Vants and Turmites

Greg Turk
Georgia Institute of Technology
School of Interactive Computing
turk@cc.gatech.edu

Abstract The two-dimensional Turing machine is a promising but under used simulation tool for Artificial Life. Single-state 2-D Turing machines exhibit a variety of interesting behaviors, some of which have already been explored. Multistate 2-D Turing machines, despite their potential for simulating even more diverse behaviors, have received little attention to date. We demonstrate the potential of such automata for studying biological phenomena by showing how they can be used to simulate self-similar growth, the spread of disease, and self-reproduction. Some of the results presented here are from investigations that were performed around the time of Dewdney (1989), but they have not been published until now.

Keywords
Simulation techniques, self-reproduction, Turing machines

1 Introduction

In this work, we explore the use of 2-D Turing machines (Dewdney, 1989; Langton, 1986) as a simulation tool for the study of biological phenomena. The investigations reported in this article were initiated by the author many years ago, and some of the more simple discoveries were communicated to Dewdney (1989).

Computer simulation is a commonly used approach to the study of Artificial Life. A simulation consists of a collection of units (e.g., molecules, cells, or organisms) that interact with one another according to a set of rules that specify how the state of each unit is updated at each time step. Frequently used simulation techniques in ALife include cellular automata, character string interactions, point masses interacting through forces, and numerical integration of partial differential equations on a grid. We suggest that 2-D Turing machines are another simulation method that can be used for ALife studies.

A classical Turing machine consists of an infinite one-dimensional strip of cells, each of which contains a symbol, and a tape head that is positioned along the tape (Turing, 1936). At a given time step, the tape head reads the current symbol on the tape and also consults its own internal state. Together, the symbol and state specify a particular transition rule that causes a symbol to be written on the tape, sets the new internal state of the head, and causes motion of the head either left or right. Instead of a one-dimensional tape, a 2-D Turing machine consists of an infinite, 2-D grid of cells, together with a tape head. Not only is the tape in a particular state but it also has a current direction. The transition rules (guided by the cell symbols, the head's state, and its current direction) cause changes to the symbols on the grid and to the internal state, and they also move the head to an adjacent cell of the grid.

We distinguish between two types of 2-D Turing machines, which we refer to as *vants* and *turmites*. We use the term *vant* to indicate a 2-D Turing machine with a head that can be in only a single state. This term was coined by Langton (1986), who (to the best of our knowledge) was the first to study such constructs. We use the term *turmite* to indicate a 2-D Turing machine with a head that can be in two or more different states. The name "turmite," coined by Dewdney (1989), both highlights the 2-D turning actions of such machines and also recognizes the foundational work of Alan Turing.

Several styles of research may be carried out with 2-D Turing machines. The naturalist approach is to observe machines' behaviors, to categorize various machines based on these behaviors (e.g., random motion, path builder, symmetric patterns), and to understand why they act as they do. Most of the early work with the single-state vants is in this style. Another research style is design, whereby a machine is constructed to simulate a particular phenomenon. Many of the turmites that we present here are purpose-built in this manner. Another research approach is to use optimization or artificial evolution to discover 2-D Turing machines that have particular properties. We do not explore this third research avenue, but we believe it is an intriguing possibility for future work. See section 2 for early work in this direction by Rucker.

In our work, we highlight three techniques that can be incorporated into a 2-D Turing machine to enhance its behavior. First is the use of four different turning directions (forward, backward, left, and right) instead of only turning right or left. Second is the use of Turing machines that take on more than one state (turmites). Third is the ability of one Turing machine to spawn another. Each of these techniques has been mentioned in prior work; however, the ramifications of using these characteristics have not been sufficiently explored to date. We demonstrate the expressive power of these simulation characteristics by presenting simulation of self-similar growth, infectious disease spread, and self-reproduction.

The remainder of the article is organized as follows. After reviewing related work, we turn to single-state 2-D Turing machines (vants) and a description of some of their behaviors. Then, we demonstrate the extended capabilities of 2-D Turing machines when they can take on multiple states (turmites). Next, we present the design of a 2-D Turing machine that is capable of universal computation. Finally, echoing von Neumann's (1966) classic construction, we extend the universal turmite to include a constructor arm and demonstrate self-reproduction. As it is based on 2-D Turing machines instead of cellular automata, our self-reproducing machine differs in many ways from those of von Neumann (1966) and Codd (1968).

2 Related Work

Alan Turing (1936) invented a class of idealized machines (now called *Turing machines*) to investigate decision procedures for mathematical proofs. Turing was searching for the answer to David Hilbert's 10th problem, namely, to find a decision procedure (*Entscheidung*) that can determine in a finite number of steps whether any given Diophantine equation with integer coefficients is solvable. (A transcript of David Hilbert's famous 1900 speech can be found in Hilbert, 1976.) Turing used his hypothetical machines to prove that in fact no such decision procedure is possible.

Work by Hartmanis and Stearns (1965) examined the possibility of Turing machines with multiple tapes and Turing machines with heads that move on a 2-D grid. They proved that such generalizations to Turing machines are no more powerful in terms of the class of problems they can solve. However, they also demonstrated that such variations can indeed sometimes reduce the number of steps needed to solve a given problem.

Michael Paterson and John H. Conway carried out a set of experiments on graph paper that were intended to mimic the pattern of the tracks of a worm that is crawling on a plane. Such a worm travels along the edges of a grid and decides whether to turn left, turn right, or go straight when it reaches an intersection. A worm is not allowed to travel along any edge more than once and will die (halt) if it reaches an intersection where there are no longer any open edges. Paterson and Conway do not seem to have published their investigations, but Beeler's (1973) technical report makes an exhaustive study of all such worms as they travel on a triangular grid. Beeler's work is in the spirit of a naturalist who is studying the behaviors of different organisms. Gardner (1973) popularized these worms in his Mathematical Games column.

Langton (1986) described a variety of ways in which different forms of automata might be used in Artificial Life studies. Although his paper calls out cellular automata in its title, several sections (sections 9, 10, and 11) are devoted instead to Turing machines that are moving on a 2-D square

grid. One such class of automata that he describes comprises *vants*. In particular, he describes the **RL** vant that turns right at a cell that it has visited an even number of times and left when it has visited the cell an odd number of times. We will return to the **RL** vant in the next section. We note that Beeler's (1973) first diagram shows a worm that crawls along a square grid, and turns right the first time it reaches an intersection and left when it revisits an intersection. The first eight steps of this worm carry out the exact same path as Langton's **RL** vant. At Step 8, Beeler's worm dies because it reaches an intersection where there are no unvisited edges, whereas the vant makes a right turn and continues on.

Dewdney (1989) presented several 2-D Turing machines that were described to him by Turk. Unaware of Langton's prior work, the first example Turk showed him was the **RL** vant. Also described (but without an accompanying figure) was a two-state Turing machine that produced a spiral pattern. We return to this example in section 4, on 2-D Turing machines with multiple states. Another example from this article is the **RRLL** vant, which exhibits bilateral symmetry. A series of articles by Gale presented investigations by Propp and others that explored this symmetric vant and other vants that always turn either left or right (Gale, 1993, 1994, 1998; Gale et al., 1995). Under the name lattice gas automata, vants have also been investigated in the physics community. Kong and Cohen (1991a, 1991b) published work on such automata both for square and for triangular grids, and Bunimovich and Troubetzkoy (1992) also studied these models. The two-dimensional version of the Busy Beaver problem, which looks for Turing machines that take the most steps (or writes the most symbols) before halting, has been investigated (Hutton, 2015).

Rucker (1993) created a software lab that includes the ability to artificially evolve turmites (or *boppers*). The simulator uses a fitness function that rewards a turmite that moves through a food source, and it is penalized when it moves in an area that has been seeded with poison. He reports that "the boppers' scores tend to improve up to a certain point, and then do not get much better" (p. 101) and that "it is uncertain whether good evolution happens in TURMITE" (p. 271).

In section 6 we demonstrate a 2-D Turing machine that is capable of self-reproduction. Computational models that carry out self-reproduction were pioneered by John von Neumann. Based on a suggestion from Ulam, he used cellular automata as the basis for his construction of a self-reproducing machine (von Neumann, 1966). The cellular automata that von Neumann described made use of a square grid whereby each cell's transition rule was based on its own state and those of its four neighbors (now known as the von Neumann neighborhood). His automaton had 29 states and carried out computation using "pulses" that traveled along virtual wires and could interact with one another to carry out logical operations. These pulses could also control a construction arm that was capable of laying down new patterns of cells. von Neumann's construction was a major landmark in Artificial Life, but it was also extremely complicated, and his book outlining the machine (completed with help from Burks) was 388 pages long. von Neumann's book gave a description of the components of his machine but did not describe the machine in its entirety. The details of his machine were worked out much later by Pesavento (1995).

Noting the complexity of von Neumann's machine, Codd (1968) set out to demonstrate self-reproduction using a more simple set of cellular automata rules. Key to Codd's construction was to propagate pulses along sheathed pathways, which allowed him to use just 8 states instead of von Neumann's 29. Like von Neumann, Codd did not present the full pattern for his self-reproducing machine. Hutton (2010) presented the first demonstration of Codd's machine in full. In the same paper, Hutton also presented two additional self-replicators that were variations on Codd's approach, due to Devore and Goucher.

One of the most complex aspects of completing von Neumann's original plan for a self-reproducing machine is the complex timing that needs to be worked out for the tape reader. Thatcher (1964) addressed this issue with a modification of the tape reader. This work appeals to the idea of a Turing machine that reads instructions from a 1-D tape, some of which directs the machine to move a construction arm over a 2-D grid of cells and write symbols on this grid. Thatcher did not in fact separate the Turing machine from the grid but instead embedded a construction similar to such a Turing machine in the cells of a cellular automaton, and this cellular

automaton made use of the same 29 symbols that von Neumann used. The "one instruction at a time" nature of Thatcher's virtual Turing machine gets around the need for signals to cross in this construction.

Both von Neumann's and Codd's self-reproducing machines use components that can be used to build a universal computer. Langton (1984) demonstrated that self-replication does not require the ability to carry out computation. Using Codd's eight-state rules, and in particular using the sheathed signal pathways, he created a simple self-reproducing machine that dispenses with computation. His self-reproducing loops encode the shape of the loop and its simple constructor arm in a short sequence of pulses that circulate around in a loop. A number of interesting variations of self-reproducing loops have been investigated, including more compact loops (Byl, 1989), loops that do not require sheaths (Reggia et al., 1993), loops that carry program payloads (Perrier et al., 1996), and loops that evolve (Sayama, 1999).

Self-reproduction has been studied in forms other than cellular automata. Dewdney (1984) introduced a game called Core War in which programs competed with each other for space. This in turn inspired Ray's (1992) self-replicating and evolving programs of Tierra. Hutton (2002) demonstrated self-reproducing molecules on a grid and extended this work to self-reproducing cells (Hutton, 2004). Sipper (1998) reviewed the literature on self-replication, covering many of the topics listed herein.

Because of the prevalence of cellular automata in the Artificial Life literature, we have included Appendix 1, which shows that cellular automata and turmites can mimic each other.

3 Vants: Single-State Machines

A classical Turing machine is described by three components: (a) the symbols on a 1-D tape, (b) a *head* that is centered on one symbol on the tape and that maintains an integer state, and (c) a rule table that describes the action of the head. Two-dimensional Turing machines are similar to their 1-D cousins, but the head moves across a 2-D grid. We first give formal definitions to these components and then turn to investigating specific instances of these machines.

We will make use of a 2-D square grid G that is infinite in all four compass directions: north, east, south, and west. A given position on the grid is specified by two integers i and j. Each cell in the grid contains exactly one symbol from a list of K symbols $Q = \{\mathbf{a}, \mathbf{b}, \ldots\}$. We will use lowercase characters to specify the symbols that may appear in a cell. In many of our examples, we assume that the grid starts out empty, that is, with cells that all contain the symbol **a**. When displaying a grid of cells in a figure, we frequently use colors rather than letters to indicate the cell symbols.

A 2-D Turing machine also has a head. The head H maintains three pieces of information: its position p, its direction d, and its state s. The position is a pair of integers i and j that specify the location of the head on the grid. The direction is one of four integers 0, 1, 2, or 3 that represent the compass directions listed in clockwise order: north, east, south, and west, respectively. The current state s of the head is one of N possible states $S = \{1, 2, \ldots N\}$. To fully describe the initial state of a 2-D Turing machine, the head's initial position, direction, and state must be specified.

The final component of a 2-D Turing machine is the transition function (or rule table). This function F describes how the grid is modified and the head's information is changed at a given time step. The transition function takes as input the current machine state $s \in S$ and the symbol $c \in K$ of the current cell. As output, it specifies the symbol to be written at the current cell, the new state of the machine, and also the action, which is a direction to turn: right (R), left (L), forward (F), or backward (B). We include one more possible action H that specifies that the machine should halt. Formally, the transition function has the form $F : S \times Q \rightarrow \{R, L, F, B, H\} \times S \times Q$.

Taken together, the grid, head, and transition function fully describe a 2-D Turing machine: $T = \{G, H, F\}$. We often use the term *vant* or *turmite* to refer to such a machine. When it is unambiguous to do so, we sometimes use one of these terms to refer to the head of a given machine.

The full configuration of the 2-D Turing machine is updated at a given time step based on the particular entry of the rule table F for the machine. Following are the steps that are taken to update the machine:

1. Based on the head state s and the symbol c of the cell at the head's current position, look up in the rule table the next state s_{new}, symbol c_{new}, and turning amount t. If the action is H rather than a turning amount, halt the machine.
2. Change the symbol at the head's current position to c_{new}.
3. Modify the head's direction based on the turning amount t, and move one cell forward in this new direction.
4. Update the head's state to be s_{new}.

The 2-D Turing machines that we consider move according to *relative* rather than *absolute* directions. That is, the head's direction is updated relative to the current direction of the head. (See Appendix 2 regarding the possibility of using absolute directions.) The actions $\{R, L, F, B\}$ that cause a head to change direction can be encoded by a turning amount t that is one of the four possible values 1, 3, 0, or 2, respectively. With this direction encoding, the new direction of the head is given by $d_{\text{new}} = (d_{\text{old}} + t) \bmod 4$. Consider the case when the head's current direction is north ($d_{\text{old}} = 0$) and when the turning direction is a right turn R so that $t = 1$. Then, the new direction will be east, according to the update $d_{\text{new}} = (d_{\text{old}} + t) \bmod 4 = (0 + 1) \bmod 4 = 1$.

We begin our exploration of 2-D Turing machines by first looking at a subset of these machines that take on only a single state. We refer to such single-state machines as *vants* (short for *virtual ants*), the name given to them by Langton (1986) in his survey of various grid-based machines. In section 4 we examine multistate 2-D Turing machines.

