Computability Equivalences and Trade-Offs in One-Dimensional Cellular Automata

Judy McDonald
Christopher Schwartz
Alison Warr

Summer Research Program
Oregon State University
Corvallis, OR 97331

August 19, 1987

Computability Equivalences and Trade-Offs in One-Dimensional Cellular Automata

The concept of computability is important when considering the equivalence of theoretical machines and attempting to differentate between competing models. The two-dimensional cellular automata has been shown to contain a Universal Turing machine with only two states and a nine neighbor rule (i.e. the Game of Life)[1]. The one-dimensional case has been shown capable of Turing machine computation[2], but equivalencies of computability among the different one-dimensional machines has not drawn much attention as of yet.

A one-dimensional cellular automaton is a recursive function phenomena and may be described as follows. Given a finite set of distinct positive integers (called the **state set**) $S = (s_1, s_2, \ldots, s_n)$ a function π is defined on the set S^* of all possible listings with repetition of the elements of S of some finite length , **l**, which is a constant. Now, $\pi : S^* \rightarrow S$ is defined for all lists of the form $(a_1, a_2, \ldots, a_l)$ where each a_i is also an element of the set S and not necessarily distinct. The number **l** is called the **window length**, each element s_i of S is a state and the number of elements, **n**, in S is termed the **number of states** of the machine (also written as **n**-states.) The function π is then applied to an infinite list (or row) of elements of S (See **figure 1**) that have relative position, that is a list that looks something like

$$\ldots, a_{j-2}, a_{j-1}, a_j, a_{j+1}, a_{j+2}, a_{j+3}, \ldots$$

where each a_j is an element of S for all integers j. <u>One</u> iteration of the **rule** will actually be an infinite number of applications of the function π of the form $\pi : (a_i, a_{i+1}, a_{i+2}, \ldots, a_{i+l}) \rightarrow b_j$ for all integers i where again b_j is in S for all integers j. π therefore is a function that maps from lists of elements of the state set S to a single element of the state set but does not necessarily have an inverse function π^{-1}, indeed for most cellular functions π there frequently does not exist an inverse. This implies that most cellular functions are an irreversable phenomena, that is given the first iteration list, it is impossible to determine the initial list. For an example of this see **plate 1** where some of the initial states

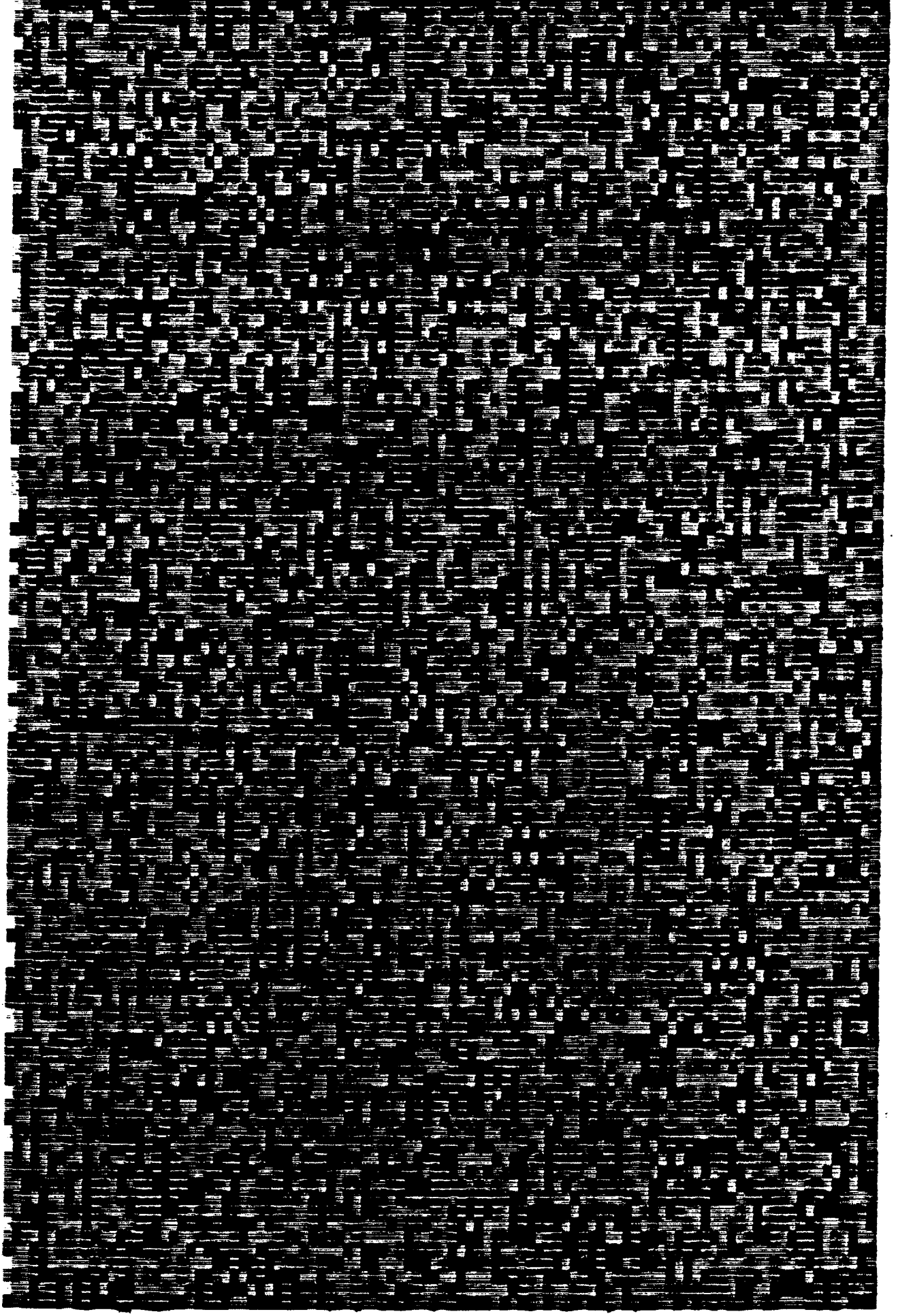


Plate 1: A typical example of a one-dimensional cellular automata with wraparound. This is a five-state, three window, sum mod three rule, with a random initial list.

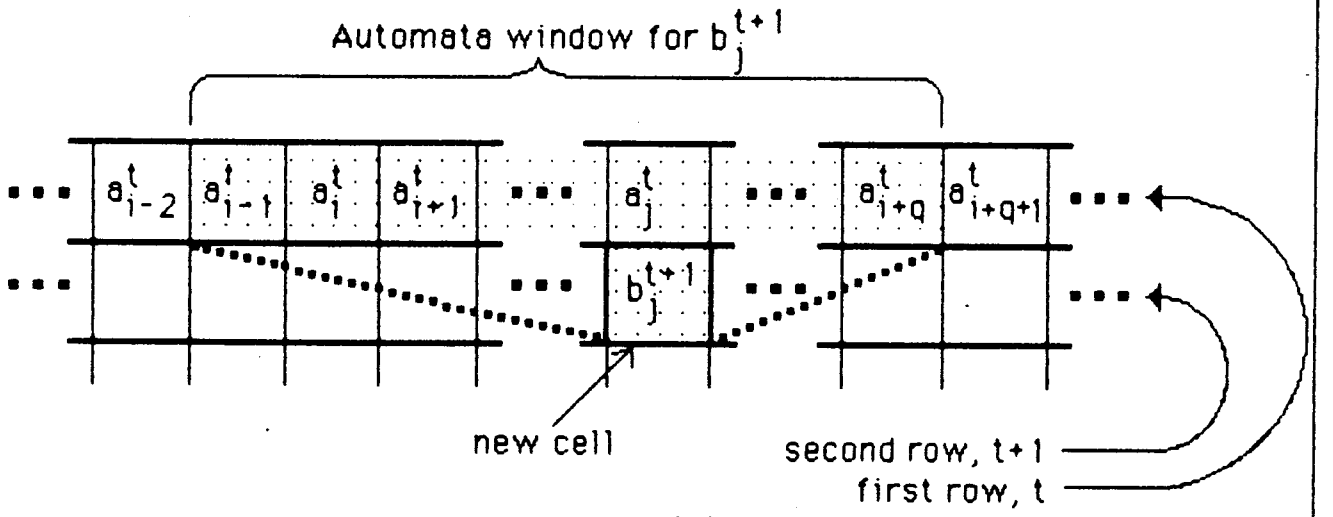
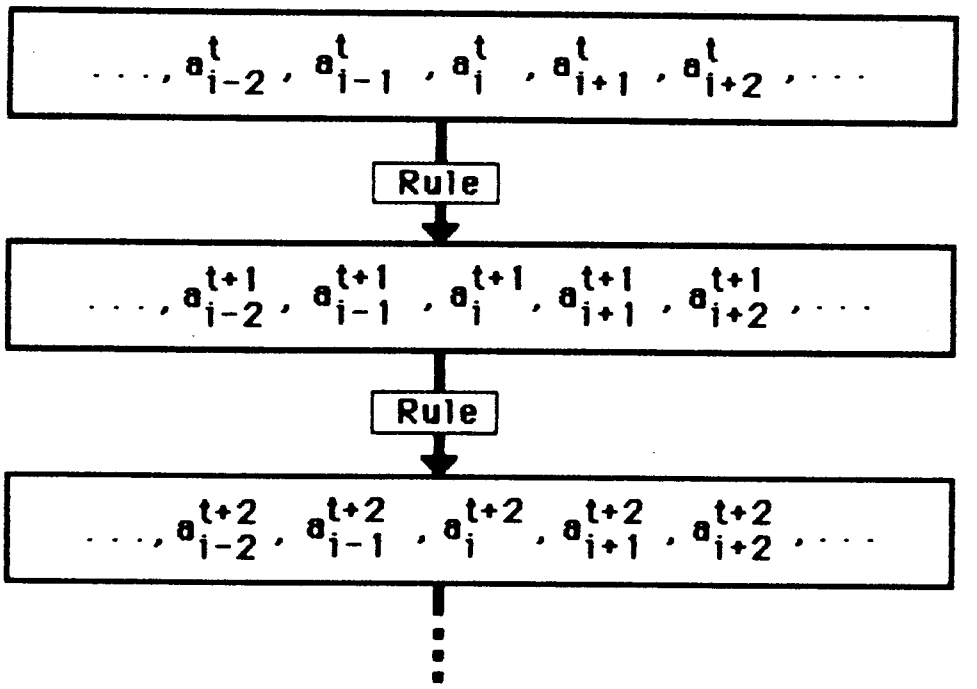


