a sufficient number of steps results in the "evaluation" of all simulated gates. In particular, after reaching a stable configuration, a specific cell of the FA is in state 1 or 0 if and only if the output of the circuit is 1 or 0, respectively.
- The initial configuration F of the FA is simple in the sense that, given the description of a circuit and an input to it, we can produce its embedding F using $O(\log n + \log |B|)$ space. Thus we have a log-space reduction from the CVP to the prediction problem for FA.

The construction consists of several layers:

Layer 0: The underlying model of fungal automata.
Layer 1: As a first abstraction we subdivide the space into *"blocks"* of 2×2 cells and always think of update *"cycles"* consisting of 4 steps of the CA, using the update sequence $(HV)^2$.
Layer 2: On top of that we will implement *"polarized circuits"* processing *"polarized signals"* that run along *"wires"*.
Layer 3: Polarized circuitry is then used to implement *"Boolean circuits with delay"*: *"bits"* are processed by *"gates"* connected by *"cables"* (we slightly deviate from the standard terminology of Boolean circuits and reserve the term *"wire"* for the more primitive wires defined in layer 2.)
Layer 4: Finally a given Boolean circuit (without delay) can be embedded in a fungal automaton (as a circuit with delay) in a systematic fashion that needs only logarithmic space to construct.

The rest of this paper has a simple organization: Each layer i will be described separately in section $i + 2$.

2 Layer 0: The Fungal Automaton

Let $\mathbb{N}_+$ denote the set of positive integers and $\mathbb{Z}$ that of all integers. For $d \in \mathbb{N}_+$, a *d-dimensional CA* is a tuple (S, N, δ) where:

- S is a finite set of states
- N is a finite subset of $\mathbb{Z}^d$, called the *neighborhood*
- $\delta\colon S^N \to S$ is the *local transition function*

In the context of CA, the elements of $\mathbb{Z}^d$ are referred to as *cells*. The function δ induces a *global transition function* $\Delta\colon S^{\mathbb{Z}^d} \to S^{\mathbb{Z}^d}$ by applying δ to each cell simultaneously. In the following, we will be interested in the case $d = 2$ and the so-called *von Neumann neighborhood* $N = \{(a, b) \in \mathbb{Z}^2 \mid |a| + |b| \leq 1\}$ of radius 1.

Except for the updating of cells the fungal automaton is just a two-dimensional CA with the von Neumann neighborhood of radius 1 and $S = \{0, 1, \ldots, 7\}$ as the set of states. For consistency, we use the same set of states as in [1]; however, the states 6 and 7 never occur in our construction, only the subset $S' = \{0, 1, \ldots, 5\}$ will be needed. A *configuration* is thus a mapping $c : \mathbb{Z}^2 \to S$.

Depending on their states cells will be depicted as follows in diagrams:

- state 0 as $\boxed{}$
- state 1 as $\boxed{\cdot}$
- state $i \in S\backslash\{0, 1\}$ as $\boxed{i}$

We will use colored background for cells in states 2, 3, and 4 since their presence determines the behavior of the polarized circuit. The state 1 is only a "side effect" of an empty cell receiving a grain of sand from some neighbors; hence it is represented as a dot. Cells which are not included in a figure are always assumed to be in state 0.

For a logical predicate P denote by $[P]$ the value 1 if P is true and the value 0 if P is false. For $i \in \mathbb{Z}^2$ denote by $h(i)$ the two horizontal neighbors of cell i and by $v(i)$ its two vertical neighbors. Cells are updated according to 2 functions H and V mapping from $S^{\mathbb{Z}^2}$ to $S^{\mathbb{Z}^2}$ where for each $i \in \mathbb{Z}^2$ the following holds:

$$H(c)(i) = c(i) - 2 \cdot [c(i) \geq 4] + \sum_{j \in h(i)} [c(j) \geq 4];$$
$$V(c)(i) = c(i) - 2 \cdot [c(i) \geq 4] + \sum_{j \in v(i)} [c(j) \geq 4].$$

Note that in every time step either all cells will use H or all cells will use V. Therefore the values computed by these will always be in the range $\{0, \ldots, 7\}$ again; and if all cells in a neighborhood are in fact from the subset $S' = \{0, \ldots, 5\}$ the new state computed will be from S' again.

The updates are similar to the sandpile model by Bak et al. [1], but toppling cells only emit grains of sand either to their horizontal or their vertical neighbors. Therefore whenever a cell is non-zero, it stays non-zero forever.

The composition of these functions applying first H and then V is denoted HV. For the transitions of a fungal automaton with update sequence HV these functions are applied alternatingly, resulting in a computation c, $H(c)$, $V(H(c))$, $H(V(H(c)))$, $V(H(V(H(c))))$, and so on. In examples we will often skip three intermediate configurations and only show c, $HVHV(c)$, etc. Figure 1 shows a simple first example.

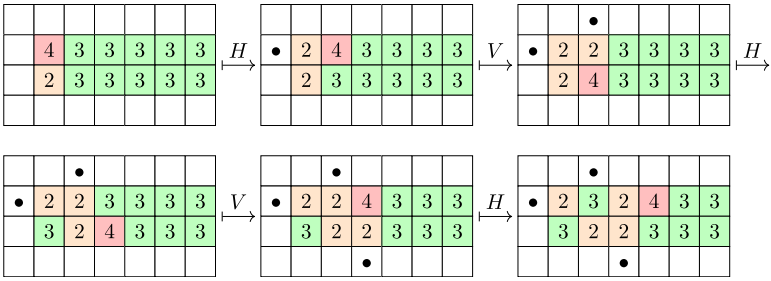


Fig. 1 Five transitions according to $HVHVH$

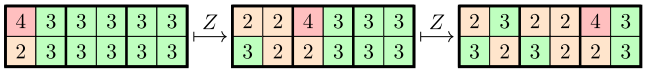


Fig. 2 Compact representation of two cycles

3 Layer 1: Coarse-Graining Space and Time

As a first abstraction from now on one should always think of the space as subdivided into *"blocks"* of 2×2 cells. Furthermore we will look at update *"cycles"* consisting of 4 steps of the CA, thus using the update sequence $HVHV$ which we will abbreviate to Z. As an example Fig. 2 shows the same cycle as Fig. 1 and the following cycle in a compact way. Block boundaries are indicated by thicker lines.

Cells outside the depicted area of a figure are assumed to be $\boxed{0}$ initially and they will never become critical and topple during the computation that is shown.

4 Layer 2: Polarized Components

We turn to the second lowest level of abstraction. Here we work with two types of signals, which we refer to as *"positive"* (denoted $\boxplus$) and *"negative"* (denoted $\boxminus$). Both types will have several representations as a block in the FA.