Let us examine the original 2-D Turing machine that Langton (1986) described. When this machine encounters an empty (white) cell, it changes the cell to red and turns right. When it arrives at a red cell, it changes the cell to white and turns left. Like all vants, this machine has only a single state ($N = 1$). Each cell in the grid may contain one of two symbols, **a** (white) or **b** (red), so $K = 2$. Figure 1 shows the transition function for this vant. In this and subsequent transition tables, each numbered row is for one state, and each column entry is for one symbol. The table entry "Rb1" that is in the column labeled with **a** specifies that when the machine is in state 1 and is on a cell with symbol **a**, the next action will be to turn right, the symbol for the current cell will be changed to **b**, and the next state will be 1.

Figure 2 shows two stages of the pattern that this vant creates when starting on an entirely empty grid of cells (all set to symbol **a**). Initially, this vant constructs a pattern of cells that has twofold rotational symmetry (Figure 2(a)). At a certain point, however, this symmetry is broken, and the pattern appears to be random. Eventually, however, the vant creates a periodic highway (right) and will continue to extend this highway indefinitely.

Propp (cited in Gale, 1994; Gale et al., 1995) showed how the transition function for many vants can be expressed more succinctly than is shown in Figure 1. First, because vants have only a single state, the next state does not need to be specified. Second, note that the symbol transitions for this particular vant form a cycle, **a** → **b** and **b** → **a**. For vants whose symbol transitions form a cycle, we can simply specify the actions for each symbol and let the symbols themselves be implied. Thus the rule string **RL** entirely specifies the same transition function as given in Figure 1.

	Symbols	
States	**a**	**b**
1	Rb1	La1

Figure 1. Transition function f for Langton's vant.

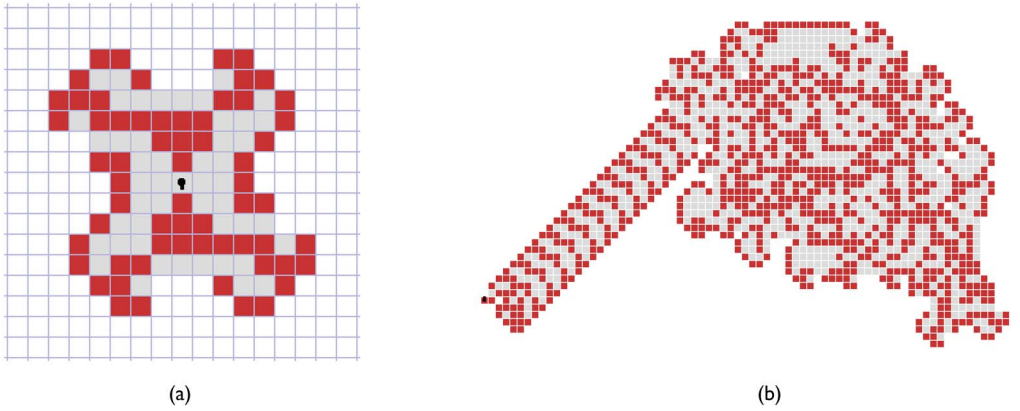


Figure 2. Two stages of the pattern produced by the **RL** vant. The vant turns right at a white cell (symbol **a**) and left at a red cell (symbol **b**). Light gray cells indicate cells with symbol **a** that have been visited.

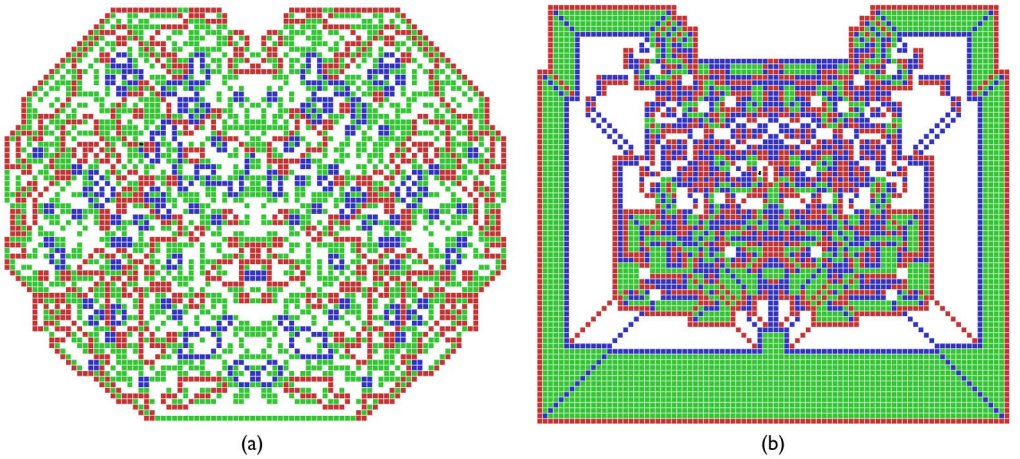


Figure 3. Patterns created by the vants (a) **RRLL** and (b) **RLLR**. The behavior of the vants causes these patterns to return to perfect bilateral symmetry repeatedly.

Whereas Langton's vant with rule string **RL** eventually breaks the symmetry of its early patterns, some other vants construct strictly symmetric patterns indefinitely. Turk noted the bilateral symmetry of the four-symbol vants with rule string **RRLL** and **RLLR** (Dewdney, 1989). Figure 3 shows the patterns created by these two vants. Rummler (cited in Gale, 1994) found that this symmetry-building behavior could be explained using Trunchet tiles, and Propp (cited in Gale et al., 1995) expanded on this approach in his studies.

As indicated earlier, our 2-D Turing machines include the actions to move forward or backward, in addition to making right turns. These forward and backward actions do not appear to have been explored much in prior work. We advocate including these actions to widen the possible behaviors of these machines. As a specific example, consider the vant with rule string **RF**. This vant turns right the first time it visits a cell, but then it goes forward on the second visit to the cell. This vant counts in binary, where the symbols **a** and **b** represent the digits 0 and 1, respectively. It creates a path that is two cells wide, but that is ever increasing in length. The growing pattern shows mirror symmetry every time the vant returns to its starting position (see Figure 4). By the same mechanism, vants with rule strings **RRF** and **RRRF** are base-3 and base-4 counters. We incorporate binary counters into our self-reproducing machine (section 6).

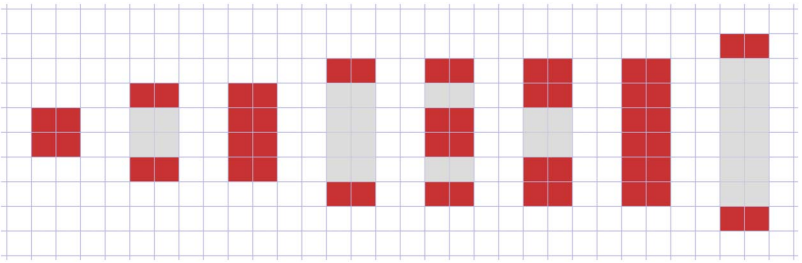


Figure 4. Binary numbers from 1 to 8. Each two-column-wide pattern shows a different time snapshot of the vant **RF**.

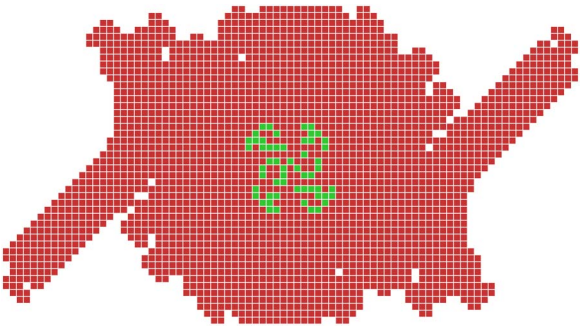


Figure 5. Pattern created by the vant **(B)RL**.

Up until now, we have considered only vants whose symbol transitions form a cycle. We can broaden the behavior of these one-state machines further if we do not require the cell's symbols to form a complete cycle. In such rule strings, we use parentheses to indicate the first symbols that are not part of the cycle of symbols. For example, the rule string **(B)RL** indicates that the vant turns backward upon its first visit to a cell but, from that point on, alternates between right and left turns on subsequent visits to this same cell. This vant creates a twofold, rotationally symmetric pattern, as shown in Figure 5. In fact, this vant recapitulates the motion of the **RL** vant repeatedly in two rotated copies (compare to Figure 2). The vant builds on this symmetric pattern in several steps:

1. Traverse a pathway from the center of the pattern to its boundary.
2. Add one newly visited cell to the pattern.
3. Turn around and retrace the path back to the center.
4. Follow a similar (reflected) pathway to reach the boundary again.
5. Add one more newly visited cell to the pattern.
6. Return to the center to start the process over again.

We will later make use of path-following behavior similar to that of the **(B)RL** vant when we create a machine that exhibits self-similar growth.[1]

4 Turmites: Multistate Machines

The vants described in the previous section demonstrate a number of intriguing behaviors, including counting, quasi-random walks, and path following. Such behaviors may be useful components for

1 Please watch the video at https://github.com/gturk1/turmites to see these vants simulated.

	Symbols	
States	**a**	**b**
1	Lb1	Fa2
2	Rb1	Rb1

Figure 6. Transition table for creating the spiral.

biology simulation, if only they could be assembled together in a deliberate fashion. Unfortunately, the single-state nature of the vant means that it has no internal memory that allows it to switch from one behavior to another. The multistate nature of *turmites*, however, allows for us to design machines that can simulate a variety of behaviors.

4.1 A Simple Turmite

We begin our study of turmites by looking at Turk's two-state, two-symbol machine (Dewdney, 1989). The simple rule table for this turmite is given in Figure 6. This turmite follows a path outward from its starting position and leaves a box-like spiral pattern in its wake (see Figure 7). This turmite traverses square regions of ever-increasing size, with each square larger than the previous one roughly by a factor of the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$. This boxy spiral pattern is closely related to the Fibonacci spiral, in which the increase in size of the squares is based on the Fibonacci sequence. In a true Fibonacci spiral, each square contains a quarter circle, but this turmite pattern instead has a diagonal line that runs through each square. Many of the spiral structures found in plants are similarly related to the Fibonacci sequence, a phenomenon known as spiral phyllotaxis (Fowler et al., 1992).

The spiral-making turmite of Figure 7 was discovered, not hand designed. It is indeed possible, however, to hand design turmites to create particular patterns. The process of creating geometric patterns is aided by the ability for a turmite to switch between multiple states, thus allowing the turmite to carry information from one location to another.

4.2 Dragon Curve

Consider the fractal known as the dragon curve (Davis & Knuth, 1970; Tabachnikov, 2014), which has a self-similar form like some serrated leaves and ferns. The dragon curve can be physically

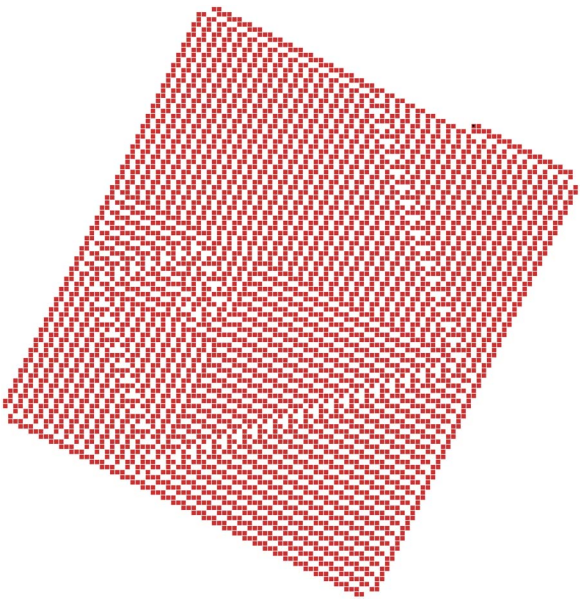


Figure 7. Spiral pattern created by a turmite.

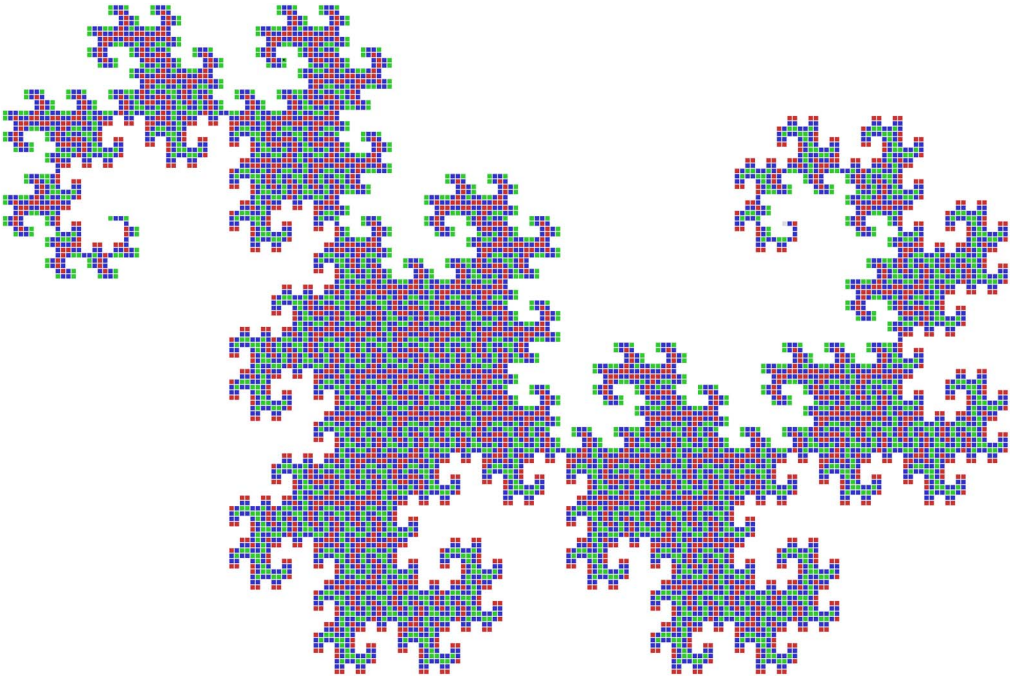


Figure 8. Dragon curve that is created by a turmite by repeatedly extending a single nonintersecting path. The construction of this pattern starts at the center of the spiral near the upper right. Red and green cells indicate right and left turns along the path, respectively, and blue cells indicate straight portions.

constructed by starting with a long, thin strip of paper, repeatedly folding it in half lengthwise (e.g., four or five times), then unfolding the strip so that each crease is a right angle. The sequence of right and left creases is a complex recursive pattern. We present a hand-designed turmite that creates a dragon curve. The behavior of this turmite is similar to that of the vant with rule string **(B)RL** in that they both repeatedly traverse long paths of cells according to the turns that are indicated by the symbols at each cell.

The dragon curve is one non-self-intersecting path of cells (see Figure 8). The right and left turns and the straight pieces of the dragon curve are encoded in the three possible symbols along its path (**b**, **c**, and **d**, respectively). The symbol **a** indicates an empty cell. The turmite extends the existing path *P* by adding one right turn to the path, then appending a flipped replica of path *P*, which roughly doubles the path length. The replica of the existing path is created by copying one cell at a time from the path *P*. The turmite keeps track of which symbols along the path it has already copied by using two variations of each symbol (unmarked, **bcd**, and marked, $\mathbf{b'c'd'}$) that indicate right, left, or straight portions of the path.