Figure 1. Computing one new cell, b_j^{t+1}, from the window cells, a_{i-1}^t through a_{i+q}^t for a total window length of q+1.



□ **Figure 2.** The generation of infinite lists by application of the rule.

actually disappear completely after the first iteration!

One application of the rule produces a second infinite list of the same form as the initial list and again comprised only of elements of **S**. One cell being formed in the second list is illustrated pictorally in **figure 2**.

The second iteration of the rule is then just the rule applied to the second list (i.e. the last list produced.) These iterations theoretically may continue indefinitely, each one producing a new infinite list from its predecessor list which for comparative purposes are best written top down with the initial list at the top and increasing iterations plotted downwards (see **figure 3**). This whole process will be referred to as a **machine**. The set of all machines with some particular number of states **m** and some window length **n** will be referred to as an automata class.

Usually, in the state set **S**, one paticular state is chosen to represent an off state or quiescent state, call this state by the symbol **0**. Often, limiting the number of non-quiescent states to a finite number in the initial list (or array) is one restriction imposed on the one-dimensional cellular automata. Another common restriction is in defining the conversion function such that it will not allow a cell to become non-quiescent with only quiescent cells in its input window, that is **f(0,0,0,...,0) = 0**. Hence, something cannot come from nothing, so both these restrictions are in the same spirit in that an infinite amount of information may not be generated in a finite number of iterations.

For visual comparisons and non-ideal modeling, the cellular automata may be programmed into a computer where different screen colors may represent different elements of the set **S** that contains all of the possible states of the paticular machine being modeled. Hopefully, the number of states of the machine is small enough so that distinguishable colors may be employed in the modeling process. Another, more severe restriction imposed by computer modeling is that of finite storage space. The cellular automata uses and produces an infinite 'database' of information and in this sense must always remain a theoretical construct assuming a finite amount of material in the universe. Nonetheless, important insights and testing may be achieved by a finite computing machine using fixed array sizes and wraparound technique in the computation process (For a pictoral representation of how wraparound functions see **figure 4**.) An example of a one-dimensional cellular automata computed with wraparound on a computer can be seen in **plate 1** and **plate 2**.

The question referred to above, probes the relations of machine equivalency in the one-dimensional case. All of the one-dimensional

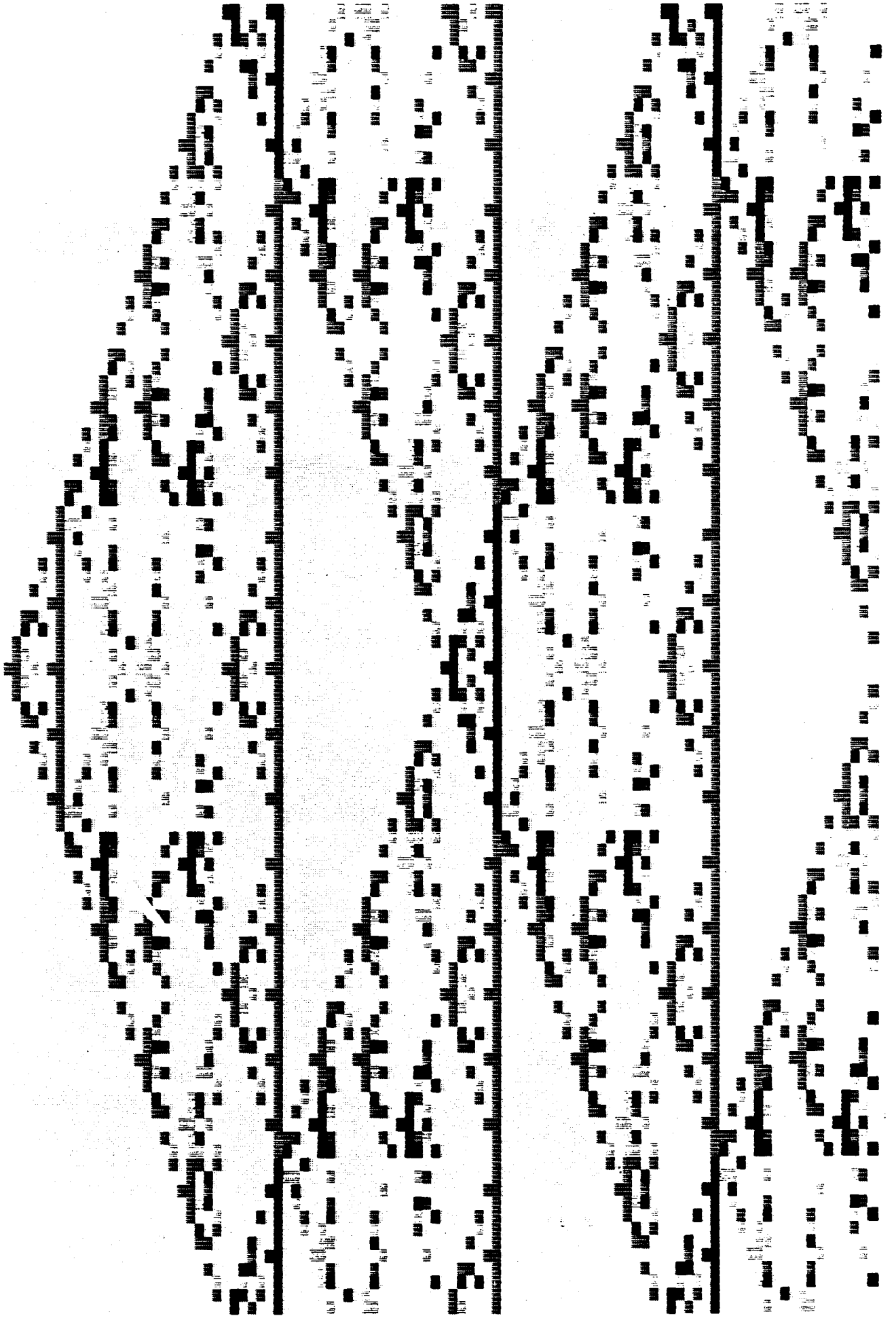


Plate 2: A typical example of a one-dimensional cellular automata with wraparound. This is a five-state, five window, sum mod five rule, with the initial list consisting of all null states except for one in the center.