- A $\boxplus$ signal is represented by the ***upper*** *left corner* of the block being a $\boxed{4}$ and the other cells being $\boxed{2}$ or $\boxed{3}$.
- A $\boxminus$ signal is represented by the ***lower*** *left corner* of the block being a $\boxed{4}$ and the other cells being $\boxed{2}$ or $\boxed{3}$.

The rules of fungal automata allow us to perform a few basic operations on these *"polarized"* signals (e.g., duplicating, merging, or crossing them under certain assumptions). The highlight here is that we can implement a (delay-sensitive) form of transistor that works with polarized signals, which we refer to as a *"switch"*.

In this section as a convention we write x and y for the inputs of a component and z, z_1, and z_2 for the outputs. Both input and output pins are shown as blue 2×2 blocks in pictures showing the structure of a component. (The reason for choosing this representation will be made clear later in Sect. 4.5.) The initial states of the four cells

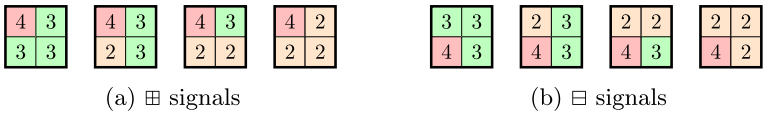


Fig. 3 Valid representations of ⊞ and ⊟ signals

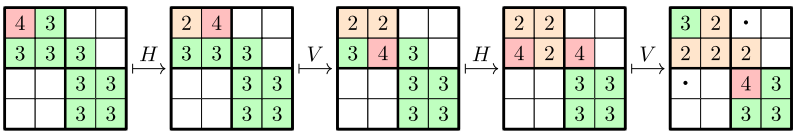


Fig. 4 A diagonal wire component with a ⊞ signal moving south east

of such a block are always 3. All polarized components have their input pins always at the left border and output pins on the right.

4.1 Polarized Signals and Wires

Figure 3 shows the most common occurrences of ⊞ and ⊟ signals. We will refer to a block initially containing a ⊞ or ⊟ signal as a ⊞ or ⊟ *"source"*, respectively. (This will be used, for instance, to set the inputs to the embedded CVP instance.)

Taking another look at Fig. 2, observe that it shows a ⊞ signal "moving from the left to the right". In general we will use *"wires"* to propagate signals. A wire is a contiguous arrangement of 2×2 *wire blocks* of cells in state 3 that carries a signal from a source or component to another component. (Wires will also be used as building blocks for certain components.) Using these wire blocks it is straightforward to route signals in all four cardinal directions. Later in Sect. 4.5 we specify exactly how wires are used to connect components with each other.

While one can use the same wire blocks for both types of signals, each block is destroyed upon use and thus can only be used once. In particular, this means every wire will be used either by a ⊞ or a ⊟ signal. For better organization, we assign each wire a *polarity* that indicates which of the two it is intended to carry and then refer to the wire as a ⊞ or ⊟ *wire*, accordingly.

For convenience we extend wires so that they may also run along a $\pm 45°$ diagonal. (Note that throughout the paper all such signals are moving "from the left to the right".) The basic building block for this extension are two diagonally neighboring wire blocks which have to be "glued together" by an additional 3 cell adjacent to both blocks. For a ⊞ signal the glue cell has to be on the left hand side in the "direction of travel", and for a ⊟ signal on the right hand side. Figure 4 shows the four steps realizing the movement of a ⊞ signal to the south east. The case of movement to the north east works analogously. For ⊟ signals one only needs to mirror the images along a horizontal line.

Stringing together several diagonal components with the same direction results in diagonal wires along which a signal moves one block horizontally *and* vertically in *one* cycle. By adding both types of glue, we can even use the same construction for both ⊞ and ⊟ signals. Figure 5 shows an example. Note that this only means that it can

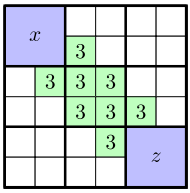


Fig. 5 A diagonal wire for ⊞ and ⊟ signals

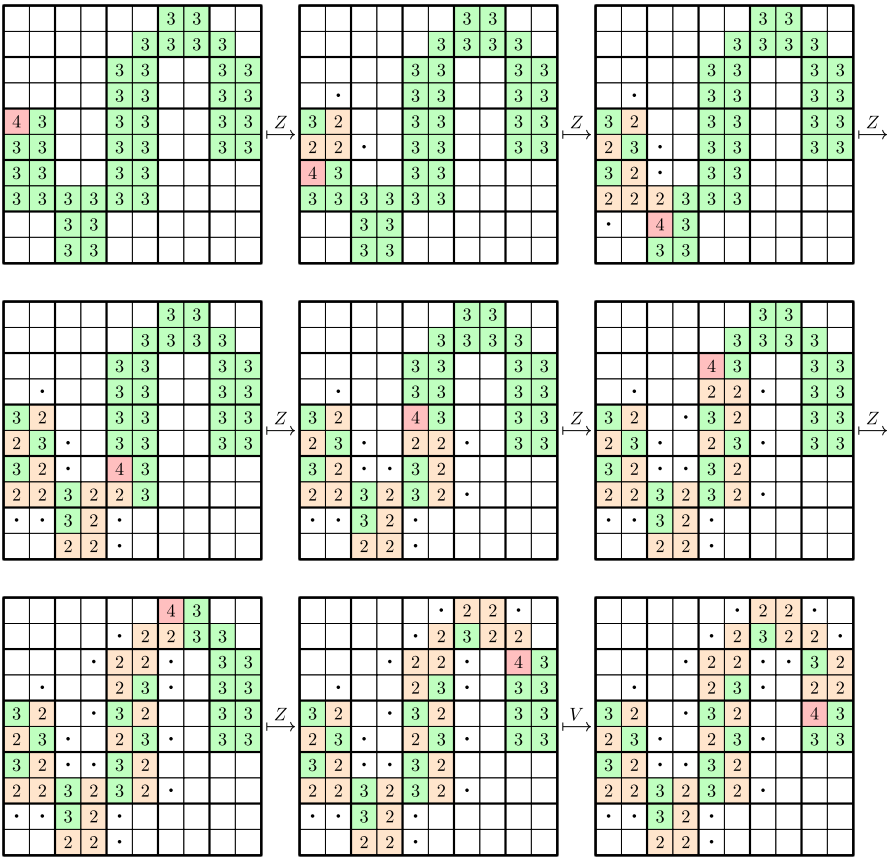
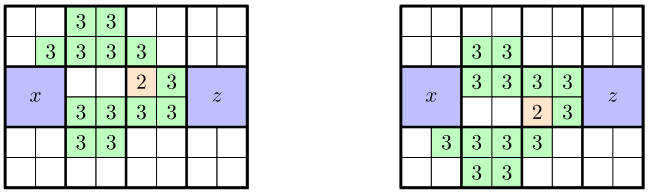


Fig. 6 A ⊞ signal moving along a bent wire. The second and third cycle realize a "turn left by 90 degrees". After moving up the next two cycles perform a U-turn to the right. Two full cycles later the signal arrives at the endpoint

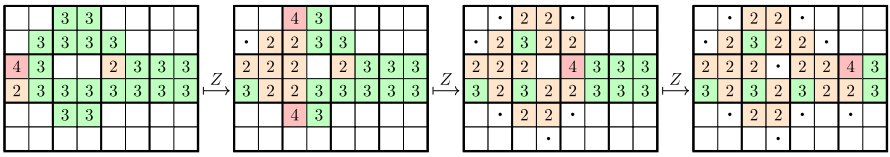
be used by both types of signals; as in all other cases it can only be used *once*, either by a ⊞ or by a ⊟ signal.