Figure 9 shows two stages of the dragon curve growth. The left portion shows the turmite copying one symbol to the growing end of the path. The diagram indicates the turmite moving forward toward the fixed end of the curve. When it arrives at a marked $\mathbf{d'}$ cell, it changes the cell to **d**, then moves back toward the growing end of the pattern. Arriving at the symbol **a** indicates that it has reached the growing end, at which point it writes the symbol **d**. The right part of the diagram continues the sequence of steps, where the turmite follows the path all the way to the fixed end. It encounters the symbol **a** that indicates that it has reached the fixed end, turns around, and marks all of the cells it encounters as it moves toward the growing end. When it reaches the growing end, it adds a new right turn to the path by changing the **a** to **b**. The turmite then returns to copying marked cells to the growing end.

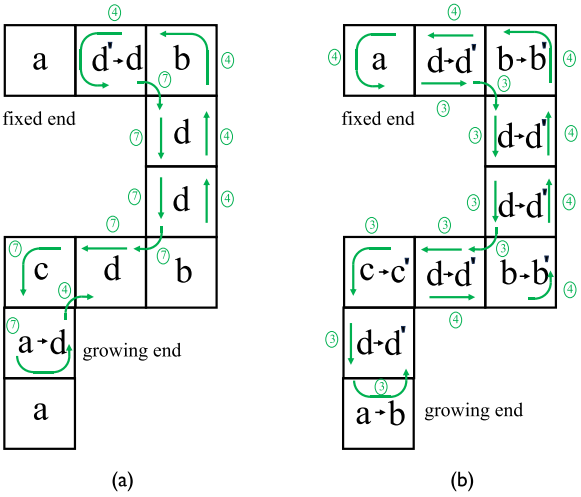


Figure 9. Dragon curve building. (a) Finding and copying symbol **d**. (b) Reaching fixed end and then marking all cells along the path.

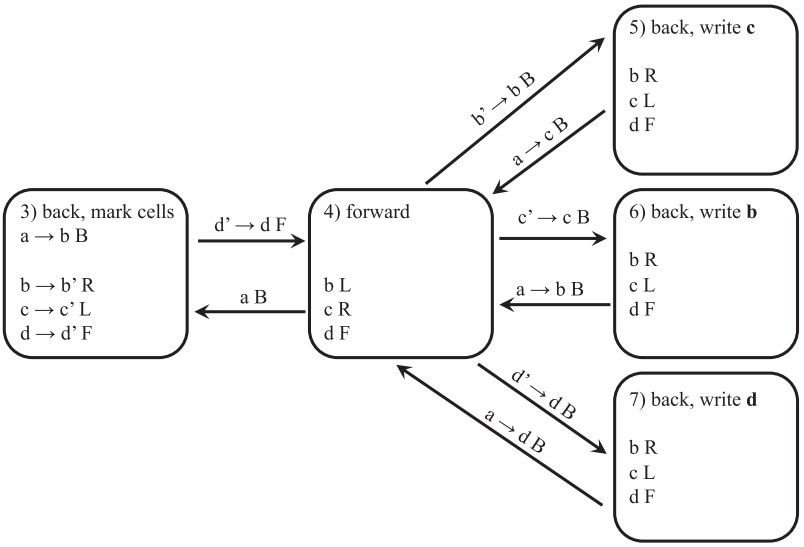


Figure 10. Transition graph for the dragon curve.

We can use a graphical representation to specify this turmite's behavior as it grows the pattern one cell at a time. In Figure 10, each rounded box in the transition graph represents one of the states of the dragon curve turmite. Each such box is given a state number in the upper left, followed by a brief description of the purpose of the state. When a symbol is encountered that will not be changed, this is represented by the symbol followed by the direction the turmite will turn. For example, in State 4, the line **bL** means the symbol **b** will be left unchanged and the turmite will turn left. If, however, a symbol will be changed, this is indicated by an arrow. For instance, in State 3, the turmite changes symbol **a** to **b**, then moves backward, which is indicated by the line **a → b B**. In the case when encountering a given symbol will cause the state to change, this is indicated by an arc that connects the box of one state to the box of another. Each such arc is also annotated to indicate whether the symbol is to be changed and by the turning of the turmite. For example, when

States	Symbols a	b	c	d	b'	c'	d'	Comments
1	Fd'2							initialize cell pattern
2	Rb7							initialize cell pattern
3	Bb3	Rb'3	Lc'3	Fd'3			Fd4	mark cells as not yet copied (**bcd** → **b'c'd'**)
4	Ba3	Lb4	Rc4	Fd4	Bb5	Bc6	Bd7	traverse forward until reaching marked **b'c'd'**, convert them to be unmarked (**b'c'd'** → **bcd**), then turn around and remember which one was seen
5	Bc4	Rb5	Lc5	Fd5				traverse back and write **c** (left)
6	Bb4	Rb6	Lc6	Fd6				traverse back and write **b** (right)
7	Bd4	Rb7	Lc7	Fd7				traverse back and write **d** (forward)

Figure 11. Transition table for creating the dragon curve.

the turmite is in State 4 and encounters the symbol **a**, the turmite moves backward and changes to State 3.

In the transition graph for the dragon curve (Figure 10), States 1 and 2 are not shown because they are used only to place the initial two symbols. State 3 is used to mark cells by changing **bcd** to **b′c′d′**. State 4 is the main workhorse of the turmite. When in this state, the turmite travels along the dragon curve's path toward the fixed end until it reaches a marked cell with one of the symbols **b′c′d′**. When it does, it unmarks the cell and then switches to one of the states 5, 6, or 7, depending on which marked cell it has reached. Each of these states causes the turmite to backtrack along the curve toward the growing end until it reaches an empty cell **a**, at which point it writes one of the symbols **bcd**.

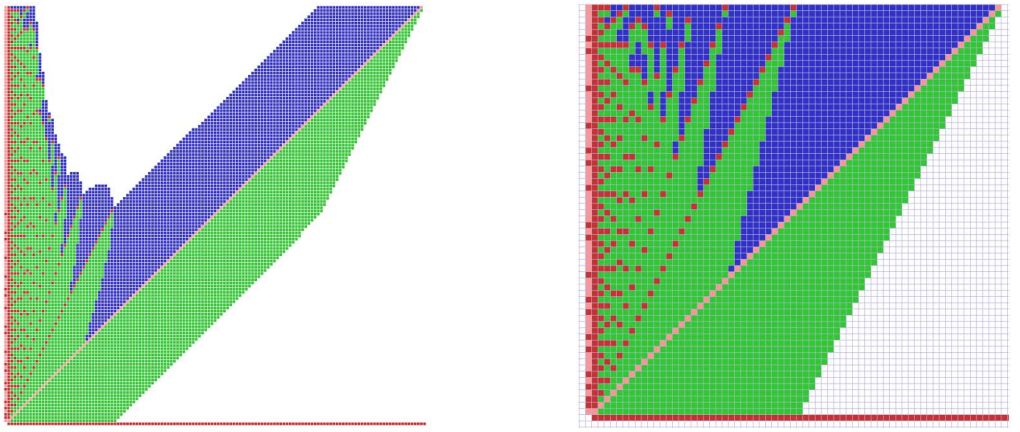
When specifying the behavior of a turmite, creating such a transition graph is typically easier than writing down the transition table directly. We made use of such transition graphs when authoring most of the turmites presented in this article. It is often convenient to create multiple transition graphs for the different operations of a single turmite. For example, when creating a counter, we might use two separate transition graphs to specify the turmite's behavior when it performs an increment or a decrement. Although these graphical representations are usually easier to follow than a transition table, they take up considerably more space on a page. Thus, owing to space considerations, we show only this one transition graph example for the dragon curve.

The transition table for the dragon curve is given in Figure 11. Empty rule entries in this table are never encountered when the machine is started on an empty grid. Transition tables are compact, but transition graphs are more convenient for authoring a turmite.

4.3 Multiple Turmites and a Prime Number Sieve

Up until this point, we have considered only the behavior of a single turmite on a grid. Just as was investigated with the original **RL** vants (Langton, 1986), we can explore the possibility of having more than one turmite on a grid at the same time. When simulating multiple turmites on a grid, we need to decide what happens when two or more different turmites move to the same grid position. In such a case, it is possible that the turmites will try to write different symbols at the cell. A consistent rule is needed for determining which symbol is written. Several possibilities give simulation repeatability: (a) Halt all of the turmites at the shared cell, (b) leave the symbol at the cell unchanged, or (c) change the symbol at the cell only if all the turmites "agree" on the symbol change. We use the last of these rules in our simulations.

By allowing multiple turmites to be active at one time, they can more easily accomplish tasks that would be complicated for a single turmite. We illustrate this by using turmites to calculate prime numbers using a sieve. A prime number sieve is an elegant, grid-based procedure for finding primes. The construction starts by placing marks in each cell in Column 1, marking every other cell in Column 2, every third cell in Column 3, and so on. The second mark along each column lies on a 45° diagonal of marked cells. A number is prime if there is a single marked cell to the left of that number's diagonal (at Column 1).



(a) (b)

Figure 12. Two snapshots of a prime sieve that is created by the work of multiple turmites. (a) A zoomed-out view of the process, in which some primes have not yet been marked. (b) A close-up in which the primes up to 61 can be seen marked in the leftmost column of cells. These numbers should be read from the bottom of the column, with the adjacent primes 2 and 3 indicated by the two lowest red cells.

Figure 12 shows the pattern of a prime number sieve that has been created by a 13-state turmite. The primes can be read off by counting from the bottom of the cells at the west edge of the pattern, and dark red cells (**b**) represent prime numbers.

Figure 13 shows the transition table for the turmites that create the prime number sieve. Spawning of a new turmite head is indicated in the table by square brackets that contain a turning amount *t* and a state *s*. The newly spawned turmite turns in this direction *t* and is placed in state *s*. Note that often the newly spawned turmite turns in a direction different from the direction of its "parent" turmite.

The one initial turmite for the sieve starts at the southwestern end of the pattern on a grid of empty cells. One of the turmites that it spawns creates the row of red cells (symbol **b**) running from west to east. Another turmite travels north to create a column of pink cells (symbol **e**) along the western edge of the pattern. A third turmite places pink cells along the diagonal. This diagonally traveling turmite also spawns one additional turmite per column, and these turmites place additional marked cells (symbol **b**) in the northern direction, spaced out according to the distance between the diagonal cell and the bottommost marked cell. This forms the characteristic pattern of cells for a sieve.

	Symbols					
States	**a**	**b**	**c**	**d**	**e**	Comments
1	Fa2 [F4]	Fb6		Fd1	Fe6	create **e**'s on diagonal and row of **b**'s along bottom
2	La3					make diagonal **e**
3	Re2 [B1]					make diagonal **e**
4	Fb5 [L9]					make column of **e**'s on left
5	Fb5 [L9]					make row of **b**'s on bottom
6	Bc7	Bb8	Fc6	Bc7	Be8	place spaced out **b**'s in a column (every 2, 3, 4...)
7	Bd1	Fb7	Fc7	Fd7	Fe7	space **b**'s
8	Bd6	Fb8	Fc8	Fd8	Fe8	space **b**'s
9	Fe9 [R10]	Fa9				create per-row heads for identifying primes
10	Ba12	Fb11			Fe10	skip over **b** and **e** at the far left of a row
11	Ba12	Hb11	Fc11	Fd11	Be13	move east, halt on **b** (identified composite number)
12	Ba10	Fb12	Fc12	Fd12	Fe12	move west after finding **a** or **d** (don't yet know number status)
13	Ha13	Fb13	Fc13	Fd13	Fb13	move west after finding **e**, halt on **a** (identified prime number)

Figure 13. Transition table for the prime number sieve.

To determine whether a given row represents a prime or a composite number, the turmite that creates the leftmost marked column of cells also spawns one new turmite per row. Each such turmite is responsible for seeing whether there are any cells with symbol **b** between the leftmost column of such marks (representing the number 1) and the diagonal. Such a turmite must wait for some of the cells along the row to become nonempty. It does so by moving east along the row until it reaches an empty cell, then traveling west until it reaches the first column of cells. Eventually during this process, the turmite will reach either a marked cell (symbol **b**) or the symbol **e** on the diagonal (a pink cell). If it finds a **b**, the number is composite, and the turmite halts. If it reaches the diagonal, the number is a prime. In the case of a prime, the turmite travels west until it reaches the leftmost column of symbol **e**, changes the symbol **e** to **b**, and halts.

4.4 Simulating the Spread of an Infectious Disease

We now turn to simulating the spread of disease using multiple turmites on a grid. We will create a grid-based Susceptible, Infectious, Recovered (SIR) model (Anderson, 1991; Nardini et al., 2021). This model will consider each individual to be in one of three states: susceptible, infectious, or recovered. Such a model will cause a susceptible individual to become infectious if they come into contact with an individual who is infectious. At first it may seem that such proximity-based rules cannot be simulated with turmites because they have no access to the state of other nearby turmites. We can resolve this issue by using the symbols at grid cells as a mechanism to pass information between turmites. We can think of this as similar to an infected individual who is leaving infectious particles on various surfaces in their environment.

For the SIR model, we make use of four cell symbols. Borrowing the wandering tendency of the **RL** vant, two of these symbols will be used to cause the turmites to move in a pseudo-Brownian manner. Half of the grid cells are randomly initialized to contain either the symbol **a** or the symbol **b**. Depending on the symbol at a given cell, the turmites will turn either left or right. The turmites will also flip the symbol in the cell between **a** or **b**. The remaining half of the cells will be used for the communication of disease and will contain either the symbol **c** or the symbol **d**. Think of the symbol **c** as indicating a "clean" cell and the symbol **d** as indicating that the cell can cause disease. A cell in state **c** will not affect a **susceptible** turmite's state, and the turmite will travel unaffected forward through the cell. If, however, a susceptible turmite moves forward through a cell in state **d**, the turmite will change its state to infectious. **Infectious** turmites change cell symbols **c** to **d**. After an infectious turmite has left a small number of infectious cells, it switches to being in the *recovered* state and no longer changes cell symbols **c** to **d**. Finally, we include some turmites, which we refer to as *cleaners*, that wander the grid and clean it of infected cells by changing the symbol **d** back to **c**. The transition table for this simulation is given in Figure 14.

Our turmite-based SIR simulation exhibits the same characteristics of other grid-based SIR models. When the simulation is initialized with just a few infectious individuals, the infection quickly dies out. When the number of infectious individuals crosses a threshold, however, the infection sweeps through the population.

	Symbols				
States	**a**	**b**	**c**	**d**	Comments
1	Rb1	La1	Fc1	Fc1	cell cleaner (d → c)
2	Rb2	La2	Fc2	Fd4	susceptible
3	Rb3	La3	Fc3	Fd3	recovered
4	Rb4	La4	Fd5	Fd4	infectious (c → d)
5	Rb5	La5	Fd6	Fd5	infectious
6	Rb6	La6	Fd7	Fd6	infectious
7	Rb7	La7	Fd8	Fd7	infectious
8	Rb8	La8	Fd3	Fd8	infectious

Figure 14. Transition table for the SIR simulation.