Pictorial Representation of a Cellular Automaton

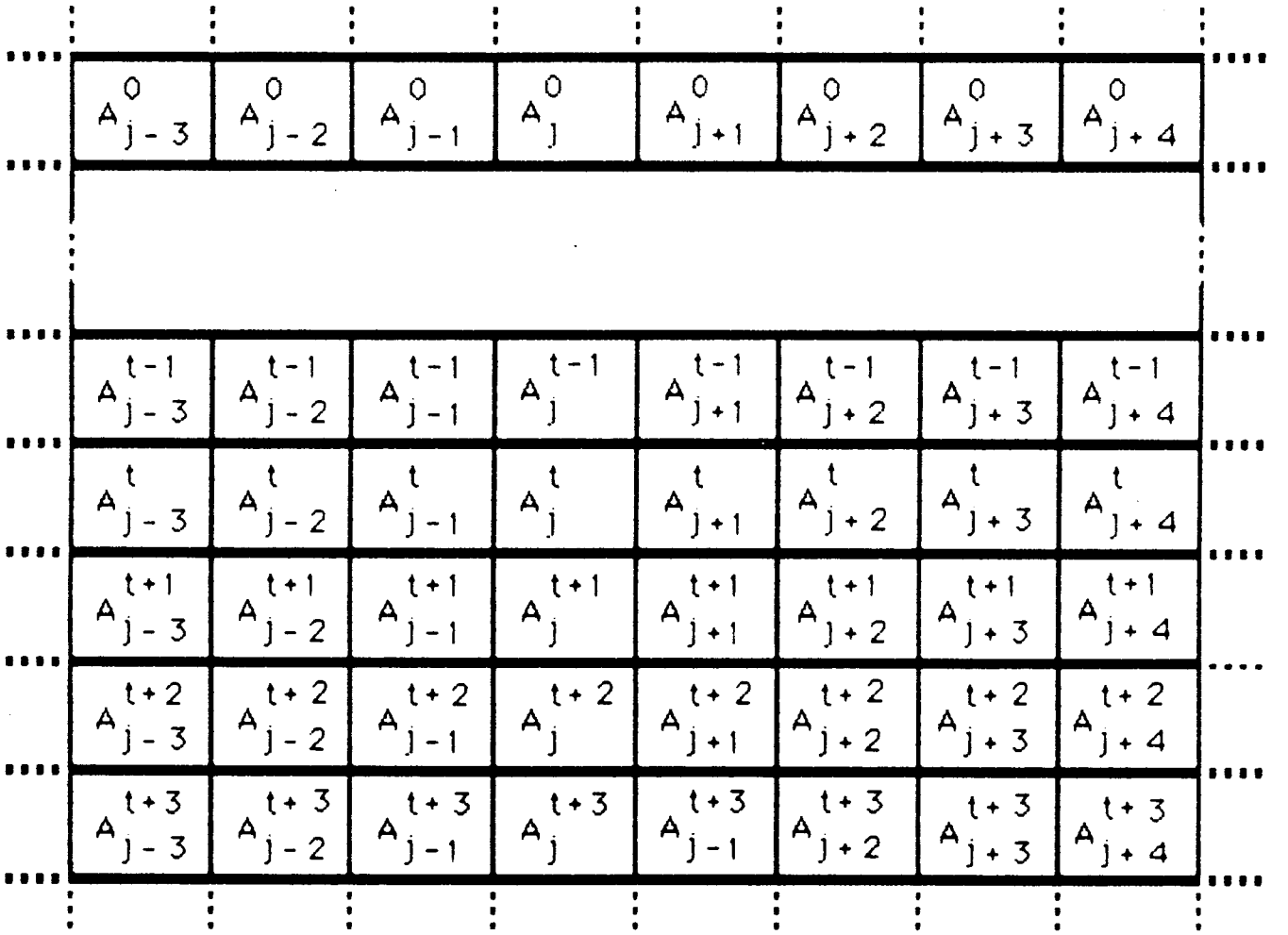
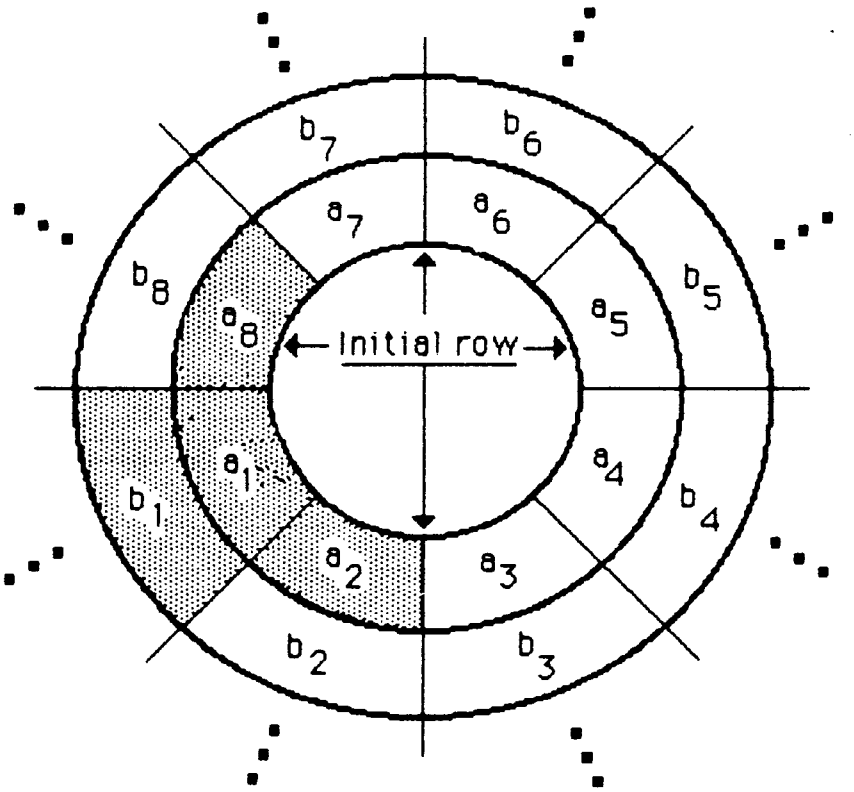


Figure 3



□ **Figure 4.** An illustration of the wraparound idea. Here, the initial array is of length eight and the window size is three. The shaded windows a_8, a_1, and a_2 are used to compute the cell b_1. This process cycles around the circle once for one iteration of the rule and then moves out one 'shell'.

cellular automata are capable of degrees of computation where this degree would seemingly depend upon the rule used, the number of states and the window length. In paticular, may these parameters be interchanged in any way and retention of computability be maintained? This is the question of simulation. Clearly, a working definition of computability for the automata is needed. First, however, some basis for comparison is needed hence we develop the idea of coverting some **n**-state list to an **m**-state list, the difficulty arising when either **n** > **m** or **m** < **n** so that one set has fewer symbols than needed forcing a conversion function that is not one-to-one.

Definition: A **conversion function,**

$$f:(a_j, a_{j+1}, \ldots, a_{j+l}) \rightarrow b_{k,j} \in (S_2 \cup \emptyset)$$

[where $a_j \in S_1$ for all integers j, $\emptyset$ is a special null symbol for no output (i.e. no mapping) and S_1 and S_2 are paticular state sets] is a function of fixed length list input, **l**, that upon application in any order to every string of that length converts one list in S_1 to a list in S_2. These lists are said to be **convertably the same** by the conversion function if the inverse conversion function exists as well.

Here the conversion is a function that, unlike the cellular function, maps to and from elements of possibly different state sets, and like the cellular function takes finite lists to single elements. The importance of the conversion function lies in the ability to compare strings of different state sets and thus lays the foundation from which we may simulate one machine by another by converting corresponding lists and checking whether they are convertably the same.

With these criteria to decide when two lists are or are not computably the same we start the question by looking at two and three state machines. In paticular, we look first at this conversion process. The interesting cases occur when the machines have a different number of states, for otherwise we have the trivial case of same-state machines where the conversion functions are obvious. The conversion function must be decided since it is the link between the two automata and is the reference by which computability equivalence is determined. Of importance is the possibility of ambiguity in the conversion, that is the conversion function must provide a list that does not depend upon the position of the starting point or order of conversion. In paticular, what is the smallest number of symbols (sequence length) of the two-state set

that are needed to have an unambiguous conversion function that maps to and from the three state set. This would imply that the conversion function f has an inverse f^{-1} and $f(f^{-1})(Q) = Q$ which is exactly the criteria needed to eliminate any possible existence of abiguity in the conversion function.

Proposition: The smallest sequence length of a conversion function with an inverse of the two state set to the three state set is at least four.

Proof: Clearly, in order to have a conversion from three states to two states a sequence of one is not enough, for all of the three-state set elements cannot be represented and therefore the inverse conversion f^{-1} does not exist. Furthermore, a sequence length of two is not enough, for with only two symbols in a sequence of two there exists no arrangement that separates the sequences. This leads to a sequence which is not translation invariant. That is to say, f^{-1} produces different list outputs dependent upon the starting position in the converted list.

To check the last possibility, assume there exists a conversion function from any three state list to any two state list such that each three state element is converted to a seqence of three two state elements

Again it is enough to show that any such conversion is not translation invariant. That is, a list of the two state elements could be converted into more than one list which are different.

It is clear, that in order for a conversion to be translation invariant there must be some sort of separating element, for otherwise there would be no way to distinguish the different blocks of three. Hence, we can reduce the number of possible conversions by only considering those conversions with the same first element, and those with the same second element, and those with the same third element. But, by the nature of the conversion function considering conversions with the same first element is equivalent to considering conversions with the same third element.

First let us consider the conversions with the same first element: Let a_1, a_2, a_3, be the three state elements.