By sticking together diagonals in an up and a down diagonal movement one can implement "U-turns". Figure 6 shows an example where a ⊞ signal first makes a left U-turn and then a right U-turn. This basic layout will be used and extended for retarders in Sect. 4.5.

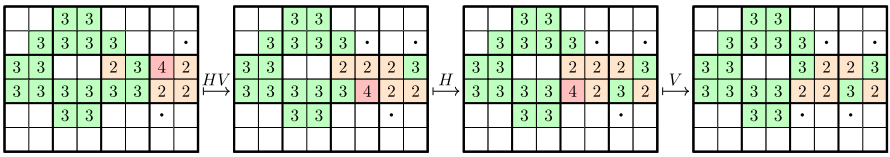


(a) diode for ⊞ wires (b) diode for ⊟ wires

Fig. 7 Diode implementations (signals can pass from left to right)



(a) ⊞ signal comes from the left



(b) ⊞ signal comes from the right

Fig. 8 Diode operation on ⊞ wires

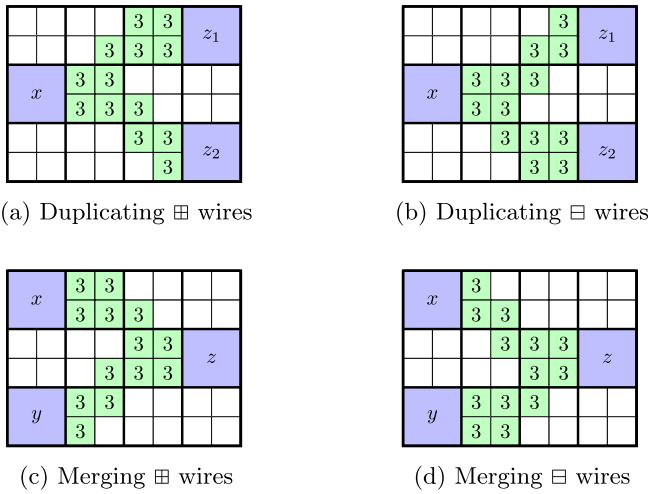
4.2 Diodes

Note that ⊞ and ⊟ signals do not encode any form of direction in them (regarding their propagation along a wire). In fact, a signal propagates in any direction a wire is placed in. In order for our components to operate correctly, it will be necessary to have a means for *preventing* a signal from propagating along a wire in a certain direction. To realize this, we use *diodes*.

A diode is an element on a horizontal wire that only allows a signal to flow from left to right. A signal coming from right to left is not allowed through. As the other components, the diode is intended to be used only once. For the implementation, refer to Fig. 7. (Recall that x denotes the component's input and z its output.)

Figure 8 illustrates the operation of a diode for ⊞ signals. (The case of ⊟ signals is similar.) Notice that, in the case where the signal comes from the left-hand side (Fig. 8a), two ⊞ signals are momentarily created and then merged in the one cell of the diode that has $\boxed{2}$ as its starting state. In turn, when the signal comes from the right-hand side (Fig. 8b), the same cell is responsible for "absorbing" the signal; that is, the state $\boxed{2}$ turns into a $\boxed{3}$, but the signal does not go any further.

For all the remaining elements described in this section, we implicitly add diodes to their inputs and outputs. This ensures that the signals can only flow from left to right (as intended) and, in addition, that every input or output of a component receives or



(a) Duplicating ⊞ wires (b) Duplicating ⊟ wires

(c) Merging ⊞ wires (d) Merging ⊟ wires

Fig. 9 Duplicating and merging wires

emits a signal at most once. This is probably not necessary for all elements, but doing so makes the construction simpler and also facilitates reasoning about its correctness; the overhead is only a constant factor blowup in the size of the elements.

4.3 Duplicating, Merging, and Crossing Wires

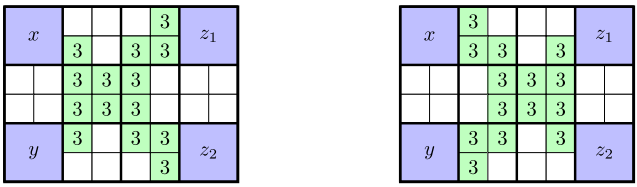
Wires of the same polarity can be *duplicated* or *merged*. By duplicating a wire we mean we create two outputs z_1 and z_2 from a single input x in such a way that, if any signal arrives at x, then this signal is duplicated and propagated on both z_1 and z_2. (Equivalently, one might imagine that $x = z_1$ and z_2 copies x.) In turn, a wire merge realizes in some sense the reverse operation: We have two wires x and y *of the same polarity* and create a wire z such that, if any signal arrives from x or y (or both), then a signal of the same polarity will emerge at z. (Hence one could say the wire merge realizes a "polarized OR" gate.) See Fig. 9 for the implementations.

As discussed in the introduction, there is no straightforward realization of a wire crossing in fungal automata in the traditional sense. Nevertheless, it turns out we *can* cross wires under the following constraints:

1. The two wires being crossed are a ⊞ and a ⊟ wire.
2. The crossing is used only once and by a single input wire; that is, once a signal from either wire passes through the crossing, it is destroyed. (If two signals arrive from both wires at the same time, then the crossing is destroyed without allowing any signal to pass through.)

To elicit these limitations, we refer to such crossings as *semicrossings*.

We actually need two types of semicrossings, one for each choice of polarities for the two input wires. The semicrossings are named according to the polarity of the top input wire: A ⊞ semicrossing has a ⊞ wire as its top input (and a ⊟ wire as its bottom



(a) ⊞ semicrossing (b) ⊟ semicrossing

Fig. 10 Semicrossing implementations

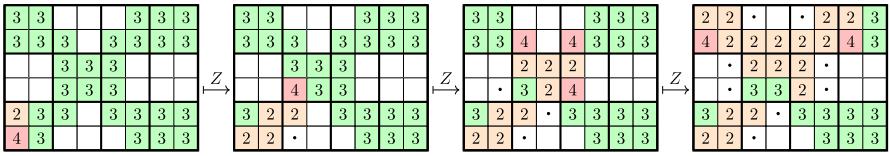


Fig. 11 ⊟ signal traversing a ⊞ semicrossing

one) whereas a ⊟ semicrossing has a ⊟ wire at the top (and a ⊞ wire at the bottom). For the implementations, see Fig. 10.