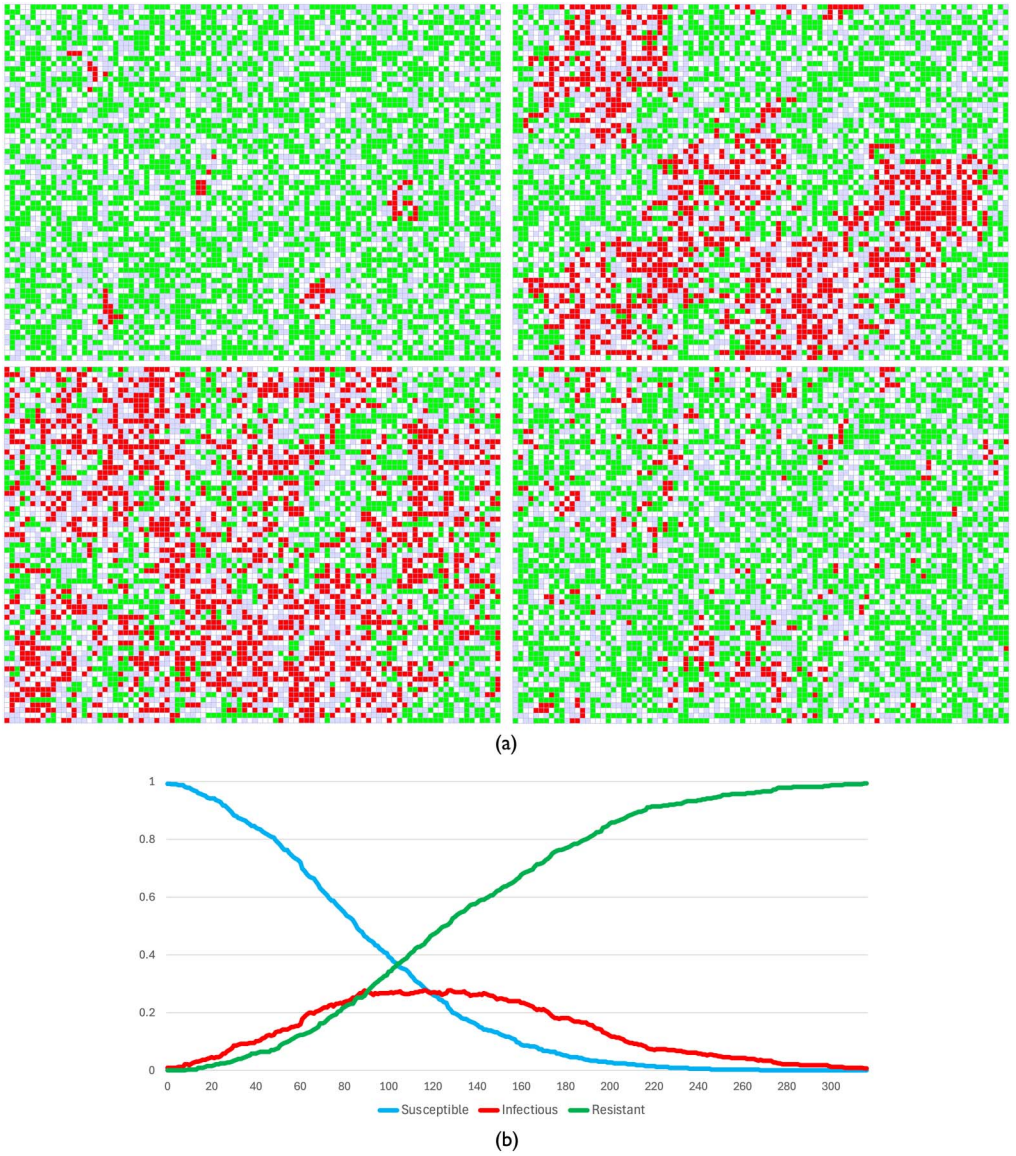


Figure 15. (a) Four snapshots showing the spread of infection. At first, just a few cells are infected (upper left). The infected cells spread more widely (upper right and lower left). Finally, the infection dies out (lower right). (b) Proportion of turmites that are susceptible (blue), infected (red), and resistant (green) as the simulation progresses.

Figure 15 shows four snapshots of an SIR simulation, at early, middle, and later stages. Light green and red cells indicate the symbols **a** and **b**. Dark green cells indicate the symbol **c** and are not infectious. Dark red indicates an infectious cell with the symbol **d**. Although many turmites are moving across the grid, the turmite positions are not shown. For this simulation, the grid is toroidal so that turmites that step off one edge will appear on the other side of the grid. The simulation is initialized with a turmite density of 0.1 (one turmite for every 10 grid cells). Initially, 99.4% of these turmites are susceptible, with just 0.6% of them starting out as being infected. The graph at the bottom of the figure shows that the number of infected individuals (red curve) starts low, grows, levels off, and finally declines. As this happens, the number of susceptible individuals declines (blue curve), while the resistant individuals increase in number (green curve).

Simulation of other biological phenomena may well be carried out using turmites in a similar manner to this simulation of infection. Other biological phenomena that may be amenable to this simulation approach include foraging, predator–prey relationships, and altruism.

5 A Universal Turmite

In this section, we present a turmite that is capable of universal computation. Alan Turing's (1936) classic result on the Halting Problem is based on the ability of a single Turing machine to mimic the computation of any other such machine. Since Turing has already presented a machine that uses a 1-D tape that is capable of universal computation, why should we repeat the exercise for 2-D turmites? We do so because we will use this universal turmite as a stepping-stone toward demonstrating self-reproduction in section 6. We will use familiar computational mechanisms, such as counters, instruction pointers, and jumps, to build this universal turmite.

The universal turmite that we construct carries out computation with the use of counters and can carry out the following four basic instructions: (a) increment a counter, (b) decrement a counter or jump if 0, (c) unconditional jump, and (d) halt. Our universal turmite will be able to use any number of counters and can execute any program that uses these four basic instructions. See Minsky (1967, chapter 11) for a proof that a machine that implements these simple instructions can be programmed to carry out universal computation. The cited chapter demonstrates that a particular program for a four-counter machine can mimic the computations of any 1-D Turing machine.

Our universal turmite has nine states and makes use of seven cell symbols (**a** through **g**). We will reserve the symbol **a** for empty cells through which the turmite will move without altering the symbol. The other six symbols will be put to various uses. Figure 20 (which appears later in this section) shows the full transition table for the universal turmite. Note that programming the universal turmite is specified entirely by the pattern of symbols placed in the grid. No modifications to the transition table are needed to change the computation that the machine will carry out.

We can convert any list of instructions into a pattern of cells on a 2-D grid. Figure 16 is a high-level diagram that shows the basic elements that the universal turmite uses to carry out computation. The turmite typically moves through this diagram in a counterclockwise manner, visiting these parts in turn: (a) instruction selector, (b) counters, (c) funnel, and (d) instruction pointer (and sometimes jump table). To avoid confusion, in what follows, we use the terms *north*, *south*, *east*, and *west* to indicate absolute directions of motion through the grid of cells, and we use *left* and *right* to indicate turns that the turmite makes relative to its current direction of motion.

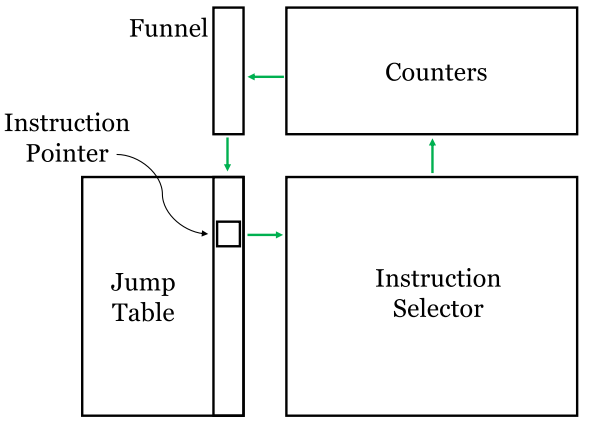


Figure 16. General layout of cell regions that control the calculations of a universal turmite. The flow of motion of the turmite roughly follows the green arrows, from instruction selector to counters to funnel to instruction pointer, and then sometimes through the jump table. See Figure 21 for a detailed pattern of cells.

A program for the universal turmite is encoded in a top-to-bottom fashion in the rows of the instruction selector and the jump table. Each row of cells in the instruction selector contains exactly one nonempty cell (symbol **f**), and the position of this cell indicates which counter is to be modified and whether it should be incremented or decremented. The turmite always enters the instruction selector while moving east in State 3. When it reaches a nonempty cell, it makes a left turn and changes to State 1. Now it is moving north, toward one of several counters. Thus each nonempty cell on a row essentially "aims" the turmite toward either the increment or the decrement cell of a particular counter. Note that once the turmite is in State 1, it passes through cells with symbol **f** without turning, which allows more than one increment (or decrement) of the same counter to be indicated in a column of the instruction selector.

We make use of binary counters to store numerical values, which were also employed by Turing (1936) in his universal machine. The 2-D nature of turmites makes it easy for us to create as many binary counters as we like and to place them so that they do not get in each other's way. The binary digits 0 and 1 are represented by the cell symbols **b** and **c**, respectively. Figure 17(a) illustrates a counter that contains the binary value 11, with binary digits 1011. In Figure 17(b), the most significant digit appears at the top of the column of digits, and the least significant digit is at the bottom. At the base of the list of digits, the symbol **d** indicates to the turmite that it is entering a counter and that it should increment the counter. Entering instead at the symbol **g** would indicate that the turmite should decrement the counter.

Let us follow the stages the turmite goes through to increment a counter (illustrated in Figure 17(b)). While the turmite is in State 1, it moves north through empty cells until it reaches the symbol **d** at the base of the counter. At this point, the transition table indicates that the turmite should move forward through **d** and change to State 2. While in State 2, the turmite continues moving forward (north) and changes any 1 it encounters to 0. In this case, it passes through two 1s (**c**) and changes them both to 0s (**b**). Once the turmite arrives at either a 0 (**b**) or an empty cell (in this case, it reaches a 0), it changes that cell to a 1 (**c**), turns back 180°, and changes to State 3. The turmite then moves south (leaving the cells it passes through untouched) until it again reaches the symbol **d**. At this point, it turns right and exits the counter moving west while still in State 3. The final state of the counter after this is shown in Figure 17(c).

The process of decrementing a counter is slightly more involved, for a couple of reasons. First, the decrement operation must be able to detect when the counter is at 0 (switching to State 9). Furthermore, the turmite has slightly different behavior depending on whether the first one it encounters is the first digit of the number. In either case, the turmite swaps 1s and 0s as it moves

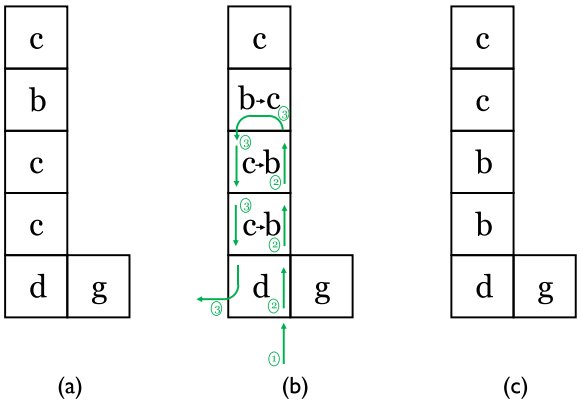


Figure 17. In a counter, zeros and ones are represented with the symbols **b** and **c**, respectively. (a) A counter containing 1011 (b) is incremented (c) to reach the new value 1100. In (b), green arrows indicate the motion of the turmite, and the accompanying numbers indicate the turmite's states. Short black arrows indicate changes to a cell's symbol. Entering the counter at the symbol **d** triggered the increment operation, and entering at **g** would have instead caused a decrement.

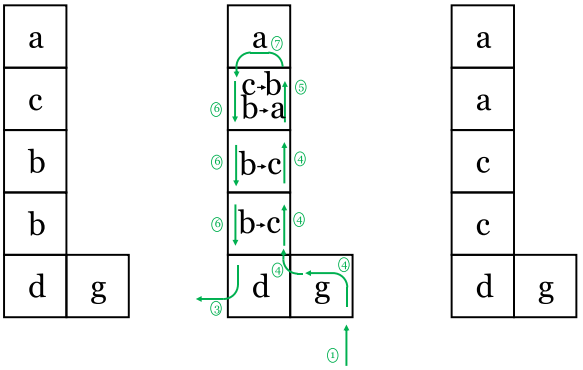


Figure 18. Decrementing a counter that contains the value 100. As the turmite rises through the digits, it swaps zeros and ones (**b**'s and **c**'s). Once it reaches the symbol **c**, it must handle the case where the next symbol is either **a** (an "empty" cell, as shown) or **c** (not shown). The final value of the counter is 11.

upward through the binary digits (while in State 4). After encountering its first 1 and changing it to 0, it switches to State 5 and moves up into the next cell. If this cell is a 0 or a 1, then the turmite descends through the digits, leaving them unchanged. If, however, it instead finds an empty cell atop the column of digits (symbol **a**), then as it descends, it changes the leading 0 to be an empty cell. This case is shown in Figure 18. Note that the top symbol **c** is first changed to **b** (1 to 0), but then is later changed from **b** to **a** (0 to empty).

When a counter with a nonzero value is decremented, the turmite exits the counter in State 3, just as it does after an increment. If the counter is already 0, however, attempting to decrement it will leave its value unchanged, but the turmite will exit the counter in State 1 instead of in State 3. This will cause a different behavior when the turmite reaches the instruction pointer, to be described next.

A turmite that has just departed a counter will be moving west and will eventually reach the "funnel," which is a column of cells containing the symbol **f**. The purpose of the funnel is to cause the turmite to turn left so that it is then moving south along a particular column of cells, no matter which of several counters it has just departed. The lower parts of counters are typically staggered vertically so that turmites can exit each of the counters without running into another counter. After leaving the funnel, the turmite will enter a vertical column of cells that maintain the instruction pointer. Most of these cells contain symbol **e**, but one of them will contain symbol **b** instead, indicating that this is the position of the instruction pointer. If no jump is indicated (by means that we will discuss soon), then the turmite will push the instruction pointer **b** south by one cell, then start moving east to carry out another counter-related instruction.

To specify an unconditional jump, one cell with the symbol **g** is placed just to the east of the instruction pointer column. If the turmite reaches such a **g** cell, it is turned backward, reenters the instruction pointer column, erases the instruction pointer (changing the cell symbol from **b** to **e**), then continues moving west into the jump table. These first actions for an unconditional jump can be seen in the upper right portion of Figure 19.