Case 1: Conversion

a_1 --> 0 0 0

a_2 --> 0 0 1

a_3 --> 0 1 0

Subsequence	Conversion	Translation

$a_1a_2a_1$	0 0 0 0 0 1 0 0 0	a_1a_3

Case 2: Conversion

a_1 --> 0 0 0

a_2 --> 0 0 1

a_3 --> 0 1 1

Subsequence	Conversion	Translation
$a_1a_1a_3$	0 0 0 0 0 0 0 1 1	a_1a_2

Case 3: Conversion

a_1 --> 0 0 0

a_2 --> 0 1 0

a_3 --> 0 1 1

Subsequence	Conversion	Translation
$a_2a_1a_1$	0 1 0 0 0 0 0 0 0	a_1a_1

Case 4: Conversion

a_1 --> 0 0 1

a_2 --> 0 1 0

a_3 --> 0 1 1

Subsequence	Conversion	Translation
a_2a_3	0 1 0 0 1 1	a_1

Similarly, we can obtain the other four case consisting of a one in the first position by replacing each 0 with a 1 and each 1 with a 0 in the above four cases.

Now, let us consider the cases consisting of conversions with the same second element.

Case 1: Conversion

a_1 --> 0 1 0

a_2 --> 0 1 1

a_3 --> 1 1 0

Subsequence	Conversion	Translation
a_1a_3	0 1 **0** **1** **1** 0	a_2

Case 2: Conversion

a_1 --> 0 1 0

a_2 --> 1 1 0

a_3 --> 1 1 1

Subsequence	Conversion	Translation
a_3a_1	1 **1** **1** **0** 1 0	a_2

Case 3: Conversion

a_1 --> 0 1 0

a_2 --> 0 1 1

a_3 --> 1 1 1

Subsequence	Conversion	Translation
a_1a_3	0 1 **0** **1** **1** 1	a_2

Case 4: Conversion

a_1 --> 0 1 1

a_2 --> 1 1 0

a_3 --> 1 1 1

Subsequence	Conversion	Translation
a_3a_1	1 **1** **1** **0** 1 1	a_2

Similarly, we can obtain the other four case consisting of a zero in the second position by replacing each 0 with a 1 and each 1 with a 0 in the above four cases.

Therefore, since all the possible conversions of three state elements to two state elements, using a sequence of only three, are not translation invariant, we conclude that this sequence length of three is not enough. Hence, we need at least a sequence length of four. □

Due to these ambiguities then a seqence length of at least four is required for the existence of the inverse conversion function. Next it is shown that four is sufficient. The proof is by way of a construction.

Proposition: There exists a conversion function with an inverse with sequence length of four from 3-state representation to 2-state representation.

Proof: Consider the conversion:

Three-state	Two-state
0	1101
1	1001
2	0001

It suffices to show that given any combination of three-state symbols, the combination can be converted to a two-state. Given a set of symbols in the two-state representation, it can only be converted in one way to the three-state representation.

i. Case 0,0: 11011101

Subsequence	Translation
11011101	0 (the correct translation)
1**1011**101	no translation (Ø)
11**0111**01	no translation (Ø)
110**1110**1	no translation (Ø)
1101**1101**	0 (the correct translation)

ii. Case 0,1: 11011001

Subsequence	Translation
11011001	0 (the correct translation)
1**1011**001	no translation (Ø)
11**0110**01	no translation (Ø)
110**1100**1	no translation (Ø)
1101**1001**	1 (the correct translation)

iii. Case 0,2: 11010001

Subsequence	Translation
11010001	0 (the correct translation)
1**1010**001	no translation (Ø)
11**0100**01	no translation (Ø)
110**1000**1	no translation (Ø)
1101**0001**	2 (the correct translation)

iii. Case 1,0: 10011101

Subsequence	Translation
10011101	1 (the correct translation)
1**0011**101	no translation (Ø)
10**0111**01	no translation (Ø)
100**1110**1	no translation (Ø)
1001**1101**	0 (the correct translation)

iv. Case 1,1: 10011001

Subsequence	Translation
10011001	1 (the correct translation)
1**0011**001	no translation (Ø)
10**0110**01	no translation (Ø)
100**1100**1	no translation (Ø)
1001**1001**	1 (the correct translation)

v. Case 1,2: 10010001

Subsequence	Translation
10010001	1 (the correct translation)
1**0010**001	no translation (Ø)
10**0100**01	no translation (Ø)
100**1000**1	no translation (Ø)
1001**0001**	2 (the correct translation)

vi. Case 2,0: 00011101

Subsequence	Translation
00011101	2 (the correct translation)
0**0011**101	no translation (Ø)
00**0111**01	no translation (Ø)
000**1110**1	no translation (Ø)
0001**1101**	0 (the correct translation)

vii. Case 2,1: 00011001

Subsequence	Translation
00011001	2 (the correct translation)
0**0011**001	no translation (Ø)
00**0110**01	no translation (Ø)
000**1100**1	no translation (Ø)
0001**1001**	1 (the correct translation)

viii. Case 2,2: 00010001

Subsequence	Translation
00010001	2 (the correct translation)
0**0010**001	no translation (Ø)
00**0100**01	no translation (Ø)
000**1000**1	no translation (Ø)
0001**0001**	2 (the correct translation)

By construction then, there exists a conversion function with an inverse which has a sequence length of four. □

With the conversion function constructed for bridging the two-state and the three-state sets, an exploration of the comparable computability of machines of two- and three-states is in order since now the output lists may be compared. Returning to the idea of the equivalence of computability, the definition is stated allowing for each single iteration of the first machine some number of iterations of the mimicking machine before appropriate output is encountered, as long as for the number of iteration, **t**, on the first machine then the iteration number in the second machine may be defined by some function $\partial(t)$. The invertable conversion function is also required. Let **I** denote the integers.

Definition: Two different one-dimensional cellular automata are said to be **computably equivalent** if for any rule on either one machine then there exists a rule on the second machine, a conversion function with an inverse and a defined function (termed a **line number function**) $\partial: I \rightarrow I$ such that for each iteration, **t**, on the first machine the output list is convertably the same as the list of the $\partial(t)$ iteration on the second machine.

By **simulate**, it is meant that if one machine is computably equivalent to a second machine, then the first is said to simulate it. With the definition of computably equivalent, and the construction for the conversion function with sequence length four (above) for the two- and three-state machines, is it possible to model or simulate a three-state machine with a two-state? To answer this requires additionally that the window length, the initial list and the three state rule be specified or that possibly for any finite window length, any arbitrary three state rule and any initial list that there always exists a two-state machine that models this. The next result concerns a three-state machine with an arbitrary window length and showing that for any rule on this machine that there exists a two-state rule of some window length that is computably

equivalent.

Theorem1: Given any three-state one-dimensional cellular automata with window length d, there exists a two-state cellular automata with window length [2(3)-2+ (d-1)(3)] which is computably equivalent.

Proof: By construction:

Conversion from three-state to two-state:

three state	two state
0	1 1 0 1
1	1 0 0 1
2	0 0 0 1

[See **figure 5** for notation and insight]

Let $\ldots A^0_{i,0}, A^0_{i,1}, A^0_{i,2}, A^0_{i,3}, A^0_{i+1,0}, A^0_{i+1,1}, A^0_{i+1,2}, A^0_{i+1,3}, \ldots$ be a line of three-state code to which the conversion function has been applied resulting in the two-state code where $A^t_{i,j} \in \{0,1\}$ for all $i,t \in Z$ and $j \in \{0, 1, 2, 3\}$.

To determine the next line in the two-state cellular automata, $A^{t+1}_{i,2} = 0$ and $A^{t+1}_{i,3} = 1$ for all i, t. Therefore, we say there exists a "buffer block" of length two and this block separates all coded information and retains a translation invariant rule, i.e. the information will not "slide" as iterations are performed. Now, all that is needed to be shown is that the two-block left, call it the information block, may always be computed correctly in the two-state machine given any arbitrary three-state rule with window length d. This amounts to showing that 1) the two-block contains the information necessary, 2) each element of the two-block may be determined uniquely and 3) each element depends strictly on the coded information that should be used to compute it. To do this, we begin with a window of length one in the three-state cellular automata and show it is true here.

It is enough to show that given any window of length 2(3)-2+(d-1)(3) in the two-state cellular automata, we have the information to uniquely determine the window in the next row, and that the information needed is used.

Let d = the window length of the three-state cellular automata.