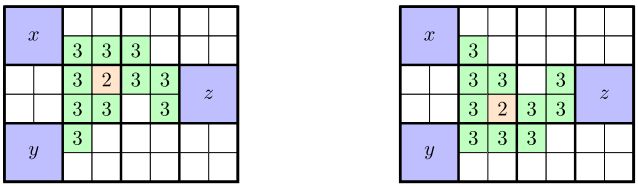
Figure 11 shows an example of a ⊞ semicrossing being traversed by a ⊟ signal. As one can see, the ⊟ signal not only destroys the semicrossing but also creates an extraneous signal that is propagated in the direction of the ⊞ input gate of the semicrossing. Fortunately (as mentioned in Sect. 4.2) we add diodes to the inputs of all gates, which prevents this other signal from causing undesirable side-effects.

4.4 Switches

A *switch* is a rudimentary form of transistor. It has two inputs and one output. Adopting the terminology of field-effect transistors (FETs), we will refer to the two inputs as the *source* and *gate* and the output as the *drain*. In its initial state, the switch is *open* and does not allow source signals to pass through. If a signal arrives from the gate, then it turns the switch *closed*. A subsequent signal arriving from the source will then be propagated on to the drain. This means that switches are *delay-sensitive*: A signal arriving at the source only continues on to the drain if the gate signal arrives (at least one time step) *previously* to the source.

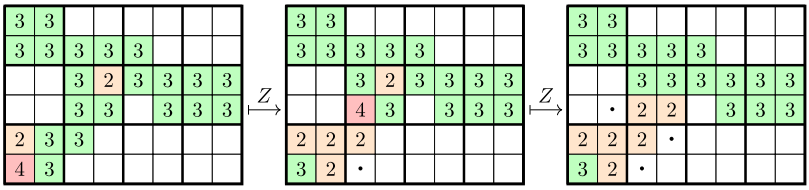
Similar to semicrossings, our switches come in two flavors. In both cases the top input is a ⊞ wire and the bottom one a ⊟. The difference is that, in a ⊞ switch, the source (and thus also the drain) is the ⊞ input and the gate is the ⊟ input. Conversely, in a ⊟ switch the source and drain are ⊟ wires and the gate is a ⊞ wire. Refer to Fig. 12 for the implementation of the two types of switches.

Figure 13 shows the behavior of the ⊞ switch. (As always, the ⊟ version is similar.) In Fig. 13a a ⊟ signal arrives and closes the switch by turning the one $\boxed{2}$ into a $\boxed{3}$. Following this, in Fig. 13b we observe a ⊞ signal traverse the closed switch. Compare this with Fig. 14, in which the ⊞ signal arrives *before* the ⊟ signal, and none of the two signals go through.

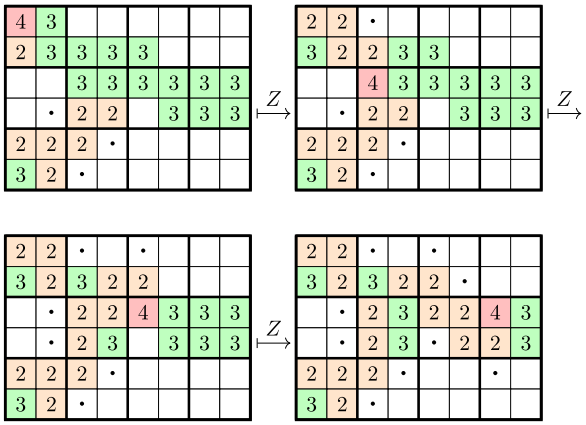


(a) ⊞ switch (b) ⊟ switch

Fig. 12 Switch implementations. In the ⊞ switch x denotes the source and y the gate. In turn, in the ⊟ switch these roles are reversed: y is the source and x the gate



(a) ⊟ signal arrives, closes switch



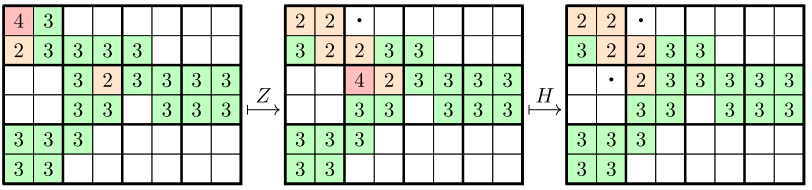
(b) ⊞ signal arrives after ⊟ signal and traverses the closed switch

Fig. 13 Behavior of the ⊞ switch when ⊟ signal arrives before ⊞ signal

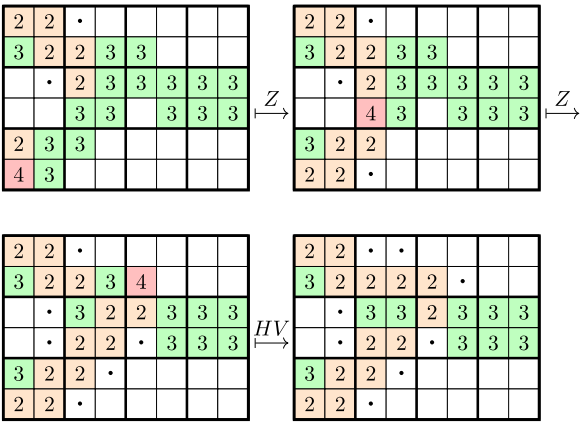
4.5 Delays and Connecting Components

As mentioned in the introduction, the circuits we construct are sensitive to the time it takes for a signal to flow from one point to the other. In the construction of the NAND gate there will be the need to know something about the *order* in which two signals arrive (if at all) at the inputs of internally used switches. This will be ensured by

- knowing upper bounds on the *delays* of some signals to be processed, and
- having so-called *retarders* which are simply "sufficiently long" meandering wires imposing long transition times on the other signals.



(a) ⊞ signal arrives before ⊟ signal, does not go through



(b) ⊟ signal arrives after and does not go through either

Fig. 14 Behavior of the ⊞ switch when ⊟ signal arrives after ⊞ signal

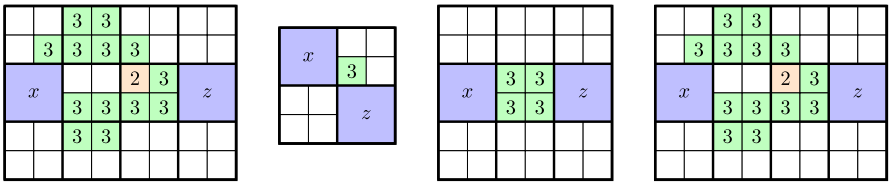
Delays will be considered in this section; retarders are the topic of the following Sect. 4.6.