Inside the jump table, it is the symbol **f** (rather than **a**) that acts as an "empty" cell. The jump table can be thought of as a collection of vertical conveyor belts that move the turmite either north or south to a different location in the list of instructions. The start of each such conveyor belt is marked with either the symbol **c** or the symbol **d**, which indicate a forward (south-moving) or backward (north-moving) jump in the instruction list, respectively. Upon reaching a **c** or **d**, the turmite makes either a left or right turn, then begins to travel along a column of cells with the symbol **g**. In Figure 19, the symbol **c** indicates a forward jump, so the turmite turns left and begins to move south. The end of a conveyor belt is an "empty" cell containing **f**, and upon reaching such a cell, the turmite will turn to move in the eastern direction. When the turmite reaches the instruction pointer column (containing the symbol **e**), it changes the symbol **e** to **b** to indicate that

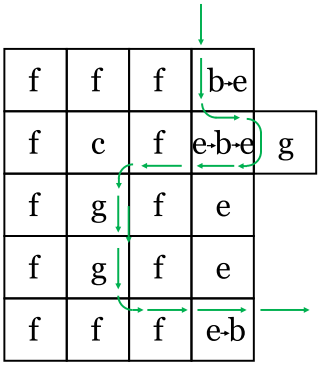


Figure 19. The motion of a turmite during an unconditional jump. The instruction pointer **b** is erased near the top, the turmite follows a conveyor belt downward (**c** and **g**), and finally the instruction pointer **b** is placed in the lower right cell of the figure.

the instruction pointer has been moved to this new location, then the turmite continues moving east and into the instruction selector region.

If a counter is decremented when it is already 0, this will trigger a conditional jump. In such a case, the turmite exits the funnel in State 8 instead of State 1. Rather than pushing the instruction pointer one cell to the south, the turmite erases the instruction pointer, turns west, and enters the jump table. From this point on, the turmite acts in exactly the same manner as it would for an unconditional jump. It catches a north or south conveyor belt, exits the conveyor belt moving east, places the instruction pointer on the current row, then continues moving east into the instruction selector.

A few aspects of the jump table require more explanation. Note that several conveyor belts may run vertically through a given row of cells, and the turmite should not be interrupted by any of them as it moves either west or east along a row. Thus, when the turmite is moving either east or west in the jump table, it passes through any cells with the symbols **f** or **g** unchanged. The turmite should begin to move along a conveyor belt only when it arrives at the start of the one particular conveyor belt for this row of cells. For this reason, there should be at most one **c** or **d** symbol on a given row of the jump table. This simply means that any one row of the instruction list should not contain both a forward and a backward jump. Note that any number of conveyor belts can end on the same row of cells; that is, more than one jump instruction can have the same jump destination.

The height of the jump table is dictated by the length of the list of instructions in the program. Choice of the width of the jump table is more subtle and depends on the structure of the program. The jump table needs to be at least as wide as the maximum number of conveyor belts that move "through" a given row. For instance, if a program contains a triple-nested loop, the jump table will have to be at least three cells wide to accommodate the conveyor belts that are associated with the three loops.

Figure 20 gives the rule table for the universal turmite. The machine has nine states, and uses seven symbols, **a** through **g**. The entries in the rule table have been color coded to indicate which operation each entry helps to implement, and the color key is at the right. For instance, portions of the rules for States 2 and 3 (indicated in green) assist in incrementing counters. Other rules for State 3 help to move the instruction pointer (light purple). Unused state/symbol combinations in the rule table are left blank (white).

Figure 21 illustrates a cell configuration for the universal turmite that calculates the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... The Fibonacci sequence begins with $F_0 = 0$ and $F_1 = 1$, then subsequent values are the sum of the two previous values, $F_i = F_{i-2} + F_{i-1}$. The turmite configuration to calculate the values of this sequence uses four counters (numbered $c1$ to $c4$) and repeatedly carries out the following steps: (a) add $c1$ to $c3$ (which zeros out $c1$), (b) move $c2$ to $c1$, (c) move $c3$ to $c4$, and (d) move $c4$ to $c2$ and $c3$.

States	Symbols a	b	c	d	e	f	g
1	Fa1	Fe3		Fd2	Fe1	Ff1	Lg4
2	Bc3	Bc3	Fb2				
3	Fa3	Fb3		Rd3	Lb3	Lf1	Bg9
4	Ba9	Fc4	Fb5	Rd4		Lf6	Fg4
5	Ba7	Bb6	Bc6			Rf6	Fg5
6		Fb6	Fc6	Rd3	Fb3	Ff6	Fg6
7	Fa7	Fa7	Fc6	Rd3	Fe8	Lf8	
8	Fa8	Re8	Lc4	Rd5	Fe8	Ff8	Fg8
9	Fa9	Fe8		Rd9	Fe3	Lf8	

Color Key	
inc	
dec	
move	
jump	
next instr	
unused	white

Figure 20. Rule table for a turmite that can carry out universal computation.

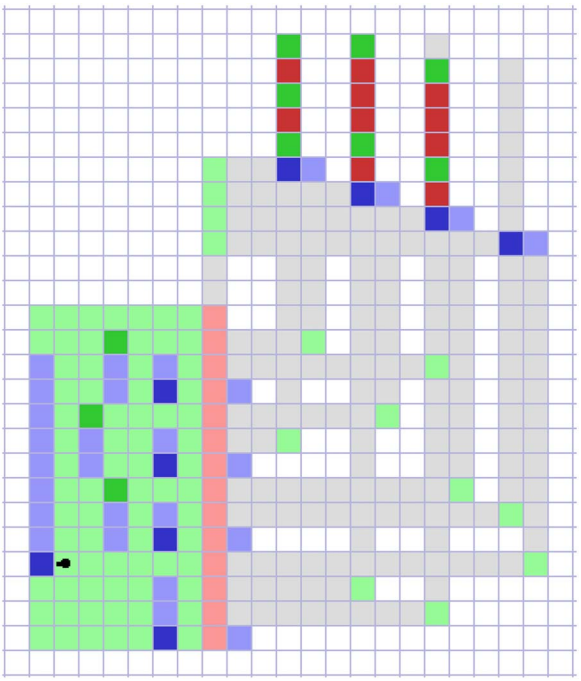


Figure 21. A turmite that calculates the values of the Fibonacci sequence. At this snapshot, the first counter is set to 21, and the second and third counters contain 34. No instruction pointer is present—it would be in the pink column. This is because the turmite (the black mark near the lower left) is about to reach a conveyor belt that causes a backward jump to place the instruction pointer at the start.

At the start of these steps, the sequence value F_{i-2} is in counter $c1$, the value F_{i-1} is stored in both $c2$ and $c3$, and $c4$ is set to zero. The addition at the first step calculates the next value of the sequence, F_i. The remaining three steps rearrange the values so that F_{i-1} is in $c1$ and the newly calculated F_i is in $c2$ and $c3$. To calculate the Fibonacci sequence from the start, the initial values of the counters $c1$ through $c4$ are 0, 1, 1, and 0, respectively.

Each of the preceding four steps (the addition of one counter to another and moving a value between counters) is carried out by a loop using a handful of low-level operations. Each such loop starts by attempting to decrement a counter and, if the counter is already zero, jumping to the instruction just after the loop. Then one or two increments are performed, and finally an unconditional jump brings the instruction pointer back to the top of the loop. For example, following are

the instructions for carrying out the addition of $c1$ to $c3$: (a) dec $c1$ or jump to 4 if zero, (b) inc $c3$, (c) jump to (a), and (d) (instructions after the loop, . . .). The total number of instructions for calculating the Fibonacci sequence is 13.

6 Turmite Constructors and Self-Reproduction

We will make a constructor turmite by following von Neumann's lead by adding a constructor arm to a universal machine. Although inspired by von Neumann's work, note that our constructor is a 2-D Turing machine, not a cellular automaton, and thus differs in several ways from prior work on self-reproduction. Our constructor arm is a connected region of cells that has two parts that we will refer to as the *base* and the *forearm*, which are joined to each other at a 90° elbow. Similar to a counter, which can carry out either increment or decrement operations, the constructor arm will be able to carry out six simple operations. Figure 22 is a diagram of the constructor arm. In Figure 22, the forearm is pointing upward, but we will later see that an arm can point downward instead.

One of the six possible arm operations is selected by causing the turmite to approach the arm from the south at a different position along the row of six **h** symbols. Similar to selecting between the increment and decrement operations for a counter, the position of an **f** symbol in a row of the instruction selection region is used to aim the turmite at a particular **h** cell. The turmite's transition table has a cascade of six states that keep track of how many **h** symbols the turmite encounters as it moves east before it reaches the portion of the base that contains **a′**. Primed symbols denote the shape of the arm in a manner that will be expanded upon shortly.

Following are the six operations that the constructor arm can carry out: (a) previous symbol (or jump if the symbol is **a**), (b) next symbol, (c) retract base, (d) extend base, (e) shorten forearm, and (f) lengthen forearm. The extend and retract operations cause the base to become longer or shorter, and the lengthen and shorten operations do the same for the forearm. The next and previous symbol operations are used to modify the symbol of the cell that is at the end of the forearm. We assume that the symbols are ordered alphabetically, starting with **a**. The next operator changes a cell with symbol **a** to the symbol **b**. Similarly, the previous operator changes the symbol **b** to **a**. If the cell at the end of the arm already contains **a** when the previous operator is executed, then the cell is left unchanged, but a jump is executed to change the location of the instruction pointer. This conditional jump is similar to when a decrement operation is performed on a counter that contains the value 0.

We introduce several new symbols that are used for the constructor arm (see Figure 22). The row of six **h** symbols at the base is used to select between the six operations that the arm can carry out, and these six cells are always left unchanged. The bulk of the arm's base is represented by a row of the symbol **a′**, and the number of such symbols is changed when the arm is extended or retracted. The elbow of the arm is marked by the symbol **i**. Finally, the forearm is represented by another group of contiguous cells with the primed symbols from the set **a′** through **i′**, this time in a column.

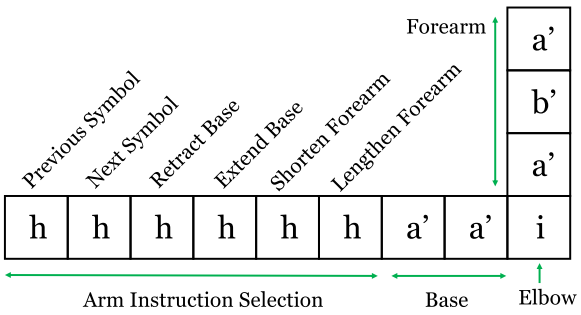


Figure 22. Constructor arm. The row of six **h** symbols is used to select between arm operations. The lower stretch of **a′** symbols is the base of the arm, followed by **i**, which marks the elbow. The forearm is marked by a column of primed symbols, in this case, **a′** and **b′**.

For both the von Neumann and the Codd self-reproducing machines, the construction arm can only move through a region containing the background symbol. While this limits the complexity of their machines, it also puts severe constraints on the motion of an arm. For our turmite constructor, the forearm is allowed to pass through a region of cells that may contain any of the first nine symbols, **a** through **i**. We accomplish this by making use of nine primed symbols **a′** through **i′**. When the forearm is moving through "empty" cells containing the symbol **a**, the forearm's cells will contain the symbol **a′**. If instead the forearm moves into a cell with the symbol **g**, for example, the forearm will use the symbol **g′** at that cell. In this way, the forearm can pass through any pattern of cells **a** through **i** in a manner that leaves both the cell symbols and the arm undisturbed.

Let us trace the steps that the constructor turmite takes to carry out the operation that lengthens the forearm. The steps for this operation are shown in Figure 23. The turmite approaches the arm from the south, having just left the instruction selector region of the machine. Let us assume that the turmite encounters the rightmost symbol **h** in the row of six such cells. Upon encountering this **h**, the turmite changes its state and turns right. The next cell the turmite encounters contains **a′**, and at this point the arm operation is "locked in" as being the forearm lengthen operation. The turmite will then move forward (in the eastern direction) through the row of **a′** symbols until it reaches the elbow that is marked with symbol **i**. The turmite turns left at this point, which takes it into the forearm. The turmite moves north along the forearm (marked with primed symbols **a′** through **i′**) until it reaches a nonprimed cell symbol (**a** through **i**). Assume that it reaches the symbol **c**. The turmite changes the **c** to **c′** (thus lengthening the forearm by one cell), turns backward, and moves into a new state. The turmite then moves south along the forearm, turns right at the elbow, moves west through the base of **a′** symbols, and continues to move west through the row of six **h** symbols; it then exits the arm, still traveling west. Just as is the case when the turmite exits a counter, the turmite will continue moving west until it encounters the funnel, which directs the turmite south and into the instruction pointer region.

The arm operations for extending and retracting the base are somewhat more complicated to carry out than the other operations. This is because they must not only add or remove one **a′** cell from the base but also shift the entire column of cells that make up the forearm one position either to the east or to the west. This is accomplished by having the turmite zigzag between the old and new columns of cells. Figure 24 shows the motion of the turmite as it performs the extend and retract operations.

The arm operations for shorten forearm, next symbol, and previous symbol (or jump) are all straightforward to implement with a small number of states in the turmite's transition table. Each of these operations is similar to the lengthen forearm operation in that it simply travels along the base and then along the forearm until it reaches the end of the forearm. At that point, the turmite will change one cell's state, then return along the arm. The shorten forearm operation changes the last primed cell at the top of the forearm to a nonprimed cell. The next operation changes the

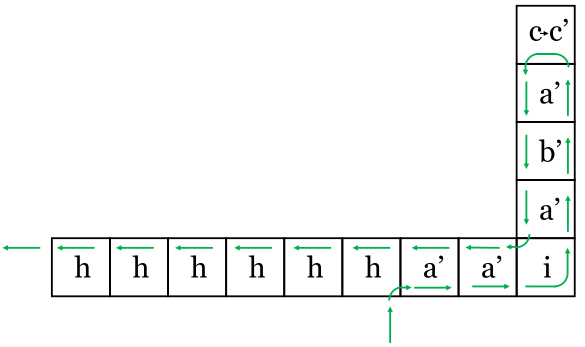


Figure 23. Lengthening the constructor arm. The lengthen operation is triggered by entering the arm at the rightmost **h**. The forearm is made longer by one cell by changing the symbol **c** that is just above the forearm to the symbol **c′**.

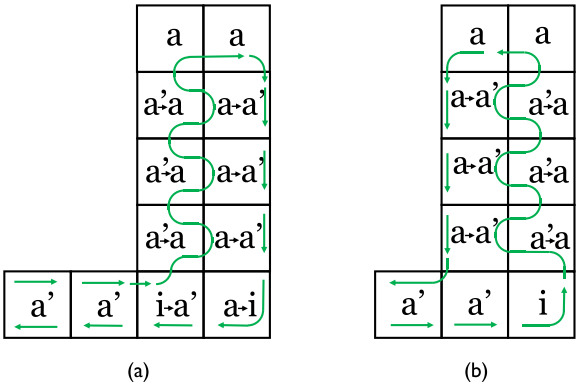


Figure 24. (a) Extending the base and (b) and retracting the base. For both operations, the turmite changes primed to unprimed symbols in the old column and does the reverse in the new column. In these examples the forearms are shown as being made entirely of **a′**, but they may in fact contain any primed symbols from **a′** to **i′**.

nonprimed cell just beyond the forearm to the "next" nonprimed symbol according to alphabetical order. The previous operator is similar, but it changes the symbol to the previous one alphabetically. If this symbol is **a**, it leaves the symbol unchanged, but the turmite will exit the arm while it is in a different state, which then causes a jump.