Notation: Label the cells in the following way:

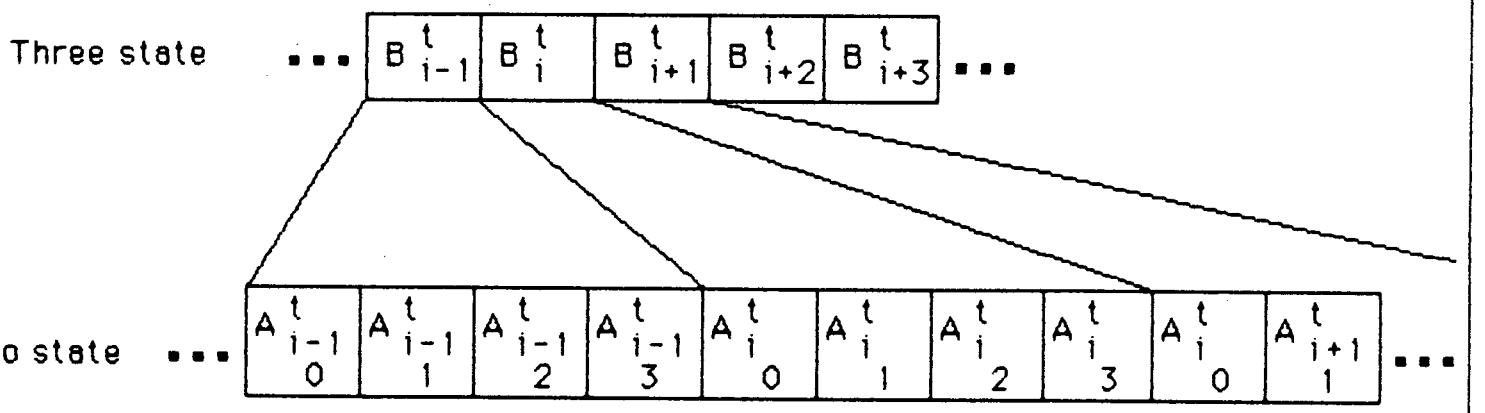


Figure 5

Let D = the window length of the two-state cellular automata.
Let i, j = the position you are at in the two-state cellular automata.
Let i = the position you are at in the three-state cellular automata.

Let d = 1. Then D = 2(3)-2+ (1-1)(3) = 4. For any given $A^t_{i,j}$:

Case 1: If j=0, then from our window we can see

$$A^t_{i-1,3},\ A^t_{i,0},\ A^t_{i,1},\ A^t_{i,2}$$

And $A^t_{i,0}, A^t_{i,1}$ along with the position of the buffer give all the information corresponding to the three-state cell B^t_i.

Case 2: If j=1, then from our window we can see

$$A^t_{i,0},\ A^t_{i,1},\ A^t_{i,2},\ A^t_{i,3}$$

And $A^t_{i,0}, A^t_{i,1}$ along with the position of the buffer give all the information corresponding to the three-state cell B^t_i.

Case 3: If j=2, then from our window we can see

$$A^t_{i,2},\ A^t_{i,3}$$

And $A^t_{i,2} = 0, A^t_{i,3} = 1$ for all i,t which tells us $A^{t+1}_{i,2} = 0$.

Case 4: If j=3, then from our window we can see

$$A^t_{i,2},\ A^t_{i,3}$$

And, $A^t_{i,2} = 0, A^t_{i,3} = 1$ for all i,t which tells us $A^{t+1}_{i,3} = 1$.

Assume this holds for k=d. Let k=d+1. Then D=2(3)-2+((d+1)-1)(3) = 3d +4.
Then, for any given $A^t_{i,j}$:

Case 1: If j=0, then by the induction hypothesis the window of

length 3d+1 encompasses

$$A^t_{i-1,3},\ A^t_{i,0},\ A^t_{i,1},\ \cdots,\ A^t_{i+d-1,1}$$

which gives all the information corresponding to the three-state cells

$$B^t_i,\ B^t_{i+1},\ B^t_{i+2},\ \cdots,\ B^t_{i+d-1}$$

Since D= 3d + 4, the window also includes

$$A^t_{i+d-1,2},\ A^t_{i+d-1,3},\ A^t_{i+d,0},\ A^t_{i+d,1}$$

thus giving the information corresponding to the last cell in the three-state d+1 window, B^t_{i+d}.

Case 2: If j=1, then an argument similar to case 1 holds.

Case 3: If j=2, then from our window we can see

$$A^t_{i,2},\ A^t_{i,3}$$

And $A^t_{i,2} = 0, A^t_{i,3} = 1$ for all i,t which tells us $A^{t+1}_{i,2} = 0$.

Case 4: If j=3, then from our window we can see

$$A^t_{i,2},\ A^t_{i,3}$$

And $A^t_{i,2} = 0, A^t_{i,3} = 1$ for all i,t which tells us $A^{t+1}_{i,3} = 1$.

Thus, for any three-state cellular automata of window length d there exists a two-state cellular automata of window length 4+(3)(d-1) = 3d + 1 which is computationally equivalent. □

The previous theorem shows that each three-state machines is computationally equivalent to a two-state machine that is to say all three-state machines can be simulated by two-state machines. Moreover, for this construction proof where the initial arrays for the two- and

three-state machines start at the same value of t, we actually have $\partial(t) = t$ in the definition form, which is actually as "nice" of a line number function as could be hoped for. Some color coded pictures of these can be seen in **plates 3 -** . Another encouraging result for the theorem above takes form in the following proposition where it is shown that not only does the conversion function used above bridge the computability equivalence between the two- and three-state machines but it has the smallest window length as well which is a lead in for the next topic: trade-off of states and symbols.

Proposition: The smallest window length for the two-state conversion of the three-state given by

three state	two state
0	1 1 0 1
1	1 0 0 1
2	0 0 0 1

where the three-state cellular automata has window length d is $D=2(3)-2+(d-1)(3)=3d+1$.

Proof: Assume d=1. Assume the smallest window length in the two-state is 3.

Suppose that there exist the following two subsequences in our two-state cellular automata:

1 1 0 1 **1 1 0** 1

1 1 0 1 **1 0 0** 1

We see that with the three window there exist two different circumstances where the window is the same. Hence we could not uniquely determine the rule for the two-state cellular automata.

Assume d=2. Assume the smallest window length in the two state is 6. In order to obtain necessary information we must have a window which looks back at least one. Consider the following subsequence of a two-state cellular automata.

?

1 1 0 1 **1** **1 0** 1 1 0 0 1

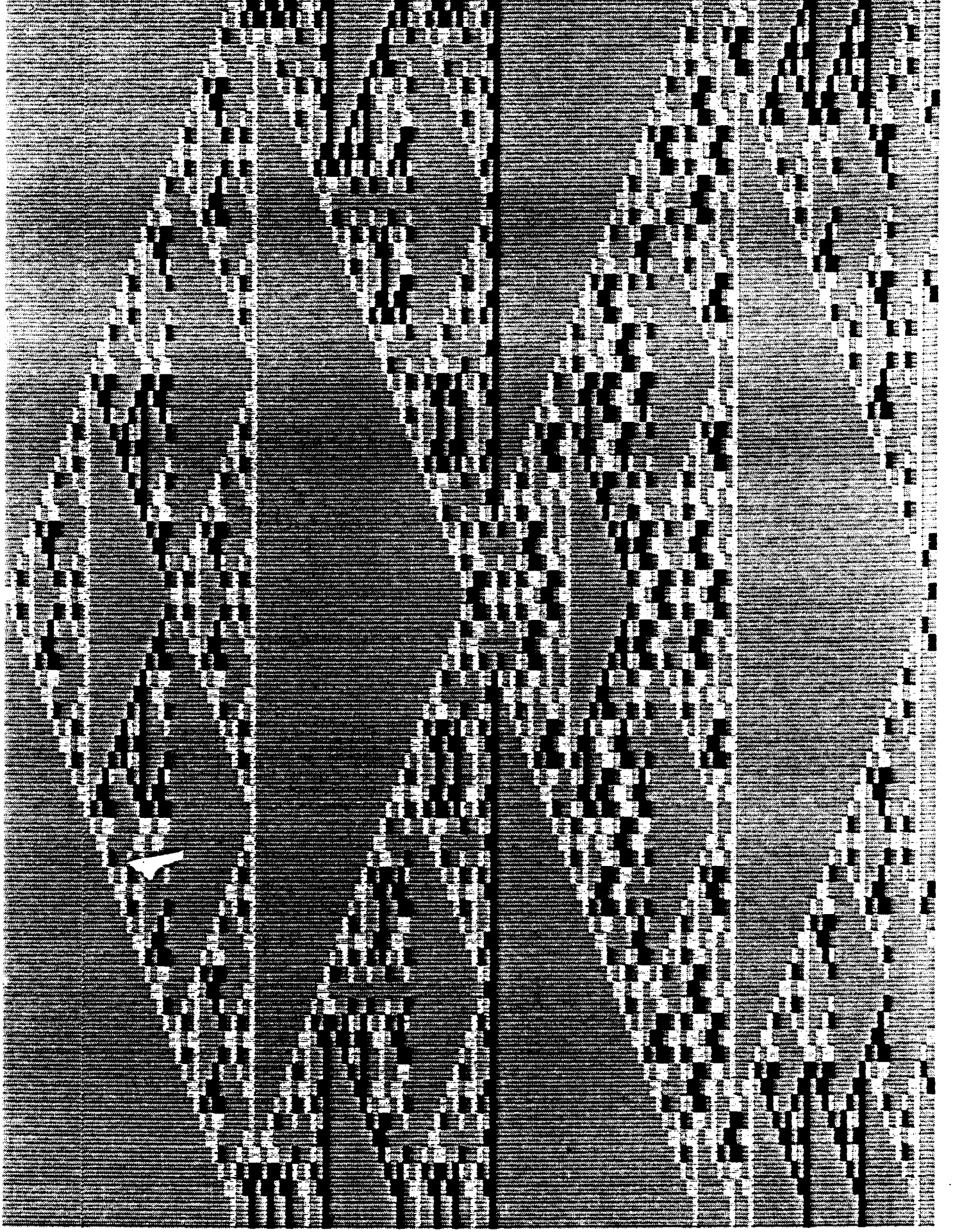


Plate 3: This is the three-state, three window, sum mod three, cellular automata.