All components described above (and also retarders) have at least one input and at least one output pin which are initialized with four cells in state $\boxed{3}$. (The only exception are ⊞ and ⊟ sources which we are initialized in the obvious way and which we consider as only an output pin.) Each wire can also be viewed as having an input and an output pin. Components and wires can be connected by superimposing the output pin of one object with the input pin of another. Figure 15 shows an example.

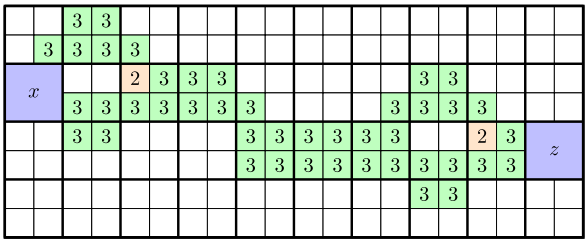
The reason for superpositioning instead of juxtapositioning is that it makes computations easier. If t_1 is the number of cycles a signal needs to travel from the input pin to the output pin of a component, and similarly t_2 for a second component, then the total travel time is simply $t_1 + t_2$. These are called *transition times*.

If it takes a cycles of the whole fungal automaton (starting from the initial configuration) before a signal arrives at the single input pin of a component (with transition time t), we call a the *arrival time* at the input. As a consequence the arrival time at the output pin(s) is $a + t$.

For diodes and wire duplications, we have $t = 3$. For wires the transition time depends on the *length* of the wire. This is defined as the number of blocks of the shortest contiguous path along the wire according to the *Moore neighborhood* $N = \{(a, b) \in \mathbb{Z}^2 \mid \max\{|a|, |b|\} \leq 1\}$. This includes retarders as defined in the next



(a) separate components to be glued together



(b) components glued together

Fig. 15 Example of two diodes and two wires to be connected (above) and their superimposition (below)

section. For example, in Fig. 16 the distance between x and z is 8. We note that all horizontal and diagonal wires have the property that signals move from one block to a next *right* neighboring block (according to the Moore neighborhood) in one cycle.

In the case of components with two inputs one has to distinguish two cases.

1. *Semicrossings and wire merges* Assume that the signals arrive at the input pins at times a_1 and a_2, where $a_i = \infty$ indicates that the signal does not arrive at all. Then the arrival time at the output pin is $\min(a_1, a_2) + 3$.
2. *Switch* Assume that the signals arrive at the gate input at time a_g and at the source input at time a_s (again $a_i = \infty$ indicates that the signal does not arrive at all).
   - If $a_g \leq a_s$ then the arrival time at the output is $a_s + 3$;
   - if $a_g > a_s$ then the arrival time at the output is ∞.

Now by definition the arrival time of sources is 0. Then for each (acyclic!) circuit the above rules define the arrival time of signals at each pin if each input is either a source or connected to a preceeding output.

4.6 Retarders

Retarders will be used to delay a signal on its travel from left to right. Therefore we will use the term *delay* instead of transition time.

The basic "meander" used in retarders has already been shown in Fig. 6. Now assume that one is given a square of $2L + 1$ by $2L + 1$ blocks, that the input pin is at the middle of the left border, and the output pin at the middle of the right border. Furthermore assume that the height of each such a meander is $2L + 1$ blocks. Then inside this square of fixed size one can place k connected meanders where k can be

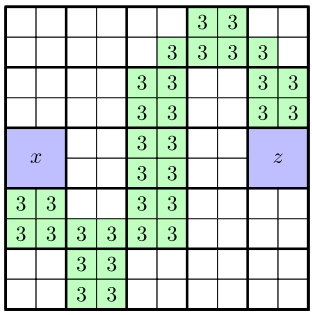


Fig. 16 Implementation of a basic meander for a ⊞ signal with transition time 8. Greater transition times can be realized by increasing (i) the height of the meanders, (ii) the number of up-down meanders, and (iii) the positions of the input and output

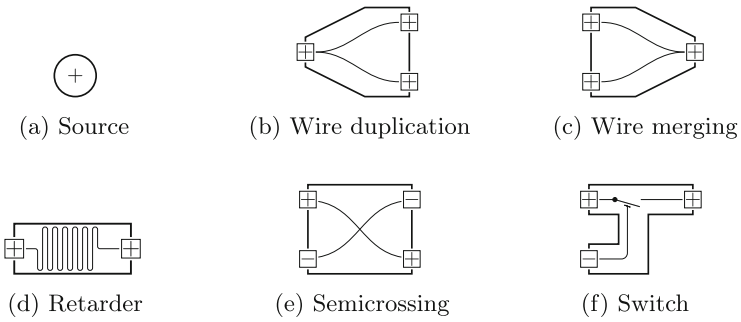


Fig. 17 Representations of the elements from abstraction layer 2 as used in layer 3. The ⊞ and ⊟ signs at input pins indicate which type of signals has to arrive; the signs at the output pins indicate which type of signal will leave. For each component there is also a dual version with all ⊞ and ⊟ exchanged

any integer up to $((2L+1)-1)/4 = L/2$ and from the last one a horizontal wire to the output pin of the square. By varying k, the delay of a signal can be varied between $\Theta(L)$ and $\Theta(L^2)$ while retaining the same area of $D = \Theta(L^2)$ cells.

In Sects. 5 and 6, we show how to construct the initial configuration for a fungal automaton that corresponds to the evaluation of a given Boolean circuit and an input to it. This will require many NAND gates which will make use of retarders. In the construction we describe, *all* retarders will have the *same size*—but the *delays will differ*. For now we mention it is sufficient to choose $L = \Theta(m^2)$ (allowing for delays up to $\Theta(m^4)$) to ensure proper timing of the whole circuit. Using retarders of a *single* size for any fixed circuit simplifies the layout significantly. (See in particular Sects. 6.3 and 6.4.)

5 Layer 3: Working with Bits

We will now draw the polarized elements from Sect. 4 in a more abstract way as shown in Fig. 17, and use them to construct planar delay-sensitive Boolean circuits. Our circuits will use NAND gates as their basis.

We use four ingredients for the realization of Boolen circuits: a representation of bits on cables, duplication of bits, NAND gates, and the crossing of cables.

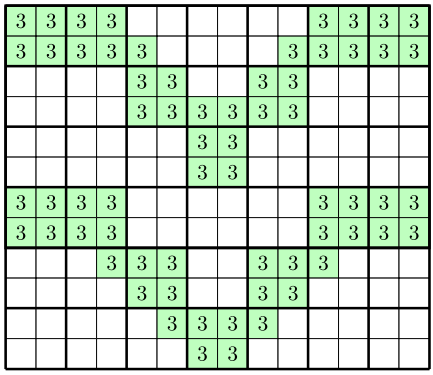


Fig. 18 Example of a simple cable with horizontal and diagonal segments. The upper wire is for the ⊞ signal representing a 1, the lower for the ⊟ signal representing a 0

5.1 Representation of Bits

For the representation of a bit, we use a pair consisting of a polarized ⊞ wire and a polarized ⊟ wire. Such a pair of polarized wires is called a *cable*. For cables only horizontal and diagonal, but never vertical, wires will be used. A signal on a cable's ⊞ wire represents a binary 1, and a signal on the ⊟ wire represents a binary 0. The construction will ensure that never both signals are present on a wire. Both wires run in parallel land are vertically separated by exactly two empty blocks everywhere; see Fig. 18 for a simple example.