This turmite that allows the use of arms can be programmed in a similar manner to the universal turmite of section 5. Both counters and construction arms of any number can be used by this turmite, allowing it both to carry out calculations and to perform construction. This turmite is a universal constructor. Altogether, this turmite operates on 18 different symbols, **a** through **i** and **a′** through **i′**, and the turmite itself has 20 states. The rule table for this turmite is given in Figure 25. The rules for the symbols **a′** through **i′** follow simple patterns and are represented in the last column of the table. The symbol **x** is shorthand for the symbols in the range **a** to **i**, and **y** is shorthand for the symbols **a′** to **i′**.

	Symbols										
States	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**a' - i'**	
1	Fa1	Fe3		Fd2	Fe1	Ff1	Lg4	Rh8	Fi1	Fx11	Color Key
2	Bc3	Bc3	Fb2					Fh7	Lj12	Fy2	
3	Fa3	Fb3	Lc3	Rd3	Lb3	Lf1	Bg9	Fh5	Ri1	Ri16	inc
4	Ba9	Fc4	Fb5	Rd4		Lf6	Fg4	Fh5	Li17	Fy4	dec
5	Ba7	Bb6	Bc6			Rf6	Fg5		Li20	Fy5	move
6	Ri11	Fb6	Fc6	Rd3	Fb3	Ff6	Fg6	Fh3		Fy6	jump
7	Fa7	Fa7	Fc6	Rd3				Fh4	Ba3	Fy7	next
8	Fa8	Re8	Lc4	Rd5	Fe8	Ff8	Fg8	Fh10	Li18	Fy8	unused
9	Fa9	Fe8	Lc9	Rd9	Fe3	Lf8		Fh9	Ri9	Fy9	
10								Fh2	Li19	Fy10	extend base
11	Fa3							Fh11	Ri11	Fy11	retract base
12	Ra14	Rb14	Rc14	Rd14	Re14	Rf14	Rg14	Fh14	Ri14	Ry13	arm next
13	Bj13	Bk13	Bl13	Bm13	Bn13	Bo13	Bp13	Bq13	Br13	Rx12	arm previous
14	Ra6	Rb6	Rc6	Rd6	Re6	Rf6	Rg6	Rh6	Ri6	Lx11	retract forearm
15	Ba14	Bb14	Bc14	Bd14	Be14	Bf14	Bg14	Bh14	Bi14	Bx16	extend forearm
16	Rj15	Rk15	Rl15	Rm15	Rn15	Ro15	Rp15	Rq15	Rr15	Ry16	pick arm op
17	Bb11	Bc11	Bd11	Be11	Bf11	Bg11	Bh11	Bi11	Bi11	Fy17	go to offspring
18	Bj11	Bk11	Bl11	Bm11	Bn11	Bo11	Bp11	Bq11	Br11	Fy18	
19	Ba1	Bb1	Bc1	Bd1	Be1	Bf1	Bg1	Bh16	Bi1	Fy19	
20	Ba9	Ba11	Bb11	Bc11	Bd11	Be11	Bf11	Bg11	Bh11	Fy20	

Figure 25. Transition table for the Universal Constructor.

Now that we have seen the six arm operations, let us put a construction arm to use. Although we could create grid patterns that contain any number of constructor arms, for simplicity, we start with a pattern of grid cells, shown in Figure 26 that has just one arm and no counters. The instructions in this grid pattern have been designed to simulate the behavior of the original **RL** vant. The construction arm starts in the middle of a region of cells that all contain **a**. The position of the virtual vant in this empty region is given by the end position of the forearm. The direction that the virtual vant is facing is encoded in the position of the instruction pointer, and the behavior of the virtual vant is encoded in the instructions for the turmite.

Four sections in the list of instructions for the turmite simulate the **RL** vant. These sections indicate that the vant's next move is in one of the four directions: north, east, south, or west (reading down the instruction list). When this machine starts, the instruction pointer is at the top, which indicates that the virtual vant is about to move north. The first operation that the turmite carries out is to lengthen the forearm, moving the virtual vant one cell north. Next, the turmite attempts to decrement the cell at the end of the arm. If this cell is **b**, it is changed to **a**, and the instruction pointer jumps to the "move west" part of the instructions (thus taking a left turn). If instead the cell is **a** (as is the case with an empty grid), the turmite carries out an increment arm operation, changing the cell from **a** to **b**. Then the turmite enters the portion of the code used when the virtual vant is facing east (thus taking a right turn). Each of the code sections for the virtual vant facing east, south, and west is similar, but with the first instruction in these sections performing extend base, shorten forearm, or retract base, respectively. The full program is 17 instructions long: four instructions for each direction plus an additional instruction to connect the last direction (west) with the first direction (north).

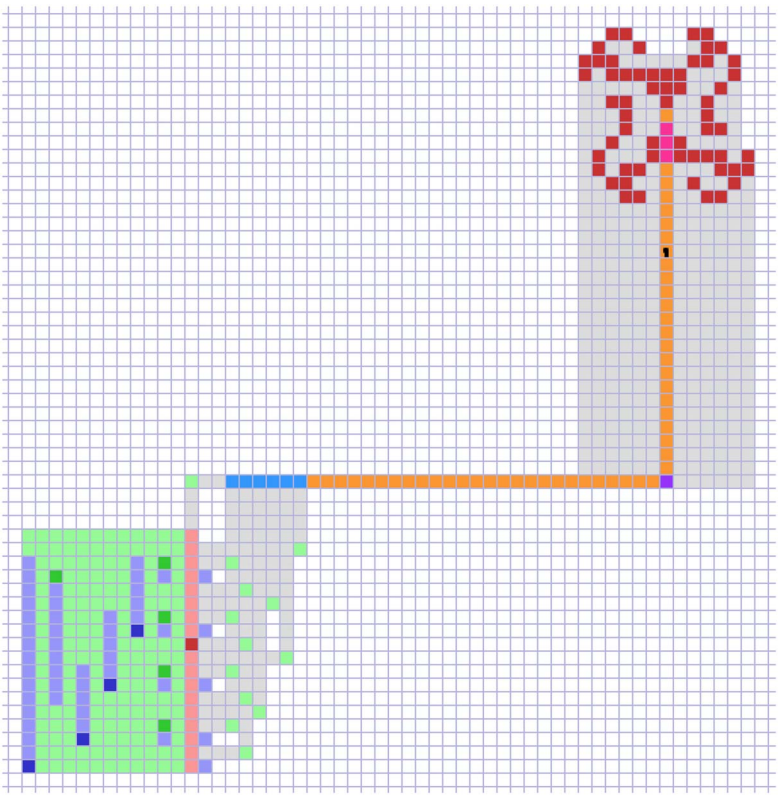


Figure 26. A pattern of cells that uses the constructor arm to simulate the **RL** vant. The turmite can be seen moving south along the forearm. The current location of the simulated vant is the cell just above the constructor arm. This simulation is in the same state as the vant shown in Figure 2(a).

In addition to showing the use of the constructor arm, the **RL** vant simulator also demonstrates that a turmite with a construction arm can simulate another 2-D Turing machine. In fact, for any 2-D Turing machine that uses nine or fewer symbols, we can create a program for our universal constructor that simulates it. If we wished to simulate a 2-D Turing machine that uses more than nine symbols, we could encode such symbols in adjacent blocks of multiple cells. This would allow us to simulate 2-D Turing machines with an arbitrary number of symbols. A slight complication arises in simulating another 2-D Turing machine because a construction arm can move only in a half plane. To allow motion of a simulated machine anywhere on the 2-D grid, either of two methods could be employed. One method would be to "fold" the 2-D grid by using a pair of columns of cells in the region above the arm to represent a single column that extends both upward and downward. Another option would be to make use of two constructor arms, one that acts in the upper part of the grid and another that acts in the lower portion. The ability of our universal constructor to simulate the behavior of other 2-D Turing machines is another way to prove that it can perform universal computation.

Now we turn to the task of programming the universal constructor to carry out self-reproduction. von Neumann's approach is to create a machine that reads a very long tape that contains the blueprints that, when followed, instructs the constructor arm to create a working replica of the machine itself. A typical image of such a replicator shows the main body of the machine, but only a tiny portion of the contents of a very long tape. Instead of using one constructor arm, a tape reader, and a very long tape, we take a somewhat different approach.

We do not make use of a tape and its associate reader. Instead, we use a second construction arm that sweeps over a copy of the machine's 2-D pattern of cells (denoted M_1). No encoding or decoding of the machine's pattern is performed. Instead, the universal constructor alternates between reading a cell symbol at the "read" constructor arm (the lower arm) and writing a cell symbol at the "write" constructor arm (the upper arm). We can make use of this strategy because of the ability of the forearm to pass over any of the symbols **a** to **i** in a manner that does not disturb the symbols. For this to work, the pattern M_1 must not contain any of the primed symbols $\mathbf{a}'$ through $\mathbf{i}'$. We accomplish this by making sure that both the base and the forearm of the constructor arm in M_1 are fully retracted so that no $\mathbf{a}'$ symbols are used. Note that the lower arm points downward and that the turmite head must approach its row of six **h** symbols from above. The symbol **i** is used in the instructor selection region to send the head downward toward this arm.

Figure 27(a) shows a high-level diagram of a universal constructor that is in the process of replicating itself. The large block of cells in the middle marked M_2 is the active machine that is performing the replication. At the bottom of this pattern is arm **B**, which reads the pattern of cells from the pattern copy M_1. At the top of M_2 is arm **A**, which is in the process of writing the symbols for the new machine M_3. When M_2 has finished copying the pattern from M_1 to M_3, it entirely retracts both arms so that neither of them contains the symbol $\mathbf{a}'$. The head of this machine is then sent into the bottom of the jump table for M_3, and the position of the head travels upward to the first instruction of M_3. At this point, M_3 begins executing the instructions to make a copy of the now-quiet M_2, and the replication process begins again. Figure 27(b) is a snapshot of the full machine in the process of replicating itself.

Figure 28(a) shows the details of the self-reproducing machine. Per-instruction annotations of this machine are given in Appendix 3. This machine makes use of six counters to carry out self-replication. Two of these counters (Counters 3 and 5) are used to store the width and height of the pattern M_1. One more counter (Counter 4) stores the horizontal shift between machines M_1 (the passive template) and M_2 (the active copier). The other three counters (1, 2, and 6) start at zero and are used to save and restore the values of the width, height, and horizontal shift, respectively.

Counter 1 is also used to help copy symbols from the lower arm **B** to the upper arm **A**. Assume that Counter 1 is set to 0. Copying a single symbol requires a loop that repeatedly issues the previous symbol operation on arm **B**. For each such iteration through the loop, the machine executes the next symbol operation on arm **A** and also increments Counter 1. When the symbol at the end of arm **B** reaches the value **a**, the loop terminates. At this point, the symbol from arm **B** has been copied to

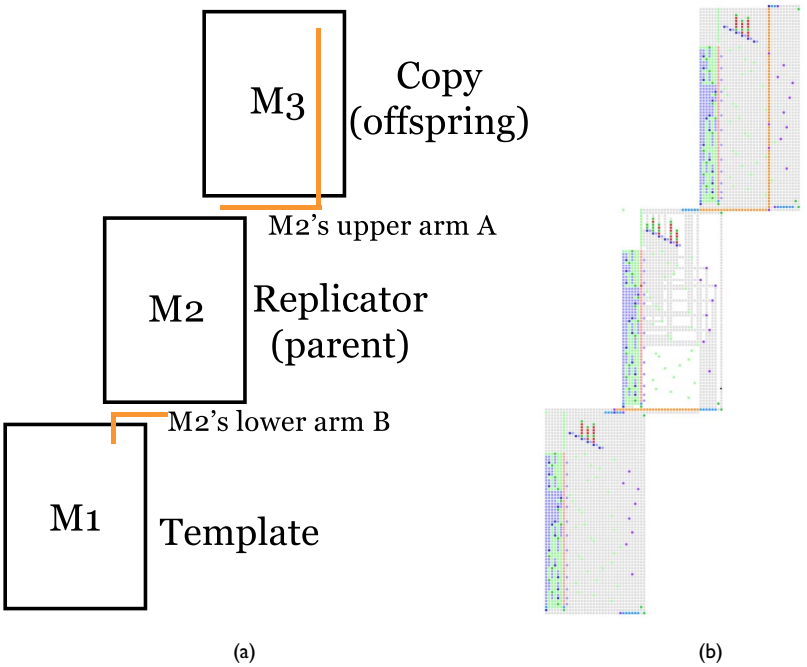


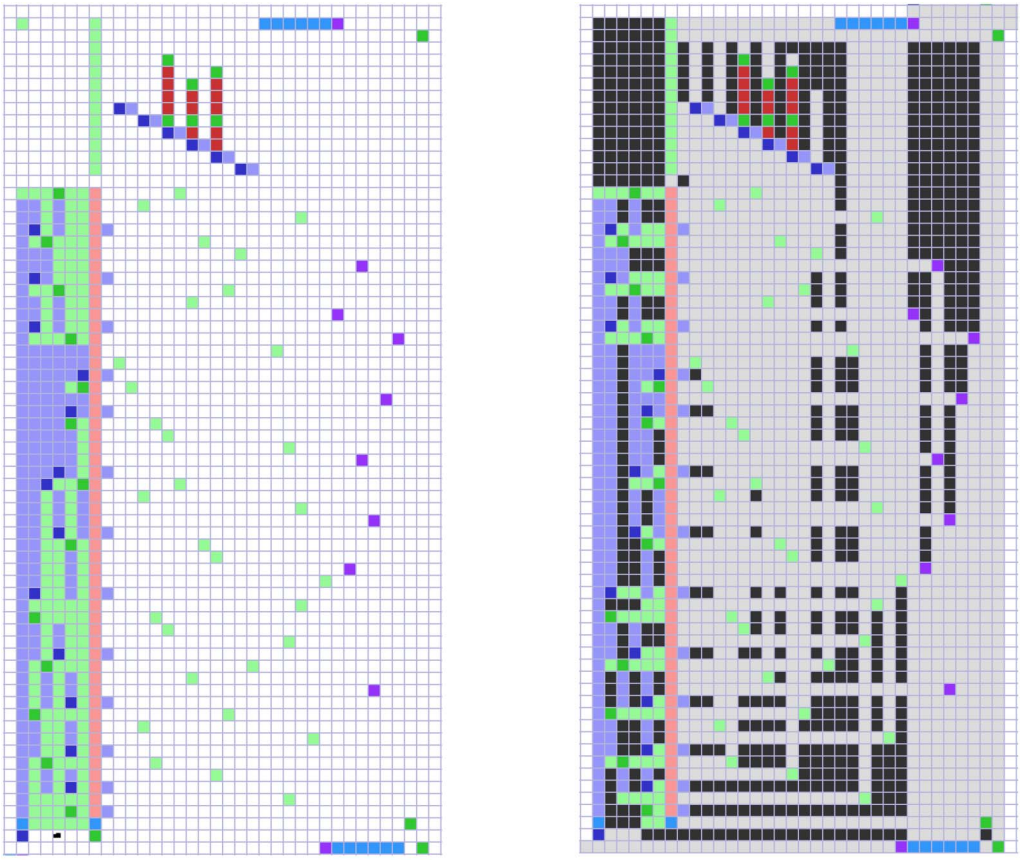
Figure 27. (a) High-level diagram of self-reproduction, and (b) the full self-reproduction pattern, including template and copy. The upper constructor arm is laying down the last few cells of the copy. After the copying is complete, both arms are retracted entirely, the counters are reset, and the active control of the turmite is transferred to the upper copy.

arm **A**. Another loop is then used to restore the original symbol at arm **B**, while decreasing Counter 1 to the value 0.