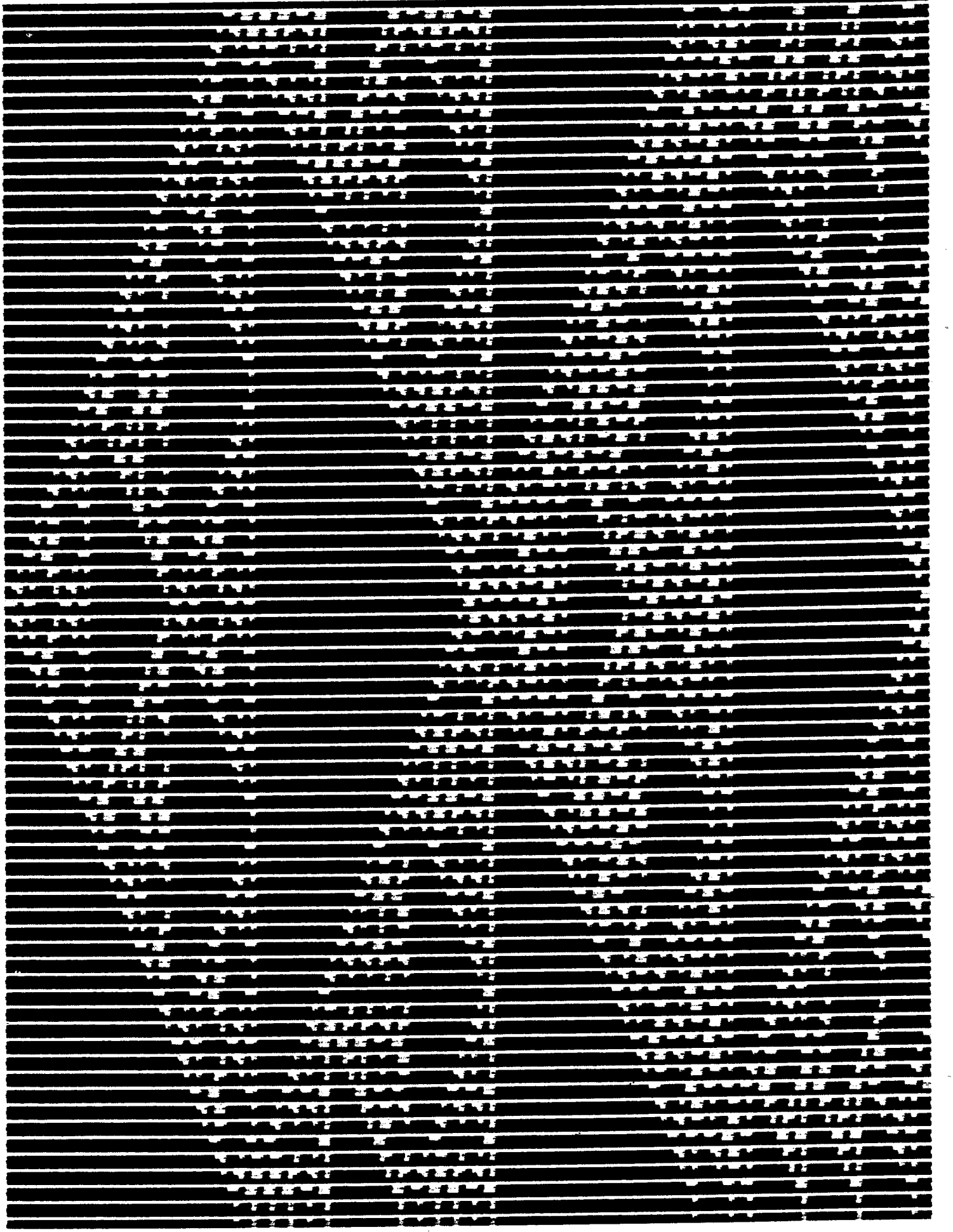


Plate 4: The two-state, ten window, cellular automata which simulates the three-state cellular automata pictured in plate 3.

Assume we at at the underlined position. Since we do not know the value in the position of the question mark (a information cell) we do not know the value of the corresponding three-state element and therefore are unable to apply the conversion function.

Similarly, for d=3,4,... we can show that there is not enough information given in a window of length (3d+1)-1.

Hence, we conclude that we need at least a window length of 3d+1. This together with the above theorem shows that a window of length 3d+1 is necessary to simulate a three-state cellular automata with a two-state cellular automata. □

Theorem2: Given any **n**-state one-dimensional cellular automata with window length d, there exists a two state one-dimensional cellular automata with window length [2(**n**)-2 + (d-1)**n**] which is computationally equivalent to the three state.

Proof: First construct the conversion function :

string length = **n**+1

$f(0) =$ 1 1 1 ... 1 0 1
$f(1) =$ 1 1 1 ... 1 0 0 1
.
.
.
$f(n-1) =$ 0 0 0 ... 0 0 1 .

The remainder of the proof is similar to the proof of the case with **n**=3.□

In the work done so far on simulating a three (or more) state cellular automata with a two state cellular automata we have considered only the cases where the window of the three state cellular automata has the form of a right hand window. But, as will be shown, any other justified window is the same as a right hand window by route of a simple shift. For example, if we have a left hand window of length two on the three state cellular automata and the following rule:

window	rule
0 0	1
0 1	0
0 2	0
1 0	2
1 1	1
1 2	0
2 0	1
2 1	2
2 2	0

And with the following initial sequence:

initial sequence 0 1 0 0 2 0 1 0 2 2 2 0 0 0 0 1 1 2 0 1 0 2 0 2 1

Then we obtain with the left hand rule (with wrap-around):

left hand rule 1 0 2 1 0 1 0 2 0 0 0 1 1 1 1 0 1 0 1 0 2 0 1 1 2

Using a right hand rule (with wrap-around) we would obtain:

right hand rule 0 2 1 0 1 0 2 0 0 0 1 1 1 1 0 1 0 1 0 2 0 1 1 2 1

It can be seen that by shifting the list generated by the right hand rule by one to the right the list generated by the left hand rule is obtained. This tidy little result is due to the invarient form of the conversion function and the properties of its inverse (i.e it exists and works unambiguously.) Therefore, the results obtained thus far for windows of any justification apply universally.

In the case of the Turing machine, two criteria are of importance. For each such machine the number of symbols and the number of states are determined. It has been found that a Universal Turing machine exists with just two states but an arbitrary number of symbols and alternatively there exists one with only two symbols but an arbitrary number of states. This seems to imply a natural trade-off between states and symbols in coding up a Universal Turing machine.

Can a similar trade-off be found for one-dimensional cellular automata? At this point any machine with a finite number of symbols and a finite window length may be simulated by a two-state machine. This represents the reduction of states to a theoretical minimum, for with one

state only, no distinction can be made between quiescent and non-quiescent states so that no computation is possible. The opposite trade-off takes a machine with a finite number of states and a finite window size and codes it into a machine with some number of states with a window length of only two. The following theorem helps with this task.

Theorem: Given any 2-state, n-window cellular automata with rule F, there exists an equivalent m-state, 2-window cellular automata with rule G, where $m = 2^n + 2^{n-1} + 2^{n-2} + \ldots + 2^1$. In this equivalence, row t of the 2-state cellular automata matches row t*n of the m-state cellular automata.

The proof of this theorem is by induction. It is helpful, however, to first prove a specific case of the theorem .

Specific Case of the Theorem: Given any 3-window, 2-state cellular automata with rule F, there exists an equivalent 2-window, 14-state cellular automata with rule G.

Proof of the Specific Case:

Assume the 2-state cellular automata has a right-hand window.

Let $T = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ where

$t_0 = 000$

$t_1 = 001$

$t_2 = 010$

$t_3 = 011$

$t_4 = 100$

$t_5 = 101$

$t_6 = 110$

$t_7 = 111.$

Let $D = \{d_0, d_1, d_2, d_3\}$ where $d_0 = 00$

$d_1 = 01$

$d_2 = 10$

$d_3 = 11.$

Let $S = \{s_0, s_1\}$ where $s_0 = 0$

$s_1 = 1.$

Let $V = T \cup D \cup S.$

Let A^0_i = cell i of the initial row of the 3-window, 2-state cellular automata.

Let B^0_i = cell i of the initial row of the 2- window, 14-state cellular automata.

Given the initial row of a 3-window, 2-state cellular automata, let $B^0_i = A^0_i$.