This way the transition time of a polarized signal from the left end of a cable to its right end will always be the same, no matter whether it is a ⊞ or a ⊟ signal. By convention the ⊞ wire will always be "above" the ⊟ wire of the same cable.

Different cables have to be separated vertically by at least 2 blocks. Polarized signals representing bits on a cable will always advance by one block from left to right during each cycle.

When referring to the inputs and outputs of a gate, we indicate the ⊞ and ⊟ components of a cable with subscripts. For instance, for an input cable x, we write x_+ for its ⊞ and x_- for its ⊟ component.

5.2 Bit Duplication

To duplicate a cable, we use the *Boolean branch* depicted in Fig. 19. The circuit consists of two wire duplications (one of each polarity) and a crossing. Obviously a Boolean branch can be realized by in *constant* size, independently of the Boolen circuit in which they are used.

When laying out a circuit in the plane it is in general necessary to have cables crossing each other. We will use XOR gates for this, and for the implementation of these we will NAND gates which are considered next.

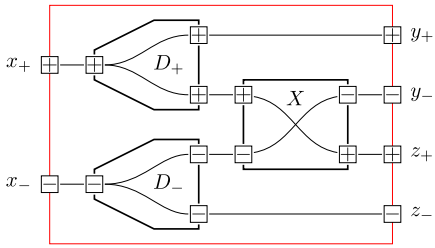


Fig. 19 Boolean branch

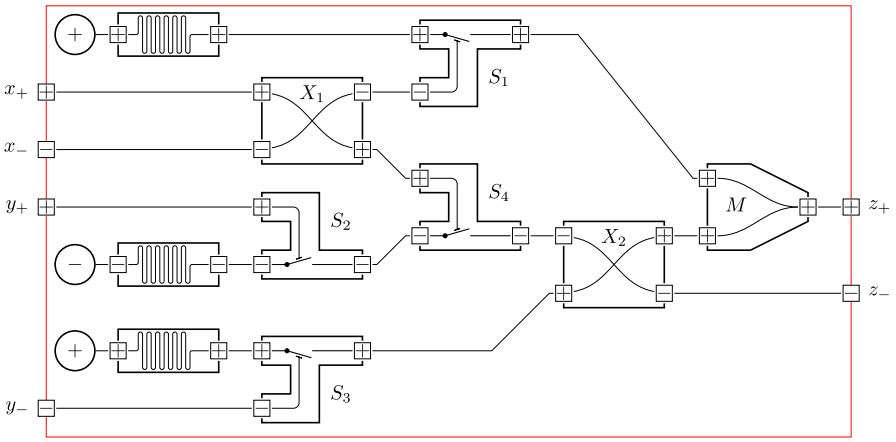


Fig. 20 NAND gate

5.3 NAND Gates

Our NAND gate is inspired by the implementation of such a gate in CMOS technology.[1] Refer to Fig. 20 for a sketch of the implementation.

Note that the four polarized inputs are connected to the gates of four switches. The sources of the switches are fed from retarders (directly or indirectly).

The use of switches implies that a NAND gate is *delay-sensitive*; it only operates correctly (i.e., computes the NAND function) if the retarders have *delays that are strictly greater the arrival times of signals at inputs* x and y of the specific NAND gate. Hence for different NAND gates in the circuit we will need to instantiate the same construction plan using *different delays* of the retarders. Nevertheless the size of all retarders (and hence of all NAND gates) stays the same as mentioned in Sect. 4.5.

We stress that Fig. 20 is only a schematic. First of all, the sizes of the components used are not to scale. On one hand all the switches, semicrossings and the wire merge require only a *constant* number of blocks (independent of the circuit for which they are used). On the other, as already mentioned in Sect. 4, the retarders are squares consisting of $\Theta(L^2)$ blocks where $L = \Theta(m^2)$ and m is the number of gates of the circuit to be simulated. (We defer the derivation of these dimensions to Sect. 6.3.) Nevertheless all retarders used in the layout of one circuit are of *the same size*.

[1] See, e.g., https://en.wikipedia.org/wiki/NAND_gate#/media/File:CMOS_NAND.svg.

Secondly, as a consequence the size of NAND gates used for different circuits also have different sizes in general, but all NAND gates used in the layout of one circuit have the same size, which is $\Theta(L) \times \Theta(L)$ blocks.

Finally, we have claimed that the $\boxplus$ and the $\boxminus$ wire of a cable are separated by exactly two empty blocks. But at least for the y_+ and y_- inputs this is not the case. To fix this and to make the whole layout more uniform, we assume that for the complete layout of a NAND gate, the part shown in Fig. 20 is extended to the left and to the right by appropriate numbers of block columns to achieve the following:

- The output pins z_+ and z_- for the output cable are exactly in the middle of the right border of the full rectangle.
- The input pins x_+ and x_- for the first input cable are at the top of the left border of the full rectangle.
- The input pins y_+ and y_- for the first input cable are at the bottom of the left border of the full rectangle.

It should be clear that the extensions to both sides require at most as many additional block columns as the NAND is high. Thus a full NAND gate requires $\Theta(L) \times \Theta(L)$ blocks.

Since NAND gates are the real core construct, we will now prove:

Claim If the retarders have larger delay than the arrival times at input cables x and y, then the circuit in Figure 20 realizes a NAND *gate.*

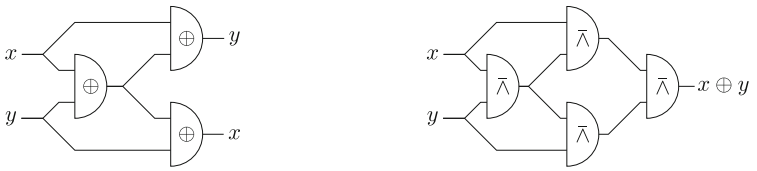
Proof Consider first the case where both x_+ and y_+ are set. Since x_- is not set, X_1 is consumed by x_+, turning S_4 on. In addition, since y_+ is set, S_2 is also turned on. Hence, using the assumption on the delay of the inputs, the negative source flows through S_2, S_4, and X_2 on to z_-. Since both the switches S_1 and S_3 remain open, the z_+ output is never set. Notice the crossings X_1 and X_2 are each used exactly once.