Self-reproduction begins by extending arm **A** of M_2 horizontally by the width of the pattern to position it at the lower right of what will eventually be copy M_3. Then, arm **B** is extended horizontally by the shift amount and vertically by the height of the pattern to bring it to the lower right of M_1. Then, both arm **A** and arm **B** sweep across horizontally right to left, and the symbols from the cell of one row of M_1 are copied to M_3. Both arms are then returned to the far right and positioned on the next row of the pattern. This process is continued until all of the cells from M_1 have been copied to M_3.

Next come the clean-up operations. Both arm **A** and arm **B** of M_2 are retracted entirely. The values of the counters are reshuffled so that the width, shift, and height values are placed back in Counters 3, 4, and 5, respectively. At this point, the pattern M_2 is now identical to the pattern in M_1 and can be used as the copy template. Finally, the active agent of the turmite (the head) is sent from M_2 into M_3 to begin the copy process again. The head is sent into the bottom of the jump table of M_2 and directed to an upward conveyor belt that takes it to the first instruction of the replication process. At this point, the replication process is repeated, this time with M_2 acting as the template and M_3 being the active replicator.

Fifty-three instructions are defined by the instruction selector and jump table regions of the replicator M_2. The size of the full pattern of the replicator M_2 is 34 × 69 cells, and the pattern for the template M_1 and replicator M_2 together is 60 × 138 cells. The detailed pattern of the replicator can be found in Appendix 3. The full replication process requires 7,544,430 time steps to complete. This replicator is smaller and replicates in fewer time steps than several other replicators that use von Neumann–style construction arms. Table 1 compares various replicators, several of which can be run using the Golly cellular automata simulator (Trevorrow & Rokicki, 2023).



(a) (b)

Figure 28. (a) A close-up of the self-reproducing machine, without the template pattern. A character-encoded version of this pattern can be found in Appendix 3. (b) Cells marked in black indicate those parts of the pattern that are not visited during self-reproduction.

Table 1. Pattern sizes and replication times for several universal self-replicators.

Replicator	Pattern size	Replication time
Devore and Hightower (1992)	365 × 258	1.02×10^{11}
Nobili and Pesavento (1996)	133 × 153	9.59×10^{9}
Buckley (2008)	272 × 132	2.61×10^{11}
Hutton (2010)	22,254 × 55,601	1.7×10^{18} (est.)
Goucher (see Hutton, 2010)	398 × 815	unknown
Turmite	34 × 69	7.54×10^{6}

Note. These pattern sizes do not include tape lengths or pattern templates.

If it seems unsatisfying that this chain of replicated machines leaves behind machines that are no longer active, there are several ways to rectify this. Of course, we could adopt the tape-reading approach that von Neumann and others have used, instead of using a template. If we wish to use a template, one option is for the machine M_2 that has just finished the copying to activate not only the new machine M_3 but also the quiet machine M_1 that has served as the template. Another possibility is to place the template M_1 to the east of the active replicator M_2 instead of to the south and have M_2 make two side-by-side copies above. This would produce an upward chain of active machines that each has an inactive template to its east.

von Neumann (1966, pp. 86–87) described how an automaton that is capable of self-reproduction would also be able to undergo heritable mutations. He never finished a description of his 29-state self-reproducing machine, but Pesavento (1995) did. Subsequently, Yinusa and Nehaniv (2011) demonstrated that this finished version of von Neumann's self-reproducing machine can indeed pass along heritable mutations to its offspring. They demonstrated first that a change to a single cell can be faithfully inherited, then showed that more complex mutations (banana- and apple-shaped collections of cells) can also be inherited.

Just like von Neumann's machine, our turmite self-reproducing machine is capable of undergoing mutations that are heritable. There are $34 \times 69 = 2{,}349$ cells in the machine's pattern, but 637 of the cells of M_2 (27% of the pattern) are not visited by the turmite's head when it is carrying out self-reproduction. This means that modifications of any of these unvisited cells will not affect the machine's operation and hence would be nonlethal heritable mutations. Figure 28(b) indicates the unvisited portions of M_2's pattern with black cells. Perhaps more interestingly, increasing the value stored in the binary counter for the pattern width (Counter 3) by changing any of the four 0s to a 1 would also be a nonlethal mutation. The consequence of such a change would be to cause a wider region of cells to be copied during reproduction (wider by 2, 4, 8, or 16 cells). This would create a larger pattern that would reproduce faithfully during subsequent generations. Moreover, this kind of mutation would open up considerable new pattern real estate for further mutations.

It would be possible to create a universal self-reproducing turmite that senses its surroundings and decides where to place a copy of itself. This could be accomplished by having the machine explore its surroundings using one or more construction arms. Such a machine could look for a large enough region of empty cells in which to put a replica (together with its template). The direction of search for empty space could be guided by calculated pseudo-random numbers, and each new replicator could have its random number generator seeded with a value that is different from its parent's. A variant of this idea would be to create self-replicating machines that are deliberately aggressive and that replicate in locations that are already occupied by other machines. The compact nature of the self-reproducing turmite and its template should be an advantage for carrying out such experiments, when contrasted with a machine that uses a long linear tape.[2]

7 Discussion and Conclusion

The idea of 2-D Turing machines has been around for decades (Dewdney, 1989; Hartmanis & Stearns, 1965; Langton, 1986), but they are relatively unexplored when compared to some other simulation models used in Artificial Life. We have highlighted three capabilities of 2-D Turing machine that we think are key to realizing their full potential for simulating biological phenomena. The first is not only to make use of left and right turns but also to allow motion in the forward and backward directions. This is key to simulating behaviors that are purposeful, such as traveling along a straight path. Second is to use 2-D Turing machines that take on more than one state, which we refer to as turmites. This, too, allows for more purposeful behaviors. Turmites can take note of their environment and alter their behavior based on the cells that they have previously visited. Finally, as

2 Please see the video at https://github.com/gturk1/turmites to watch simulation of the turmite pattern that performs self-reproduction.

Langton (1986) first suggested, all sorts of interactions are possible when multiple Turing machine heads (vants or turmites) are crawling on the same grid.

We have demonstrated some of the capabilities of turmites by simulating several different biological phenomena. Forms from nature, such as bilateral symmetry and spiral patterns, can be created by turmites. We have also shown how a turmite can build a self-similar shape, the dragon curve. Self-similarity is found everywhere in nature, from the leaves of ferns and branches of trees to the systems of veins and arteries in vertebrates. We have also demonstrated that multiple turmites can be used to model the spread of disease. Finally, we presented a turmite that carries out self-reproduction, borrowing ideas from the classic work of von Neumann.

Why should we ever use turmites to simulate biological phenomena? Turmites can be viewed as a restricted case of agent-based modeling (Gilbert, 2019; Railsback et al., 2006). In general, *agent-based modeling* refers to any computational method that can be described as having multiple interacting agents that are situated in a simulated environment. Agents may live in a discrete (e.g., grid-based) environment or may have continuous positions and orientations. Agents may carry any amount and form of information, discrete or continuous. Agents can use a variety of ways to sense the state of their environment. Agents may communicate either directly or indirectly with other agents. Each agent's behavior can be the result of an arbitrarily sophisticated set of computations. Turmites are a special case in which the agents have a discrete state, move on a discrete grid, and can communicate only indirectly with other agents by changing the state of a grid cell. Their actions are entirely dictated by a small transition table. We argue that simulating a given phenomenon with turmites allows us to create a stripped-down simulation of the given system. This allows us to learn what aspects of a simulation are absolutely needed to model the phenomenon and which ones are superfluous. If we want to build a simulation that can be used to closely match a real-world system, the full power of a general agent-based model might often be the best approach. If, however, we want to study the essence of a phenomenon, then a very simple simulation framework, such as turmites, may be attractive.

There are a number of interesting directions for future work using turmites. Many forms of biological phenomena might be investigated using turmites, including foraging behavior (Panait & Luke, 2004), predator–prey relationships (Durrett & Levin, 2000; Sun et al., 2012), and the spatial development of altruism (Nowak & May, 1992). It would be interesting to allow turmites to have transition rules that are nondeterministic. Another possibility is to explore 3-D turmites. A 3-D turmite would have a heading in one of six directions and would be in one of four possible "roll" orientations. We also note that to date, there has not yet been a systematic investigation of the behaviors of simple turmites, similar to the work of Beeler (1973) for worms and of Wolfram (1983) for 1-D cellular automata. Finally, it would be interesting to see what behaviors might develop by allowing artificial evolution of turmites under various rules for evaluating fitness.

Data and Code Availability

We encourage the reader to watch a video that shows the behavior of the turmites described in this article. The video is hosted at https://github.com/gturk1/turmites. The same site also provides the code that created the figures for the article.

References

Anderson, R. M. (1991). Discussion: the Kermack–McKendrick epidemic threshold theorem. *Bulletin of Mathematical Biology*, *53*, 1–32. https://doi.org/10.1007/BF02464422

Beeler, M. (1973). *Paterson's worm* (Memo Number 290). MIT AI Laboratory.

Berlekamp, E. R., Conway, J. H., & Guy, R. K. (2004). *Winning ways for your mathematical plays* (Vol. 4). AK Peters/CRC Press. https://doi.org/10.1201/9780429487309

Buckley, W. R. (2008). Signal crossing solutions in von Neumann self-replicating cellular automata. In Adamatzky, A., Alonso-Sanz, R., Lawniczak, A., Martinez, G. J., Morita, K., & Worsch, T. (Eds.), *AUTOMATA-2008, Theory and Applications of Cellular Automata* (pp. 453–503). Luniver Press.

Bunimovich, L. A., & Troubetzkoy, S. E. (1992). Recurrence properties of Lorentz lattice gas cellular automata. *Journal of Statistical Physics*, *67*(1), 289–302. https://doi.org/10.1007/BF01049035

Byl, J. (1989). Self-reproduction in small cellular automata. *Physica D*, *34*(1–2), 295–299. https://doi.org/10.1016/0167-2789(89)90242-X

Codd, E. F. (1968). *Cellular automata*. Academic Press.

Davis, C., & Knuth, D. E. (1970). Number representations and dragon curves. *Journal of Recreational Mathematics*, *3*(2), 66–81.

Devore, J., & Hightower, R. (1992). *The Devore variation of the Codd self-replicating computer* [Presentation]. Third Workshop on Artificial Life, Santa Fe, New Mexico.

Dewdney, A. K. (1984). In the game called Core War hostile programs engage in a battle of bits. *Scientific American*, *250*(5), 14–22. https://doi.org/10.1038/scientificamerican0584-14

Dewdney, A. K. (1989). Two-dimensional Turing machines and turmites make tracks on a plane. *Scientific American*, *261*(9), 180–183.

Durrett, R., & Levin, S. (2000). Lessons on pattern formation from planet WATOR. *Journal of Theoretical Biology*, *205*(2), 201–214. https://doi.org/10.1006/jtbi.2000.2061, PubMed: 10873432

Fowler, D. R., Prusinkiewicz, P., & Battjes, J. (1992). A collision-based model of spiral phyllotaxis. In *Proceedings of the 19th annual conference on computer graphics and interactive techniques* (pp. 361–368). SIGGRAPH. https://doi.org/10.1145/133994.134093

Gale, D. (1993). The industrious ant. *Mathematics Intelligencer*, *15*(2), 54–58. https://doi.org/10.1007/BF03024194

Gale, D. (1994). Further ant-ics. *Mathematics Intelligencer*, *16*(1), 37–44. https://doi.org/10.1007/BF03026614

Gale, D. (1998). *Tracking the automatic ant: And other mathematical explorations*. Springer Science & Business Media.

Gale, D., Propp, J., Sutherland, S., & Troubetzkoy, S. (1995). *Further travels with my ant*. ArXiv. https://arxiv.org/abs/math/9501233

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American*, *223*(4), 120–123. https://doi.org/10.1038/scientificamerican1070-120

Gardner, M. (1973). Mathematical games: Fantastic patterns traced by programmed worms. *Scientific American*, *229*(5), 116–123. https://doi.org/10.1038/scientificamerican1173-116

Gilbert, N. (2019). *Agent-based models*. SAGE. https://doi.org/10.4135/9781506355580

Hartmanis, J., & Stearns, R. E. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, *117*, 285–306. https://doi.org/10.1090/S0002-9947-1965-0170805-7

Hilbert, D. (1976). Mathematical problems [Lecture delivered before the International Congress of Mathematicians at Paris in 1900]. In *Mathematical developments arising from Hilbert problems* (Vol. 28, pp. 1–34). American Mathematical Society. https://doi.org/10.1090/pspum/028.1/9813

Hutton, T. J. (2002). Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, *8*(4), 341–356. https://doi.org/10.1162/106454602321202417, PubMed: 12650644

Hutton, T. J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In *Proceedings of the ninth international conference on the Simulation and synthesis of living systems (ALIFE9)* (pp. 444–449). MIT Press. https://doi.org/10.7551/mitpress/1429.003.0075

Hutton, T. J. (2010). Codd's self-replicating computer. *Artificial Life*, *16*(2), 99–117. https://doi.org/10.1162/artl.2010.16.2.16200, PubMed: 20067401

Hutton, T. J. (2015). *Two dimensional turing machines*. https://github.com/GollyGang/ruletablerepository/wiki/TwoDimensionalTuringMachines

Johnston, N., & Greene, D. (2022). *Conway's Game of Life: Mathematics and construction*. Nathaniel Johnston.

Kong, X. P., & Cohen, E. G. D. (1991a). Diffusion and propagation in triangular Lorentz lattice gas cellular automata. *Journal of Statistical Physics*, *62*(3), 737–757. https://doi.org/10.1007/BF01017981

Kong, X. P., & Cohen, E. G. D. (1991b). A kinetic theorist's look at lattice gas cellular automata. *Physica D*, *47*(1–2), 9–18. https://doi.org/10.1016/0167-2789(91)90273-C

Langton, C. G. (1984). Self-reproduction in cellular automata. *Physica D*, *10*(1–2), 135–144. https://doi.org/10.1016/0167-2789(84)90256-2

Langton, C. G. (1986). Studying Artificial Life with cellular automata. *Physica D*, *22*(1–3), 120–149. https://doi.org/10.1016/0167-2789(86)90237-X

Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall.