The rule for the 14-state cellular automata is the following. In general terms, $G: V \times V \rightarrow V$. $G(B^t_i, B^t_{i+1}) = B^{t+1}_i$

Independent of the 2-state rule, the following portions, which are really conversions, of the 14-state rule can be defined. **Table 1** gives an illustration of this.

$G(s_0,s_0) = d_0$
$G(s_0,s_1) = d_1$
$G(s_1,s_0) = d_2$
$G(s_1,s_1) = d_3$

(Conversion from a pair of symbols, each representing 1 cell, to a symbol representing a block of 2 cells)

$G(d_0,d_0) = t_0$
$G(d_0,d_1) = t_1$
$G(d_1,d_2) = t_2$
$G(d_1,d_3) = t_3$
$G(d_2,d_0) = t_4$
$G(d_2,d_1) = t_5$
$G(d_3,d_2) = t_6$
$G(d_3,d_3) = t_7$

(Conversion from a pair ofsymbols, each representing a block of 2 cells, to a symbol representing a block of 3 cells)

Depending on the 2-state rule, the remaining portions of G can be defined. Given the 2-state cellular automata rule F:

$F(0,0,0) = a_0$
$F(0,0,1) = a_1$
$F(0,1,0) = a_2$
$F(0,1,1) = a_3$

EQUIVALENT BLOCK OF 3 CELLS IN ROW 0	EQUIVALENT SYMBOL PAIRING IN ROW 1	EQUIVALENT SYMBOL (REPRESENTING A BLOCK OF 3 CELLS) IN ROW 2
0 0 0	d_0 d_0	t_0
0 0 1	d_0 d_1	t_1
0 1 0	d_1 d_2	t_2
0 1 1	d_1 d_3	t_3
1 0 0	d_2 d_0	t_4
1 0 1	d_2 d_1	t_5
1 1 0	d_3 d_2	t_6
1 1 1	d_3 d_3	t_7

Table 1

REPRESENTION OF CONVERSION

One set of conversions are from a pair of symbols, each representing one cell to a symbol representing a block of 2 cells.(This is column 1 to column 2.) For example, in the shaded row, 10 is converted to d_2 ; 01 is converted to d_1 .There are also conversions from a pair of symbols, each representing a block of two cells, to a symbol representing a block of 3 cells. (This is column 2 to column 3). For example, in the shaded row, d_1 d_2 is converted to t_5. Note that t_5 corresponds to 101 , which is as is should be.

$F(1,0,0) = a_4$

$F(1,0,1) = a_5$

$F(1,1,0) = a_6$

$F(1,1,1) = a_7$ where $a_i \in \{0,1\}$ for i= 0, ... ,7.

Define the 14-state rule as follows.

$G(t_0, t_i) = a_0 \in S$

$G(t_1, t_i) = a_1 \in S$

$G(t_2, t_i) = a_2 \in S$

$G(t_3, t_i) = a_3 \in S$

$G(t_4, t_i) = a_4 \in S$

$G(t_5, t_i) = a_5 \in S$

$G(t_6, t_i) = a_6 \in S$

$G(t_7, t_i) = a_7 \in S$ for i=0, ... ,7.

In summary, read B^t_i and B^t_{i+1}. Then,

If $B^t_i \in S$, then $B^{t+1}_i = d \in D$, according to G.

If $B^t_i \in D$, then $B^{t+1}_i = t \in T$, according to G.

If $B^t_i \in T$, then $B^{t+1}_i = s \in S$, according to G.

Figure 6 illustrates how the 14-state cellular automata works.

In this way, at time t, $A^t_i = B^{3*t}_i$. So the two cellular automata are equivalent.

So given any 3-window two-state cellular automata with rule F, there exists an equivalent 2-window 14-state cellular automata with rule G.

And now the inductive proof of the theorem follows.

Proof:

Let A^t_i represent the i^{th} cell of the 2-state cellular automata at time t.

Let B^t_i represent the i^{th} cell of the m-state cellular automata at time t.

Let n=1.

Given a 2-state, 1-window cellular automata, does there exists a

Schematic Representation of the Two Window Fourteen State Cellular Automata

Initial row of 14 state (Equivalent to initial row of two state)

Equivalent to row 2 of two state

Equivalent to row 3 of two state

Equivalent to row 4 of two state

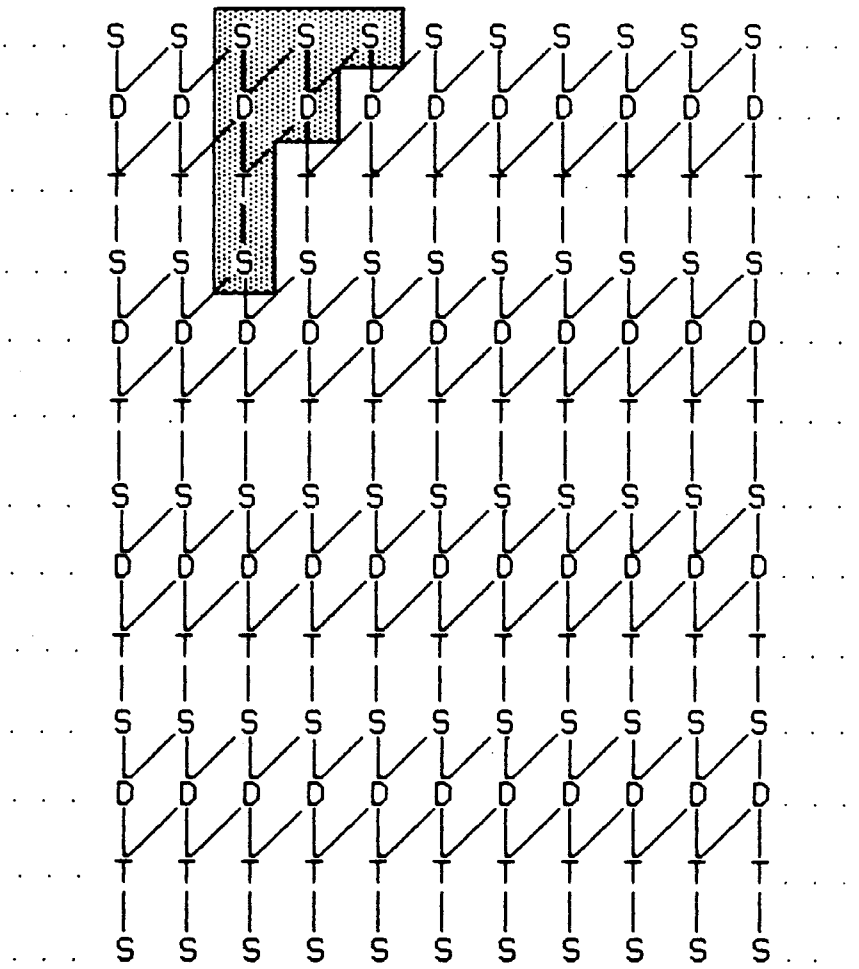


Figure 6

2-state 2 window cellular automata?

Given the 2-state 1-window cellular automata rule F as follows

$F(0) = a_0$

$F(1) = a_1$, where $a_0, a_1 \in \{0,1\}$

Create the 2-state, 2-window cellular automata rule G as follows

$G(0,0) = a_0$

$G(0,1) = a_0$

$G(1,0) = a_1$

$G(1,1) = a_1$.

Let $A^0_i = B^0_i$.

So there is an equivalent 2-state, 2 window cellular automata.

Assume that given any 2-state, (k-1)-window cellular automata with rule F, there exists an equivalent m-state, 2-window cellular automata with rule G.

Let n= k.

By assumption, we have the conversions:

from pairs of symbols, each representing a block of 1 cell, to a symbol representing a block of 2 cells;

from pairs of symbols, each representing a block of 2 cells, to a symbol representing a block of 3 cells;

from pairs of symbols, each representing a block of 3 cells, to a symbol representing a block of 4 cells;

.

.

.

from pairs of symbols, each representing a block of (k-2) cells, to a symbol representing a block of (k-1) cells.

Consider the conversion from pairs of symbols, each representing a block of (k-2) cells, to a symbol representing a block of (k-1) cells. Let $gamma_i$ represent the (k-2)-digit binary equivalent of the base ten number i, where $i=0,1,.. 2^{k-2}-1$. Let rho_i be the symbol corresponding to the (k-1)-length binary conversion of the base ten number i, where $i=0,...,2^{k-1}-1$. Explicitly, the conversion can be expressed by

$$G(gamma_{(idiv2)}, gamma_{(imod2^{k-2})}) = rho_i.$$

Table 2 illustrates this conversion.

We can create the conversion from pairs of symbols ε { rho0, rho1,..., rho $_{2^{k-1}-1}$ } in a similar way. Let phi $_i$ is the symbol corresponding to the k-length binary conversion of the base ten number i, where i=0,...,2k-1. Explicitly, the conversion is

$$G(\text{rho}_{(i \text{div} 2)}, \text{rho}_{(i \bmod 2^{k-1})}) = \text{phi}_i.$$

Table 3 illustrates this conversion.