Let now x_- or y_- (or both) be set. Then either S_2 or S_4 is open, which means z_- is never set. As a result, X_2 is used at most once (namely in case y_- is set). If x_- is set, then S_1 is closed, thus allowing the positive source to flow on to M. The same holds if y_- is set, in which case M receives the positive source arriving from S_3. Hence, at least one positive signal will flow to the M gate, causing z_+ to be set eventually. □

5.4 Cable Crossings

There is a more or less well-known idea to cross two bits using three XOR gates which can for example be found in the paper by Goldschlager [8]. Figure 21 shows the idea. Each XOR gate can be implemented using five NAND gates and the layout of the whole circuit can be made planar.

This construction can be used in FA. Because of the delays, there is not *the* crossing gate, but a whole family of them. Depending on the position in the whole circuit layout, each crossing needs NAND gates with specific builtin delays (which will be derived in Sect. 6.3). But the sizes of both, XOR gates and bit crossings, are the same everywhere in the layout of one circuit and they can be realized using $\Theta(L) \times \Theta(L)$ blocks.



(a) Crossing two cables using 3 XOR gates and 3 bit duplications

(b) Implementation of XOR using 5 NAND gates and 4 bit duplications

Fig. 21 Implementation of cable crossings as in [8]

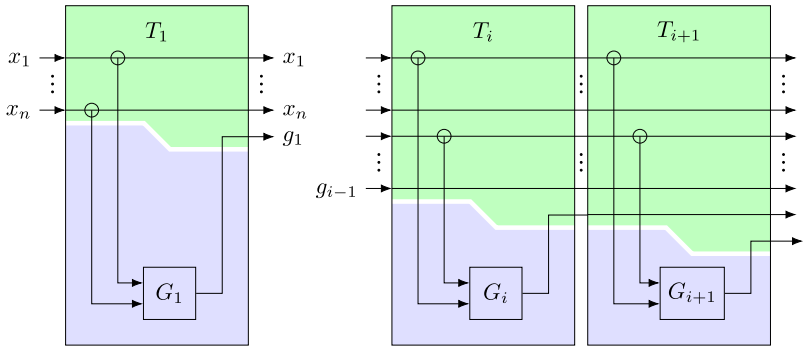


Fig. 22 Overview of the construction of F. The equal-sized tiles each realize one gate of the original circuit C. The meaning of the green and blue areas is given in Sect. 6.2 (Color figure online)

6 Layer 4: Layout of a Whole Circuit

Finally we describe one possibility to construct a finite rectangle F of cells of a FA containing the realization of a complete circuit, given its description B. The important point here is that, in order to produce F from B, the constructor only needs logarithmic space. Therefore the simplicity of the layout has precedence over any form of "optimization".

6.1 Arranging the Circuit in Tiles

Let C be the circuit that is to be embedded as an FA configuration F. Letting n be the number of input bits to C and m its number of gates, there is an upper bound of m on the circuit depth of C. Without restriction, we may assume $m \geq n$, which also implies an upper bound of $O(m)$ on the number of cables (and wires) of C (since all gates have bounded fan-in). The logical gates of C are denoted by $G_1, \ldots, G_m$ and we assume that G_i has number $n + i$ in the description B of C (recall Sect. 1.1).

In the configuration F we have cables $x_1, \ldots, x_n$ originating from the input gates as well as cables $g_1, \ldots, g_m$ coming from (the embedding of) the gates of C. The x_i and g_i flow in and out of equal-sized *tiles* $T_1, \ldots, T_m$, where in the i-th tile T_i we implement the i-th gate G_i of C. (See Fig. 22 for an illustration.) The inputs to T_i are $I_i = \{x_1, \ldots, x_n, g_1, \ldots, g_{i-1}\}$ and its outputs are $O_i = I_i \cup \{g_i\}$; hence $I_{i+1} = O_i$.

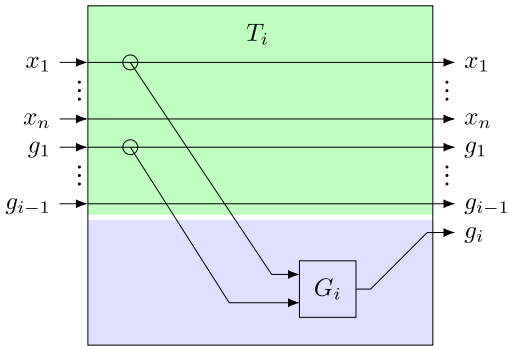


Fig. 23 Overview of the tile T_i. The *upper part* of the tile has green background, the *lower part* has blue background (Color figure online)

Recall that, unlike standard circuits, the behavior of our layer 3 circuits is subject to spatial considerations, that is, to both gate placement and cable length. For the sake of simplicity, each tile is shaped as a square and all tiles are of the same size. In addition, the tiles are placed in ascending order from left to right and with no space in-between. The only objects in F that lie outside the tiles are the inputs and output of C itself. The inputs are placed immediately next to corresponding cables that go into T_1 whereas the output is placed next to its corresponding cable g_m at the outgoing end of T_m.

6.2 Layout for Tile *i*

As depicted in Fig. 23, each tile is subdivided into two areas. The *upper part* contains the cables that pass through it, while the *lower part* implements the gate G_i proper.

We give a broad overview of the process for constructing T_i:

1. First determine the cables $y_1, y_2 \in I_i$ that provide the inputs to G_i.
2. Duplicate the bits on cables y_1 and y_2 (as in Sect. 5.2) and cross the copies over to the lower part of the tile. These crossings use multiple NAND gates and require setting adequate delays, which we will adress in the next section. (In case $y_1 = y_2$, simply duplicate the cable twice and proceed as otherwise described.)
3. Instantiate G_i with a proper amount of delay (again, see the next section) and plug in y_1 and y_2 as inputs into G_i.
4. Finally connect all inputs in I_i as well as the output cable g_i of G_i to their respective outputs.

Notice the tile contains $O(m)$ crossings and thus also $O(m)$ NAND gates in total.

6.3 Choosing Suitable Delays for All Gates

The two details that remain are setting the dimensions and the delays for the retarders in all NAND gates. This requires certain care since we may otherwise end up running into a chicken-and-egg problem: The retarders' dimensions are determined by the required delays (in order to have enough space to realize them); in turn, the delays depend on the aforementioned dimensions (since some input wires in the NAND gates must be laid so as to "go around" the retarders).

The solution is to assume we already have an upper bound D on the maximum delay in F. This allows us to fix the size of the components as follows:

- The retarders and NAND gates have side length $\ell_{\text{NAND}} = O(\sqrt{D})$.
- Each tile has side length $\ell_T = O(m\sqrt{D})$.
- The support of F fits into a square with side length $O(m^2\sqrt{D})$.