Nardini, J. T., Baker, R. E., Simpson, M. J., & Flores, K. B. (2021). Learning differential equation models from stochastic agent-based model simulations. *Journal of the Royal Society Interface*, *18*(176), Article 20200987. https://doi.org/10.1098/rsif.2020.0987, PubMed: 33726540

Nobili, R., & Pesavento, U. (1996). Generalised von Neumann's automata I: A revisitation. In E. Besussi & A. Cecchini (Eds.), *Artificial worlds and urban studies*. IUAV, Daest-Dipartimento di analisi economica e sociale del territorio.

Nowak, M. A., & May, R. M. (1992). Evolutionary games and spatial chaos. *Nature*, *359*(6398), 826–829. https://doi.org/10.1038/359826a0

Panait, L., & Luke, S. (2004). Ant foraging revisited. In *Proceedings of the ninth international conference on the Simulation and synthesis of living systems (ALIFE9)* (pp. 569–574). MIT Press. https://doi.org/10.7551/mitpress/1429.003.0096

Perrier, J.-Y., Sipper, M., & Zahnd, J. (1996). Toward a viable, self-reproducing universal computer. *Physica D*, *97*(4), 335–352. https://doi.org/10.1016/0167-2789(96)00091-7

Pesavento, U. (1995). An implementation of von Neumann's self-reproducing machine. *Artificial Life*, *2*(4), 337–354. https://doi.org/10.1162/artl.1995.2.4.337, PubMed: 8942052

Railsback, S. F., Lytinen, S. L., & Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *Simulation*, *82*(9), 609–623. https://doi.org/10.1177/0037549706073695

Ray, T. S. (1992). *Evolution, ecology and optimization of digital organisms* [Unpublished manuscript]. School of Life & Health Sciences, University of Delaware. https://faculty.cc.gatech.edu/~turk/bio_sim/articles/tierra_thomas_ray.pdf

Reggia, J. A., Armentrout, S. L., Chou, H.-H., & Peng, Y. (1993). Simple systems that exhibit self-directed replication. *Science*, *259*(5099), 1282–1287. https://doi.org/10.1126/science.259.5099.1282, PubMed: 17732248

Rucker, R. (1993). *Artificial Life lab*. Waite Group Press.

Sayama, H. (1999). Toward the realization of an evolving ecosystem on cellular automata. In *Proceedings of the fourth international Symposium on Artificial Life and robotics (AROB 4th'99)* (pp. 254–257). Science and International Affairs Bureau, Japan Ministry of Education, Culture, Sports, Science and Technology.

Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, *4*(3), 237–257. https://doi.org/10.1162/106454698568576, PubMed: 9864438

Sun, G.-Q., Jin, Z., Li, L., Haque, M., & Li, B.-L. (2012). Spatial patterns of a predator-prey model with cross diffusion. *Nonlinear Dynamics*, *69*(4), 1631–1638. https://doi.org/10.1007/s11071-012-0374-6

Tabachnikov, S. (2014). Dragon curves revisited. *Mathematical Intelligencer*, *1*(36), 13–17. https://doi.org/10.1007/s00283-013-9428-y

Thatcher, J. W. (1964). Universality in the von Neumann cellular model. In A. W. Burks (Ed.), *Essays on cellular automata* (pp. 132–186). University of Illinois Press.

Trevorrow, A., & Rokicki, T. (2023). *Golly: Game of Life simulator*. https://golly.sourceforge.io/

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Mathematics*, *58*(345-363), 58–90. https://doi.org/10.1093/oso/9780198250791.003.0005

von Neumann, J. (1966). *Theory of self-reproducing automata* (A. W. Burks, Ed.). University of Illinios Press.

Wolfram, S. (1983). Statistical mechanics of cellular automata. *Reviews of Modern Physics*, *55*(3), Article 601. https://doi.org/10.1103/RevModPhys.55.601

Yinusa, A. R., & Nehaniv, C. L. (2011). Study of inheritable mutations in von Neumann self-reproducing automata using the Golly simulator. In *2011 IEEE symposium on Artificial Life (ALIFE)* (pp. 211–217). MIT Press. https://doi.org/10.1109/ALIFE.2011.5954654

Appendices

Appendix 1: Equivalence Between Turmites and Cellular Automata

Cellular automata have been used in a wide variety of Artificial Life studies. Two-dimensional cellular automata and turmites both carry out computations at discrete time steps on a grid in a manner that is given by a rule table. It is tempting to consider turmites to be cellular automata, but this is not the case. Cellular automata update each cell in the grid at each time step, whereas turmites only update the grid at the location of the head. The rule table for a cellular automaton is based on the symbols in a neighborhood around a given cell. The rule table for a turmite does not take into account the values of neighboring cells but instead is based on the head's state as well as the single symbol at the head's current location.

Two-dimensional Turing machines and cellular automata are both capable of universal computation, and thus they are of equal computational power. Moreover, a machine of one of these classes can mimic the behavior of a machine from another one of these classes. Let us first consider how a cellular automaton C can mimic a turmite T. Each cell in the cellular automaton C must encode in its state the symbol from the corresponding grid cell of T. In addition, each cell of C must indicate whether the head of T is at that grid cell and, if so, the value of that head's state and direction. The cellular automaton C just requires the use of a von Neumann neighborhood (four neighbors). A major task of the rule table is for C to recognize when a given cell p is adjacent to another cell q where the virtual head is located and decide that the head should be moved instead to p. The von Neumann neighborhood gives enough information so that this task can be accomplished.

Let us count the number of states of a cellular automaton C that mimics the behavior of the **RL** vant. The state of each cell C must encode the color of each cell (two possible values) and must also encode the state of the head (one value, as is the case for all vants) and the direction of motion of the vant's head (four directions, plus one more to indicate no head equals five). Altogether, this means each cell of C can take on $2 \times 1 \times 5 = 10$ states. The number of entries in the rule table of a four-neighborhood cellular automaton with 10 values is $10^5 = 100,000$. This rule table could be reduced to perhaps a few thousand entries based on symmetries. For the more complicated turmite that creates the dragon curve, the cells of C would encode the number of cell symbols (7), the number of states (also 7), and the directions plus the presence or absence of a head (5). Each cell would have any of $7 \times 7 \times 5 = 245$ states. The transition table would have $245^5 = 882,735,153,125$ entries, prior to reduction based on symmetries.

Let us now turn to the task of cloning the behavior of a cellular automaton C by a turmite T. One way to approach this is to create a grid that contains one turmite head on each cell of the grid. Each turmite is responsible for visiting its four or eight neighbors and remembering their collective states in a manner that is encoded by the state of its head. Upon returning to its home cell, the turmite would then consult its transition table to decide what symbol to write. For a two-state totalistic cellular automaton such as Conway's Game of Life (Berlekamp et al., 2004; Gardner, 1970; Johnston & Greene, 2022), the turmite would only have to remember the number of live neighbors (0 to 8) that it has visited, for nine different possibilities. Such a turmite would need to take 10 steps to traverse all of the neighbors and return to its home cell, and each such step in its path would also have to be encoded in its state. Thus such a turmite would require $9 \times 10 = 90$ states. Because each cell can take on just one of two states, the rule table would have $2 \times 90 = 180$ entries. Even if a cellular automaton were to use a von Neumann neighborhood, it still would require 10 steps for a turmite to visit four neighbors and return to its starting position. To mimic Codd's eight-state, nontotalistic cellular automaton (Codd, 1968), a turmite would need to remember $8^4 = 4,096$ possible neighbor state combinations, requiring $10 \times 4,096 = 40,960$ states, and thus $8 \times 40,960 = 327,680$ entries in its rule table.

Another way to mimic the behavior of a cellular automaton C with a turmite is to make use of the universal constructor turmite from section 5. Let us assume for simplicity that the grid cells of C can take on nine or fewer states. The turmite would make use of three constructor arms to mimic

C. One of these arms would sweep over a region of cells that represent the grid of *C*. As this arm sweeps over the region, it examines and records the values of the five or nine cells for the current cell's neighborhood. For totalistic cellular automata, this simply means summing the values of the neighbors, plus recording the state of the center cell. The turmite then consults an encoded rule table of *C* with a second arm, which specifies the value to which the current cell should be changed. These new cell contents are then recorded by a third constructor arm in a grid that represents the next state for all the cells of *C*. When all the cells of the virtual grid of *C* have been examined, the updated cells are copied from the third arm's grid to the first's. If new nonempty cells have been created at the boundary of the virtual grid for *C*, then the bounds of the region over which arms 1 and 3 are swept should be widened.

Appendix 2: NESW Turmites

Let us return to the manner in which the transition function indicates a change in the direction of motion of a turmite. The actions that we use are relative to the current direction *d* of the turmite. We could instead consider 2-D Turing machines that make use of *absolute* instead of *relative* directions. That is, the rule table could indicate the new direction of motion explicitly. Instead of specifying the relative directions {*R*, *L*, *F*, *B*}, such a rule table would contain the new direction {*N*, *E*, *S*, *W*}, where NESW refer to the compass directions north, east, south, and west. When the rule table indicates an action of *E*, for instance, the turmite would face east and move one step in that direction, regardless of its previous direction. To disambiguate between relative and absolute directions, let us call 2-D Turing machines that use relative directions *RLFB turmites*, and let us call those that use absolute directions *NESW turmites*. Note that the machines that Alan Turing (1936) originally introduced used absolute directions, namely, west and east, although he referred to them as left and right. Both RLFB turmites and NESW turmites are capable of universal computation. Neither of these classes of machines is more powerful or expressive than the other.

For any given RLFB termite *R*, there exists a NESW turmite *N* that mimics the behavior of *R* exactly. Because *N* does not maintain information about the direction that its head is facing, this information must be encoded in the head's state. For this reason, *N* will require four times as many states as *R*, and the rule table for *N* will thus contain four rows for each individual row of the table for *R*. These four rows together encode the current direction of the head of the *R* turmite.

For any given NESW turmite *N*, there also exists an RLFB termite *R* that mimics the NESW turmite exactly. The RLFB termite can only make relative changes to its direction of motion, yet it must be able to turn to a particular direction (e.g., east) if the rule table for *N* indicates this. To do this, *R* will have four rows of its rule table for every one row of the table for *N*. The four rows of *R* essentially translate between the four possible directions that its head may be facing to each absolute direction specified by the rule table for *N*.

It may seem paradoxical that to mimic a turmite from one of these classes with the other requires four times the number of states, regardless of whether we are mimicking NESW with RLFB or RLFB with NESW. To resolve this apparent paradox, note that the sketches for how to mimic the behavior of one machine by the other give an automatic process for creating such a behavior clone. This automatic process requires multiplying the number of states by 4, but in certain circumstances, there are also behavior clones that will require fewer than 4 times the number of states. If we were to use the automatic process to clone the **RL** vant that gives us a NESW turmite *N*, and then use another automatic process to clone *N* in an RLFB turmite *R*, then *R* would have 16 states. But we know there is also a one-state RLFB turmite that also mimics the behavior of *N*, namely, the original **RL** vant.

Why have RLFB turmites garnered more attention than NESW termites? One reason is that one-state RLFB turmites (vants) have more interesting behaviors than one-state NESW turmites when placed on an empty grid. A one-state NESW turmite that is placed on an empty grid will always travel in a straight line in one of the four compass directions, based on which absolute

direction is indicated in the rule table entry for symbol **a** and State 1. In contrast, we have seen in section 3 that the behavior of vants that are placed on an empty grid can be quite complex.

Another reason to prefer using RLFB turmites over NESW turmites is when the phenomenon that is being studied does not distinguish between spatial directions. If we think of turmites as being creatures crawling on the plane, we would typically use RLFB turmites if the creatures had no preferred direction of motion. If instead certain directions are singled out, such gravity or direction to a light source, then NESW turmites might be the better choice. The examples of the dragon curve, the SIR model for disease spread, and self-reproduction all are easier to embody using relative directions of motion. For example, due to there being no preferred direction, we can create two different constructor arms for self-replication, one that points north and another that points south. The prime sieve, in contrast, might just as easily have been created using NESW turmites, because most of the turmites that it spawns carry out specific tasks along certain rows or columns.

Appendix 3: Self-Reproduction Details

Following is the full description of the cell pattern for the self-reproducing machine. This is the character-encoded version of the pattern shown in Figure 28. The top and bottom lines contain the upward- (A) and downward-pointing (B) arms. Near the top are the six counters that are used. The long column of **f** symbols to the left of the counters forms the funnel that directs the motion of the head down toward the column that contains the instruction pointer. Below the counters are the instructions for carrying out self-reproduction, and each instruction is annotated at the right.

```
f     f             hhhhhhi
      f                          c
      f
      f     c
      f     b   c
      f     b c b
      f     b b b
      f dg  b b b
      f   dgc c c
      f     dgb b
      f       dgb
      f         dg
      f           dg

fffcffe      f                    decrement 3          place arms
ggfgffe   f                       increment 2
ggfgffe                f          lengthen forearm A
gdfgffeg                          jump
gfcffffe       f                  decrement 4
gggffffe          f               increment 6
gggffffe                    i     lengthen forearm B
gdgffffeg                         jump
gffcffe          f                decrement 5
ggfgffe       f                   increment 4
ggfgffe                  i        lengthen B
gdfgffeg                          jump                 end placing arms
gffffcfe                      i   decrease B           start copying
ggggggge             f            increase A
ggggggge f                        increment 1
gggggdeg                          jump
ggggfce  f                        jump if zero or decrement 1
ggggggge                      i   increase B
```

```
ggggdgeg                          jump
ggggcfe    f                      jump if zero or decrement 2
gggggfe     f                     increment 3
gggggfe               f           shorten forearm A
gggggfe                     i     lengthen forearm B
gggdgfeg                          jump
ggdffce      f                    jump if zero or decrement 3
ggfgfge   f                       increment 2
ggfgfge                f          lengthen forearm A
ggfgfge                      i    shorten forearm B
ggfdfgeg                          jump
ggffcfe        f                  jump if zero or decrement 4
ggfffgfe        f                 increment 5
ggfffgfe                    i     shorten B
ggfffgfe                  f       lengthen A
gdffgfeg                          jump                end copying
gffffe                 f          lengthen forearm A  fix arm lengths
gcffffe    f                      decrement 2
ggfgffe     f                     increment 3
ggfgffe               f           shorten forearm A
ggfdffeg                          jump
gfcfffe            f              decrement 6
gfgfgfe       f                   increment 4
gfgfgfe                     i     shorten forearm B
gfgfdfeg                          jump
gcffffe          f                decrement 5
ggfffgfe   f                      increment 2
ggfffgfe                 f        shorten A
ggffdfeg                          jump
gfcffffe    f                     decrement 2
gfgfgfe         f                 increment 5
gfgfdfeg                          jump
gffffe               f            shorten forearm A
gfffcfeg                          jump forward one instruction
hffffh                         c  begin execution in new constructor
d     c
                        ihhhhhh c
```