All that remains is to define the portions of the rule G which depend on the rule F.

Given the original rule F:

$F(0,0,0,\ldots,0,0,0) = a_0$

$F(0,0,0,\ldots,0,0,1) = a_1$

$F(0,0,0,\ldots,0,1,0) = a_2$

.

.

.

$F(1,1,1,\ldots,1,1,1) = a_{2^k-1}$

Create the additional portion of the rule G in the following way.

$G(\text{rho}_0, \text{rho}_i) = a_0$

$G(\text{rho}_1, \text{rho}_i) = a_1$

$G(\text{rho}_2, \text{rho}_i) = a_2$

.

.

.

$G(\text{rho}_{2^{k-1}}, \text{rho}_i) = a_{2^{k-1}}$, for $i=0,\ldots,2^{k-i}$.

So for any 2-state, n-window cellular automata, there exists an equivalent m-state, 2-window cellular automata, where $m = 2^n + 2^{n-1} + 2^{n-2} + \ldots + 2^1$ and row t of the 2-state cellular automata matches row t*n of the m-state cellular automata.

EXAMPLE: The following is an example of how a 2-state, 3-window cellular automata can be converted to a 14-state 2-window cellular automata. Given the 2-state, 3-window (right-hand window) rule F:

EQUIVALENT BLOCK OF (K-1) CELLS IN ROW 0	EQUIVALENT SYMBOL PAIRING IN ROW K-3	EQUIVALENT SYMBOL (REPRESENTING A BLOCK OF (K-1) CELLS) IN ROW K-2
k - 2 : 000...000 0	$\gamma_0\ \gamma_0$	P_0
k - 3 : 000...000 01	$\gamma_0\ \gamma_1$	P_1
k - 3 : 000...000 10	$\gamma_1\ \gamma_2$	P_2
k - 3 : 000...000 11	$\gamma_1\ \gamma_3$	P_3
⋮	⋮	⋮
k - 2 : 111...111 1	$\gamma_{2^{k-2}-1}\ \gamma_{2^{k-2}-1}$	$P_{2^{k-1}-1}$

Table 2

REPRESENTATION OF CONVERSIONS

The conversions are from a pair of symbols, each representing a block of (K-2) cells to a symbol representing a a block of (K-1) cells. Take, for example, the shaded table entry. The first (K-2) symbols in column 1 correspond to γ_0. Now shift to the right 1 symbol; this block of (k-2) symbols corresponds to γ_1.

Now consider the pair $\gamma_0\ \gamma_1$. This pair is converted to P_1. Note that P_1 corresponds to the block of (K-1) cells in the first column, which is as it should be.

EQUIVALENT BLOCK OF K CELLS IN ROW 0	EQUIVALENT SYMBOL PAIRING IN ROW K-2	EQUIVALENT SYMBOL (REPRESENTING A BLOCK OF K CELLS) IN ROW K-1
k - 2 0 000...000 0	$P_0\ P_0$	Φ_0
k - 3 0 000...000 01	$P_0\ P_1$	Φ_1
k - 3 0 000...000 10	$P_1\ P_2$	Φ_2
k - 3 0 000...000 11	$P_1\ P_3$	Φ_3
⋮	⋮	⋮
k - 2 1 111...111 1	$P_{2^{k-1}-1}\ P_{2^{k-1}-1}$	Φ_{2^k-1}

Table 3

REPRESENTATION OF CONVERSIONS

The conversions are from a pair of symbols, each representing a block of (K-1) cells to a symbol representing a a block of (K) cells. Take, for example, the shaded table entry. The first (K-1) symbols in column 1 correspond to P_1. Now shift to the right 1 symbol; this block of (K-1) symbols corresponds to P_2.

Now consider the pair $P_1\ P_2$. This pair is converted to Φ_2. Note that Φ_2 corresponds to the block of (K) cells in the first column, which is as it should be.

0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	1
1 1 1	1

An arbitrary initial row and three applications of the rule (with wrap-around) give the following.

```
1 0 1 1 0 1 0 0 1 0 1 1 1 0  Row 0
0 0 1 0 0 1 1 0 0 0 1 1 0 0  Row 1
1 0 1 1 0 1 1 1 1 0 1 1 1 1  Row 2
0 0 1 0 0 1 1 1 0 0 1 1 1 1  Row 3
```

The rule F can be converted to the 2-window (right-hand window), 14-state rule G which follows.

0 0	|	d_0	d_0 d_0	|	t_0	t_0 t_i	|	1					
0 1	|	d_1	d_0 d_1	|	t_1	t_1 t_i	|	1					
1 0	|	d_2	d_1 d_2	|	t_2	t_2 t_i	|	0					
1 1	|	d_3	d_1 d_3	|	t_3	t_3 t_i	|	0					
			d_2 d_0	|	t_4	t_4 t_i	|	1					
			d_2 d_1	|	t_5	t_5 t_i	|	0					
			d_3 d_2	|	t_6	t_6 t_i	|	1					
			d_3 d_3	|	t_7	t_7 t_i	|	1	for i=0,...,7				

The identical initial row and nine applications of rule G (with wrap-around) give the following.

```
1   0   1   1   0   1   0   0   1   0   1   1   1   1      Row 0
d2  d1  d3  d2  d1  d2  d0  d1  d2  d1  d3  d3  d2  d1     Row 1
```

t_5	t_3	t_6	t_5	t_2	t_4	t_1	t_2	t_5	t_3	t_7	t_6	t_5	t_2	Row 2
0	0	1	0	0	1	1	0	0	0	1	1	0	0	Row 3
d_0	d_1	d_2	d_0	d_1	d_3	d_2	d_0	d_0	d_1	d_3	d_2	d_0	d_0	Row 4
t_1	t_2	t_4	t_1	t_3	t_6	t_4	t_0	t_1	t_3	t_6	t_4	t_0	t_0	Row 5
1	0	1	1	0	1	1	1	1	0	1	1	1	1	Row 6
d_2	d_1	d_3	d_2	d_1	d_3	d_3	d_3	d_2	d_1	d_3	d_3	d_3	d_3	Row 7
t_5	t_3	t_6	t_5	t_3	t_7	t_7	t_6	t_5	t_3	t_7	t_7	t_7	t_6	Row 8
0	0	1	0	0	1	1	1	0	0	1	1	1	1	Row 9

Note that row 0 of the 14-state cellular automata and row 0 of the 2-state match; row 3 and row 1; row 6 and row 2; row 9 and row 3, respectively.

This reduces two-state machines of any window length to machines with only two windows that are computably equivalent with the appropriate increase in the number of states needed in the new machine. The two-state automata prove to be the pivoting ground from which we may twist from higher state, multiple window machines to two-state then back to higher states with only two windows. The following corollary ties this up.

Corollary: For every **m**-state, **n**-window one-dimensional cellular automata there exists a 2-window, **p**-state one-dimensional cellular automata that is computably equivalent.

Proof: Given any **m**-state, **n**-window one-dimensional automata, **theorem 2** yields a 2-state machine with window length $[2(\mathbf{m})-2 + (\mathbf{n}-1)\mathbf{m}] = \mathbf{y}$, and this machine is computably equivalent to the origional. With this new machine apply **theorem 3** to it to get a one-dimensional cellular automata with window length 2 and the number of states being

$$p = \sum_{k=1}^{y} 2^k$$

where **y** is the number of windows in the first converted two-state. This machine by the theorem is computably equivalent to the two-state but this impies that it is computably equivalent to the origional by the nature of

the functional inverses. Hence, by construction there exists a computably equivalent two-window machine. □

In conclusion, the one-dimensional cellular automata displays characteristics similar to those of the Universal Turing machine in as far as the automata can be shown to exibit a trade-off phenomena for the number of states and the length of the window. In particular, any one-dimensional cellular automaton with finite states and finite window length may be simulated by an automaton with just two states or by an automaton with a window length of just two.

That the two-state machine was the connecting point for these two different forms of computing should not come as a large surprise, for the simple analogy is easily drawn between the modern binary computer and the computational power it contains (i.e. the ability to code in arbitrary programs.)

Another important point to consider for future inquiries, is that of equality of computing power. It has been shown here that the two-state contains the computing power of any higher state cellular automaton, it is obvious that this containment runs the other direction, for given any higher state machine, simply do not use all but two of the symbols to run the equivalent of the two-state machine. It would then seem natural that all the finite-state, finite window length automata have the potential of equal computing power. Continuing in this vein, if any particular automaton can be shown capable of Universal Computation, then every **m**-state, **n**-window automata class contains a Universal Turing machine.

Acknowledgements

We are grateful to several people that provided important motivation, ideas, direction and encouragement for this research project, in particular Dr. B. Burton, Dr. P. Cull, and Dr. R. Robson.

References

[1] S. Wolfram, "Universality and complexity in cellular automata", Physica 10D (1984)

[2] Paul Cull, Department of Computer Science, Oregon State University.