With this in place, we determine upper bounds on the delays of the upper gates in a tile (i.e., the gates in the upper part of the tile), then of the lower gates G_i, then of the tiles themselves, and finally of the entire embedding of C. In the end we obtain an upper bound for the maximum possible delay in F. Simply setting D to be this large concludes the construction.

Upper gates In order to set the delays of a NAND gate G in a tile T_i, we first need an upper bound d_{input} on the delays of the two inputs to G. Suppose the origins O_1 and O_2 of these inputs (i.e., either a NAND gate output or an input to T_i) have both delay at most d_{origin}. Then certainly we have $d_{\text{input}} \leq d_{\text{origin}} + d_{\text{wire}}$, where d_{wire} is the maximum wire distance needed to connect the wires of either one of O_1 and O_2 and the respective switches that we are connecting them to inside G.

Let us now say G is in the j-th *layer* of T_i if any path from the inputs I_i to T_i leading to G goes through at most j gates (and j is minimal with this property). (Hence if O_1 originates from a j_1-th layer gate and O_2 from a j_2-th layer one, G will be in the $\max(j_1, j_2)$-th layer.) Now the key observation is that the layout of T_i together with the fact that a NAND gate has $\ell_{\text{NAND}} = O(\sqrt{D})$ side length gives us $d_{\text{wire}} = O(\sqrt{D})$. This is because, by using a sensible placement of the gates (as in, e.g., Fig. 21), we can arrange the upper gates so that the distance between an upper gate in one layer and its successors in the next layer is at most $O(\ell_{\text{NAND}}) = O(\sqrt{D})$. In other words, each layer of NAND gates incurs an $O(\sqrt{D})$ additional delay. Hence, if G is in the j-th layer of T_i, then we may upper-bound its delay by $d_i + O(j\sqrt{D})$, where d_i is the maximum over the delays of the inputs to T_i.

Lower gates Since there are $O(m)$ cables inside a tile, there are $O(m)$ cable crossings and thus $O(m)$ NAND gates realizing these crossings. Recalling that $\ell_T = O(m\sqrt{D})$ is the side length of T (and also the maximum cable length needed to connect the last of the upper gates with G_i), we conclude that the inputs to the gate G_i in the lower part of T_i have delay at most

$$d_i + O(m) \cdot d_{\text{wire}} + O(\ell_T) = d_i + O(m\sqrt{D}).$$

Tiles Clearly the greatest delay amongst the output cables of T_i is that of g_i (since every other cable originates from a straight path across T_i). As we have determined in the last paragraph, at its output g_i has delay $d_{i+1} \leq d_i + O(m\sqrt{D})$. Since the side length of a tile is $\ell_T = O(m\sqrt{D})$, this gives us an upper bound of $i \cdot O(m\sqrt{D})$ on the delays of the inputs of T_i.

Support of F Since there are m tiles in total, it suffices to choose a maximum delay D that satisfies $D \geq cm^2\sqrt{D}$ for some adequate constant c (that results from the considerations above). In particular, this means we may set $D = \Theta(m^4)$ independently of C.

6.4 Constructor

In this final section we describe how to realize a logspace constructor R which, given a CVP instance consisting of the description of a circuit C and an input x to it, reduces it to an instance as in Theorem 1. Due to the structure of F, this is relatively straightforward.

The constructor R outputs the description of F column for column. (Computing the coordinates of an element or wire is clearly feasible in logspace.) In the first few columns R sets the inputs to the embedded circuit according to x. Next R constructs F tile for tile. To construct tile T_i, R determines which cables are the inputs to G_i and constructs crossings accordingly. To estimate the delays of each wire, R uses the upper bounds we have determined in Sect. 6.3, which clearly are all computable in logspace. (Recall, in particular, that the maximum delay $D = \Theta(m^4)$ is polynomial in m.)

Finally R also needs to produce y and T as in the statement of Theorem 1. Let c_i be the cable of T_m that corresponds to the output of the embedded circuit C. Then we let y be the index of the cell next to the $\boxplus$ wire of c_i at the output of T_m. (Hence y assumes a non-zero state if and only if c_i contains a 1, that is, $C(x) = 1$.) As for T, certainly setting it to the number of cells in F suffices (since a signal needs to visit every cell in F at most once).

7 Summary

We have shown that, for fungal automata with update sequence HV, the prediction problem is P-complete, solving an open problem of Goles et al. [5].

Acknowledgements Augusto Modanese is supported by the Helsinki Institute of Information Technology (HIIT). Much of this work was done while he was affiliated with the Karlsruhe Institute of Technology (KIT).

Author Contributions A.M. and T.W. wrote the main manuscript, prepared all figures and reviewed the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Bak, P., Tang, C., Wiesenfeld, K.: Self-organized criticality: an explanation of the $1/f$ noise. Phys. Rev. Lett. **59**(4), 381–384 (1987). https://doi.org/10.1103/PhysRevLett.59.381
2. Formenti, E., Perrot, K.: How hard is it to predict sandpiles on lattices? A survey. Fundam. Inform. **171**(1–4), 189–219 (2020). https://doi.org/10.3233/FI-2020-1879
3. Miltersen, P.B.: The computational complexity of one-dimensional sandpiles. Theory Comput. Syst. **41**(1), 119–125 (2007). https://doi.org/10.1007/s00224-006-1341-8
4. Moore, C., Nilsson, M.: The computational complexity of sandpiles. J. Stat. Phys. **96**(1), 205–224 (1999)
5. Goles, E., Tsompanas, M.-A.I., Adamatzky, A., Tegelaar, M., Wosten, H.A.B., Martínez, G.J.: Computational universality of fungal sandpile automata. Phys. Lett. A **384**(22), 126541 (2020). https://doi.org/10.1016/j.physleta.2020.126541
6. Modanese, A., Worsch, T.: Embedding arbitrary boolean circuits into fungal automata. In: Castañeda, A., Rodríguez-Henríquez, F. (eds.) LATIN 2022: Theoretical Informatics—15th Latin American Symposium, Guanajuato, Mexico, November 7–11, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13568, pp. 393–408. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-20624-5_24
7. Ruzzo, W.L.: On uniform circuit complexity. J. Comput. Syst. Sci. **22**(3), 365–383 (1981). https://doi.org/10.1016/0022-0000(81)90038-6
8. Goldschlager, L.M.: The monotone and planar circuit value problems are log space complete for P. SIGACT News **9**(2), 25–29 (1977). https://doi.org/10.1145/1008354.1008356
9. Dymond, P.W., Cook, S.A.: Hardware complexity and parallel computation (preliminary version). In: 21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13–15 October 1980, pp. 360–372. IEEE Computer Society, Syracuse (1980). https://doi.org/10.1109/SFCS.1980.22
10. Gajardo, A., Goles, E.: Crossing information in two-dimensional sandpiles. Theor. Comput. Sci. **369**(1–3), 463–469 (2006). https://doi.org/10.1016/j.tcs.2006.09.022